River Publishers Series in Information Science and Technology

Digital System Design — Use of Microcontroller

Dawoud Shenouda Dawoud
R. Peplow

River Publishers

Digital System Design — Use of Microcontroller

RIVER PUBLISHERS SERIES IN SIGNAL, IMAGE & SPEECH PROCESSING

Volume 2

Consulting Series Editors

Prof. Shinsuke Hara
Osaka City University
Japan

The Field of Interest are the theory and application of filtering, coding, transmitting, estimating, detecting, analyzing, recognizing, synthesizing, recording, and reproducing signals by digital or analog devices or techniques. The term "signal" includes audio, video, speech, image, communication, geophysical, sonar, radar, medical, musical, and other signals.

- Signal Processing
- Image Processing
- Speech Processing

For a list of other books in this series, see final page.

Digital System Design — Use of Microcontroller

Dawoud Shenouda Dawoud
R. Peplow

University of Kwa-Zulu Natal

River Publishers

Routledge
Taylor & Francis Group
LONDON AND NEW YORK

Published 2010 by River Publishers
River Publishers
Alsbjergvej 10, 9260 Gistrup, Denmark
www.riverpublishers.com

Distributed exclusively by Routledge
4 Park Square, Milton Park, Abingdon, Oxon OX14 4RN
605 Third Avenue, New York, NY 10017, USA

Digital System Design — Use of Microcontroller / by Dawoud Shenouda Dawoud, R. Peplow.

Routledge is an imprint of the Taylor & Francis Group, an informa business

ISBN 978-87-92329-40-0 (print)

Dedication

To Nadia, Dalia, Dina and Peter

D.S.D

To Eleanor and Caitlin

R.P.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Preface

Electronic circuit design is not a new activity; there have always been good designers who create good electronic circuits. For a long time, designers used discrete components to build first analogue and then digital systems. The main components for many years were: resistors, capacitors, inductors, transistors and so on. The primary concern of the designer was functionality however, once functionality has been met, the designer's goal is then to enhance performance.

Embedded system design is not just a new label for an old activity; embedded system designers today face new challenges. They are asked to produce increasingly complex systems using the latest technologies, but these technologies are changing faster than ever. They are asked to produce better quality designs with a shorter time-to-market. They are asked to implement functionality but more importantly to satisfy numerous other constraints. It is not enough, for example, to have a mobile phone that you can communicate with; it must also be small, light weight, have a reasonable cost, consume minimal power and secure the contents of the messages while communicating etc. To communicate is the functionality, while the size, weight, power consumption and the use of encryption for secure data exchange are constraints that must be achieved for an acceptable mobile handset. To achieve the current goals of design, the designer must be aware with such design constraints and more importantly, the factors that have a direct effect on them.

This book consists of three parts. The first part comprises Chapters 1–3. Chapters 1 and 2, cover the important concepts of designing under constraints. In Chapter 1, we introduce the reader to a good amount of needed knowledge about design constraints and design metrics while in Chapter 2, we give the reader a complete idea of the design flow process and the challenges he will

face. More importantly, how he is to select one of the many options available to him.

In Chapter 3, we introduce the reader to the rest of the book; the other two parts. We use it to familiarize the reader with terminologies such as; microprocessor, microcontroller, microprocessor-based and microcontroller-based systems, the organization and the building blocks of the microprocessor and microcontroller, etc. We briefly introduce the reader to the main building blocks of the microcontroller that will be the subject of the rest of the book; timers, counters, the watchdog timer, ADC, DAC, serial communication, memory, need of programming, etc.

Part 2 comprises Chapters 4 and 5 which cover the programming part of microcontrollers. The two chapters give the reader a very clear idea of instructions and the instruction set, instruction cycles, addressing modes, assembly language, how to write a program, etc. We use for demonstration the instruction sets of the Atmel AVR series and Intel 8051 family.

The remaining Chapters 6–9 together form Part 3 of the book. Each chapter deals with one subsystem that can be considered, from the embedded system design point of view as a single purpose processor. For example, timers and counters if implemented as an IC chip, then, they are single-purpose processor, or subsystem. Similarly for: ADC, DAC, UART, CAN, I2C, etc. To make our book useful for a wider class of readers, we introduce each item as if it was a discrete subsystem and then we discuss its implementation as part of the resources of a microcontroller. The use of a microcontroller to implement the functionality of each subsystem and how to use it in many possible applications is then given. This explains the main reason behind calling our book "Digital System Design" and not "Embedded System Design"; the use of the microcontroller as the processor in the system. In Chapter 6, we discuss the memory system; Chapter 7 considers timer/counters, while Chapter 8 treats the main components of any Data Acquisition System. Finally Chapter 9 considers the communication between microcontrollers and the outside world. Each chapter in this part can be considered as a complete unit that covers a topic. The reader can read them in any order he prefers.

Contents

List of Abbreviations

ACK	Acknowledge
ADC	Analogue-to-Digital Converter
ALU	Arithmetic and Logic Unit
AU	Arithmetic Unit
ASIC	Application Specific Integrated Circuit
ASIP	Application-specific instruction-set processor
BCD	Binary Coded Decimal
CAD	Computer Aided Design
CAN	Controller Area Network
CISC	Complex Instruction Set Computer
CMOS	Complementary Metal oxide Semiconductor
CPI	Cycles per Instruction
CPLD	Complex PLD
CPU	Central Processing Unit
CRC	Cyclic Redundancy Checks
CSMA	Carrier-Sense Multiple Access
CSMA/CA	Carrier-Sense Multiple Access with Collision Avoidance
DAC	Digital-to-Analogue Converter
DAS	Data Acquisition System
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DRO	Destructive Read Out
DSP	Digital Signal Processor
EA	Effective Address
EDA	Electronic Design Automation
EEPROM	Electrically Erasable Programmable Read Only Memory

EPROM	Erasable Programmable Read Only Memory
EPLD	Erasable PLD or electrically PLD
FIFO	First-In First-Out
FLOPS	Floating-Point Operation per Second
FP	Floating-Point
FPU	Floating Point Unit
FPGA	Field Programmable Gate Array
FSM-D	Finite-State Machine with Datapath
GPP	General Purpose Processor
HLL	High-level Language
IC	Integrated Circuit
IC	Instruction Count
IR	Instruction Register
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
I/O	Input/Output
I2C	Inter-Integrated Circuits
LCA	Logic Cell Array
LED	Light Emitting Diode
LFSR	Linear-Feedback Shift Register
LIFO	Last-In First-Out
LLL	Low Level Language
LU	Logical Unit
MAR	Memory Address Register
MBR	Memory Buffer Register
MCU	Microcontroller Unit
MFLOPS	Million Floating Point Operation per Second
MIPS	Million (Mega) Instruction per Second
MPU	Microprocessor Unit
NRE	Non Recurring Engineering (cost)
OTP ROM	One-Time Programmable ROM
PAL	Programmable Array Logic
PC	Program Counter
PCB	Printed Circuit Board
PIT	Programmable Interval Timer
PLA	Programmable Logic Array
PLD	Programmable Logic Device

PRBS	Pseudo-random bit Sequence
PROM	Programmable Read Only Memory
PSR	Processor Status Register
PSW	Processor Status Word
PWM	Pulse Width Modulator
RAM	Reliability, Availability and Maintainability
RAM	Random Access Memory
RAS	Reliability, Availability and Serviceability
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RTC	Real Time Clock
RTL	Register Transfer Language
RTN	Register Transfer Notation
RTOS	Real Time Operating System
RWM	Read-write Memory
SCL	Serial-Clock-Line
SDA	Serial-Data-Line
SFR	Special Function Register
S/H	Sample and Hold
SMM	System Management Mode
SOC	System On a Chip
SP	Stack Pointer
SPECS	System Performance Evaluation Cooperative
SPI	Serial Peripheral Interface
SPP	Special Purpose Processor
SRAM	Static Random Access Memory
SSI	Small Scale Integrated circuit
SWT	Software Timer
TDM	Time Division Multiplexing
TOS	Top of Stack
TWI	Two-Wire Interface
UART	Universal Asynchronous Receiver and Transmitter
USART	Universal Synchronous/Asynchronous Receiver Transmitter
USI	Universal Serial Interface
VDU	Visual Display Unit
VLSI	Very Large Scale Integration
WDT	Watchdog Timer

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

1

Processor Design Metrics

THINGS TO LOOK FOR…

- The meanings of "Design" and "Designing under Constraints"
- The meaning of design constraints.
- Designing for Performance: What is performance? Can we measure it? Which constraints do we consider while designing for performance?
- Why performance analysis and optimization is important in digital/embedded systems.
- Designing for Cost: Which constraints must we consider while designing in order to minimize the cost? How can these constraints affect the final cost?
- Designing for Power Consumption: Is it possible to reduce the power consumption?

1.1 Introduction

Design is the task of defining a system's functionality and converting that functionality into a physical implementation, while satisfying certain constrained design metrics and optimizing other design metrics. The functionality of modern microelectronics and embedded systems is becoming more and more complex. Getting such complex functionality right is a difficult task because of the huge variety of possible environment scenarios that must be responded to properly. Not only is getting the correct functionality difficult, but also creating a physical implementation that satisfies all constraints is difficult as there are so many competing, tightly constrained metrics.

The designer, while converting requirements into a workable implementation passes through many phases, each with its own constraints. The constraints

may be grouped as followed:

- **Generic constraints or rules:** These constraints are normally known to the designer before starting the design; they are not flexible and must be fulfilled. For example for some particular kind of objects the designer may have no option but to using NAND, NOR and NOT gates to implement his design. Any complex component in his design must be converted to these three gates. Another generic rule may be that the system under design should use two-level logic (i.e. binary numbers); multi-valued or continuous logic not being allowed.
- **Constraints directly related to the circuit under design:** These types of constraints are important if the unit under design will be a part in an existing system or will be located at a certain place, etc. In the first case the customer may require the designer to use a specific operating system or certain types of components. The second situation may put constraints on the design as to the size of the final product, the location of power and ground paths, etc.
- **Normal design constraints:** These are the constraints that occur while the design is in progress. The design process, as shown in Chapter 2, consists of number of stages each with its own constraints. As the design process proceeds through the different stages, the designer has to consider more and more constraints and has to implement some of these constraints by selecting one out of many available options. The good designer leaves sufficient options to select out of them at the last stages of the design. Without doing that the designer will find great difficulties in designing the last stages.

While trying to satisfy some constraints, the designers may face difficulties from constraints that require trade-offs. Such constraints compete with each other; improving one leads to worsening the other. It is difficult to find optimum solution for such competing constraints. The best example of such trade-offs is between performance, size and power constraints. It is impossible to simultaneously design for high performance, low power consumption and small size. These three constraints cannot be optimised simultaneously; reducing the size causes performance to suffer; improving performance increases power dissipation. The designer must find schemes that help to satisfy some metrics without degrading others. Design may thus be said to be a matter of optimising constraints while maintaining full system functionality (the system requirements and specifications).

It is expected then that more than one design can fulfill the required functionality. The real challenge facing any designer is not merely to design for functionality but to find the implementation that can simultaneously optimises a large number of design metrics.

During the development phases of a design, many challenges arise that require a large number of decisions arise. Some of these decisions require knowledge of the most suitable way of approaching the solution; others require knowledge of the available IC technologies in the market and so on. Out of the many decisions, there are three fundamental decisions that have a direct effect on the way of optimising the design metrics. These decisions are related to the technologies the designer will use while developing the system. These technologies are:

- **Design technology**: Which design flow and which top-down model can we use to speed up the design process?
- **Processor technology**: Which processor technology (software, hardware, combination between hardware and software, etc.) can we applying in order to implement each functional block?
- **IC technology**: Which type of IC technology (VLSI, PLD, FPGA, etc) is suitable to implement the chosen processor?

The selection of the proper technology at each stage determines the efficiency with which the design can be converted from concept to architecture, to logic and memory, to circuit, and ultimately to physical layout.

Analysis versus Design

The best way to understand the effect of metrics and constraints on design is to understand the difference between analysis and design. In school, we mostly faced analysis problems. Consider for example the following problem. If a can of beans costs $1.00, a loaf of bread costs $1.50, a litre of milk costs $1.00, and a kilogram of meat costs $5.00, calculate how much money Ms. Smith will spend to buy four cans of beans, a loaf of bread, a litre of milk and two kilograms of meat?

This is what is termed an analysis problem. A teacher is giving it to a child to test his ability to use multiplication and addition. It has a single answer.

This example can be converted into a design problem. In this case the problem will read as follows: Ms Smith invited four guests. She wants to spend only $20.00 in buying food for them. The ingredient for meal must contain beans, milk, bread and meat. It is required from Ms Smith to construct an appropriate shopping list using a given price list for different items of food.

The goal (the function) in this example is to purchase groceries that cost $20.00. The constraints are:

- There is enough to serve four people, and
- The meal must include items from the four basic groups.

This is not a single answer problem; it has many answers depending on the price of the individual items, the appetites of the visitors etc. In real-life everyone faces such design problems; they are in fact more common than analysis problems. To solve this design problem we must use analysis. We need to use multiplication and addition as tools to get possible answers to the design problem. Solving a design problem needs an ability to analyse and takes place normally by trial-and-error, until an acceptable solution is achieved.

1.2 Common Design Metrics

The following discussion assumes that the system implementation combines hardware and software components. The hardware may be microprocessor-based, microcontroller-based or by means of specialised hardware. A design metric is a measurable feature of a system's implementation. The design constraints represent the non-functional requirements that are placed on the system. The IEEE standard 83–1993 lists the following non-functional requirements:

- Performance requirements
- Interface requirements
- Operational requirements
- Resource requirements
- Verification requirements
- Acceptance requirements
- Documentation requirements
- Security requirements
- Quality requirements
- Portability requirements
- Reliability requirements
- Maintainability requirements
- Safety requirements

It is possible to add the following requirements:

- Legal requirements
- Interoperability requirements

The most common constraints that cover these requirements are:

- **NRE (Non Recurring Engineering) cost:** This is a one-time monetary cost for designing the system. Once the system is designed, any number of units can be manufactured without incurring any additional design cost; hence the term nonrecurring. When designing in VLSI, the term "first silicon cost" is normally used as alternative to NRE cost.
- **Unit cost:** The monetary cost of manufacturing each copy of the system, excluding NRE cost.
- **Size**: The physical space required by the system, often measured in bytes for software, and gates or transistors for hardware.
- **Performance:** The execution time of the system or the processing power. It is usually taken to mean the time required to complete a task (latency or response time), or as the number of tasks that can be processed per unit time (throughput). Factors which influence throughput and latency include the clock speed, the word length, the number of general purpose registers, the instruction variety, memory speed, programming language and the availability of suitable peripherals.
- **Power:** This is the amount of power consumed by the system. It has a direct effect on the lifetime of a battery. It determines the cooling requirements of the IC; more power means more heat and hence the need for more cooling. Other effects are:
    1. Heat generation is the major factor that limits the performance of a system. More computation power and more speed mean more power consumption. One of the main concerns of any designer of a new processor is to keep it cool.
    2. Where physical size of the end product is a primary design constraint, the designer will in most cases end up designing a product that results in crowding of components into a small space. In such applications, reducing the power usage of the components will be a primary target. Such applications are very sensitive to heat problems.
    3. Having hundreds of millions of computers in use worldwide and sometimes, when thousands of them are located in the same enterprise, the power consumption and energy conservation becomes a serious issue.
    4. The consumed power by a device affects the overall system reliability and many other design metrics.

- **Flexibility:** This is the ability to change the functionality of the system without incurring a high NRE cost. Software is typically considered to be very flexible. Many digital systems are created not just to solve a single problem but rather, to provide a device that can be used in a variety of applications to achieve a reasonable solution.
- **Time-to-prototype:** This is the time needed to build a working version of the system which may be bigger or more expensive than the final goal but can be used to verify the system's usefulness and correctness and to refine its functionality.
- **Time-to-market:** This is the time required to develop a system to the point where it can be released and sold to customers. The main contributors are design time, manufacturing time and testing time.
- **Reliability:** Reliability is the probability that a machine or product can perform continuously, without failure, for a specified interval of time when operating under standard conditions. Increased reliability implies less failure and consequently less downtime and lost production.
- **Availability:** Availability refers to the probability that a system will be operative (up). Sometimes it indicates the possibility of sourcing the same product from more than one manufacturer. This meaning came from the fact that many devices are made by more than one manufacturer. It is preferable to call such type of availability "Second source suppliers." Availability of "second source suppliers" is important where the system designer needs to consider the reliability and longevity of supply.
- **Serviceability:** Refers to how easily the system is repaired. This metric is sometimes linked to "Manufacturer's support" which covers the provision of a range of services from the development system and its associated software through the documentation, maintenance of the development system and providing answers to technical queries.
- **Maintainability:** The ability to modify the system after its initial release, especially by designers who did not originally design the system. It is also the ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment
- **Design adequacy:** It is the probability that a system will successfully accomplish its mission given that the system is operating within its design specifications throughout the mission.

- **Range of complementary hardware:** For some applications the existence of a good range of compatible ICs to support the microcontroller/microprocessor may be important.
- **Special environmental constraints:** The existence of special requirements, such as military specifications or minimum physical size and weight, may well be overriding factors for certain tasks. In such cases the decision is often an easy one.
- **Ease of use:** This will affect the time required to develop, to implement, to test it, and to start using the system. These three factors - design time, manufacturing time and testing time are the main factors defining the time-to-market merit which is very important if the system is designed for commercial use. In commercial applications, as mentioned before, entering the market in time means the full use of the market windows that are available for the product and this will guarantee the profitability. This is a very important factor because the time-to-market factor defines the profitability.
- **Software Support:** Newer, faster processors enable the use of the latest software. In addition, new processors enable the use of specialized software not usable on earlier machines. Easier language means shorter time to learn and better maintainability
- **Correctness:** Confidence that the functionality of the system has been implemented correctly. We can check the functionality throughout the process of designing the system, and we can insert test circuitry to check whether the manufacturing was correct.
- **Safety:** The probability that the system will not cause harm.

As mentioned before, the metrics typically compete with one another; improving one often leads to worsening another. For example, if we improve the performance, the power consumption will increase. Figure 1.1 is a graphical representation of the trade-off between metrics. To face such challenges and achieve the optimum solution, the designer requires a wide background and expertise in the different technologies available. This knowledge must cover both hardware and software technologies.

The above design metrics can be divided into five broad groups based on the design phenomena that they measure. The five proposed groups are:

1. **Performance Metrics:** The metrics of this group deal with the speed of the system and how long it takes to execute the desired application.
2. **Cost Metrics:** The metrics of this group measure the product cost, the unit cost and the price of the product. NRE cost, Time-to-prototype,

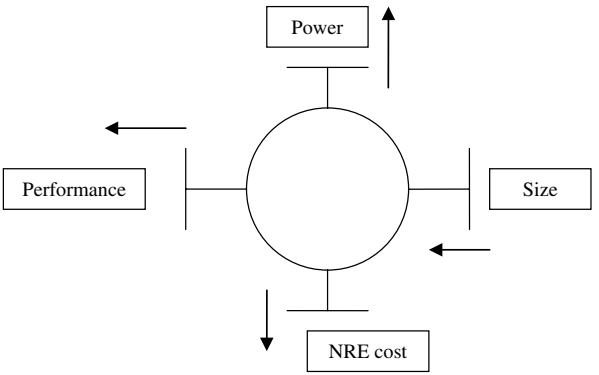


Figure 1.1 Design metric competition — improving one may worsen others.

Time-to-market and Ease of use are some of the metrics that affect the cost and price. Sometimes, a cost-performance metric may be more important than cost and performance separately.

3. **Power Consumption Metrics:** Metrics of this group measure the power consumption of the system. These metrics are gaining importance in many fields as battery powered mobile systems become prevalent and energy conservation becomes more significant.
4. **System Effectiveness Metrics:** In many applications such as military applications, how effective the system is in implementing its target is more important than cost. Reliability, Maintainability, Serviceability, design adequacy and flexibility are related to the metrics of this group.
5. **Others**: These are metrics that include those that may guide the designer to select from many off-the-shelf components that can do the job. Ease of use, software support, safety and the availability of second source suppliers are some of the metrics of this group.

1.3 Performance Design Metrics

Performance of a system is a measure of how long a system takes to execute a desired application. For example, in terms of performance, a computer user cares about how long a digital computer takes to execute a programme. For the user the computer is faster when the execution of his programme takes less time. For the IT manager, a computer is faster when it completes more tasks within a unit time. For the manager as well as for the user, the clock frequency of the computer (instructions per second) is not the key issue; the architecture

of one computer may result in a faster programme execution even though it has a lower clock frequency than another. We note here that, the main concern of the user is to reduce the *response time* (or *latency, or execution time*), while the main concern of the manager is how to increase the *throughput*.

With that said, there are several measures of performance. For simplicity, suppose we have a single task that will be repeated over and over, such as executing a programme or producing a car on an automobile assembly line. The two main measures of performance are latency (or response time) and throughput. However, we must note that throughput is not always just the number of tasks times the latency. A system may be able to do better than this by using parallelism, either by starting one task before completing the next one (pipelining) or by processing tasks concurrently. In case of automobile production factory, the assembly line consists of many steps. Each step contributes something to completing the car. Each step operates in parallel with the other steps, though on a different car. Thus, our assembly line may have a latency of 4 hours but a throughput of 120 cars per day. The throughput in our example here is defined as the number of cars produced per day. In other words, the throughput is determined by how often a completed car exits the production line.

Definitions:
Latency or Response time: The time between the start of the task's execution and the end. For example, producing one car takes 4 hours.
Throughput: The number of tasks that can be processed per unit time. For example, an assembly line may be able to produce 6 cars per day.

The main concern in the two cases, throughput and response time, is time. The computer that performs the same amount of work in the least time is the fastest. If we are speaking of a single task, then we are speaking of response time, while if we are speaking of executing many tasks, then we are speaking about throughput. The latency metric is directly related to the execution time while throughput measures the rate of implementing a given task. We can expect many metrics measuring throughput based on the definition of the task. The task may be an instruction as in case of MIPS (see 1.3.2.2), or it may be floating-point operations as in case of MFLOPS (see 1.3.2.3) or any other task. Besides execution time and rate metrics, there is a wide variety of more specialised metrics used as indices of computer system performance. Unfortunately, as we shall see later, many of these metrics are often used but interpreted incorrectly.

Performance metrics can be divided into two classes;

a. **Metrics that measure the performance of working systems:** Latency and throughput are examples of this group. We use them to measure the performance of something already implemented irrespective of whether such performance fulfils the design requirements of this metric or not. Such performance metrics that measure what was ready done are called *means-based* metrics.
b. **Metrics that measure performance during the design process:** i.e. before implementing the system. This group of metrics, for example, enables the designer to compare different algorithms against the expected computation time. All the algorithms may achieve the same functionality but it is necessary to decide which one is optimum from the point of view of execution time and hardware complexity. As will be discussed in Chapter 2, to reach an optimum system, the designer has to optimise some of the design metrics at each stage of the design process; performance at one stage, power consumption at another stage and so on. In the early stages of the design process the target of the designer is to measure what is actually achieved after each stage. The Big-O notation is an example of such performance metrics.

This leads us to the following question: What are the characteristics of a good performance metric?

1.3.1 Characteristics of a Good Performance Metric

There are many different metrics that have been used to describe the performance of a computer system. Some of these metrics are commonly used throughout the field, such as MIPS and MFLOPS (which are defined later in this chapter), whereas others are introduced by manufacturers and/or designers for new situations as they are needed. Experience has shown that not all of these metrics are 'good' in the sense that sometimes using a particular metric out of context can lead to erroneous or misleading conclusions. Consequently, it is useful to understand the characteristics of a 'good' performance metric. This understanding will help when deciding which of the existing performance metrics to use for a particular situation and when developing a new performance metric.

A performance metric that satisfies all of the following requirements is generally useful to a performance analyst in allowing accurate and detailed comparisons of different measurements. These criteria have been developed by

researchers through observing the results of numerous performance analyses over many years. While they should not be considered absolute requirements of a performance metric, it has been observed that using a metric that does not satisfy these requirements can often lead the analyst to make erroneous conclusions.

1. **Linearity**: The metric is linear if its value is proportional to the actual performance of the machine. That is, if the value of a metric changes by a certain ratio, the actual performance of the machine should change by the same ratio. For example, suppose that you are upgrading your system to a system whose speed metric (i.e. execution-rate metric) is twice as large as on your current system. You then would expect the new system to be able to run your application programmes in half the time taken by your old system. Similarly, if the metric for the new system were three times larger than that of your current system, you would expect to see the execution times reduced to one-third of the original values. Not all types of metrics satisfy this proportionality requirement e.g. logarithmic metrics are nonlinear metrics.
2. **Reliability**: A performance metric is considered to be reliable if system A always outperforms system B when the corresponding values of the metric for both systems indicate that system A should outperform system B. For example, suppose that we have developed a new performance metric called WPAM that we have designed to compare the performance of computer systems when running the class of word-processing application programmes. We measure system A and find that it has a WPAM rating of 128, while system B has a WPAM rating of 97. We then can say that WPAM is a reliable performance metric for word-processing application programmes if system A always outperforms system B when executing these types of applications. While this requirement would seem to be so obvious as to be unnecessary to state explicitly, several commonly used performance metrics do not in fact satisfy this requirement. The MIPS metric, for instance, which is described latter, is notoriously unreliable. Specifically, it is not unusual for one processor to have a higher MIPS rating than another processor while the second processor actually executes a specific programme in less time than does the processor with the higher value of the metric. Such a metric is essentially useless for summarizing performance, and we say that it is unreliable.
3. **Repeatability:** A performance metric is repeatable if the same value of the metric is measured each time the same experiment is performed. Note that this also implies that a good metric is deterministic.

4. **Ease of measurement:** If a metric is not easy to measure, it is unlikely that anyone will actually use it. Furthermore, the more difficult a metric is to measure directly, or to derive from other measured values, the more likely it is that the metric will be determined incorrectly. The only thing worse than a bad metric is a metric whose value is measured incorrectly.
5. **Consistency:** A consistent performance metric is one for which the units of the metric and its precise definition are the same across different systems and different configurations of the same system. If the units of a metric are not consistent, it is impossible to use the metric to compare the performances of the different systems. While the necessity for this characteristic would also seem obvious, it is not satisfied by many popular metrics, such as MIPS (Section 1.3.2.2) and MFLOPS (Section 1.3.2.3).
6. **Independence:** Many purchasers of computer systems decide which system to buy by comparing the values of some commonly used performance metric. As a result, there is a great deal of pressure on manufacturers to design their machines to optimise the value obtained for that particular metric and to influence the composition of the metric to their benefit. To prevent corruption of its meaning, a good metric should be independent of such outside influences.

Many metrics have been proposed to measure performance. The following sections describe some of the most commonly used performance metrics and evaluate them against the above characteristics of a good performance metric.

1.3.2 Some Popular Performance Metrics

Many measures have been devised in an attempt to create standard and easy-to-use measures of computer performance. One consequence has been that simple metrics, valid only in a limited context, have been heavily misused such that using them normally results in misleading conclusions, distorted results and incorrect interpretations. Clock rate, MIPS and MFLOPS are the best examples of such simple performance metrics; using any of them results in misleading and sometimes incorrect conclusions. These three metrics belong to the same family of performance metrics that measure performance by calculating the rate of occurrence of an event. In Section 1.3.2.8 we give an example that highlights the danger of using the wrong metric (mainly the use of means-based metrics or using the rate as a measure) to reach a conclusion about computer performance. In most cases it is better to use metrics that use the execution time as a base for measuring the performance.

1.3.2.1 Clock Rate

The clock rate is the best example of how a simple metric of performance can lead to very wrong conclusions. The clock frequency of the central processor is used by many suppliers as an indication of performance. This may give one the impression that using a 1.5 GHz system would result in a 50% higher throughput than a 1.0 GHz system. This measure however ignores how much work the processor achieves in each clock cycle; that some complex instructions may need many clock cycles and that I/O instructions may take many clock cycles. More importantly, they ignore the fact that in many cases the processor may not be the performance bottleneck; it may be the memory subsystem or the data bus width.

Some of the characteristics of the clock rate metric are:

- it is repeatable since it is a constant for a given system
- it is easy to determine since it is most likely stamped on the box
- it is consistent across all systems
- it is difficult for a manufacturer to misrepresent it

The above characteristics make using the clock rate as a measure of performance appear advantageous but as it is a nonlinear measure (doubling the clock rate does not always mean doubling the resulting performance) it is therefore, an unreliable metric. This point will be clarified later when considering the execution time equation. Thus, we conclude that the processor's clock rate is not a good metric of performance.

1.3.2.2 MIPS (Mega Instruction per Second)

The MIPS metric belongs to a group of metrics called 'rate metrics'; an execution-rate performance metric. It is regularly used to compare the speed of different computer systems and it measures the number of CPU instructions performed per unit time.

$\text{MIPS} = \text{Instruction count}/(\text{Execution time} * 10^6)$

This definition shows MIPS to be an instruction execution rate metric; specifying performance inversely to execution time. It is important to note here that both the instruction count (IC) and the execution time are measurable, i.e., for a given programme (normally a benchmark), it is possible to calculate MIPS and accordingly to get an idea of the performance. It is possible to rewrite the definition of MIPS to include other measurable parameters:

$$\begin{aligned}\text{MIPS} &= \text{Instruction count}/(\text{Execution time} * 10^6)\\ &= \text{Instruction count}/(\text{CPU clocks} * \text{Cycle time} * 10^6)\end{aligned}$$

$$
\begin{aligned}
&= (\text{Instruction count} * \text{clock rate})/(\text{CPU clocks} * 10^6) \\
&= (\text{Instruction count} * \text{Clock rate})/(\text{Instruction count} \\
&\quad * \text{CPI} * 10^6) \\
&= \text{Clock rate}/(\text{CPI} * 10^6) \qquad (1.1)
\end{aligned}
$$

In the above equation:

$$
\begin{aligned}
\text{CPU clocks} &= \text{the total number of clock cycles used to execute} \\
&\quad \text{the programme, and} \\
\text{CPI} &= \text{the average cycles per instruction.}
\end{aligned}
$$

Knowing MIPS, it is possible to calculate the execution time:

$$\text{Execution time} = \text{Instruction count}/(\text{MIPS} * 10^6) \qquad (1.2)$$

Problems that may arise of using MIPS as measure of performance:

Taking the instruction as the unit of counting makes MIPS easy to measure, repeatable, and independent; however MIPS suffers from many problems that make it a poor measure of performance. It is not linear, since a doubling of the MIPS rate does not necessarily cause a doubling of the resulting performance and it is neither reliable nor consistent since it does not correlate well to performance at all.

The problem with MIPS as a performance metric is that different processors can do substantially different amounts of computation with a single instruction. For instance, one processor may have a branch instruction that branches after checking the state of a specified condition code bit. Another processor may have a branch instruction that first decrements a specified count register and then branches after comparing the resulting value in the register with zero. In the first case the single instruction does one simple operation whereas in the second case, one instruction performs several operations. The failing of the MIPS metric is that each instruction counts as one unit, irrespective of the work performed per instruction. This difference is the core of the difference between RISC and CISC processors and this renders MIPS essentially useless as a performance metric. Even when comparing two processors of the same architecture, this metric can be inconsistent when instruction rate changes and may not carry through to all aspects of a processor performance such as I/O or interrupt latency.

As a matter of fact, sometimes MIPS can vary inversely with performance. One of the examples that can clearly shows this fact is the case, when we

calculate MIPS for two computers with the same instruction set but one of them has special hardware to execute floating-point operations and another machine using software routines to execute the floating-point operations. The floating-point hardware needs more clock cycles to implement one floating point operation compared with the number of clock cycles needed to implement an integer operation. This increases the average value of the CPI (cycles per instruction) of the machine which in turn, according to equation (1.1), results in a lower MIPS rating. On the other hand however, the software routines that were needed to execute floating point operation consisted of many simple instructions, now being replaced by a single hardware instruction and thus executing much faster. Hence, the inclusion of floating point hardware will result in a machine that has a lower MIPS rating but can do more work, thus highlighting the drawback of MIPS as a metric. Example 1.4 further illustrates this effect.

1.3.2.3 FLOPS (Floating-Point Operation per Second)

The primary shortcoming of the MIPS metric is due to counting instructions irrespective of the amount of work the instruction does. MIPS metrics counts a NOP instruction equal to a multiplication instruction that operates on data at memory; both are counted as one instruction. Some applications contain intensive processing of floating-point operations and MIPS leads to very wrong conclusions in such cases. To account for the floating point execution which may be critical in some applications, the FLOP (FLoating point OPeration) metric was generated which counts the number of floating point operations per second. This metric is impossible to measure directly as it only measures floating point instructions which are almost impossible to use on their own; they always need other instruction classes around them. However the metric is easy to determine from the machine hardware specifications and can be measured approximately using code sequences that are predominantly floating point.

MFLOPS, (Mega FLOPS) as MIPS, can be calculated for a specific programme running on a specific computer. MFLOPS is a measure of millions of floating point-operation per second:

$$\text{MFLOPS} = \text{Number of floating-point operations}/(\text{Execution time} * 10^6) \tag{1.3}$$

The metric considers floating-point operations such as addition, subtraction, multiplication, or division applied to numbers represented by a single or double

precision floating-point representation. Such data items are heavily used in scientific calculations and are specified in programming languages using key words like float, real, double, or double precision.

MFLOPS is not a dependable measure as it depends on the type of floating-point operations present in the processor (availability of floating point instructions). For example some computers have no sine instruction while others have. In the first case, the calculation of a sine function needs to call the software sine routine that would perform several floating-point operations, while the second case would require only one operation. Another weakness of MFLOPS comes from treating floating point addition, subtraction, multiplication and division equally. This is incorrect as floating point division is significantly more complex and time consuming than floating point addition.

1.3.2.4 SPECS (System Performance Evaluation Cooperative)

To avoid the shortages of the simple performance metrics mentioned above, several computer manufacturers formed the System Performance Evaluation Cooperative (SPEC) to produce a standardized performance metric. To reach the target, they choose benchmark programmes that contained both integer and floating-point operations, which reflected common workstation usage. They also defined, as follows, a standard methodology for measuring and reporting the performance obtained when executing these programmes:

- Measure the time required to execute each benchmark programme on the system being tested.
- Divide the time measured for each programme by the time required to execute the same programme on a standard basis machine in order to normalise the execution times.
- Average these normalised values using the geometric mean to produce a single performance metric.

While the SPEC methodology is certainly more rigorous than using MIPS or MFLOPS as a measure of performance, it still gives a problematic performance metric. One of its shortcomings is that averaging together the individual normalized results with the geometric mean produces a metric that is not linearly related to a program's actual execution time. Thus, the SPEC metric is not intuitive. Furthermore, and more importantly, it has been shown to be an unreliable metric in that a given programme may execute faster on a system that has a lower SPEC rating than it does on a competing system with a higher rating.

Finally, although the defined methodology appears to make the metric independent of outside influences, it is actually subject to a wide range of

tinkering. For example, many compiler developers have used these benchmarks as practice programmes, thereby tuning their optimisations to the characteristics of this collection of applications. As a result, the execution times of the collection of programmes in the SPEC suite can be quite sensitive to the particular selection of optimisation flags chosen when the programme is compiled. Also, the selection of specific programmes that comprise the SPEC suite is determined by a committee of representatives from the manufacturers within the cooperative. This committee is subject to numerous outside pressures since each manufacturer has a strong interest in advocating application programmes that will perform well on their machines. Thus, while SPEC is a significant step in the right direction towards defining a good performance metric, it still falls short of the goal.

1.3.2.5 Comments

As mentioned before any performance metric must be reliable. The majority of the above mentioned metrics are not reliable. The main reason that makes them unreliable is that they measure what was done whether or not it was useful. Such metrics are called means-based metrics. The use of such metrics may lead to wrong conclusions concerning the performance of the system.

To avoid such problems, we must use metrics that are based on the definition of performance, i.e. the execution time. Such metrics are ends-based metrics and measure what is actually accomplished. The difference between the two classes of performance metrics is highlighted in Section 1.3.2.8.

1.3.2.6 The processor performance equation

The performance of any processor is measured, as mentioned above, by the time required to implement a given task, in our case a given programme. The execution time of a programme is normally referred to as CPU time. It is expressed as follows:

$$\text{CPU time} = \text{CPU clock cycles for a programme} * \text{Clock cycle time}$$

For any processor the clock cycle time (or clock rate) is known and it is possible to measure the CPU clock cycles. CPU time can also be expressed in terms of number of instruction executed (called instruction count IC), and the average number of clock cycles per instruction (CPI):

$$\text{CPU time} = \text{IC} * \text{CPI} * \text{Clock cycle time} \tag{1.4}$$

Where:

$$\text{CPI} = \text{(CPU clock cycles for a programme)/IC}$$

Equation (1.4) can be written as:

$$CPU\ time = \frac{Seconds}{program} = \frac{Instruction}{Program} \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle} \quad (1.5)$$

In the general case, executing the programme means the use of different instruction types each of which has its own frequency of occurrence and CPI.

Example 1.1

When running specific programme on a given computer we measured the following parameters:

- Total instruction count (IC): 30 000 000 instructions
- Average CPI for the programme: 1.92 cycles/instruction
- CPU clock rate: 350 MHz.

Calculate the execution time for this programme.

Solution:

We use equation (1.5) to calculate the execution time:

$$\begin{aligned}
\text{CPU time} &= \text{(Seconds/ Programme)} \\
&= \text{(Instructions/ Programme)} \times \text{(Cycles/Instruction)} \\
&\quad \times \text{(Seconds/ Cycle)} \\
\text{CPU time} &= \text{Instruction count} \times \text{CPI} \times \text{Clock cycle} \\
&= 30,000,000 \times 1.92 \times 1/\text{clock rate} \\
&= 30,000,000 \times 1.92 \times (1/350) \times 10^{-6} \\
&= 0.1646\text{seconds}
\end{aligned}$$

1.3.2.7 Speed-up Ratio

In embedded systems, there is high possibility of having many design alternatives. In such cases it is important to compare the performance of such design alternatives to select the best. Speedup is a common method of comparing the performance of two systems. The speedup of system A over system B is determined simply as:

$$\text{Speedup of A over B} = \text{performance of A/performance of B.}$$

Performance could be measured either as latency or as throughput, depending on what is of interest. Suppose the speedup of assembly line A over

assembly line B is 2. Then we also can say that A is 2 times faster than B and B is 2 times slower than A.

Another technique for comparing performance is to express the performance of a system as a percent change relative to the performance of another system. Such a measure is called relative change. If, for example, the throughput of system A is R1, and that of system B is R2, the relative change of system B with respect to A, denoted $\Delta_{2,1}$ (that is, using system A as the base) is then defined to be:

$$\text{Relative change of system B w.r.t. system A} = \Delta_{2,1} = \frac{R_2 - R_1}{R_1}$$

Typically, the value of $\Delta_{2,1}$ is multiplied by 100 to express the relative change as a percentage with respect to a given basis system. This definition of relative change will produce a positive value if system B is faster than system A, whereas a negative value indicates that the basis system is faster.

An example of how to apply these two normalization techniques, the speedup and relative change of the systems is shown in Table 1.1 are found using system 1 as the basis. From the raw execution times, we can see that system 4 is the fastest, followed by systems 2, 1, and 3, respectively. However, the speedup values give us a more precise indication of exactly how much faster one system is than the other. For instance, system 2 has a speedup of 1.33 compared with system 1 or, equivalently, it is 33% faster. System 4 has a speedup ratio of 2.29 compared with system 1 (or it is 129% faster). We also see that system 3 is actually 11% slower than system 1, giving it a slowdown factor of 0.89.

Normally, we use the speedup ratio and the relative change to compare the overall performance of two systems. In many cases it is required to measure how much the overall performance of any system can be improved due to changes in only a single component of the system. Amdahl's law can be used, in such cases, to get the impact of improving a certain feature on the performance of the system.

Table 1.1 An example of calculating speedup and relative change using system 1 as the basis.

System X	Execution time T_x(s)	Speedup $S_{x,1}$	Relative change $\Delta_{2,1}$(%)
1	480	1	0
2	360	1.33	+33
3	540	0.89	−11
4	210	2.29	+129

Amdahl's Law

The performance gain that can be obtained by improving some portion of a computer can be calculated using Amdahl's Law. Amdahl's Law states:

"*The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used*". Amdahl's law can be expressed mathematically by the equation:

$$\frac{T_{total}}{T_{enhanced}} = \frac{T_{total}}{(T_{total} - T_{component}) + (T_{component}/n)}$$

Where:
T_{total} = System metric prior to enhancement,
$T_{enhanced}$ = System metric after enhancement,
$T_{component}$ = Contribution of the component to be improved to the system metic,
n = The amount of the enhancement.

Example 1.2 Use of Amdahl's law

Consider a system with the following characteristics: The task to be analysed and improved currently executes in 100 time units, and the goal is to reduce execution time to 80 time units. The component that is to be enhanced in the task is currently using 40 time units. Calculate the amount of enhancement needed.

Solution:

Use the above equation we get:

$$\frac{100}{80} = \frac{100}{(100 - 40) + (40/n)}$$

From which we can get $n = 2$. This means that to achieve the required task, the component to be enhanced must be improved by speed up ratio of 2. In our case its execution time must go from 40 time units to 20 time units.

1.3.2.8 Rate vs. Execution-time based metrics

One of the most important characteristics of a performance metric is that it be reliable. One of the problems with many of the rate metrics discussed above that makes them unreliable is that they measure what was done whether or not it was useful. What makes a performance metric reliable, however, is that it accurately and consistently measures progress towards a goal. Metrics that measure what was done, useful or not, and which have been called means-based metrics are not recommended to be used in many situations because they

Table 1.2

Instruction type	Frequency	Clock Cycle Count
(a) Before optimization		
Arithmetic & Logic	43%	1
Loads	21%	2
Stores	12%	2
Branches	24%	2
(b) After optimization		
Arithmetic & Logic	27.4%	1
Loads	26.8%	2
Stores	15.3%	2
Branches	30.5%	2

can lead to wrong conclusions. The use of ends-based metrics that measure what is actually accomplished are more accurate and lead to correct decisions. The reason is that ends-based metrics uses the execution time to measure the performance.

To understand the difference between the rate metrics and the execution-time based metrics we are considering here two examples.

Example 1.3 Effect of Compiler Variations on MIPS Rating and Performance:

Table 1.2 shows the instruction usage before and after optimizing the compiler of a load-store machine. The system clock is running on 500 MHz. This development results in reducing the instruction count IC as follows:

$$\mathrm{IC}_{optimised} = 0.785\,\mathrm{IC}_{unoptimised}$$

Calculate the MIPS rating before and after optimization and also the speedup ratio. Comment on the results.

Solution: From the tables we can get:

$$\begin{aligned}\mathrm{CPI}_{\mathrm{unoptimised}} &= 0.43 \times 1 + 0.21 \times 2 + 0.21 \times 2 + 0.24 \times 2 = 1.57\\ \mathrm{CPI}_{\mathrm{optimised}} &= 0.274 \times 1 + 0.268 \times 2 + 0.153 \times 2 + 0.305 \times 2 = 1.726\end{aligned}$$

Calculation of MIPS:
From equation (1.1), we get:

$$\begin{aligned}\mathrm{MIPS}_{\mathrm{unoptimised}} &= 500\,\mathrm{MHz}/1.57 * 10^6 = 318.5\\ \mathrm{MIPS}_{\mathrm{optimised}} &= 500\,\mathrm{MHz}/1.726 * 10^6 = 289.0\end{aligned}$$

Calculation Speedup ratio:

Use equation (1.4) to calculate the performance of unoptimised code, we get

$$\begin{aligned}\text{CPU time}_{\text{unoptimised}} &= \text{IC}_{\text{unoptimised}} * 1.57 * (2 * 10^{-9}) \\ &= 3.14 * 10^{-9} * \text{IC}_{\text{unoptimised}}\end{aligned}$$

For the optimised case:

$$\begin{aligned}\text{CPU time}_{\text{optimised}} &= \text{IC}_{\text{optimised}} * 1.73 * (2 * 10^{-9}) \\ &= 0.785\text{IC}_{\text{unoptimised}} * 1.73 * 2 * 10^{-9} \\ &= 2.72 * 10^{-9}\text{IC}_{\text{unoptimised}}.\end{aligned}$$

Discussions:
Taking MIPS as metric:
MIPS before optimization = 318
MIPS after optimization = 289

Conclusion 1:
The optimised code is lower than the unoptimised: 289 versus 318.
When taking Speedup as performance metric:
The speedup of the optimised code relative to the unoptimised code = 3.14/2.72 = 1.15

Conclusion 2:
If we take speedup ratio as a measure for performance, then the system after optimizing the compiler is 1.15 faster than before optimization.

Conclusion 3:
We reach completely different conclusions when using two different types of metrics. We reduced the total CPU time from $3.14 * 10^{-9}$ IC to $2.72 * 10^{-9}$ IC but the MIPS shows that the unoptimised is better. This is a misleading conclusion.

Example 1.4
The programme given in Figure 1.2 calculates the vector dot-product. The programme executes N floating-point addition and multiplication operations for a total of $2N$ floating-point operations. This programme may be modified to take the form given in Figure 1.3. For each version calculate:

- the execution time,
- the MFLOPS,

```
s = 0;
for (i = 1; I<N: i++)
        s = s + x[i] * y[i]
```

Figure 1.2 A vector dot-product programme example.

```
s =0;
for (i = 1; i <N; i++)
        If (x[i] != 0 && y[i] != 0)
            S = s + x[i] * y[i];
```

Figure 1.3 Vector dot-product programme modified to calculate only non zero elements.

as a function of the data size N. Assume that the floating-point addition operation needs T_{add} cycles and the floating-point multiplication operation needs T_{mult} cycles.

Solution

Programme 1:

This programme executes N floating-point addition and multiplication operations for a total of $2N$ floating point operation.

The total time required to execute the programme $= N(T_{add} + T_{mult})$ cycles.

The execution rate in terms of the number of Floating-point operations (FLOPS) per cycle:

$$\frac{R_1 = \frac{2N}{N(T_{add}+T_{mult})} = \frac{2}{T_{add}+T_{mult}} FLOPS}{Cycle}$$

Programme 2

This option of the programme takes into consideration the fact that there is no need to perform the addition or multiplication operations for elements whose value is zero. In such a case it may be possible to reduce the total execution time if many elements of the two vectors are zero. The programme given in Figure 1.3 performs the floating-point operations only for those nonzero elements. If the conditional if statement requires T_{if} cycles to execute, the total time required to execute this programme is

$$T_{total\ 2} = N[T_{if} + f(T_{add} + T_{mult})]\text{cycles}$$

where f is the fraction of N for which both $x\ [i]$ and $y\ [i]$ are nonzero.

Since the total number of additions and multiplications executed in this case is $2Nf$, the execution rate in FLOPS/cycle for this programme is

$$\frac{R_1 = \frac{2NF}{N[T_{if}+f(T_{add}+T_{mult})]} = \frac{2f}{T_{if}+f(T_{add}+T_{mult})} FLOPS}{Cycle}$$

For the designer to select one of the two options, he can consider practical values for the parameters T_{add}, T_{mult}, T_{if} and the fraction f. For example, if $T_{if} = 4$ cycles, $T_{add} = 5$ cycles, $T_{mult} = 10$ cycles, $f = 10\%$, and the processor's clock rate is 250 MHz (i.e. one cycle is 4 ns), he gets:

$T_{total1} = 60N$ ns and

$T_{total2} = N\ [4 + 0.1(5 + 10)] * 4$ ns $= 22N$ ns.

The speedup of programme 2 relative to programme 1 then is found to be:

$S_{2,1} = 60N/22N = 2.73$.

The speed up ratio $\Delta_{2,1} = N(60 - 22)/22N = 172.7\%$

Calculating the execution rates realized by each programme with these assumptions produces:

$R_1 = 2/(60 \text{ ns}) = 33$ MFLOPS and

$R_2 = 2(0.1)/(22 \text{ ns}) = 9.09$ MFLOPS.

Thus, even though we have reduced the total execution time from $T_{total1} = 60N$ ns to $T_{total2} = 22N$ ns, the means-based metric (MFLOPS) shows that programme 2 is 72% slower than programme 1. The ends-based metric (execution time), however, shows that programme 2 is actually 172.7% faster than programme 1.

We reach completely different conclusions when using these two different types of metrics because the means-based metric unfairly gives programme 1 credit for all of the useless operations of multiplying and adding zero. This example highlights the danger of using the wrong metric to reach a conclusion about computer-system performance.

1.3.2.9 Discussions and conclusions

In Example 1.3 the difference came from the fact that the rate metrics is unreliable because it measures what was done whether or not it was useful. What makes a performance metric reliable, however, is that it accurately and consistently measures progress towards a goal. Metrics that measure what was done, useful or not, are called means-based metrics and they are not recommended to be used in many situations because they can lead to wrong conclusions. The use of ends-based metrics that measure what is actually

accomplished are more accurate and lead to correct decisions. The reason is that ends-based metrics uses the execution time to measure the performance.

1.3.3 Analysing Algorithms

The performance measures mentioned above can be used to measure the performance or compare the performance of working machines. The result has no effect on the design since it is ready completed, implemented and working. The designer can do nothing if the measured performance does not fulfill the original requirement. The designer thus needs a performance measure that can be used during the design stages and before the implementation. Such performance measures can guide the designer while selecting between software or hardware implementation, which algorithm to use, which IC technology that will help fulfill the requirement etc. The importance of having such a performance measure is related directly to the nature of the design cycle.

The design cycle, to be covered in Chapter 2, has different levels of abstractions. The designer refines the system while moving from one abstraction level to the next one. The requirements (functional and non-functional) list can be considered as the highest level of abstraction in which we use natural language (like English) to describe the system. The designer refines this level of abstraction to get the next level; the system level of abstraction. The process of refining the system continues to the lowest abstraction level which is the implementation. The implementation level consists of machine-code for general-purpose processors and a gate-level netlist for single-purpose processor. For the final product to fulfill the design metrics, e.g. performance, it is important for the designer to analyse and optimise these metrics at each level of abstraction.

On the highest level of abstraction we normally use algorithms to convert the functional requirements (describing the behaviour of the system) into structural requirements. In this stage the designer has no idea about the processor technology (software or hardware) to use to implement each functional block or about the IC technology needed. On such a level of abstraction the absolute values of the metrics, e.g. how many micro seconds difference in the execution time or how many microwatts of power consumption are not relevant. As a matter of fact at this level of abstraction it is practically impossible to define exact values for execution time or power consumption since neither the processor technology nor the IC technology has been chosen. On such a level of abstraction we focus on the relative performance of different algorithms and

designs. The algorithms and designs at this level are completely independent of the hardware or physical implementation used in the latter stages.

1.3.3.1 Analysing algorithms (complexity of algorithms)

The main target of analysing any algorithm is to predict the resources needed to perform it. In computers, by resources, we mean: storage capacity; communication (bus) bandwidth; hardware needed (number of gates required); and computational time. In most cases the computational time represents the major factor the designer wants to measure. If the designer identifies several algorithms that can be used to solve the problem under consideration, analysing these algorithms will help find the most efficient one. If the analysis indicates that more than one algorithm is viable, the designer keeps them as options while going through the different design stages. While going from one abstraction level (design stage) to another, other design metrics may force the designer to drop some of the candidate algorithms. The designer will end, at some stage, with the optimum algorithm. For example, when moving from system level to instruction level, the metrics analysis will be at a much finer grain. On such a level of abstraction, the analysis will focus on specific machines and limited blocks of code. At the lower level, the objective changes to optimise a specific algorithm or identify upper and lower bounds on a sequence through a piece of code. The designer then normally ends with the optimum algorithm that optimises the design metrics, usually performance.

Complexity analysis provides a high-level or heuristic view of algorithms and software. Complexity analysis measures the execution time as a number of time steps or number of time units. The hardware complexity is measured by the number of storage units needed. To use such metrics, the designer has to start by quantifying what comprises an elementary operation. The assumption is that each elementary operation takes one step and that each elementary object will occupy one unit of memory. At some stage, the designer must also decide on the IC technology to be used for implementing the device and thence their cost. Normally the designer assumes that the design will end with a random-access machine (RAM) in which instructions are executed sequentially.

Suppose M is an algorithm, and suppose n is the size of the input data. The time and space used by the algorithm M are the two main measures for the efficiency of M. The time is measured by counting the number of time units needed to run the algorithm and the space is measured by counting the maximum number of memory units needed by the algorithm.

The complexity of an algorithm M is the function $f(n)$ which gives the running time and/or storage space requirement of the algorithm in terms of the size n of the input data. Frequently, the storage space required by an algorithm is simply a multiple of the data size n. Since we are discussing in this section the performance of algorithms, we will henceforth use the term "complexity" to refer to the running time of the algorithm.

We will start our discussion by considering three simple examples. The first one illustrates the meaning of the function $f(n)$ and the other two establish the meaning of "worst case", "best case" and "average case". The "worst case", "best case" and "average case" are normally used in comparing algorithms.

Example 1.5

In this example we analyse a simple algorithm that accepts as input, an array of n integers and returns the sum of the elements of the array.

The algorithm is given in the second column of Table 1.3

Table 1.3 Simple summation algorithm analysis.

No.	Algorithm	Cost (number of time units needed to execute each operation in the statement)	Times	Total Cost of executing the statement
1	int total (int myArray[ ], int n)	1 c_1	1	$2c_1$
		1 c_1	1	
2	{			
3	Int SUM = 0;	1 c_2 (assign)	1	c_2
4	Int i = 0	1 c_2 (assign)	1	c_2
5	For (i = 0; i<n; i++)	1 c_2 (assign)	1	$c_2 + n(c_3 + c_4)$
		1 c_3 (compare)	n	
		1 c_4 (increment)	n	
6	{			
7	SUM = SUM + myArray[i]	1 c_2 (assign)	n	$n(c_1 + c_2 + c_3)$
		1 c_3 (add)	n	
		1 c_1	n	
8	}			
9	Return SUM;	1 c_5	1	c_5
10	}			
		Total number of time units needed to execute the algorithm		$5n + 6$

In this algorithm we can identify the following operations: PUSH into the stack, assign, increment, add, compare and the index operation (e.g. myArray[i]). We start our analysis by identifying a "cost" for each operation. "Cost" means the time unit needed to implement each operation. The costs of executing the operations are:

- c_1 for PUSH in stack or index operation,
- c_2 for the assignment operation,
- c_3 for compare operator
- c_4 for increment/add
- c_5 for Return operation

The analysis of the algorithm is summarized in the last three columns of Table 1.3. "Cost" represents the time units required to implement each operation in the statement and "Times" represents the number of times each statement is executed when running the algorithm. Line 5 for example, there are three operations;

- initialize i (assign 0 to i): costs c_2 and takes place once,
- compare costs c_3 and takes place n times, and
- increment costs c_4 and takes place also n times.

The last column of Table 1.3 represents the total executing time of each statement. It represents the contribution of each statement to the total execution time of the algorithm. If the statement takes c_i units of time to execute and is executed n times then it will contribute $c_i n$ units of time to the total execution time of the algorithm. The execution time of the algorithm is the sum of all the elements of the last column of Table 1.3.

$$\text{The total running time} = 2c_1 + c_2 + c_2 + c_2 + n(c_3 + c_4) + n(c_1 + c_2 + c_3) + c_5$$

For simplification, it is better to define one time unit that can be used to execute any of the elementary operations. In other words let us assume that the cost of any of the above mentioned operations is the same and equals to one unit time, i.e.

$$c_1 = c_2 = c_3 = c_4 = c_5 = 1$$

In this case the cost of running the algorithm will be $5n + 6$.

The expression for the total number of time units in the code fragment can be interpreted as a function of n; that is the total number of time units required to execute the algorithm depends directly on the size of the input data. Thus, one can write a complexity function for the algorithm:

$$f(n) = 5n + 6$$

Example 1.6

Suppose we are given an English short story TEXT, and suppose we want to search through TEXT for the first occurrence of a given 3-letter word W. If W is the 3-letter word "the" then it is likely that W occurs near the beginning of TEXT, so $f(n)$ will be small. On the other hand, if W is the 3-letter word "zoo," then W may not appear in TEXT at all, so $f(n)$ will be large.

The above discussion leads us to the question of finding the complexity function $f(n)$ for certain cases. The two cases one usually investigates in complexity theory are as follows:

1. *Worst case*: the maximum value of $f(n)$ for any possible input
2. *Average case*: the expected value of $f(n)$

Sometimes we also consider the minimum possible value of $f(n)$, called the *best case*.

The analysis of the average case assumes a certain probabilistic distribution for the input data; one such assumption might be that all possible permutations of an input data set are equally likely. The average case uses the following concept from probability theory. Suppose the numbers $n_1, n_2, \ldots n_k$ occur with respective probabilities $p_1, p_2, \ldots p_k$. Then the expectation or average value E is given by

$$\mathrm{E} = n_1 p_1 + n_2 p_2 + \cdots + n_k p_k$$

These ideas are illustrated in the following example.

Example 1.7: Linear Search

Suppose a linear array DATA contains n elements, and suppose a specific ITEM of information is given. We want either to find the location LOC of ITEM in the array DATA, or to send some message, such as LOC = 0, to indicate that ITEM does not appear in DATA. The linear search algorithm solves this problem by comparing ITEM, one by one, with each element in DATA. That is, we compare ITEM with DATA[l], then DATA[2], and so on, until we find LOC such that ITEM = DATA[LOC]. A formal presentation of this algorithm follows (Figure 1.4).

Solution

The complexity of the search algorithm is given by the number C of comparisons between ITEM and DATA[K]. We seek $C(n)$ for the worst case and the average case.

Algorithm: (Linear Search) A linear array DATA with N elements and a specific ITEM of information are given. This algorithm finds the location LOC of ITEM in the array DATA or sets LOC= 0.

1. [Initialize] Set K : =1 and LOC : = 0.
2. Repeat Steps 3 and 4 while LOC = 0 and K ≤ N.
3. If ITEM = DATA[K], then: Set LOC:= K.
4. Set K := K + 1. [Increments counter.]
   [End of Step 2 loop.]
5. [Successful?]
   If LOC =0, then:
   Write: ITEM is not in the array DATA.
   Else:
   Write: LOC is the location of ITEM.
   [End of If structure]
6. Exit.

Figure 1.4 Linear search algorithm.

Worst Case

Clearly the worst case in this example occurs when ITEM is the last element in the array DATA or is not there at all. In either situation, we have

$C(n) = n$

Accordingly, $C(n) = n$ is the worst-case complexity of the linear search algorithm.

Average Case

Here we assume that ITEM does appear in DATA, and that it is equally likely to occur at any position in the array. Accordingly, the number of comparisons can be any of the numbers $1, 2.3, \ldots, n$, and each number occurs with probability $p = 1/n$. Then

$$\begin{aligned} C(n) &= \frac{1.1}{n} + \frac{2.1}{n} + \cdots + n \cdot \frac{1}{n} \\ &= (1 + 2 + \cdots + n) \cdot \frac{1}{n} \\ &= \frac{[n(n+1)/2.1]}{n} = \frac{n+1}{2} \end{aligned}$$

This agrees with our intuitive feeling that the average number of comparisons needed to find the location of ITEM is approximately equal to half the number of elements in the DATA list.

Remark: The complexity of the average case of an algorithm is usually much more complicated to analyse than that of the worst case. Moreover, the probabilistic distribution that one assumes for the average case may not actually apply to real situations. Accordingly, in many cases and unless otherwise stated or implied, the complexity of an algorithm shall mean the function

which gives the running time of the worst case in terms of the input size. This is not too strong an assumption, since the complexity of the average case for many algorithms is proportional to the worst case.

Order of growth: Big O Notation or Θ Notation

In the last analysis we ended with some expressions that take, in general, the form of a polynomial of n, e.g.

$$C(n) = an^2 + bn + c$$

Where a, b, and c are some constants, which depend on the statement cost c_i. It is possible to make some more simplifications on such expressions to get what is known as the "*rate of growth*" or "*order of growth*". It is actually the rate of growth or the order of growth, of the running time that interests the designer. To get the rate of growth, we consider only the first term of the polynomial (e.g. an^2). The rest of the formula, the lower order terms (e.g. $bn + c$), are relatively insignificant for large n. The order of growth ignores also the constant coefficient of the leading term, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs. We end with measuring the rate of growth by comparing $f(n)$ with some standard function, such as:

$$log_2 n, n, n\ log_2 n, n^2, n^3, 2^n$$

The rates of growth for these standard functions are indicated in Table 1.4, which gives their approximate values for certain values of n. Observe that the functions are listed in the order of their rates of growth: the logarithmic function $\log_2 n$ grows most slowly, the exponential function 2^n grows most rapidly, and the polynomial functions a^c grow according to the exponent c. One way to compare the function $f(n)$ with these standard functions is to use the functional O notation defined as follows:

Suppose $f(n)$ and $g(n)$ are functions defined on the positive integers with the property that $f(n)$ is bounded by some multiple of $g(n)$ for almost all n. That is, suppose there exist a positive integer n_0 and a positive number M such that, for all $n > n_0$, we have

$$|f(n)| \leq \mathrm{M}|g(n)|$$

Then we may write:

$$f(n) = O(g(n))$$

Table 1.4 Growth rates of exponential functions.

g(n)						
n	$\log n$	N	$n \log n$	n^2	n^3	2^n
5	3	5	15	25	125	32
10	4	10	40	100	10^3	10^3
100	7	100	700	10^4	10^6	10^{30}
1000	10	1000	10,000	10^6	10^9	10^{300}

Table 1.5 Complexity of common search/sort algorithms.

Algorithm	Complexity
Linear search	$O(n)$
Binary search	$O(\log n)$
Merge–sort	$O(n \log n)$
Bubble sort	$O(n^2)$

which is read "$f(n)$ is of order $g(n)$." For example, any polynomial P(n) of degree m can be written using O-notation as P(n) = $O(n^m)$; e.g.,

$$8n^3 - 576n^2 + 832n - 248 = O(n^3)$$

To indicate the convenience of this notation, we give the complexity of certain well known searching and sorting algorithms in Table 1.5.

1.4 Economic Design Metrics

It is important for a system designer to be able to predict the cost and time required to design the system. This can guide the choice of an implementation strategy. This section will summarize a simplified approach to estimate the cost and time values. The section differentiates between cost and price and shows the relationship between them: price is what you sell a finished product for, and cost is the amount spent to produce it, including overheads. The section will introduce the reader to the major factors that affect cost of a processor design and discuss how these factors are changing over time.

1.4.1 Time-to-Market

Time-to-market and product lifetime (market window): Each product has a lifetime cycle which, in general, consists of a start up period beginning when the product is first released to market, a maturity period (market-window) where sales reach a peak and an end-of-life period where the product sales decline until it goes out of the market (see Figure 1.5). The fast growth of

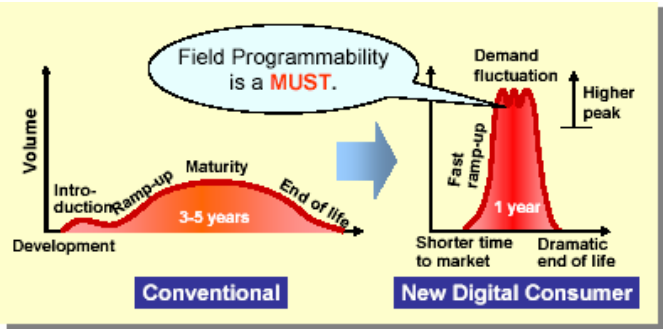


Figure 1.5 Dramatic change in product life cycle (market window).

microelectronic industries and the new fields of applications that use microelectronic components reflect directly on the lifetime of the products. Some of the effects are:

- Life cycle: Till recently products had a life cycle of 3 to 5 years. The lifetime of the microelectronic products is now often only one year.
- The start up period has been shortened and the product reaches maturity faster.
- Higher and narrower peak.
- The product reaches its end of life cycle faster.

This means that we are now in an era that is characterised by short market-windows. Missing this short market window means a great loss in sales. It is reported that in some microelectronic industries one day delay in introducing the product to the market can cause a loss of a one-million dollars. One way for the industry to respond to these trends is to move faster towards programmable technologies.

Loss of revenue due to delayed entry: Let's investigate the loss of revenue that can occur due to a delayed entry of a product in the market. We'll use a simplified model of revenue that is shown in Figure 1.6(b). This model assumes the market peaks mid way through the product life-cycle (denoted by W) and that the market peaks at the same time, even for a delayed entry. The revenue for an on-time market entry is the area of the triangle labelled On-time, and the area of the triangle labelled Delayed represents the revenue for delayed entry product. The difference between the areas of the two triangles represents the loss in the revenue due to delayed entry. The percentage revenue loss can

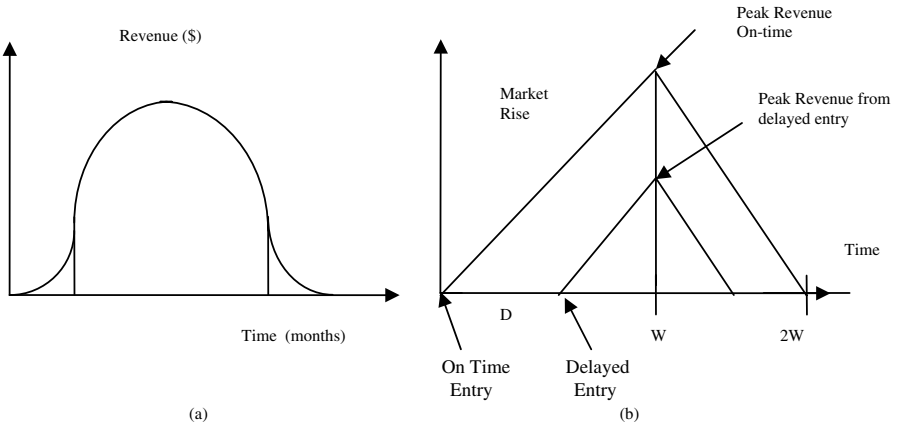


Figure 1.6 Time-to-market: a) Market window; b) simplified revenue model.

be calculated as follows:

Percentage loss = {[(area of on-time triangle) – (area of delayed triangle)]/ [area of on-time]} × 100%

If we consider the lifecycle of Figure 1.6 we get (assuming that triangle has an angle of 45°:

$$\begin{aligned}\text{Percentage loss} &= \{(1/2 \times 2W \times W) - 1/2(W - D + W) \\ &\quad *(W - D)\}/[1/2 \times 2W \times W]\} * 100\% \\ &= \{D(3W - D)/2W2\} \times 100\end{aligned}$$

Consider a product whose lifetime is 52 weeks, so W = 26. According to the preceding equation, a delay of just D = 5 weeks results in a revenue loss of 27%, and a delay of D = 10 weeks results in a loss of 50%.

1.4.2 Design Economics

In Section 1.2 we defined the NRE cost and the unit cost as parts of the design metrics. The NRE and the unit costs represent two of the factors that define the final total cost of the product. The total product cost, in general, consists of two parts:

- Fixed costs or Cost of the "first silicon" or the "cost of the first working prototype", and
- Variable costs (proportional mainly to the number of products sold i.e. the sales volume).

The product cost is related to the fixed and variable costs by the equation:

$$\begin{aligned}\text{Total product cost} &= \text{fixed product cost} + \text{variable product cost} \\ &\quad * \text{products sold} \end{aligned} \tag{1.6}$$

If a product made from parts (units), then the total cost for any part is

$$\begin{aligned}\text{Total part cost} &= \text{fixed part cost} + \text{variable cost per part} \\ &\quad * \text{volume of parts} \end{aligned} \tag{1.7}$$

The selling price S_{total} of a single part, an integrated circuit in our case, may be given by

$$S_{total} = C_{total}/(1 - m) \tag{1.8}$$

Where

- C_{total} is the manufacturing cost of a single IC to the vendor
- m is the desired profit margin

The profit margin has to be selected to ensure a gross profit after overhead and the cost of sales (marketing and sales costs) have been considered. Normally a profit model representing the profit flow during the product lifetime is used to calculate the profit margin m.

Each term in the cost equations ((1.6 and 1.7) has an effect on the final unit price (C_{total} and S_{total}) and also plays an important role in selecting the best technology that can be used to implement the product to get the best performance with minimum unit price. We start by understanding the meaning of fixed and variable costs.

1.4.2.1 Fixed costs

Fixed costs represent the one-time costs that the manufacturer must spend to guarantee the successful development cycle of the product and which are not directly related to the production strategy or the volume of production. Producing any quantity of the product after designing will not add any extra value to the fixed cost. Fixed costs include:

- Non-recurring Engineering Costs (NREs) also called "first Silicon" costs:
  - engineering design cost E_{total}
  - prototype manufacturing cost P_{total}
- Fixed costs to support the product

These costs are amortized over the total number of ICs sold. F_{total}, the total non recurring cost, is given by

$$F_{total} = E_{total} + P_{total} \tag{1.9}$$

The NRE costs can be amortized over the lifetime volume of the product. Alternatively, they can be viewed as an investment for which there is a required rate of return. For instance, if \$1 M is invested in NRE for a chip, then \$10 M has to be generated for a rate of return of 10.

1.4.2.2 Engineering costs

The cost of designing a digital system (or IC) E_{total} should occur only once during the product design process. This shows importance of having exact, complete and correct specifications from the first beginning. The engineering costs include:

- personnel cost
- support costs
- The personnel costs might include the labour for:
- architectural design
- logic capture
- simulation for functionality
- layout of modules and chip
- timing verification
- test generation: Includes:
    - production test and design for test
    - test vectors and test-programme development cost
- second source

The support costs amortized over the life of the equipment and the length of the design project include

- Software Costs:
    - computer costs
    - programming costs (in case of FPGA)
    - CAD software costs
- Education or re-education costs. This includes training cost for a new electronic design automation (EDA) system

Costs can be drastically reduced by reusing modules (e.g. off the shelf modules) or acquiring fully completed modules from an intellectual property

vendor. As a guide the per annum costs might break down as follows (these figures are in US dollars for engineers in the USA around 2004):

Salary:	\$50—\$100 K
Overhead:	\$10—\$30 K
Computer:	\$10 K
CAD Tools (digital front end):	\$10 K
CAD Tools (analogue):	\$ l00 K
CAD Tools (digital back end):	\$ 1 M

The cost of the back-end tools clearly must be shared over the group designing the chips. Increasing the productivity of the members of the design team is an effective way of reducing the engineering costs.

1.4.2.3 Prototype manufacturing costs

These costs (P_{total}) are the fixed costs, which are required to get the first working unit of the product (first ICs in case of designing IC) from the vendor. These include the costs of all the hardware and software needed to convert the design into the first working unit. In case of designing an IC, for example, this cost includes:

- the mask cost
- test fixture costs (hardware costs)
- package tooling

The photo-mask cost is proportional to the number of steps used in the process. A mask currently costs between \$500 and \$30,000 or more. A complete mask set for 130 nm CMOS costs in the vicinity of \$500 K to \$1 M.

The test fixture consists of a printed wiring board probe assembly to probe individual die at the wafer level and interface to a tester. Costs range from \$1000 to \$50,000, depending on the complexity of the interface electronics.

If a custom packaging is required, it may have to be designed and manufactured (tooled). The time and expense of tooling a package depends on the sophistication of the package. Where possible, standard packages should be used.

Example 1.8

You are starting a company to commercialise your brilliant research idea. Estimate the cost to prototype a mixed-signal chip. Assume you have seven digital designers (each of salary \$70 K and costs an overhead of \$30 K), three analogue designers (each of salary \$100 K and costs an overhead of \$30 K),

and five support personnel (each of salary \$40 K and an overhead of \$20 K) and that the prototype takes two fabrication runs and two years.

Solution:

Total cost of one digital Engineer per year
= Salary + Overhead + Computer used
+ Digital front end CAD tool
= \$70 K + \$30 K + \$10 K + \$10 K = \$120K
Total cost of one analogue Engineer per year
= Salary + Overhead + Computer used
+ Analogue front end CAD tool
= \$100 K + \$30 K + \$10 K + \$100 K = \$240 K
Total cost of one support staff per year
= Salary + Overhead + Computer used
= \$40K + \$20K + \$10K = \$70K
Total cost per year:
Cost of 7 digital engineers = 7 ∗ \$120 K = \$ 780 K
Cost of 3 analogue engineers = 3 ∗ \$240 K = \$ 720 K
Cost of 5 support staff = 5 ∗ \$70 K = \$ 210 K
Two fabrication runs = 2 ∗ \$1 M = \$ 2 M
Total cost per year = \$ 3.91 M
The total predicted cost here is nearly \$8 M.

Figure 1.7 shows the breakdown of the overall cost.

It is important for the project manager to find ways to reduce fabrications costs. Clearly, the manager can reduce the number of people and the labour cost. He might reduce the CAD tool cost and the fabrication cost by doing multi-project chips. However, the latter approach will not get him to a pre-production version, because issues such as yield and behaviour across process variations will not be proved.

1.4.2.4 Fixed costs to support the product

Once the system has been designed and put into manufacturing, the cost to support the product from an engineering viewpoint may have few sources. Data sheets describing the characteristics of the system (or the IC) have to

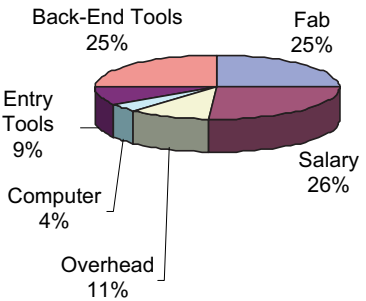


Figure 1.7 Breakdown of prototyping costs for a mixed signal IC.

be written. This is valid also even for application-specific ICs that are not sold outside the company that developed them. From time to time, application notes describing how to use the system or the IC may be needed. In addition, specific application support may have to be provided to help particular users. This is especially true when the product is an Application Specific IC (ASIC). In such systems the designer usually becomes the main player who knows everything on the circuit.

Of course, every product designed should have accompanying documentation that explains what it is and how to use it. This even applies to chips designed in the academic environment because the time between design submission and fabricated chip can be quite large and can tax even the best memory.

1.4.2.5 Variable costs and recurring costs

Variable Costs:

Represent all the costs after the development (or design) stage. It is directly related to the production phase (including production strategy) and production volume. Some examples of the variable costs are profit margin, price per gate, part cost plus the "cost of sales" which is the marketing, sales force, and overhead costs associated with selling each product.

Cost Equation and Selecting the Best Technology

To understand the importance of each term in the total cost equation (equation 1.6) and the rule it plays in selecting the best technology that can be used for implementing any product, assume that it is required to implement an ASIC circuit using one of three possible technologies: FPGA, MGA, or

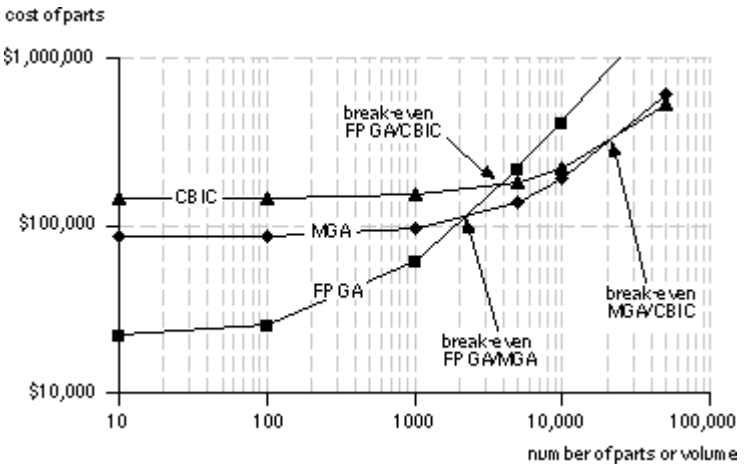


Figure 1.8 Break even graph.

CBIC. Suppose we have the following (imaginary) fixed and variable costs (the meaning of fixed and variable costs will be discussed later):

- FPGA: $21,800 (fixed) and $39 (variable)
- MGA: $86,000 (fixed) and $10 (variable)
- CBIC $146,000 (fixed) and $8 (variable)

Ignoring all other design metrics, like time-to-market, the best technology choice will depend on the number of units we plan to produce (more accurately, the volume of parts). In Figure 1.8, for each of the three technologies, we plot total cost versus the number of parts sold. From the plot we can calculate the following break-even volumes:

- FPGA/MGA @ 2000 parts
- FPGA/CBIC @ 4000 parts
- MGA/CBIC @ 20,000 parts

The above results mean that, of the three technologies, FPGA technology yields the lowest total cost for low volumes, namely for volumes between 1 and 2000. MGA technology yields the lowest total cost for volumes between 2000 and 20,000 and CBIC technology yields the lowest cost for volumes above 20,000.

Recently, the rapid rise in costs associated with the use of fine design rules has become an issue at semiconductor manufacturing plants. Similarly, the cost of producing the first sample of a new chip, the “first silicon", has been

increasing every year. Manufacturers noted that the cost for the “first silicon" in the latest 90 nm process is six times that of the earlier 0.35 um process.

To handle these increasing costs, industry has concentrate on producing dynamically programmable devices, which allow the LSI functions to be determined after the chip is produced, rather than letting the end users to create high-cost custom LSIs for his application. As a matter of fact the trends towards increased performance and reduced costs in programmable devices (reconfigurable LSIs) are progressing surely from the fact that the difference in cost between FPGA and ASIC solutions continues increase every year

1.5 Power Design Metrics

In the early days of computers, there was no large concern about how much power a processor used. At that time the number of computers was small and the amount of work needed from them was not that much. Now with hundreds of millions of computers plus the possibility that one organization may have thousands, things have changed completely; power consumption has become a major concern. An additional change that has driven the power issue is the massive change to mobility. Mobile systems are everywhere today (e.g. mobile communication, mobile computing, medical implants, deep space applications, etc.) almost exclusively using batteries as the power source. The limited and costly power offered by batteries has placed another constraint on the design of such mobile systems; how to reduce the power consumption?

In general, the power usage and the voltage support of a processor are important for the following reasons:

- Power consumption equates largely with heat generation, which is a primary enemy in achieving increased performance. Newer processors are larger and faster, and keeping them cool can be a major concern.
- In the embedded domain, applications cannot use a heat sink or a fan. A cellular phone with a fan would probably not be a top seller.
- Reducing power usage is a primary objective for the designers of notebook computers and embedded systems, since they run on batteries with a limited life and may even rely on “super capacitors” for remaining operational during power outages.

Newer processors strive to add additional features, integrate more and more peripherals and to run at faster speeds, all of which tend to increase power

consumption. This trend causes ICs to be more sensitive to heat problems since their components are crammed into such a small space.

1.5.1 Reducing Power Consumption

There are three conditions that have to be considered when planning for a microcontroller's power consumption in an application.

1. The "intrinsic power," which is the power required just to run the microcontroller.
2. The "I/O drive power," which takes into account the power consumed when the microcontroller is sinking/sourcing current to external I/O devices.
3. The power consumed when the microcontroller is in "sleep" or "standby" mode and is waiting with clocks on or off for a specific external event.

Taking these three conditions into account when planning an application can change an application that would have only run for a few hours to literally several months.

1.5.1.1 Reducing the intrinsic power

Most embedded microprocessors have three different modes: fully operational; standby or power down; and clock-off (or sleep). (Some vendors use different names, but they mean basically the same thing.) ***Fully operational*** means that the clock signal is propagated to the entire processor and all functional units are available to execute instructions. In ***standby*** mode, the processor is not actually executing an instruction, but all its stored information is still available—for example, DRAM is still refreshed, register contents are valid, and so forth. In the event of an external interrupt, the processor returns (in a couple of cycles) to fully operational mode without losing information. In ***clock-off*** (***sleep***) mode, the system has to be restarted, which takes nearly as long as the initial start-up. The "intrinsic power" consumption reaches its maximum value when the processor is in fully operational mode.

1.5.1.2 Effect of Clock Speed on the Power Consumption

Reducing the power consumption starts by using lower-power semiconductor processor. All microcontrollers built today use "complementary metal oxide semiconductor" (CMOS) logic technology to provide the computing functions and electronic interfaces. CMOS-based devices require significantly less power than older bipolar or NMOS-based devices. "CMOS" is a "push-pull"

technology in which a "PMOS" and "NMOS" transistor are paired together. During a state transition, a very small amount of current will flow through the transistors. As the frequency of operation increases, current will flow more often in a given period of time. In other words, the average current will be going up. This increased current flow will result in increased power consumption by the device. This type of power that is consumed by the microcontroller when it is running and nothing is connected to the I/O pins is called "Intrinsic power". As mentioned before, this consumption of power is largely a function of the CMOS switching current, which in itself is a function of the speed of the microcontroller.

Lowering the clock speed used in an application will reduce the required power but may increase the time required to run software programmes. Lowering the clock speed can be used in applications where battery life is more important than a slight change in the running time of the software. As a result we can say that a CMOS device should be driven at the slowest possible speed, to minimize power consumption. ("Sleep" mode can dramatically reduce a microcontroller's power consumption during inactive periods because if no gates are switching, there is no current flow in the device.) A more recent addition to lowering the power consumption of processors is due to many processors now being equipped with run time programmable oscillators. These enable the application itself to slow down a processor's clock during periods of low demand. For instance a small processor with a full clock speed power consumption of 100 mW can slow down the clock significantly to get the power consumption to 100 μW.

In some applications reducing the speed is not acceptable. This is why most of the new processors focus on reducing power consumption in fully operational and standby modes. They do so by stopping transistor activity when a particular block is not in use. To achieve this, such designs connect every register, flip-flop, or latch to the processor's clock tree. The implementation of the clock therefore becomes crucial, and it often must be completely redesigned. (In traditional microprocessor design, the clock signal is propagated from a single point throughout the chip.)

1.5.1.3 Effect of reducing the voltage on the power consumption

The power consumption, mainly the intrinsic power, can be further reduced by reducing the supply voltage to the microcontroller/microprocessor (which may or may not be possible, depending on the circuitry attached to the microcontroller). A CPU core voltage of 1.8V is the state-of-the-art processor technology.

The addition of variable clocks, sleep mode and low supply voltage has tended to shift the power issue away from the CPU toward peripheral devices and other components. The IC now represents a complete system due to the increase of the integration. Many ICs now includes on chip many power-consuming peripherals alongside embedded cores. This means that when speaking on power consumption, we must consider the power consumption of the entire system. Overall power consumption differs considerably depending on the system design and the degree of integration. Increasingly the processor core is only a small part of the entire system.

1.5.1.4 Reducing the I/O drive power

The I/O drive power is a measurement of how much power is sourced/ sunk by the microcontroller to external devices and is unique to the application. In many applications, the microcontroller is the only active device in the circuit (i.e., it gets input from switches and outputs information via LCD displays). If the microcontroller is driving devices continually at times when they are not required, more current (which means more power) than is absolutely necessary is being consumed by the application.

1.5.1.5 Reduce power during standby mode

The last aspect of power consumption to consider when developing an application is the consumption during the sleep or standby mode. The processor usually enters this mode by executing a special instruction which then shuts down the oscillator and waits for some event to wake it again (such as a watchdog timer to count down or an input to change state).

Using this mode can reduce the power consumption of a microcontroller from milliwatts to microwatts.

Using the sleep mode in a microcontroller will allow the use of a virtual "on/off" switch that is connected directly to the microcontroller. This provides several advantages. The first is cost and reliability; a simple momentary on/off switch is much cheaper and much less prone to failure than a slide or toggle switch. Second is operational; while sleep mode is active, the contents of the variable RAM will not be lost or changed.

There is one potential disadvantage of sleep mode for some applications, and that is the time required for the microcontroller to "wake up" and restart its oscillator. This can be as long as the initial start-up time of several milliseconds which may be too long for some applications. Such delays are usually

a problem when interfacing to electronic components but are generally not a problem when interfacing with a person.

One thing to remember with sleep mode is to make sure there is no current drawn when it is active. A microcontroller sinking current from an LED connected to the power rail while in sleep mode will result in extra power being consumed.

1.5.1.6 Use of power management

Spurred on primarily by the goal of putting faster and more powerful processors in laptop computers, Intel has created power management circuitry to enable processors to conserve energy use and lengthen battery life. This feature set is called SMM, which stands for “System Management Mode".

SMM circuitry is integrated into the physical chip but operates independently to control the processor’s power use based on its activity level. It allows the user to specify time intervals after which the CPU will be powered down partially or fully, and also supports the suspend/resume feature that allows for instant power on and power off, used mostly with laptop PCs.

1.6 System Effectiveness Metrics

It is very important for a design group to measure the performance and the capabilities of the product they just finished and to determine that it is capable of accomplishing its mission. There are a number of metrics that indicate the overall performance of a system. The most popular are reliability, availability and system effectiveness. Reliability (as will be explained latter) is the probability of continued successful operation, whereas availability (see later) is the probability that the system is operational and available when it is needed. System effectiveness is the overall capability of a system to accomplish its mission and is determined by calculating the product of reliability and availability.

The concept of system effectiveness was introduced in the late 1950s and early 1960s to describe the overall capability of a system for accomplishing its intended mission. The mission (to perform some intended function) was often referred to as the ultimate output of any system. Various system effectiveness definitions have been presented. The definition in ARINC is:

”the probability that the system can successfully meet an operational demand within a given time when operated under specified conditions.”

To define system effectiveness for a one-shot device such as a missile, the definition was modified to:

> *"the probability that the system [missile] will operate successfully [destroy the target] when called upon to do so under specified conditions."*

Effectiveness is obviously influenced by equipment design and construction (mainly ***reliability***). However, just as critical are the equipment's use and maintenance (***maintainability***). Another famous and widely used definition of system effectiveness is from MIL-STD-721B;

> *"a measure of the degree to which an item can be expected to achieve a set of specific mission requirements and which may be expressed as a function of* ***availability***, ***dependability*** *and* ***capability.****"*

From the above simple discussion, it is clear that any model for system effectiveness must include many different attributes. Many of the design metrics given in Section 1.2 represent some of the attributes. Beside reliability, maintainability, serviceability, availability and design adequacy mentioned in Section 1.2:

- ***Repairability***: the probability that a failed system will be restored to operable condition in a given active repair time
- ***Capability:*** a measure of the ability of an item to achieve mission objectives given the conditions during the mission
- ***Dependability:*** a measure of the item operating condition at one or more points during the mission, including the effects of reliability, maintainability and survivability, given the item conditions at the start of the mission
- ***Human performance:*** Human performance and environmental effects are attributes used in some system effectiveness models.

In the following, we discuss some of the attributes (also design metrics) used in many of the system effectiveness models.

1.6.1 Reliability, Maintainability and Availability Metrics

Digital systems, as with all other sophisticated equipment, undergo a cycle of repair, check-out, operational readiness, failure and back to repair. When the cost of a machine being out of operation is high, methods must be applied to reduce these out-of-service or downtime periods. The cost of downtime is not simply the lost revenue when the system is not used, but also the cost

of having to rerun programmes that were interrupted by the ailing system (especially if the system is a computer), retransmitting or requesting other terminals to resend (if possible) copies of real-time data lost, loss of control of external processes, opportunity costs, and costs related to user inconvenience, dissatisfaction and reduced confidence in the system. Other costs are related directly to the diagnosis and corrective repair actions, and associated logistics and book keeping.

Due to the complexity of digital systems, many users often decide not to maintain the system themselves but rather to have a maintenance contract with the system manufacturer. The cost of a maintenance contract over the useful life of the system in relation to its capital cost is quite high. Some literature suggests that roughly 38% of the life cycle cost is directed toward maintainability issues. Such high costs due to unreliability and maintenance provide a strong argument for designing for reliability, maintainability and serviceability.

1.6.1.1 Reliability

Reliability is an attribute of any component (software, hardware, or a network, for example) that consistently performs according to its specifications. It has long been considered one of three related attributes that must be considered when making, buying or using any product or component. Reliability, availability and maintainability (RAM for short) are considered to be important aspects to design into any system. (Note: sometimes the term serviceability is used instead of maintainability. In this case RAS is used instead of RAM). Quantitatively, reliability can be defined as the probability that the system will perform its intended function over the stated duration of time in the specified environment for its usage. In theory, a reliable product is totally free of technical errors; in practice however vendors frequently express a product's reliability quotient as a percentage.

Software bugs, instruction sensitivity and problems that may arise due to durability of the EEPROM and Flash memories (the nature of the EEPROM architecture, limits the number of updates that may be reliably performed on a single location — called the durability of the memory.) are some of the possible reasons for the failure of embedded systems.

Reliability of a system depends on the number of devices and connections used to build the system. As the number of devices and the number of interconnections increases, the chance of system unreliability becomes greater, since the reliability of any system depends on the reliability of its components. The relationship between parts reliability and the system reliability depends

mainly on system configuration and the reliability function can be formulated mathematically to varying degrees of precision, depending on the scale of the modelling effort. To understand how the system reliability depends on the system configuration and to understand how to calculate the system reliability given the reliability of each component, we consider two simple examples of a system consisting of *n*components. These components can be hardware, software or even human.

Let $\rho(A_i)$, $(1 \leq i \leq n)$ denote the probability of event Ai that component i operates successfully during the intended period of time. Then the reliability of component i is $R_i = \rho(A_i)$. Similarly, let $\rho(\overline{A}_i)$ denote the probability of event $\overline{A}_i$ that component i fails during the intended period. In the following calculations, we are going to assume that the failure of any component is independent of that of the other components.

Case 1: ***Serial Configuration***:

The series configuration is the simplest and perhaps one of the most common structures. The block diagram in Figure 1.9 represents a series system consisting of n components.

In this configuration, all n components must be operating to ensure system operations. In other words, the system fails when any one of the n components fails. Thus, the reliability of a series system R_s is:

$$\begin{aligned} R_s &= \rho \text{ (all components operate successfully)} \\ &= \rho(A_1 \cap A_2 \cap \ldots \cap A_n) \\ &= \prod_{i=1}^{n} \rho(A_i) \end{aligned}$$

The last equality holds since all components operate independently. Therefore, the reliability of a series system is:

$$R_s = \prod_{i=1}^{n} r_i \tag{1.10}$$

Case 2: ***Parallel Configuration***

In many systems, several paths perform the same operation simultaneously forming a parallel configuration. A block diagram for this system is shown

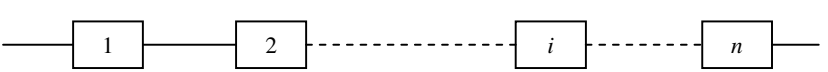


Figure 1.9 Serial component configuration.

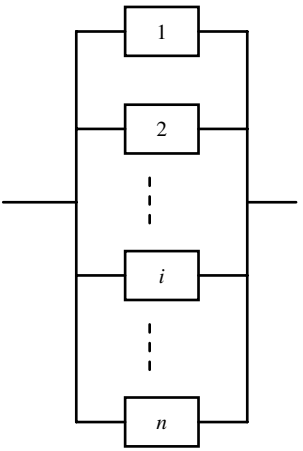


Figure 1.10 Parallel components configuration.

in Figure 1.10. There are n paths connecting input to output, and the system fails if all the n components fail. This is sometimes called a redundant configuration. The word "redundant" is used only when the system configuration is deliberately changed to produce additional parallel paths in order to improve the system reliability. Thus, a parallel system may occur as a result of the basic system structure or may be produced by using redundancy specifically to enhance the reliability of the system.

In a parallel configuration consisting of n components, the system is successful if any one of the n components is successful. Thus, the reliability of a parallel system is the probability of the union of the n events $\mathrm{A}_1, A_a, \ldots, A_n$. which can be written as:

$$\begin{aligned} R_p &= \rho(A_1 \cup A_2 \cup \ldots \cup A_n) \\ &= 1 - \rho(\overline{A_1} \cap \overline{A_2} \cap \ldots \cap \overline{A_n}) \\ &= 1 - \prod_{i=1}^{n} \rho(\overline{A_i}) = 1 - \prod_{i=1}^{n} \{1 - \rho(A_i)\} \end{aligned}$$

The last equality holds since the component failures are independent. Therefore, the reliability of a parallel system is:

$$R_s = \left(1 - \prod_{i=1}^{n} (1 - r_i) \right. \tag{1.11}$$

Equations (1.10) and (1.11) show how the configuration of the components affects the system reliability. In addition, it is possible to recognize two distinct

and viable approaches to enhance system reliability; one on the level of the components and the second on the level of the overall system organization.

1. *Component technology*: The first approach is based on component technology; i.e., manufacturing capable of producing components with the highest possible reliability, followed by parts screening, quality control, pretesting to remove early failures (infant mortality) etc.
2. *System organization*: The second approach is based on the organization of the system itself i.e., *fault-tolerant* architectures that make use of protective *redundancy* to mask or remove the effects of failure, and thereby provide greater overall system reliability than would be possible by the use of the same components in a simplex or non-redundant configuration.

When designing for reliability, the primary goal of the designer is to find the best way to increase system reliability. Accepted principles for doing this include:

1. to keep the system as simple as is compatible with performance requirements;
2. to increase the reliability of the components in the system;
3. to use parallel redundancy for the less reliable components;
4. to use standby redundancy (hot standby) components that can be switched in when failures occur;
5. to use maintenance repair where failed components are replaced but not automatically switched in;
6. to use preventive maintenance such that components are replaced by new ones whenever they fail, or at some fixed time interval, whichever comes first;
7. to use better arrangement for exchangeable components;
8. to use large safety factors or management programmes for product improvement.

1.6.1.2 Maintainability

A qualitative definition of maintainability M is given by Goldamn and Slattery (1979) as:

> *"…The characteristics (both qualitative and quantitative) of material design and installation which make it possible to meet operational objectives with a minimum expenditure of maintenance effort (manpower, personnel skill, test equipment, technical data, and maintenance support facilities) under operational environmental conditions in which scheduled and unscheduled maintenances will be performed"*

Recently, maintainability is described in MIL-HDBK-470A dated 4 August 1997, "Designing and Developing Maintainable Products and Systems" as:

> *"The relative ease and economy of time and resources with which an item can be retained in, or restored to, a specified condition when maintenance is performed by personnel having specified skill levels, using prescribed procedures and resources, at each prescribed level of maintenance and repair. In this context, it is a function of design."*

Based on the qualitative definitions, maintainability can also be expressed quantitatively by means of probability theory. Thus quantitatively, according to Goldamn and Slattery,

> *"…maintainability is a characteristic of design and installation which is expressed as the probability that an item will be restored to specified conditions within a given period of time when maintenance action is performed in accordance with prescribed procedures and resources."*

Mathematically, this can be expressed as:

$$M = 1 - e^{-t/\mathrm{MTTR}}$$

Where t is the specified time to repair, and MTTR is the mean time to repair.

The importance of focusing on maintainability is indicated by some articles which suggest that roughly 38% of the life cycle cost is directed toward maintainability issues.

Supportability has a design subset involving ***testability*** a design characteristic that allows verification of the status to be determined and faults within the item to be isolated in a timely and effective manner. This can occur via build-in-test equipment so that the new item can demonstrate its status (operable, inoperable or degraded).

Maintainability is primarily a design parameter. Designing for maintainability determines how long equipment will be down and unavailable in times of failure or maintenance. This time can be affected by the quality of the maintenance workforce, by their training and also by the availability of spares when needed. However, few things reduce the maintenance time (and cost) as much as having the system effectively designed for maintenance.

1.7 Summary of the Chapter

In this chapter, we studied five groups of metrics namely; time, space (size), cost (to the customer), power (consumed or battery life) and effectiveness

that have a direct effect on the performance and the efficiency of a system. We studied several measures of performance and compared them. These measures can start at a high level with big O analysis and then moved to a lower level of detail when examining the performance of the system. We introduced a several metrics for assessing embedded/digital performance. We introduced some ways for reducing the power consumption including power management.

1.8 Review Questions

1.1 What is meant by the performance of embedded application?
1.2 What is the difference between an optimization and a trade-off?
1.3 List a pair of design metrics that may compete with one another, providing an intuitive explanation of the reason behind the competition.
1.4 Does performance optimization apply equally to the hardware and software components forming an embedded system?
1.5 Using the revenue model of Figure 1.6(b), derive the percentage revenue loss equation for any rise angle, rather than just 45 degrees.
1.6 The design of a particular disk drive has an NRE cost of $100,000 and a unit cost of $20. How much will we have to add to the cost of each product to cover our NRE cost assuming we sell: (a) 100 units, and (b) 10,000 units?
1.7 Describes the methods by which you can perform a time loading analysis of an embedded application. Discuss the advantage and disadvantages of each.
1.8 The quick sort algorithm has a worst case complexity of O ($N\log_2 N$) where N is the size of the search range. Provide the complexity analysis to show this is correct.
1.9 Suppose that we are considering an enhancement that runs 10 times faster than the original machine but is only usable 40% of the time. What is the overall speedup gained by incorporating the enhancement?
1.10 Implementation of floating-point (FP) square root varies significantly in performance. Suppose FP square root (FPSQR) is responsible for 20% of the execution time of a critical benchmark on a machine. One proposal is to add FPSQR hardware that will speed up this operation by a factor 10. The other alternative is just to try to make all FP instructions run faster; FP instructions are responsible for a total of 50% of the execution time. The design team believes that they can make all FP instructions run two times faster with the same effort as required to the fast square root. Compare these two design alternatives

1.11 A programme runs in 100 seconds on a machine with multiply operations responsible for 80 seconds of this time. By how much must the speed of multiplication be improved to make the programme four times faster?

1.12 For the RISC machine with the following instruction mix:

Operation	Frequency	Cycles	CPI(i)	% Time
ALU	50%	1	0.5	23%
Load	20%	5	1.0	45%
Store	10%	3	0.3	14%
Branch	20%	2	0.4	18%

If a CPU design enhancement improves the CPI of load instructions from 5 to 2, use Amdahl's law to calculate the resulting performance improvement from this enhancement?

2

A System Approach to Digital System Design

THINGS TO LOOK FOR…

- Structured Design: The design life cycle of medium scale engineering systems.
- Traditional product life cycle.
- Design Process Technology: How to select the right design cycle for a project.
- Processor Technology: Are you going to select a software, hardware or hybrid solution for implementing your project?
- Hardware: What hardware solutions may be available?
- IC Technology: Which IC technologies are available and which one to select?
- How to use tools that can assist in the design process.

2.1 Introduction

In Chapter 1 we defined the design of a system as "the task of defining the functionality of a system and converting that functionality into a physical implementation, while satisfying certain constrained design metrics and optimizing other design metrics". We mentioned also that the functionality of modern microelectronics and embedded systems are becoming more and more complex and creating a physical implementation that satisfies constraints is also very difficult. The reasons of this design complexity are coming from the hundreds of options that the designers can use to design and to implement the target system. The design is not a one step process; it is a multi-steps operation. At each step there are challenges for the designer that require some decisions.

Some of the challenging questions that the designer will face are: which IC technology he is going to use? Is the software solution better or the hardware

solution? Is he going to use the off-shelf components (can be software or hardware components) or he has to design his own hardware components and his software programmes? To take such decisions the designer must gain knowledge about the most suitable way for approaching the solution; others require knowledge on the available IC technologies in the market, and so on.

In case of designing small digital circuit, it is possible to design it using ad hoc, intuitive method, using pen and paper, and a bit of brain cells. However, if the system is of moderate to high complexity, ad hoc methods do not usually lead to working systems (definitely not right-first-time). The design of moderate and complex digital systems consists of many stages; if the designer at any stage, especially at the early stages, selects a wrong option, it is very possible that he will end with a product that has functionality that differs from the required functionality or at least with a product that does not satisfy the design constraints. This normally represents a huge loss in money and time wasting. The only way to avoid such a situation is to find a "design methodology and tools" that helps the designer to go through well defined steps and guides him to the points where he has to take a decision and how to select one option.

The embedded systems engineers learned a lot from software engineers; they learned from structured, top-down design method developed for software engineering. Using such design approach, hardware engineers are now designing in a much more systematic way than before. Two other factors helped in that: computers are getting cheaper and more powerful and Computer Aided Design (CAD) tools are also becoming more available. It is now possible to design complex digital circuits systematically on a computer, simulate the entire design down to component level and verify that everything works before even considering making any hardware.

Selecting the design flow and the type of top-down model represents one of the decisions that the designer engineer has to decide at the beginning of the design process. In general, there are three fundamental decisions that have direct effect on the way of optimising the design metrics. These decisions are related to the technologies, the designer will to use while developing the system. These technologies are:

- **Design technology:** Which design flow and which top-down model he is going to use to speed up the design process?
- **Processor technology:** Which processor technology (software, hardware, combination between hardware and software, etc.) he is going to use to implement each functional block.

- **IC technology:** Which type of IC technology he is going to use (VLSI, PLD, FPGA, etc) to implement the chosen processor.

The selection of the proper technology at each stage determines the efficiency with which the design can be converted from concept to architecture, to logic and memory, to circuit, and ultimately to physical layout.

In this chapter we are introducing the design methodology and tools that any designer can use to convert the client requirements into a workable system.

2.2 System Design Flow

The design process takes the form of a sequence of steps called "***design flow***". Each step in the sequence receives inputs from the higher level and produces an output that goes to the next level. This step uses tools to speed up the design and ways to verify the correctness of the functionality of step.

When the process involves the implementation of a system or a processor, we sometimes refer to the process as a "*life cycle*" or a "*development cycle*". Thus, the software development process is sometimes called "*Software life cycle*" and the digital (in general the embedded) system development process is sometimes called "*embedded system development cycle*". "*Design flow*" is also a common name for the process.

This design flow considered in this chapter is that outlined in Figure 2.1. This is a simplified representation of what is known as the "*traditional embedded system development cycle*". The life cycle (or development cycle) is purely a descriptive representation. It contains a sequence of blocks. Each block identifies one of the development cycle phases, with its dominant activity indicted inside. The primary outcome of each phase is shown as output of the block. Each outcome is represented by documentation that describes the system at that stage of development. Producing accurate and complete documentation of the results from each phase of development and for the final system is extremely important. The same development cycle is applicable whether a single person or a team designs a system. Understanding the phases of the development cycle and completing them in a methodical fashion is critical to successful system design.

The meaning and the target of each block in the development cycle of Figure 2.1 are discussed in the following subsections.

2.2.1 Requirement Analysis

Requirements analysis involves a detailed examination of the needs of the end user — the problem to be solved. The needs of the end user, or customer, are

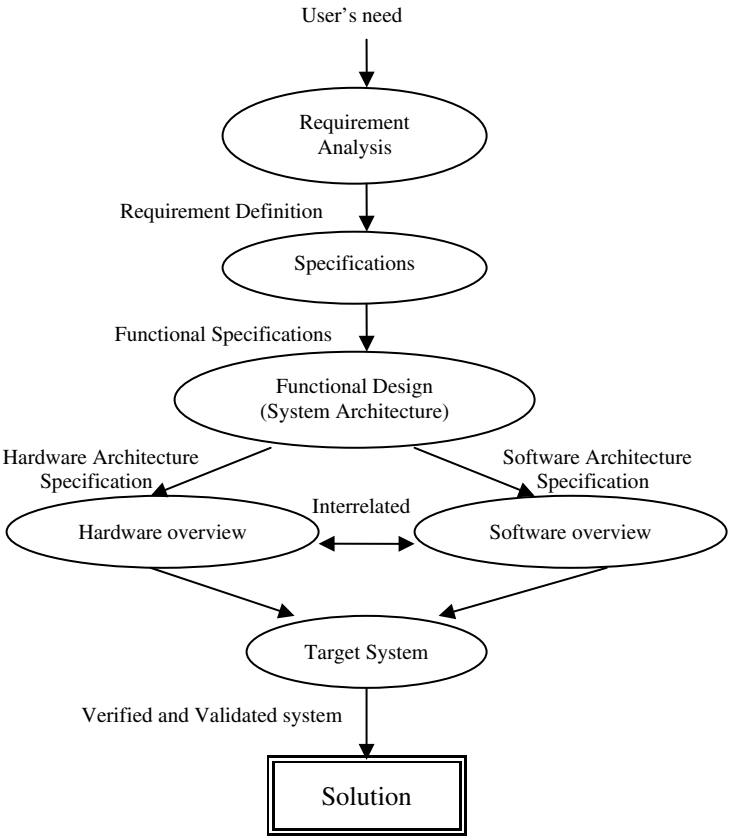


Figure 2.1 System design flow.

studied to determine what the user wants or needs the system to do. It may be necessarily for the designer at this stage to talk with everyone involved directly and sometimes indirectly while the system under design (i.e. all the stakeholders); to listen to different opinions; to see how the design might affect the people who have to work with it and what they are expecting from the new system. It is important for the designer to take his time to look at different views of the problem, to look at it from both the inside and the outside. This will end with a list of requirements. This requirement is defined from the user's point of view, and without any reference to possible solutions. Some non technical matters that may also be part of the requirements analysis are the determination of the overall schedule and budget. The results of the analysis are documented in a requirements definition.

2.2.2 Specifications

Based on the requirements definition, the essential functions of, and operational constraints for, the system are determined. Functional specifications define system features that can be directly tested. In addition to required functions, goals and objectives are also specified. What the system should do is determined, but not how it will be done. The user interface to the system is also defined. The desired functions and user interface are clearly documented in a functional specification.

The functional specifications outline the functional requirement of the entire system as supplied by the user. This may be altered during the design process to accommodate additional detailed specifications. However, the specification should be complete and formalized before the final stages of the design process take place, and should be considered to be unalterable. Sometimes it may be absolutely necessary to make changes to the specification but this should not be a normal requirement.

2.2.3 Functional Design: System Architecture

From the functional specification, the overall architecture or framework of the system is developed in a top down fashion. The architecture is broken down into manageable functional elements. This is followed by determining the relationships between the major functional elements. An objective is to define the functional elements so that the interactions among them are limited, resulting in a modular structure. Each functional element is first described independently of its implementation in either hardware or software.

In case of small systems, trial and error techniques may be used to convert the specifications into system architecture. Such a technique cannot be used with large systems. In large systems the designer has to use well defined techniques and tools to convert the specifications in a systematic way to system architecture. Studying such techniques and tools is an important part of embedded system design.

The techniques and tools needed to convert specifications into system architecture are called in this book "Design Technology". We are introducing this subject in Section 2.4.

Separating the functional specifications into Hardware and Software Components

The major functional elements are then partitioned into hardware subsystems and software modules and the interfaces between them are defined.

The designer in this step decides which "processor technology" he is intended to use to implement each subfunction.

The separation into hardware and software is a difficult task as there are many possible solutions and, depending upon the specifications, one product may implement a function in software, another may implement it in hardware. An example might be the conversion of binary coded decimal (BCD) into signals for a seven segment display. This function can be performed using a software look-up table, or by hardware using the appropriate logic decoder.

The software/hardware separation is determined by optimizing a variety of parameters (design metrics), such as speed of execution, cost, power consumption, size, memory size, programme size and so on, as determined by the user. In many applications the number of levels in the top-down structure represents a serious parameter in deciding the implementation of the interfaces between the different blocks. The decisions in this stage reflect what portion of each subfunction is implemented primarily in hardware and what portion is implemented primarily in software.

The system architecture is documented as a hardware and a software architecture specification. Selection of essential algorithms and data representations are part of the software architecture specification.

With separate hardware and software architecture specifications, it is possible for the development of the system to be splitted into separate, and possibly, concurrent hardware and software phases, as indicated by the split in the diagram.

In Section 2.6 we shall discuss in detail the "Processor Technology" and explain the factors (the constraints) that guide the designer to select the optimum processor technology for his project.

2.2.4 Hardware Overview

The hardware overview will consider various different alternatives, and decide which might be the optimum solution. This will involve some detailed overviews of the required metrics and constraints, before the selection of one particular solution. The hardware design process has been split into three sections as illustrated in Figure 2.2 called design, implementation and prototype testing.

2.2.4.1 Hardware design

Once the hardware requirement has been identified, its specification can be detailed. This will describe what functions have to be performed, what interfaces

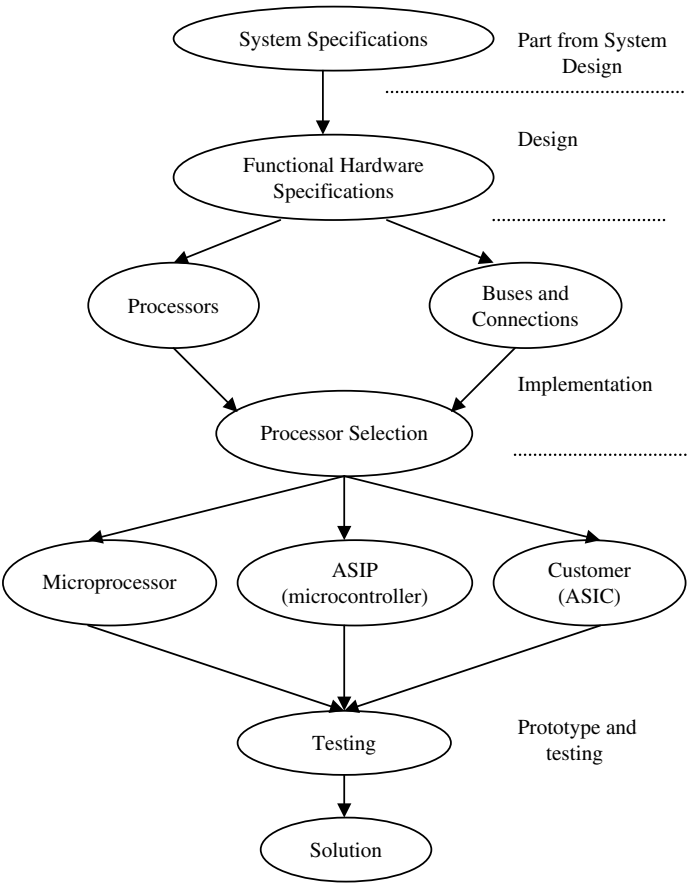


Figure 2.2 Hardware systems implementation.

will be used, how fast they are and so on. This enables the test procedure that will be used on the completed prototype to be identified, and the test parameters determined. Testing should always be performed to the initial specification, and not just to verify that the hardware is working.

2.2.4.2 Hardware implementation

a) Processor technology selection

Having decided what functions the hardware should perform, the next stage is to decide what components should be used. Two important decisions are to be taken at this stage: the type of processor to be used to implement

each functional block and which IC technology we are going to use for implementation. For example, the microprocessor can be selected as the execution unit based upon an optimization of various parameters. These might include speed of execution, data bus size, and so on, which can be considered to be the technical considerations. These are relatively easy to decide unless the product being designed is a very special one requiring, for example, a very high speed of execution and/or a very high accuracy. In such cases we have to use a custom (single-purpose) processor.

However, there are non-technical parameters which influence these decisions and often these parameters are considered first, and can be termed the product production considerations. These include the non-ideal parameters related to cost. If an investment has previously been made in a particular type of processor development aid, then unless there is some extremely important overriding technical reason, such as those given above, then that processor will be used in all new products.

b) IC Technology selection

This stage starts by selecting the IC technologies (VLSI, ASIC, PLD, FPGA, etc.) that will be used in implementation. The microprocessor, memory, and peripheral ICs are chosen and their interconnection to implement the subfunctions is specified. Timing analysis is performed. Detailed analogue and digital circuit design is carried out. Schematic diagrams and timing diagrams documents are implemented. A hardware prototype is then constructed.

The selections of the processor technology and the IC technology during the implementation stage are one of the main topics of "embedded system design". The reader can get more about that from any book on embedded system.

We note here that hardware design and implementation may involve design at any or all of the following levels:

- component
- printed circuit board
- system

Some systems are designed entirely at the ***component level***. At this level of design, you must select and specify the interconnection of discrete ICs. This step is typical for small embedded systems consisting of only one or a few printed circuit boards.

The hardware for other systems may be implemented at the ***board level***. At this level, one can select commercially available printed circuit boards that provide the required subsystems. These boards are combined by plugging

them into a backplane that carries the system bus connections. The backplane and bus signals may correspond to one of numerous bus standards. These bus standards are independent of a specific microprocessor. If a subsystem function is not commercially available, then it is necessary for you to design, from the component level, a board that provides the required function and is compatible with the system bus.

At the ***system level***, hardware is purchased as a complete, or essentially complete, system; if the complete system hardware cannot be purchased, it may still be necessary to design one or more special function circuit boards, compatible with the system bus, to complete the system.

Successful hardware design at the component and board levels requires that you have a thorough understanding of logic and circuit design principles and an understanding of the function and operation of commonly used ICs. The design of a board for use at the system level requires an understanding of bus interface design and the details of the particular system bus used.

In Section 2.4 we shall discuss "IC Technology" in details and introducing the design constraints that guide the designer to select the suitable IC technology for implementing his design.

2.2.4.3 Hardware prototyping and testing

Once the devices have been selected the initial prototype has to be put together and tested. This can involve two levels of prototyping: the functional and the physical, as illustrated in Figure 2.3.

Functional prototyping involves implementing the functions of the product to ensure that it operates correctly. This may not be in the final physical form. For example, a complex electronic and logical function containing many logic gate equivalents may be implemented using discrete logic gates, whereas for the final production version a single custom component would be made. This would incorporate the same functional logic devices but in a more compact and cost-effective package. These devices are generally known as application-specific integrated circuits (ASICs) or single-purpose processors. This type of component is not often used for functional prototyping due to the cost of making only a small number: the functional prototyping may identify some problems requiring a re-design and hence a new ASIC. It is more cost-effective to perform functional prototyping using standard components, which can then be condensed into an ASIC when correct.

The use of hardware simulators when designing ASICs is becoming more important as the simulators become more accurate. This method uses a computer to simulate the physical and logical characteristics of all components

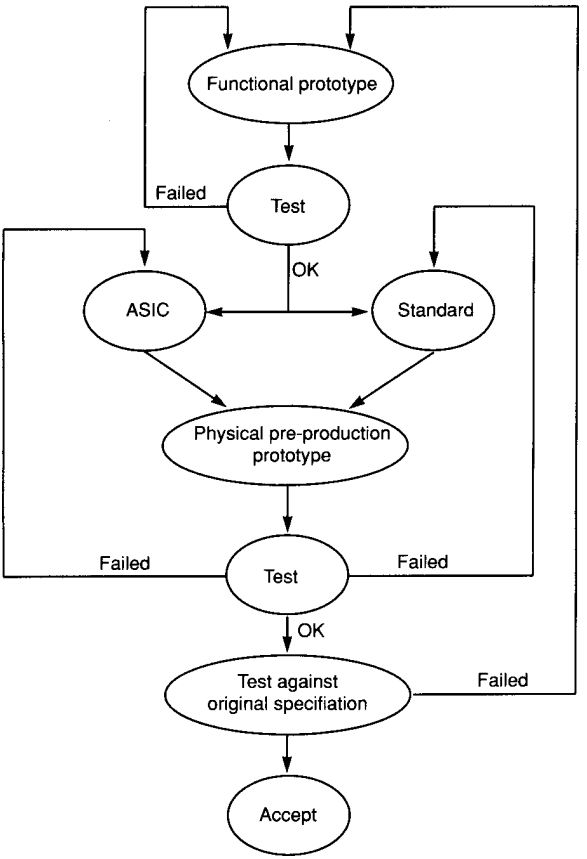


Figure 2.3 Prototyping and testing.

in the product, in order to perform functional testing. Some simulators also allow physical simulation of devices such as ASICs. In this way functional prototyping can be performed without any physical devices being put together. This has the advantage of flexibility as the circuits can be changed easily if a problem is identified, and avoids a physical functional prototype. It is also possible to go directly to an ASIC after successful simulation.

The disadvantages are that the computer systems used for the simulation are expensive and complex, and that the success of the simulation depends upon the accuracy of the modelling of the physical and logical characteristics of the components. However, due to the increasing use of ASICs in products, simulation enables the physical functional prototyping stage to be avoided. This enables the operation of the ASIC design to be tested without constructing

either a functional equivalent or a test ASIC, so that faults and errors can be detected. The ASIC can then be made with a high degree of confidence that it will work first time.

Physical prototyping considers the electrical and mechanical connections. Should the circuit be soldered together using a standard printed circuit board or should wire wrap be used are considered here. Wire wrap is easier to construct, quicker and easier to make changes to, but the sockets are expensive and the method has a maximum useful frequency of 8–10 MHz before signals become corrupted. It is also a method which cannot be used for the final product as once the design is finalized and no more changes are to be made, it takes a long time to produce a completely connected circuit, when compared to using a PCB.

Alternatively a custom printed circuit board (PCB) could be produced if high frequency operation is required, although the cost involved means that it would usually only be used for the production version, and not just for functional testing. This would mean producing any ASICs before final prototype testing which would require parts of the product to be developed in isolation, converted into ASICs, tested, and then placed into the complete product for final prototyping and testing. Finally the product has to be tested against the original specification, as that is the measure of the success of the technical design. Commercial considerations involved in the marketing of products are not considered here. At this stage testing will revert to the black box approach, see Figure 2.4, where the product is considered only in terms of its inputs and outputs. Following this stage of testing, there will be further product testing when the software and hardware are combined into the final complete product.

2.2.5 Software Overview

Software design has much the same approach as hardware design in that the functional software description is derived from the system specification. This is then followed by the implementation stage and finally the prototype and testing stages.

2.2.5.1 Functional software specification

The functional software specification details the functions to be performed by the software and the interfaces between functions. As a general rule it can be said that software is best designed using a block structured approach, independent of the language used.

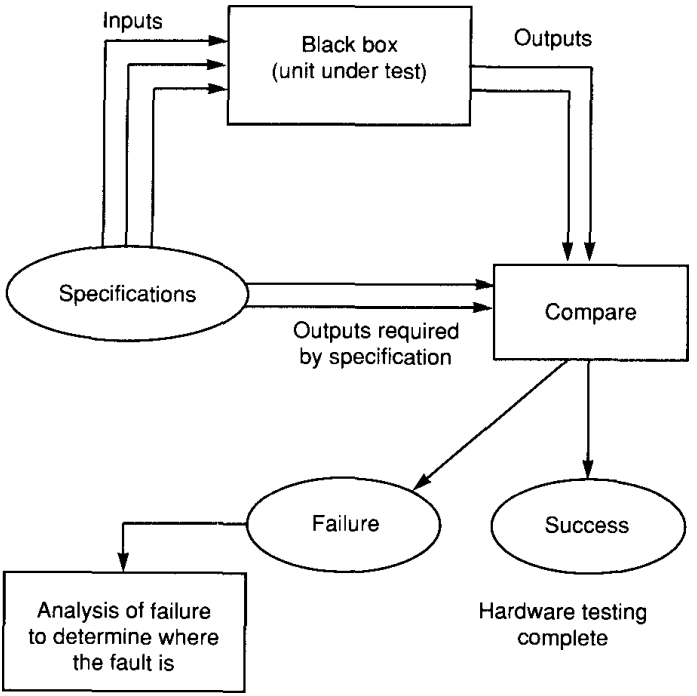


Figure 2.4 Black box testing.

When using a block structured approach, it is only necessary to specify what actions are to be performed by a particular section of software, what goes in, and what comes out, as in Figure 2.5. The internal operation of the software should be invisible to the rest of the programme and should only communicate to the rest of the programme via the defined interface.

Block structuring makes testing easier as each software black box can have its internal operation tested in isolation and then when working, it can be added to the system. When system testing takes place the data and results passing through the software programme then need only be tested for correctness on entering and exiting the individual software black boxes. In this way the number of tests required can be reduced and errors pinpointed quickly. There is also the advantage that if a problem has been identified in a particular software black box, then that piece of software can be replaced with an improved version, without affecting the rest of the programme provided that the interface is maintained the same.

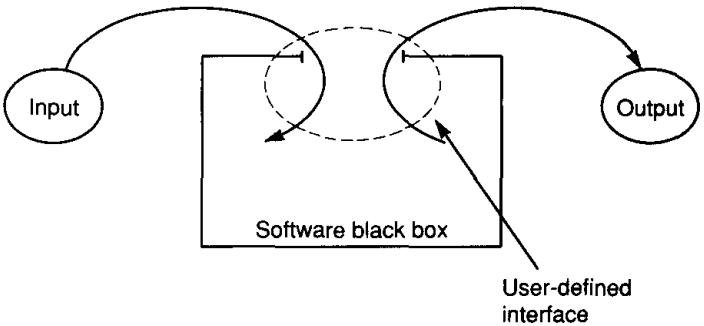


Figure 2.5 Block-structured software.

Software design and implementation can be carried out at any of several language levels:

1. Assembly language (low level language LLL).
2. High-level language (HLL).
3. Mixed assembly and high-level.

2.2.5.2 Implementation

Once the functional design has been completed then the implementation can be performed. This involves the selection of the appropriate assembler and/or compiler(s). A microprocessor will generally have only one assembly language so that most assemblers will be similar. However, as there are a variety of High Level Languages (HLL), a decision has to be made as to which HLL to choose, and then which compiler to select and from which company. Although, nominally all compilers for a particular HLL are the same, there are considerable differences in speed of compilation, the size and speed of execution of the code produced, and other factors, such as the ability to link easily to assembly language programmes. Some of these decisions are also similar to those used for selecting the microprocessor and its development aids, and are based on commercial as well as technical considerations.

Each HLL has its own advantages; some of them are good for programmes involving large amount of calculations, some are good for block structured programmes, and others (e.g. C and C++) is good for maintaining close control of the hardware of the computer, and for producing compact, fast-executing programmes.

A large programme, which has several of these characteristics, could be written in several HLLs as well as a LLL, i.e. a mixed assembly and HLL.

This would enable the most efficient language to be used in different parts of the programme in order to produce the optimum solution. However this would be unusual and normally the most appropriate HLL would be chosen for the complete programme.

2.2.5.3 Prototyping and testing

If two or more types of language are being used each would be produced separately and then linked together. Once this has been achieved system testing can be performed. There are a variety of testing strategies for testing each software black box as it is produced and then gradually connecting the blocks together to build up the complete system. The system testing compares the complete programme operation against that required by the initial functional software specification. As with the hardware this can be achieved using two different techniques, simulation and emulation.

When simulating the operation of a programme the target microcomputer is not used, instead the target system is simulated by producing a model that is equivalent. There are two basic levels of simulation, functional and detailed.

A **functional simulation** will have the same interfaces between blocks as the real hardware but they may be implemented in a different way. For example, if the target hardware has a keyboard connected this may be simulated by a list of characters stored in a file. When the programme requests an input from the keyboard a character from the file is substituted. Even in a functional simulator the microprocessor will be accurately simulated to enable register and memory contents to be examined at any point. The timing of the execution of the programme can usually be obtained from the simulation although it is not possible to be 100 per cent confident of the results.

For **detailed simulation**, the internal operations of the hardware blocks are also accurately modeled. This is unusual as a great deal of extra programming is required to produce an accurate simulation and there is not a significant improvement in the quality of the results obtained.

Software simulation is not quite the same as for the hardware, in that, programmes can usually be made to execute on several different types of computers by using a compiler suitable for each. The operation of the programme can then be tested on whatever computer is available, and any hardware functions are either simulated using software equivalents, or emulated if hardware equivalents are available. This does not completely test the software as emulation or simulation can never be 100 per cent accurate, but it is a useful method if the target system is not available.

When the designer uses software option for implementing a component of the system, it will be easy for him to test the design. The designer can test the

actual programme by executing it to test its functionality. Hardware solution is not as simple as such. In some cases it will be very expensive and time consuming to implement the real component or at least a prototype in order to test the correctness of the design. Simulation is the technique normally used to test the hardware design. During simulation, the designer can refine the design to meet the requirements. When the refined version of the design satisfies the entire requirements, the system, or a prototype, can be implemented and tested.

2.2.6 Target System and Solution

This consists of three phases (Figure 2.6); System integration, system validation and solution.

2.2.6.1 System integration

If the preceding software and hardware phases were performed separately, the results of these efforts must now be merged. At this point the system software and prototype hardware are integrated and complete system testing can be performed involving both the hardware and software to ensure three things:

(a) The hardware meets the functional hardware specification;
(b) The software meets the functional software specification;
(c) The system meets the original system specification.

Once the prototype has been tested, the preproduction prototype is made. In the pre-production prototype the design is re-engineered so that functionally it

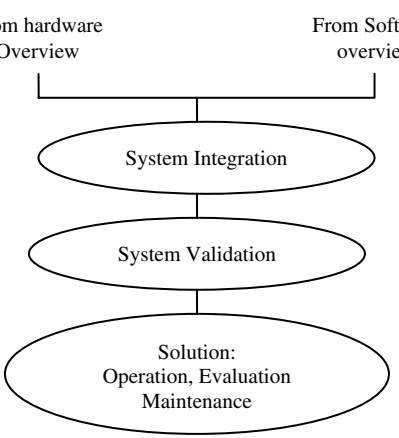


Figure 2.6 Target system flow.

is the same, but that the hardware of the design is adapted for production. This may involve designing a PCB or using programmable logic arrays (PLA) rather than discrete logic and using surface mount components so that automated assembly techniques can be employed and so on.

The pre-production prototype is then tested against the specifications again to ensure that it is still functionally correct. Finally the production version is made which should be identical to the pre-production version, but will be made in the numbers required. It is only when large numbers are being made that difficulties in the production version may occur, due to variations in the production processes themselves. It is the job of the production engineer to ensure that these difficulties are overcome.

2.2.6.2 System validation

Deficiencies or invalid assumptions in the requirements analysis and specification phases may have resulted in an inadequate or inaccurate functional specification. The system is tested to determine whether it actually meets the user's needs, in contrast to meeting simply the functional specification.

It is likely that there will still be faults somewhere in the product as 100 per cent testing is difficult and expensive to achieve. These faults would either be due to an incorrect specification or an incorrect implementation of the specification, which was not detected during any of the tests performed. In practice, these will be gradually discovered by users of the product and if this information can be obtained by the manufacturer an enhanced and refined version can be produced when thought appropriate.

2.2.6.3 Solution: Operation, maintenance and evaluation

Errors detected after the system is put into service are corrected. The system is modified to meet new needs and is enhanced by adding new features. Customer support is provided including continued correction of errors.

A development cycle model like that of Figure 2.1 is often referred to as the life cycle model because it describes the existence of the system from initial concept (birth) to obsolescence (death). A narrow view of system design considers only the phases of functional design and the hardware/software overviews as design. In this view, the phases of requirement and specifications are considered analysis; the phase of getting the target system is considered as testing; and the phase of solution is considered maintenance. This narrow view tends to result in less successful designs. It is advantageous to consider the entire development cycle as design.

2.3 Technologies Involved in the Design Process

The above section shows that at each level of the design flow cycle there are a number of design options that can be selected to solve a particular problem or particular function. For instance it is possible to use software (general purpose processor) to implement a given task or, on the other side, it is possible to use a single purpose processor (pure hardware solution) to implement the same task. The designer here has to select the type of processor he is going to use to execute each sub function. This is called "***processor technology***". After this, the designer will decide which "***IC technology***" he is going to use for the implementation of each processor. For example, whether he is going to use FPGAs, PLA or PAL IC technology for implementation? The designer, before selecting the "processor technology" or the "IC technology", has to select how he will convert the system specifications into an *architecture design.* The ways and the tools to do that are known as "***design technology".*** Figure 2.7 is a part of the traditional development cycle of Figure 2.1 with these technologies shown on it.

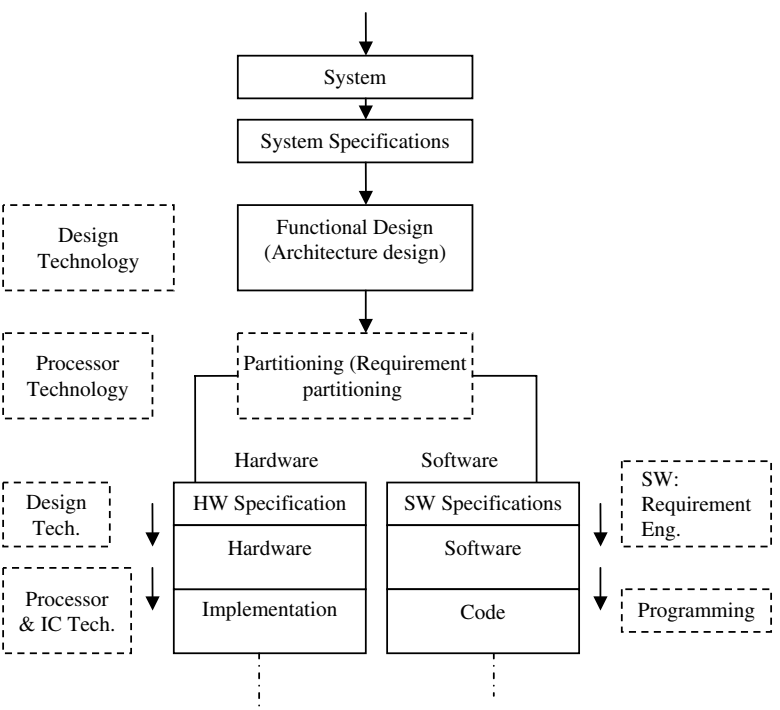


Figure 2.7 Use of different technologies within the development cycle.

In summary, during the different phases of designing and implementing of a given task, the designer needs three types of technologies:

- Design technology (or design flows),
- Processor technology, and
- IC technology.

The selection of the proper technology at each stage determines the efficiency with which the design can be converted from concept to architecture, to logic and memory, to circuit, and ultimately to physical layout. The efficiency of accomplishing any task or subtask is measured, as mentioned before, by the design metrics. The decision of the designer to select one among many available technologies is guided by the design metrics and also which metrics he has to fulfill and which he can optimize.

In the following sections the various technologies the designer uses during the different design phases are considered.

2.4 Design Technology

Design technology deals with the sequence and steps which the designer will follow to convert the desired system functionality (or system requirements) into an implementation. During the design and implementation process, it is not enough to optimize the design metrics, but it is important also to do so quickly. This has direct effect on the productivity and the time-to-market. Any delay in the time-to-market reduces the revenue and it may let the manufacturer incurs losses rather than the expected profit. This explains why improving design technology to enhance productivity and reaching the market in time has been the main concern of the software and hardware engineers for some decades now.

Different tools and design approaches are available for the reader in many embedded systems books. One of the main design tools that help the designer of a complex digital system, which will briefly introduced here, is the concept of "*design partitioning*". Two general design approaches are introduced briefly here; *structured design approach*, and *multiple description domains* (the Y-chart) approach. The first approach uses the concepts of hierarchy, regularity, modularity and locality to handle the complexity of any system. The Y-chart or behavior/structural/physical domains system descriptions describes the system in three domains; behavior, structure and physical.

2.4.1 Design Partitioning

Design partitioning is one of the main design tools to deal with when considering complex systems. The design is partitioned into a number of inherently interdependent tasks or levels of abstractions. The levels of design abstraction start at a very high level and descend eventually to the individual elements that need to be aggregated to yield the top level function. Digital VLSI design, for example, is often partitioned into the following interrelated tasks: architecture design, microarchitecture design, logic design, circuit design, and physical design.

Architecture design: This task describes the functions of the system. For example, the x86 microprocessor architecture specifies the instruction set, register set, and memory model.

Microarchitecture design: This task describes how the architecture is partitioned into registers and functional units. The 80386, 80486, the various versions of Pentium, Celeron, Cyrix Mu, AMD KS, and Athlon are all microarchitectures offering different performance / transistor count tradeoffs for the $\times 86$ architecture.

Logic design: describes how functional units are constructed. For example, various logic designs for a 32-bit adder in the x86 integer unit include ripple carry, carry look-ahead, and carry select.

Circuit design: describes how transistors are used to implement the logic. For example, a carry lookahead adder can use static CMOS circuits, domino circuits, or pass transistors. The circuits can be tailored to emphasize high performance or low power.

Physical design: describes the layout of the chip.

These elements are inherently interdependent. For example, choices of microarchitecture and logic are strongly dependent on the number of transistors that can be placed on the chip, which depends on the physical design and process technology. Similarly, innovative circuit design that reduces a cache access from two cycles to one can influence which microarchitecture is most desirable. The choice of clock frequency depends on a complex interplay of microarchitecture and logic, circuit design, and physical design. Deeper pipelines allow higher frequencies but lead to greater performance penalties when operations early in the pipeline are dependent on those late in the pipeline. Many functions have various logic and circuit designs trading speed for area, power, and design effort. Custom physical design allows more compact, faster circuits and lower manufacturing costs, but involves an enormous labor cost.

2.4.2 Use of Multiple Views (Multiple Description Domains): The Y-Chart

The first common technique that can be used to design any complex system is the use of multiple views to provide differing perspectives. Each view contains an ***abstraction*** of the essential artifact, which is useful in aggregating only the information relevant to a particular facet of the design. In microelectronics, for example, a VLSI circuit can be viewed ***physically*** as a collection of polygons on different layers of a chip, ***structurally*** as a collection of logic gates, or ***behaviorally*** as a set of operational restrictions in a hardware-description language. A good designer is the one who can switch between the different views when building his circuit. This is because each view has its own advantages and merits that can help the design process. The concept of multiple views is not closed on the microelectronic system design, but it is in use in many other disciplines. For example a mechanical engineer will use the multiple views concept for better understanding of the project under design. Another example is the use of this concept in the civil engineering field. We cannot say that the set of floor-plans produced by an architect are enough to build the house. More accurately, these floor-plans are reflecting the view of the architect to the project. It is necessary to know how the civil engineer will see the project (the design details of the structure), to know the view of the plumbers (plumbing plans), electrical engineers (electrical wiring plans), etc. In many cases it is possible to use some views to derive from them other views, but it is more useful to consider the different views separately to have different way of thinking which will be a form of a “double check" for the designer.

To make the concept of views more understandable in case of designing digital system, Gajski developed the Y-chart shown in Figure 2.8. The radial lines on the Y-chart represent three distinct design domains (view is replaced by domain in the Gajski Y-chart): **behavioral, structural,** and **physical**. These domains can be used to describe the design of almost any artifact and thus form a very general taxonomy for describing the design process. Within each domain there are a number of levels of design abstraction that start at a very high level and descend eventually to the individual elements that need to be aggregated to yield the top level function (i.e., in the case of chip design and transistors).

The **behavioral domain** describes what a particular system does or what we wish to accomplish with a system. For instance, at the highest level we might state that we desire to build a chip that can generate audio tones of specific frequencies (i.e., a touch-tone generator for a telephone). This behavior

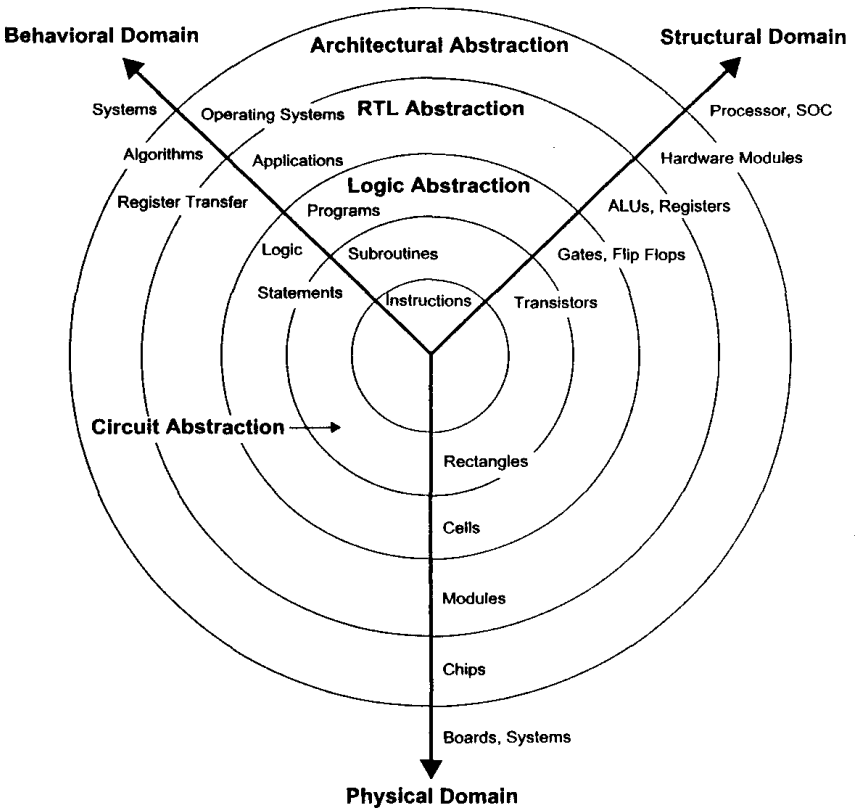


Figure 2.8 Gajski-kuhny chart.

can be successively refined to more precisely describe what needs to be done in order to build the tone generator (i.e., the frequencies desired, output levels, distortion allowed, etc.).

At each abstraction level, a corresponding **structural** description can be described. The structural domain describes the interconnection of components required to achieve the behavior we desire. For instance, at the highest level, the touch-tone generator might consist of a keyboard, a tone generator, an audio amplifier, a battery, and a speaker. Eventually at lower levels of abstraction, the individual gate and then transistor connections required to build the tone generator are described.

For each level of abstraction, the **physical domain** description explains how to physically construct that level of abstraction. In other words, the physical domain specifies how to arrange the components in order to connect them,

which in turn allows the required behavior. At high levels this might consist of an engineering drawing showing how to put together the keyboard, tone generator chip, battery, and speaker in the associated housing. At the top chip level, this might consist of a floor plan, and at lower levels, the actual geometry of individual transistors.

Using the Y-chart, the design process can be viewed as making transformations from one domain to another while maintaining the equivalency of the domains. Behavioral descriptions are transformed to structural descriptions, which in turn are transformed to physical descriptions. These transformations can be manual or automatic. In either case, it is normal design practice to verify the transformation of one domain to the other by some checking process that compares the pre- and post-transformation design. This ensures that the design intent is carried across the domain boundaries. Hierarchically specifying each domain at successively detailed levels of abstraction allows us to design very large systems.

The Y-chart is rearranged in Table 2.1 to show the tools that may be used at each abstraction level in the different domains.

We need to describe the design in one way or another at each and every stage. The representation and notations used for describing the design are extremely important because they can affect how designs can be view and refined. The most appropriate description for a design depends on which level of design the design working in and on the type of system he is designing. As an example, as shown in Table 2.1, top-level specification could be in natural language (as in many formal specifications), in equations form (such as

Table 2.1 Levels of Design/Y-chart.

Design Levels	Behavioural Domain/Design Descriptions	Structural Domain/ Primitive Components	Theoretical Techniques
Algorithmic or Architecture abstraction	Specifications High-level language. Mathematical equations	Functional blocks 'black boxes'	Signal processing theory Control theory Sorting algorithm
Functional or RTL abstraction	VHDL, Verilog, AHDL FSM language C/C++/Pascal	Registers Counters ALU	Automata theory Timing analysis
Logic abstraction	Boolean equations Truth tables Timing diagrams	Logic gates Flip-flops	Boolean algebra K-map Boolean minimization
Circuit abstraction	Circuit equations Transistor netlist	Transistors Passive comp.	Linear/non-linear eq. Fourier analysis

Table 2.2 Comparison between using hardware language and block diagrams.

	Schematic capture	Hardware Description Languages
Advantages	— Good for multiple data flow — Give overview picture — Relates to hardware better — Does not need to be good in computing — High information density — Back annotations possible — Mixed analogue/digital possible	— Flexible and can be described through parameters — Excellent for optimisation & synthesis — Direct mapping to algorithms — The best technique for designing datapaths — Readily interfaced to optimiser — Easy to handle and transmit (electronically)
Shortages	Not suitable for designing datapaths Does not interface well in optimiser Not good for synthesis software Difficult to reuse Parameters cannot used to describe it	Essentially serial representation May not show overall picture Often need good programming skill Divorce from physical hardware Need special software

difference equation for a digital filter), in data-flow diagrams (as in software), in behavioural languages such as AHDL or VHDL. On the other side if designer is designing a finite-state machine, then the easiest way to describe it is the use of a special-state machine language such as AHDL when using Altera's Maxplus-II. Some form of graphical block diagram is usually very useful and helpful to describe some designs. Roughly speaking, design descriptions can be divided into graphical description using schematic capture or language description using some form of hardware description language. Table 2.2 gives a comparison between the use of hardware language for description and the use graphical block diagrams.

2.4.3 Use of Structured Design: Functional Block-Structured Top-Down Design (Structural Hierarchy)

A critical tool for managing complex designs is hierarchy. Functional block-structured approach views an object as parts composed of subparts in a recursive manner. A large system is an object that can be partitioned into many units or parts. Each unit or part in turn is composed of multiple functional blocks. These blocks in turn are built from cells, which ultimately are constructed from an available prebuilt component for the particular function. In other words, the functional block-structured approach separates common

actions and functions into self- contained blocks with defined interfaces to the other blocks. Each block is then subjected to a top-down design which considers the actions required of each block. These actions are then broken down into smaller sub-actions which are effectively simpler blocks with simpler interfaces. These simpler blocks are then broken down into even simpler blocks. This process is repeated until the blocks being produced contain only single actions which can be implemented directly. As the blocks are completely defined and the interfaces between all the blocks are completely defined, by implementing the lowest level of blocks the product is also implemented completely. This approach is taken from a well known software strategy; to handle a large software programme split it into smaller and smaller sections till we reach simple functions and simple subroutines that can be easily interfaced.

The idea behind this approach is the fact that any system can be more easily understood at the top level by viewing units as black boxes with well-defined interfaces and functions rather than looking at each individual basic component, e.g. transistor. Consider for example a radio system: a radio may have hundreds of parts in it, but they are more easily viewed when divided into groups such as the tuner, amplifier, power supply, and speaker. Each group can then be viewed in subgroups; for example, by dividing the amplifier into its first stage and its second stage. The bottom of the structural hierarchy is reached when all the parts are basic physical components such as transistors, resistors, and capacitors. This hierarchical composition enables a designer to visualize an entire related aspect of some object without the confusing detail of subparts and without the unrelated generality of superparts. For example, the hierarchical view of an automobile fuel-supply system is best visualized without the distracting detail of the inner workings of the fuel pump and also without the unrelated generality of the car's acceleration requirements and maximum speed.

Use of hierarchical structure approach is the best way that can be used for designing any moderately complex system. **Top-down** and **bottom-up** are the most common hierarchical organizations. In top-down a circuit is designed with successively more detail; in bottom-up a circuit is viewed with successively less detail.

2.4.3.1 Top-down design

The term top-down design comes from programming and relates to the concept of structured programming. Niklaus Wirth, the developer of the Pascal programming language, offered the following definition of structured programming.

"Structured programming is the formulation of programmes as hierarchical, nested structures of statements and objects of computation."

This definition implies a decomposition or breaking down of a large problem into component parts. These parts are then decomposed into smaller problems with this successive refinement continuing until each remaining task can be implemented by simple programme statements or a series of statements. As each statement or structure is executed, a part of the overall objective is accomplished. The programme is decomposed into units called modules, which are decomposed into control structures or a series of statements. The statement is the principal unit of the programme.

A very significant point in applying top-down design is that the modules should be selected to result in minimum interaction between these units. In general, complexity can be reduced with the weakest possible coupling between modules. Minimizing the connections between the modules also minimizes the paths along which changes and errors can propagate into other parts of the system. Although complete independence is impossible, as all modules must be harnessed to perform the overall programme objective, interaction is minimal in a well-designed programme.

Top-Down Design of Digital Systems

When applied to digital systems, top-down design proposes a broad, abstract unit to satisfy the given system functions and is characterized by a set of specifications. The overall system is then decomposed into modules, which are partitioned in such a way to be as independent of one another as possible, but work together to satisfy the function of the system. Just as in programming, in which higher levels of structure contain the major decision-making or control elements of the programme, one of the partitioned modules will be the control unit of the system. Successive refinement of the modules leads to operational units, which will often be MSI circuits or groups of circuits. This refinement continues, with an accompanying increase in detail, until all sections can be implemented by known devices or circuits.

There are certain characteristics of the digital field that make top-down design an easy method to apply. For example, the many SSI and MSI circuits available allow operational units and even modules to be realized directly with chips. These chips are generally designed to require a minimal number of control inputs, and thus at least minimal interaction between modules is automatically achieved.

Another advantage of the top-down method in digital design is that the control module can be realized as a state machine, which allows well-known design procedures to be used in designing a reliable control unit. Although directed toward state machine controllers, this method can be easily extended to microprocessor controller design. The differences between using conventional state machine controller and microprocessor state machine controller are given latter.

Example 2.1 Top-Down Design an example

To demonstrate the idea of the top-down approach, we shall use the design of a digital voltmeter (DVM). The design of this instrument is well known, and it illustrates some of the principles of design. The problem statement begins with a set of specifications that the DVM should meet. The DVM is then broken down into a set of modules as shown in Figure 2.9. These modules are chosen according to their function. The DVM must measure and display the results. A control module is required to direct the operation of the measurement and display modules.

Figure 2.9 reflects a high level of abstraction (the architecture abstraction level of the Y-chart). At this step of the design, almost no detail appears. Modules indicate only broad functions to be performed, with no indication of how those functions should be carried out. Before proceeding to the next step, the designer must choose an approach. In this example, the designer may choose a dual-slope technique for the measurement module and three 7-segment LED displays for the display module. A control module will always be present at this level.

All modules except the control module are then divided into operational units, as shown in Figure 2.10. It is at this point that the inexperienced designer or student becomes the most frustrated. There is no plug-in procedure that can be used here, as this step requires creativity and experience along with trial-and-error methods to be completed. As the modules are broken down into

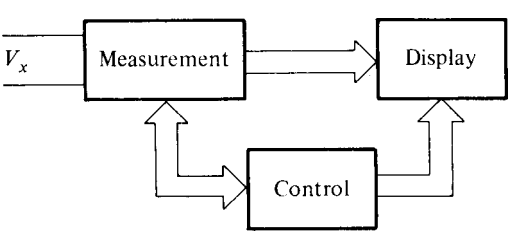


Figure 2.9 Function modules.

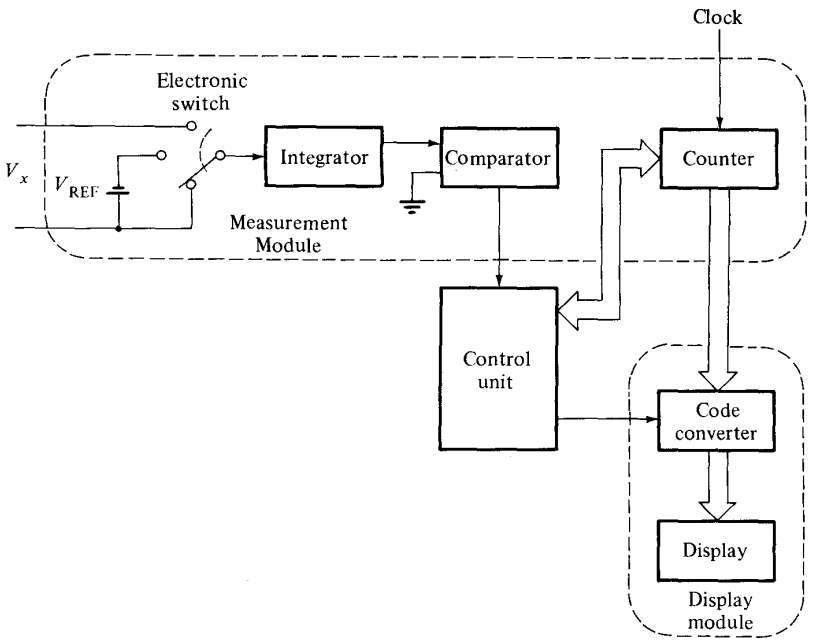


Figure 2.10 Operational units.

operational units, we attempt to realize them in existing MSI chips or circuits that can easily be fabricated. Knowledge of available MSI chips is helpful at this step of the procedure.

Figure 2.10 represents a more detailed picture of the system. Modules have now been broken down into units that perform specific operations; hence the name operational units. Each of these units can be implemented by a circuit or a series of circuits using well-known design procedures or presently available IC chips. The comparator, counter, code converter/driver, and 7-segment displays are off-the-shelf items. The electronic switch and integrator may also be purchased or designed to meet the given specifications.

Figure 2.11 shows some of the basic circuits used to make up the counting, code- converting, and display units. The system is now completely specified, with the exception of the control unit.

To design the controller using top-down, the designer of such simple system can start the design process by constructing a timing chart, showing the relationship among all necessary control signals, inputs, and outputs. The controller is then designed to produce the control signals necessary to drive the basic circuits. This completes the design process.

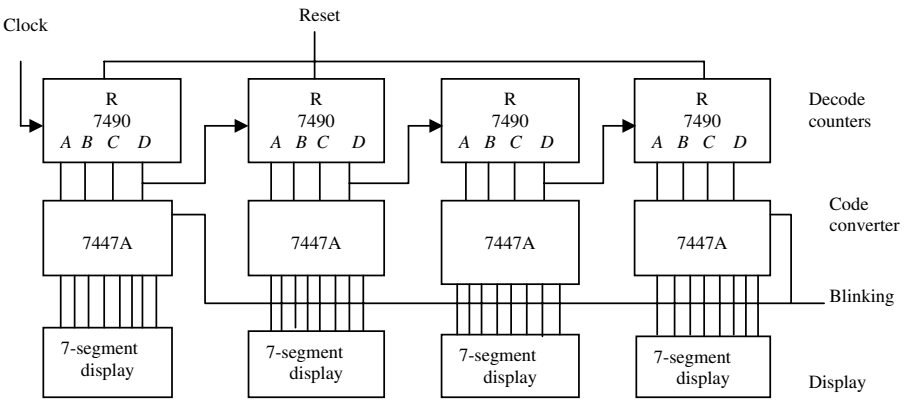


Figure 2.11 Basic circuits.

Starting the controller design by constructing the timing chart represents one of the possible techniques that can be used for this purpose. This approach looks to the controller as a state machine. It is suitable when the designer decides to use discrete components to build the processing and execution unit of the system under design. Later in this section we are going to show that in many digital system applications, the use of microprocessor or microcontroller as the processing and execution unit represents an advantage. The microprocessor/microcontroller can be used also to implement the rule of the controller. Such approach will be discussed in Section 2.7 and will be used in Chapter 3 to complete the design of our digital voltmeter.

Example 2.2 Top-Down Design example

In this example we are using the concept of top-down to design a circuit that receives at its inputs two numbers A and B and gives at its output the greater value (see Figure 2.12). The above statement represents the most abstraction level; describes solely the relation between the inputs and output. To get the structured hierarchical organization that represents the complete circuit, a top-down design should be done.

We start from the top, we compose the circuit into sub-circuits each represent a functional block. This stage split the required circuit into two functional blocks: subtraction, and select the bigger number. In the third level, the subtraction is decomposed into negation followed by addition. The select circuit can be decomposed (one possible solution) into test the sign of the result of subtraction and select 1 out of circuit. In the fourth level we decompose the two's complement negation into inversion followed by an increment. We can

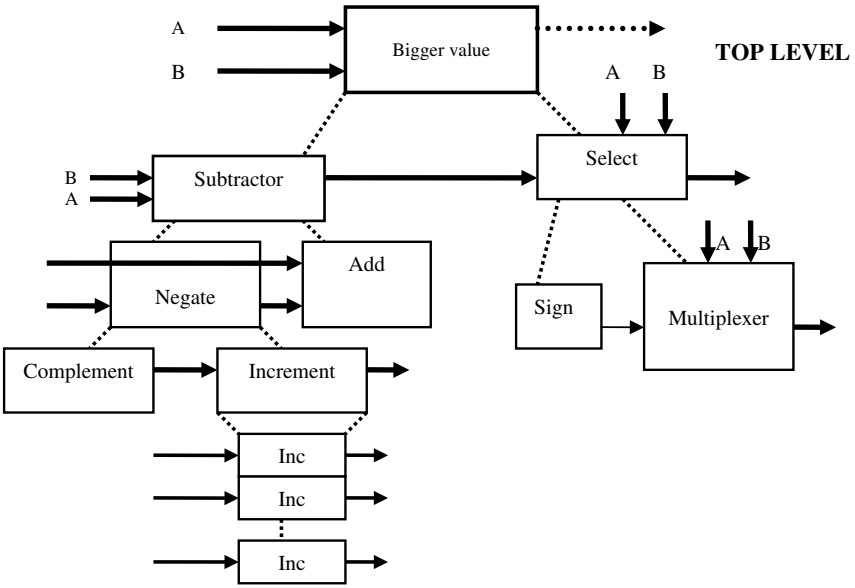


Figure 2.12 Top-down hierarchical organization of a bigger-value engine.

use a 2×1 multiplexer for the select circuit. The decomposition continues till we end with components that can be find on-shelf. The design is complete when the layout for implementing all the components is given.

Example 2.3 Computer System Hierarchy

This example considers the hierarchy of a typical computer system. The hierarchy is shown in Figure 2.13. At the top level, the computer system contains a hardware part and a software part. The hardware part consists of five blocks; Input block, storage block, arithmetic and logical block, output block and control block. The software side includes the firmware and operating system, etc. The ALU consists of a working register file, a shifter, logical unit (LU) and arithmetic unit (AU). The AU consists of adder, multiplication unit, floating-point unit, etc. Similarly, the storage unit can be divided into internal memory and external memory. The internal memory may be divided into cache memory, inner semiconductor memory, ROM, etc. The same can be followed for the other blocks.

The diagram given in Figure 2.13 illustrates how a relatively complex component can be rapidly decomposed into simple components within a few levels of hierarchy. Each level only has a few modules, which aids in the understanding of that level of the hierarchy.

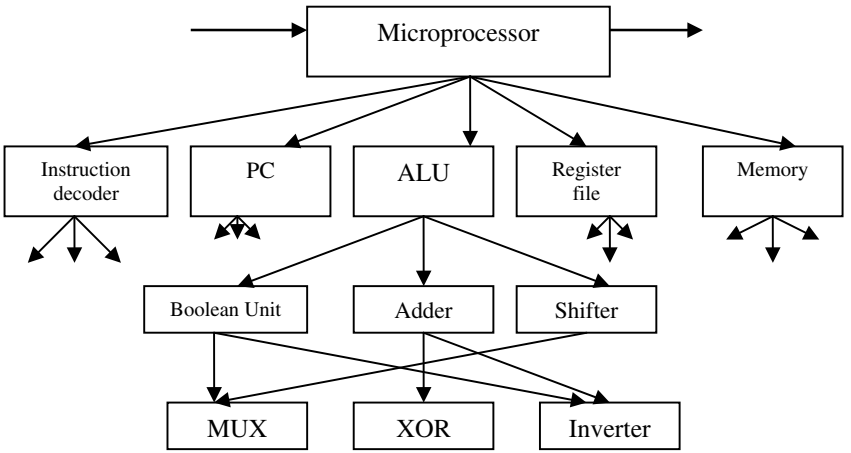


Figure 2.13 Possible top-down hierarchy of a microprocessor.

2.4.3.2 Bottom-Up Hierarchy

Bottom-up hierarchy is the opposite of up-down hierarchy explained in the previous examples. This technique of design is better used when the designer really understood and knows the details of the system under consideration. More importantly he knows that such system has been properly implemented before. For example any designer knows that the basic storage element is the flip-flop that stores a single bit and he knows that such basic cell is already done. Suppose now that someone asked that engineer to design a 4 K memory unit. Since of the size of the single bit cell is the most important factor in the chip, all other circuitry must be designed to accommodate such a cell. Therefore, the design composition must proceed in a bottom-up manner. The design starts by the single bit of memory (the basic cell). From the cell the designer forms words each consists of eight bits that form a row; combines vertically four of the words to form a block of 4×8 bits; combines eight of such block at the next level; and so on (see Figure 2.14). The highest level of the hierarchy, level six in our example, shows four arrays, each containing 32×32 bits. For proper operation, memory-driving circuitry with the correct spacing is then placed around the bits.

2.4.3.3 Hierarchy and concepts of regularity, modularity and locality

In example 2.3 we tried to show how to use the concept of hierarchy to solve a complex problem as computer system. If we continued dividing all the blocks

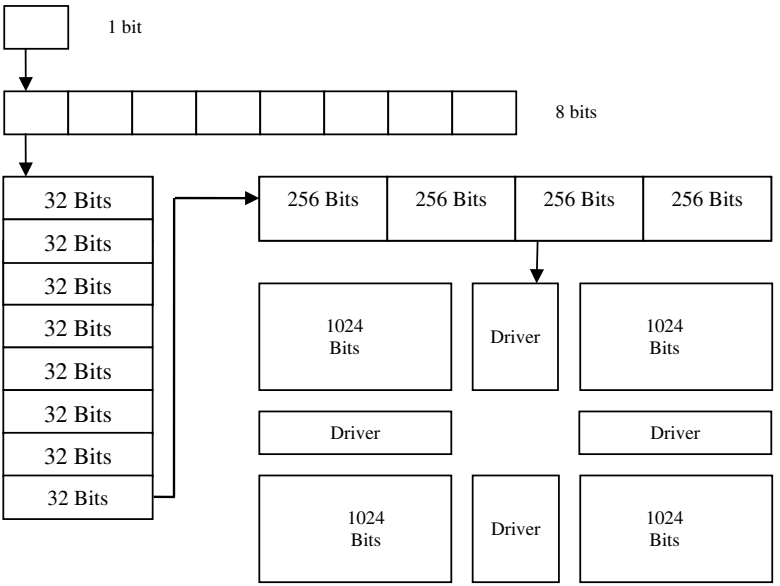


Figure 2.14 Bottom-up hierarchical composition of a 4 k memory array.

into submodules till we reach a stage that we cannot divide more, it is very possible to end with a huge number of different submodules. The design and the implementation of such large number of submodules represent a new complex problem. This indicates that using the concept of hierarchy alone does not solve such a complex problem. Other design concepts and design approaches are also needed to further simplify the design process. Normally the concepts of regularity, modularity and locality are used together with hierarchy. The reader can get more information on this subject from any embedded system book.

2.4.4 Design Procedure Based on Top-Down Approach

A procedure for digital system design that implements the top-down concept discussed in the preceding section consists of the following steps:

- Determine the goals of the system from a thorough study of the specifications of the system.
- Decompose the system into the functional modules required to implement these goals. One module will be the controlling state machine.
- Determine the next lower level of tasks that must be accomplished by each module.

- Decompose all modules except the controlling state machine into the operational units designed to accomplish these tasks. Often these subsections will correspond to MSI circuits.
- Decompose the operational units until the entire system is reduced to components that can be realized with basic circuits.
- Determine all the control signals necessary to operate the circuits.
- Draw up a timing chart to show the time relationship of all signals referenced to the clock signal.
- Design the controlling state machine to implement the system.
- Document your work.

Step 1 describes the system in the behavior domain of the Y-chart, while step 2 represents the description of the system in the structural domain. The remaining steps represent different abstractions levels of the three domains of the Y-chart.

Two main issues related to this procedure are:

- How to convert the behaviour specifications into structure specification. In other words, how to go from step 1 to step 2.
- How to design and the implementation of the controller.

In small systems, trial-and-error methods may be used in going from step 1 to step 2. Trial-and-error methods may play a part in several steps of this procedure especially in case of small digital systems. Furthermore, two or more different designs may be equally good; there is generally no single "best" design. Experience improves the efficiency of applying this or any other design method. In case of designing a complex digital system, we cannot use trial-and-error, but we must use a more systematic techniques that can convert the behavior specifications (step 1) into structure specifications (step 2). One of the popular techniques is the use of "Finite-State Machine with Datapath (FSM-D)" to implement the conversion. This technique is based on the fact that a basic processor consists of a controller and a datapath.

Concerning the design and the implementation of the controller, two techniques are possible; the use of conventional state machine and the use of microprocessor state machine. If the conventional combinational and sequential logic design techniques are used to design the controller and the functional units, we end with what is called "single purpose processor". On the other hand, if microprocessor is used to implement the controller and the majority of the functional units, we ended with what is called "general-purpose processor" and the complete system is called in this case "microprocessor- based system".

2.4.5 Programmable Digital Systems Design Using Block Structured Design

In Section 2.6 we will discuss the topic "processor technology"; the type of the processor that the designer is going to use to implement the data execution and manipulation unit in his design. A wide class of designers uses standard off-the-shelf components. Three of the main standard off-the-shelf components are: microprocessor, microcontroller and digital signal processor (DSP). The use of these components results in a programmable digital system; the system functionality can be changed by reprogramming without the need, in most cases, for major hardware changes. The use of microprocessor/microcontroller as the core building block of the digital system results in what is called "microprocessor-based system" and "microcontroller-based system". Using such technique helps the designer to achieve: short time-to-market, less NRE, and more flexibility for the system.

Because of the wide spread of such digital systems, we are considering in this section how to use the structured top-down approach for designing such systems; designing a programmable digital system.

Designing microprocessor/microcontroller-based system starts, as for any other system, by creating the specifications of the system under design. This can be obtained by considering what actions are required of the system and then detailing all of them and their side-effects completely. The specifications are then implemented. If there are any errors in the specifications they will be implemented in the end product (the system), so it is very important to ensure that the specification is error-free. One method of producing the specifications is to consider what actions and functions the system should perform. Some will be directly defined by the nature of the system; others will have to be deduced.

Microprocessor/microcontroller-based systems as any programmable system have several basic and common functions that normally required performing:

- Executing user programmes;
- Manipulating and saving data;
- Displaying results;
- Responding to user intervention.

Because of these common features of the two systems, it is expected that the operational block diagram, and in many cases, the functional block diagram, on the higher level of abstraction will be the same for them.

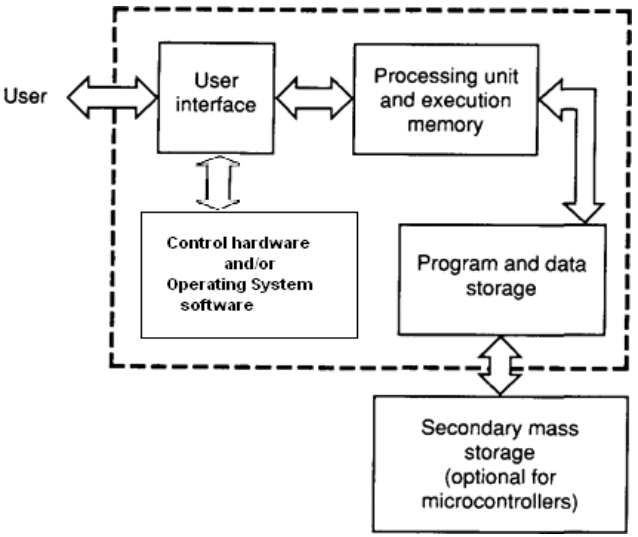


Figure 2.15 Operational blocks.

To start the block-Structured Design:

1. The designer starts normally by the functional block diagram. The designer, where possible, should gather together the operations and functions that are similar and a fixed boundary drawn around them.
2. The functions can be normally separated into three blocks as illustrated in Figure 2.15, which are:
    - User interface;
    - Processing unit and execution memory;
    - Programme and data storage.
    - There three operational units are gathered together to operate as a coherent whole by the controller which can be either hardware or as an operating system software. The use of operating system with programmable systems has its advantages and limitations especially if we are looking for real-time implementation (this will be shown during the following discussions).

In terms of a top-down design, the diagram in Figure 2.16 would be more appropriate as each function has clearly defined boundaries and identifiable interfaces.

The diagram of Figure 2.16 illustrates some important features of a block structured top-down design especially the rule and the importance of

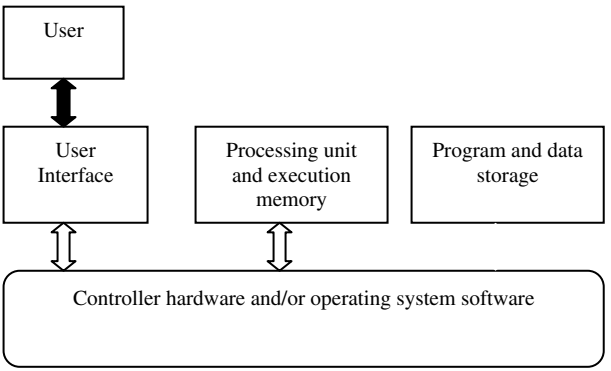


Figure 2.16 Top-down design.

the interface:

- The functions inside the boundary are only allowed to communicate with functions outside the boundary through defined interfaces.

An interface can be considered to be the connection between dissimilar systems and in this context it will be a combination of hardware and software.

The use of interfaces enables the functions inside the boundary to be invisible to those outside so that the block functions can be specified in terms of the interfaces. A block may then be replaced at any time with an alternative block, provided the interface is maintained.

If the system under design deals with a lot of input data coming from different input units, then there is a possibility of having some errors in the data enters. To avoid the effect of such situation, it is recommended to use the interfaces that contain error detection procedures to ensure that errors are not transmitted throughout the entire system and detected errors are handled in a specified and controlled manner. Testing can be reduced to testing at the interfaces to ensure that if correct data enters then correct solutions are produced and that incorrect data is detected and prevented from propagating throughout the entire system. If a system is found to be producing incorrect results it is only necessary to trace the erroneous data at the interfaces until the block which has an error-free input but a faulty output is detected. The error has then been isolated within that particular block.

If the functions inside the boundary do not generate errors and the designer discovers the presence of errors, then errors must enter via the interfaces. Therefore, and as a general rule, to minimize the possibility of erroneous input, the number of interfaces per block in Figure 2.16 should be minimized. In order

to maintain block-to-block communication, the operating system block has to have several interfaces, one to each other block and this indicates that the operating system introduces errors into the system.

The use of the operating system as the medium for inter-block communication has advantages, in that a common communication channel can be established and all block interfaces can be designed to match that standard. This allows blocks to be added or subtracted without having to re-design the system or the operating system. Due to the standardization of the interfaces the operating system is able to perform more error detection operations on the data and messages being communicated than would be economical for individually designed interfaces, as illustrated in Figure 2.17.

Top-Down Design

In terms of the top-down design concept described earlier, the functions of the system in Figure 2.16 can be represented as shown in Figure 2.18,

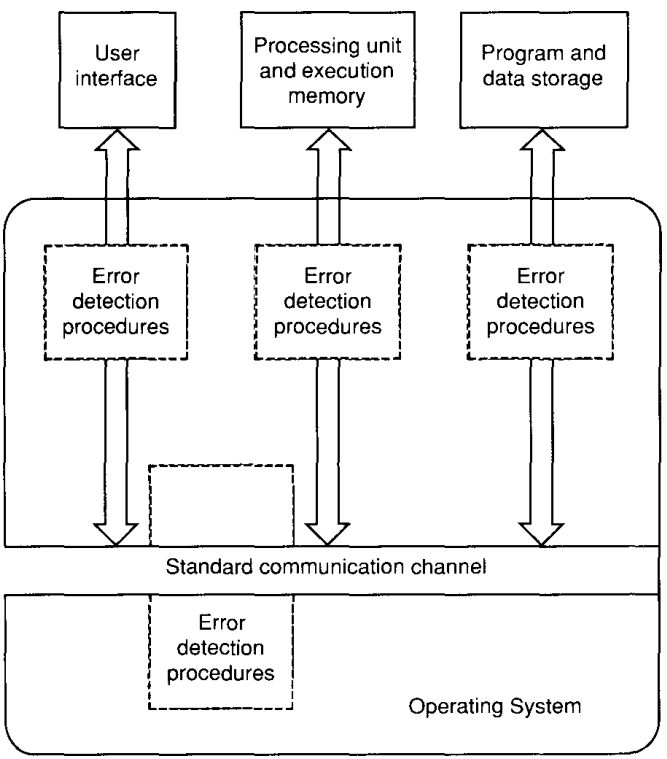


Figure 2.17 Refine block system: contains more details.

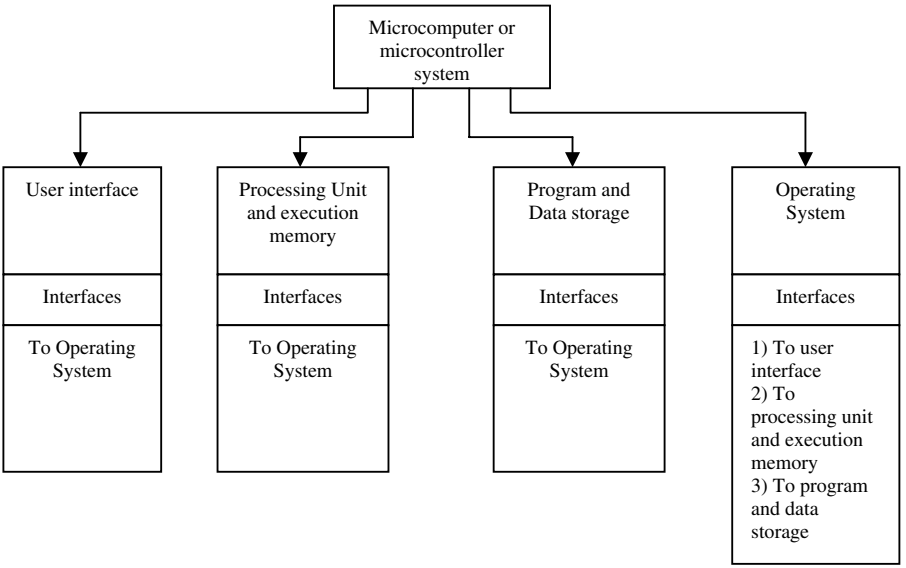


Figure 2.18 Top-down hardware design.

which identifies each operational block and the interfaces each requires. In this example, each of the blocks has a single interface to the operating system, except for the operating system which has three interfaces corresponding to the three blocks. Each block is complete in itself with the defined interfaces indicating the how, when and why of data and information communication with the other blocks. The interfaces are the only means of transferring data and information and can be hardware, software, or a combination of the two. The number of the levels in the top-down hierarchy defines the number of interfaces required. This, in turn, defines the type of interface used; software, hardware, or a combination.

Top-down approach is then used with each of the four blocks. In the following we are considering, as an example, the user interface block (the rest will be considered in the Case Study). The user interface performs two basic operations, input of data and commands from the user which form one sub-function, and output of data and results as a second sub-function. The user interface, accordingly, can be divided into a number of sub-functions depends on the number and types of the input and output devices connected to the system. In Figure 2.19 we divided the user interface into four sub-functions which make up the operations of the user interface. The main sub-function is the link to the operating system and three interfaces to specific I/O devices.

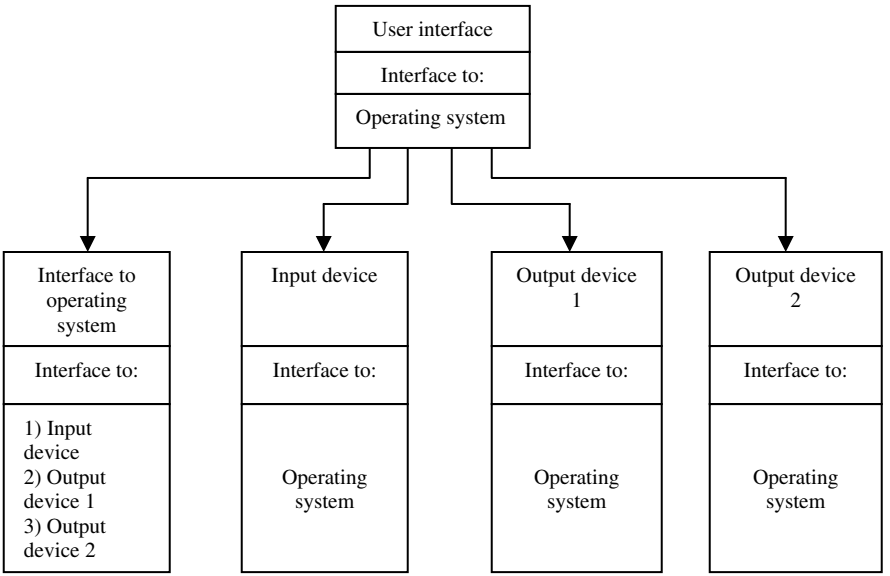


Figure 2.19 Next level of user interface.

In this way, the blocks can communicate in a structured way with the higher levels. It is clear that in Figure 2.19 we assumed one input unit and two output units. As there are many forms of output, such as visual display units (VDUs), graphics displays, serial communication links and parallel printer ports and printers, each of which needs to be handled in a different way, a block is required for each one. Similarly for the input devices, if there are several input devices, each one will require a separate sub-functional block. Input devices can be sensors (in case of data acquisition systems), keyboard, etc. To determine exactly the number and types of input and output devices and what actions each block has to perform, a detailed specification has to be available.

Effect of the number of top-down levels on the performance

In top-down approach, if a sub-block has to communicate with the higher level, this has to be performed via the interface link defined at the same level. This achieves the error checking and correct formatting of data required, but unfortunately it results in slowing down the communication process as each additional interface takes a finite time to execute. This is one factor which has to be optimized for a particular design. The use of well-structured communication interfaces produces error free, robust and reliable links, but with the drawback that each level introduces additional delays.

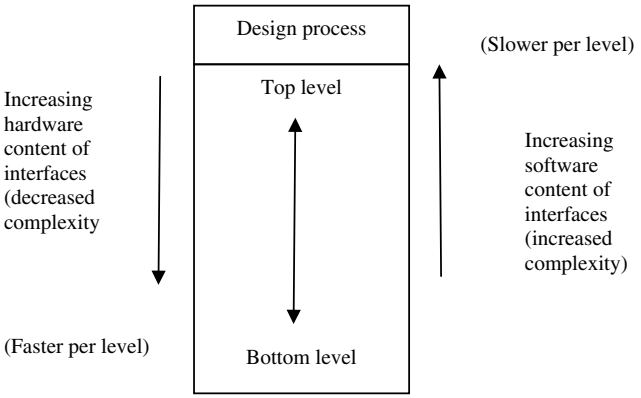


Figure 2.20 Hardware/software optimization.

As a general rule the more levels the top-down design introduces the slower the communication process is. This fact is very important when the designer is designing for high performance. It is very possible in the cases when the performance is the main design constraint, e.g. in real time systems, the designer avoids using programmable systems as a solution and goes to another approaches that implements the performance constraint. In many cases, it is possible to improve the performance by using hardware, which is much faster, to implement the interfaces at the lower levels. This is illustrated in Figure 2.20 which also indicates that the higher levels are implemented mainly in software which is correspondingly slower. However, the data at the higher levels is more significant and there is less of it and this tends to reduce the impact of slow high level communication interfaces.

The reason the low level functions are implemented in hardware is that the interfaces become simpler and more easily implemented in hardware. As more levels are created in the design process, the additional delay in communication links increases significantly, as seen in Figure 2.21. This only applies to products containing both hardware and software, such as microcontrollers and microcomputers.

To determine the optimum combination of hardware/software interfaces the response time of the system is considered. This is known as the real-time response and a system can be said to have a real-time response when there is no user-perceivable delay from the initiation of an action to the response of the system with the desired data and/or results. This makes the assessment of real-time response subject to the user and different users may well view the same system in different ways. A user is not necessarily a

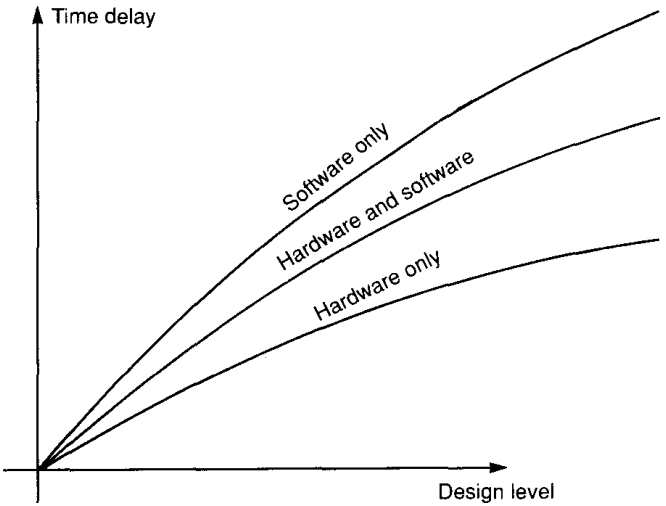


Figure 2.21 Interface time delay against design level.

person but can be other computers and hardware systems. In general, good requirement engineering solves this situations; good requirement engineering means unambiguous requirement. Accordingly, the initial requirements from the system will define exactly what the client means by real time and gives quantified values for the performance.

Figure 2.22 illustrates how the acceptable response time (the performance) represents the limit that to be placed onto the interface execution time and accordingly on the design levels.

The design level required is mainly determined by the complexity of the product as any product has to be reduced to similar simple blocks. The more complex a product the more levels are required, assuming compatible complexity of each level. Some variation in the number of levels is possible by increasing the number of blocks per level to achieve the individual block simplicity. This reduces the number of levels needed but it also reduces the advantage of using top-down design and an alternative design process may be required.

Once the design level and the interface execution times have been determined, a range of solutions from mostly hardware to mostly software can then be implemented.

If the response time cannot be achieved, it may be necessary to implement a special interface linking a low level functional block with a block much higher up in the design, as illustrated in Figure 2.23. However, this way is not

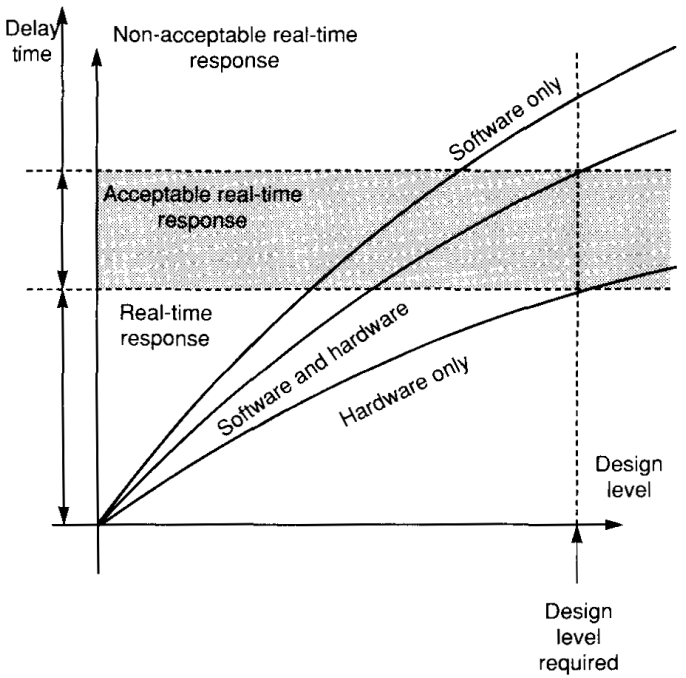


Figure 2.22 Limiting interface execution time.

recommended in many cases since it is against the basic principle of top-down block-structured design.

The nearer the top of the top-down design the more complex the interface links become and the more likely they are to be implemented using software. Software has the ability to implement complex functions relatively easily and can implement any changes by updating the software. This is a requirement as the more complex a link the more likely it will be that changes, upgrades and error removal will be necessary. If implemented in hardware, complex interface links are difficult to change. Low level links are suitable for implementing in hardware as they are simpler and stable, requiring few upgrades and can be used in a wide variety of applications.

2.5 IC-Technology; Implementation Technology

The day when the microelectronic industry introduced the first IC, the electronic systems designers started to change their way of designing and implementing the systems. Before IC technology, the designers were using

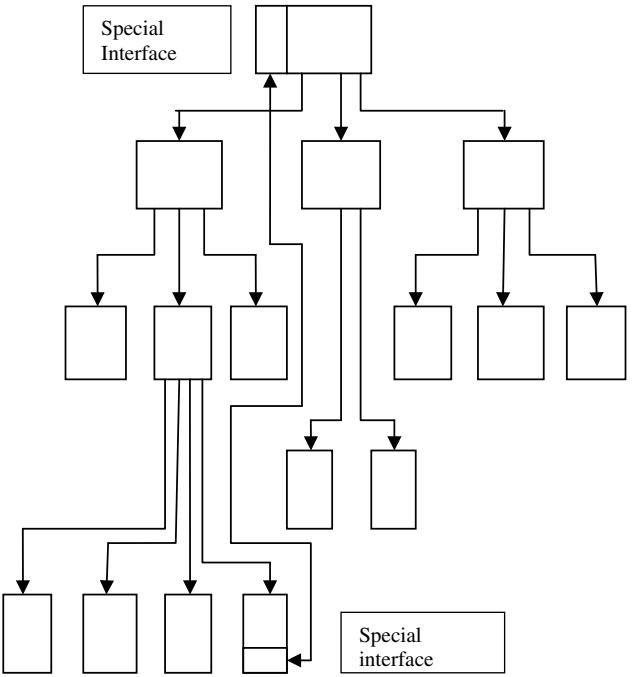


Figure 2.23 Special interfaces.

discrete elements (transistors, resistors, capacitors, inductors, etc.); they were designing to implement some metrics as performance, accuracy, etc. without giving large attention to the values and the tolerance of the discrete component available in the market. We can refer this to two reasons; the component-count was (the area required for implementation) not that vital factor in the design; and the fact that by connecting some components in parallel or in series he would get the values he wants. With ICs in the market and with the huge demand on the electronic equipment as a result of the appearance of new wide fields of applications all of them need such equipment, some new constraints started to put pressure on the system designers. For example the size of the end product started to be important, accordingly the component-count constraint started to be important. With mobility, the size of the end product gained more importance and the power consumption constraint started to appear, and so on. To face such growing needs of the market, the designers started to change their way of designing; they started to study at first the market; which ICs are available in the market and how to use them in their designs and implementation.

Directly after the first IC appeared, the IC technology started to develop very fast. The microelectronic industries started to increase the capabilities of the ICs by increasing the degree of integration; more and more transistors on one chip. From few hundreds of transistors on Small Scale Integrated circuit (SSI) chip, when IC technology started, to millions of transistors on the chip (VLSI) now. The microelectronic industries used in that Moor's law. We must note here that, for long time the microelectronic industry were introducing to the market ICs with specific functions that cannot be changed after manufacturing. Here another problem between the microelectronic industry and the needs of the system designers started to appear; the designers need some freedom. This problem started to appear heavily when the time-to-market constraint started to put heavy pressure on the designers and also when the market window of the product started to shrink. The designers need now ICs that they can define their final function. This started the era of programmable IC technology. The microelectronic industries started to produce ICs with such facility. For example, in the field of memory, they started by ROM, then PROM, then EPROM followed by EEPROM, flash EEPROM, and so on. In the field of logic circuits we have programmable logic arrays (PLA), programmable logic devices (PLD), etc. Now we have more advanced forms of programming, e.g. FPGA. The programmability facility is the result of the technology of producing the ICs. Any IC consists of large number of layers. The bottom group of layers forms the transistors, the middle group of layers forms the logic components and the last group, the top one, is used to connect these components to form the final product. The type of the IC technology is defined by how many layers the designer of the processor is going to optimize and complete during the fabrication of the IC and how many left to the user to optimize and connect during the phase of implementing his application. Three types of IC technologies can be recognized: Full-custom (or VLSI), semicustom application-specific IC (ASIC), and programmable logic devices (PLD). A fourth type known as "platform-based" or a "system on a Chip" (SOC) started to have popularity in the implementation of the systems.

Full-Custom/VLSI: The designer of the full-custom IC optimizes and connects all the layers to be suitable for the implementation of a particular embedded system. Full-custom IC design, often referred to as very large scale integration (VLSI) design, has a very high nonrecurring engineering (NRE) cost, but can yield excellent performance with small size and power. The microprocessors and microcontrollers are, in general, VLSI circuits.

Semicustom/Application Specific Integrated Circuits (ASIC): In ASIC technology, the transistors layers are fully built, the middle layer partially

built leaving the rest of it together with the upper layers (the connection layers) to finish them according to the application. ASIC comes with arrays of gates already formed during the fabrication phase. In a standard-cell ASIC technology, logic-level cells (forming the middle layers) are ready on-chip, and the task of the user is to arrange and connect the cells in a way suitable for the implementation of his application.

Programmable Logic Devices (PLD): In a programmable logic device (PLD) technology, all layers already exist. The layers implement a programmable circuit. Programming, in this case, means creating or destroying connections between the wires that connect the gates. This takes place by either blowing a fuse or setting a bit in a programmable switch. PLD has two main types: Programmable logic arrays (PLA) (consists of programmable array of AND gates and programmable array of OR gates) and programmable array logic (PAL) (consists of just one programmable array). During the past decade a complex form of PLD known as field programmable gate array (FPGA) started to be very popular between embedded system designers. FPGAs offer more general connectivity among blocks of logic, rather than just arrays of logic as with PALs and PLAs.

In the next subsection a brief idea on PLD technology is given. For ASIC design we refer the reader to any VLSI textbook for the details of VLSI technology.

2.5.1 Programmable Logic Device (PLD)

Often, the cost, speed, or power dissipation of a microprocessor may not meet the system goals and the designer has to find an alternative solution. In such cases, normally, going to full-custom IC technology to implement the required task is not the possible solution because of the increased design time and the very high NRE cost. In many of such cases, the solution is to use programmable IC technology. A variety of programmable ICs, known as *programmable logic devices* (PLD), are available now days that can give a solution that can be more efficient than general purpose microprocessor and at the same time faster to develop than dedicated IC and does need much of investment. In contrast to the dedicated IC that comes with fixed-function logic determined at the time of manufacture and could not be changed, the PLD is supplied to the user with no logic function programmed at all. It is up to the designer to make the PLD perform in whatever way a design requires; only those functions required by the design need be programmed. PLD offers the digital circuit designer

the possibility of changing design function even after it has been built. PLD can be programmed, erased, and reprogrammed many times, allowing easier prototyping and design modification.

In a programmable logic device (PLD) technology, the word programming has a lower-level meaning than a software programme. The programming that takes place may consist of creating or destroying connections between wires that connect gates, either by blowing a fuse, or setting a bit in a programmable switch. PLDs can be programmed from a personal computer (PC) or work station running special software.

PLD is a generic term. There is a wide variety of PLD types, including PAL (programmable array logic), GAL (generic array logic), EPLD (erasable PLD or electrically PLD), CPLD (complex PLD), FPGA (field-programmable gate array) as well as several others.

CPLD means, normally, a device based on a programmable sum-of-product (SOP) array, with several programmable sections that are connected internally. In effect, a CPLD is several interconnected PLDs on a single chip. This structure is not apparent to the user.

2.5.1.1 Programmable logic array (PLA) and programmable array logic (PAL)

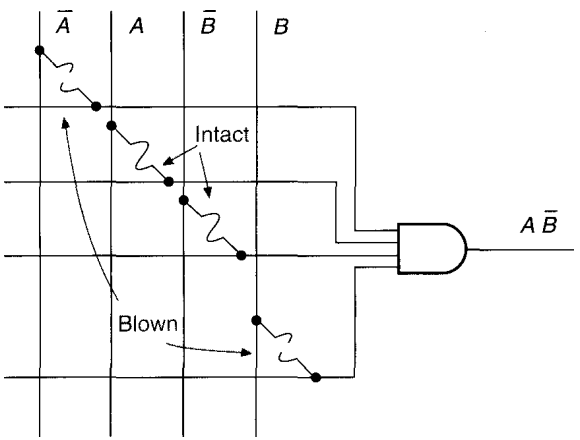
Programmable logic array (PLA) and programmable array logic (PAL) are two forms of the simple EPLD. Programmable logic array (PLA) consists of two planes of logic arrays: a programmable array of AND gates and a programmable array of OR gates. The AND plane and the OR plane give the possibility to compute any function expressed as a sum of products (SOP). The capability of programming the arrays can be achieved by fully populating the AND and OR plane with a NOR structure at each PLA location. Each node is programmed with a floating-gate transistor, a fusible link, or a RAM-controlled transistor. The first two versions were the way these types of devices were programmed when device densities were low. The second form of the simple EPLD is the programmable array logic (PAL), which uses just one programmable array (fixed OR matrix and a programmable AND matrix) to reduce the number of expensive programmable components. Programmable logic devices based on PLAs allowed a useful product to be fielded and well-established techniques allowed logic optimization to target PLA structures, so the associated CAD tools were relatively simple. These devices are generally used for low-complexity problems (a few hundred gates), which require fairly high speed (100's MHz).

Programmable Sum-of-Product arrays, an example

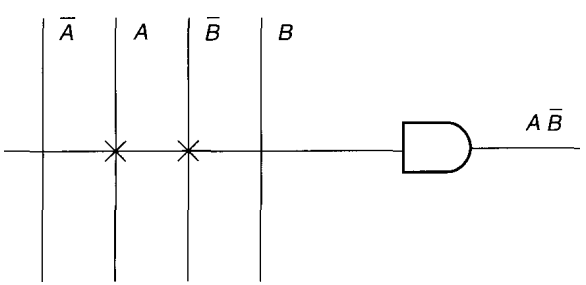
The original programmable logic devices (PLDs) consisted of a number of AND and OR gates organized in sum-of-products (SOP) arrays in which connections were made or broken by a matrix of fuse links. An intact fuse allowed a connection to be made; a blown fuse would break a connection.

Figure 2.24a shows a simple fuse matrix connected to a 4-input AND gate. True and complement forms of two variables, A and B, can be connected to the AND gate in any combination by blowing selected fuses. In Figure 2.24a, fuses for A and B are blown. The output of the AND gate represents the product term $A\overline{B}$, the logical product of the intact fuse lines.

Figure 2.24b shows a more compact notation for the AND-gate fuse matrix. Rather than showing each AND input individually, a single line, called the product line, goes into the AND gate, crossing the true and complement input



a. Cross-point fuse matrix (A and /B intact)



b. PLD 'notation for fuse matrix

Figure 2.24 Cross-point fuse matrix.

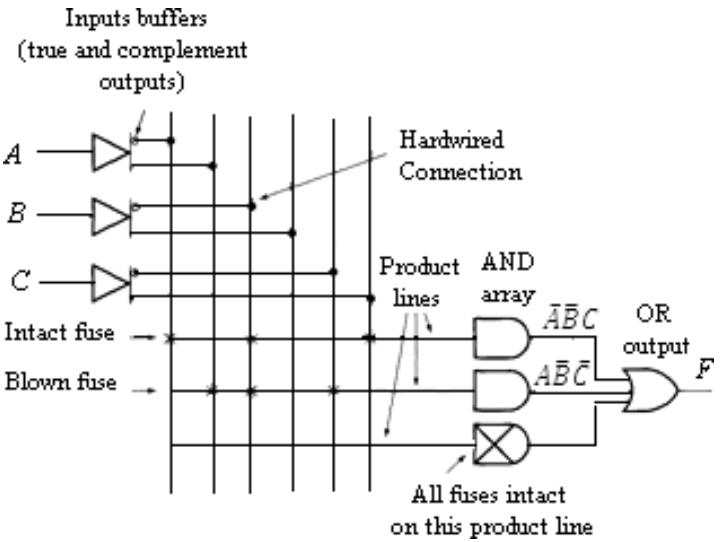


Figure 2.25 PLD symbolic representation.

lines. An intact connection to an input line is shown by an "X" on the junction between the input line and the product line.

A symbol convention has been developed for programmable logic. The circuit shown in Figure 2.25 is a sum-of-products network that implements the Boolean expression:

$$F = \overline{A}\,\overline{B}C + A\overline{B}\,\overline{C}$$

The product terms are accumulated by the AND gates as in Figure 2.24b. A buffer having true and complement outputs applies each input variable to the AND matrix, thus producing two input lines. Each product line can be joined to any input line by leaving the corresponding fuse intact at the junction between the input and product lines.

If a product line, such as for the third AND gate, has all its fuses intact, we do not show the fuses on that product line. Instead, this condition is indicated by an "X" through the gate. The output of the third AND gate is a logic 0 since $\overline{A}A\overline{B}B\overline{C}C = 0$. This is necessary to enable the OR gate output:

$$\overline{A}\,\overline{B}C + A\overline{B}\,\overline{C} + 0 = \overline{A}\,\overline{B}C + A\overline{B}\,\overline{C}$$

Unconnected inputs are HIGH (e.g., $\overline{A}.1.\overline{B}.1.1.C = \overline{A}\,\overline{B}C$ for the first product line). If the unused AND output was HIGH, the function F would be:

$$F = \overline{A}\,\overline{B}C + A\overline{B}\,\overline{C} + 1 = 1$$

The configuration in Figure 2.25, with a programmable AND matrix and a hard-wired OR connection, is called PAL (programmable array logic) architecture.

Since any combinational logic function can be written in SOP form, any Boolean function can be programmed into these PLDs by blowing selected fuses. The programming is done by special equipment and its associated software. The hardware and software selects each fuse individually and applies a momentary high-current.

Erasable (also Electrically) PLD (EPLD)

The main problem with fuse-programmable PLDs, which is used in the previous example, is that they can be programmed one time only; if there is a mistake in the design and/or programming or if the design is updated, we must program a new PLD. More recent technology has produced several types of erasable PLDs (EPLD), based not on fuses but on floating-gate metal-oxide-semiconductor transistors.

The electrically (also erasable) programmable logic devices (EPLD) have broadened the range of the applications for which the PLD technology can be used. The earliest form of EPLD was the PROM (Programmable Read-Only Memory) but a much higher degree of logical functionality can now be obtained by the adoption of more advanced architectures. These are similar in many ways to the Gate Array but with the interconnections made by electrically fuse able links. Such devices are now capable of replacing sizeable amounts of standard logic in a form that can be programmed via a personal computer in a matter of seconds. The programming, as mentioned before, consists of creating or destroying connections between wires that connect gates, either by blowing a fuse, or setting a bit in a programmable switch. This means that the design can be evaluated and changed many times in a very short space of time.

The available EPLD options are:

- Chips with programmable logic arrays.
- Chips with programmable interconnect.
- Chips with reprogrammable

The first two are simple forms of EPLD while the third is a complex form. The system designer should be familiar with these options for the following reasons:

- First, it allows the designer to competently assess a particular system requirement for an IC and recommend a solution, given the system

complexity, the speed of operation, cost goals, time-to-market goals, and any other top-level concern.
- Second, it familiarizes the IC designer with methods of making any chip reprogrammable at the hardware level and hence both more useful and of wider spread.

2.6 Processor Technology

Processor technology relates to the architecture of the computation engine used to implement a system's desired functionality. Although the term *processor* is usually associated with programmable software processors, we can think of many other, nonprogrammable, digital systems as being processors also. Each such processor differs in its specialization towards a particular function (e.g., image compression), thus manifesting design metrics different than other processors. Figure 2.26 shows how to classify the processors based on the field

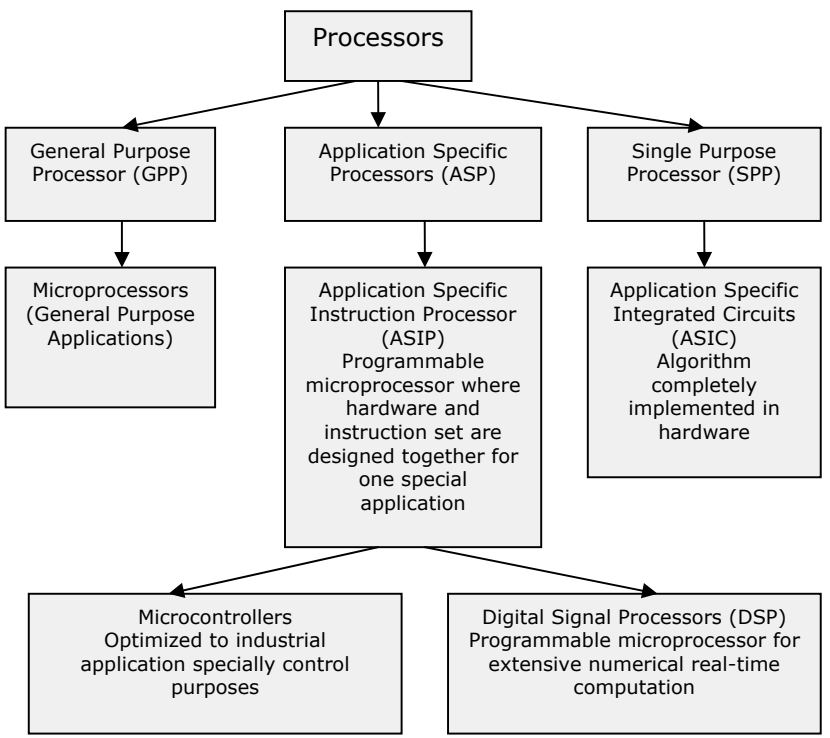


Figure 2.26 Classification of digital processors.

of application. We can recognize the following three types:

1. General purpose processor (GPP).

The general purpose microprocessors are the best example of this type. The use of GPP is not necessarily an advantage all the time. In some applications it is too general to be used; its use will not be effective. One example is using GPPs in video processing; video processing requires huge number of buffers and includes a lot of processing large arrays. GPP is not effective in such a field of applications.

2. Single purpose processor or **Application-specific integrated circuits** (ASIC).

It is a digital circuit designed to achieve one task only. It is used to execute one programme and its hardware includes only the components needed to execute this single programme. Because it is not programmable, ASIC does not include any programmable memory but includes only a data memory. This type of processors is fast, low power, and has small size.

3. Special purpose processor (SPP) or **Application-specific instruction-set processor** (ASIP):

This is a programmable processor optimized for a particular class of applications having common characteristics. We can recognize two types:

- **Microcontroller Unit (MCU):** This is a processor optimized for embedded control applications. Such type of applications needs:
  - Receive analogue inputs from sensors and output signals to set actuators.
  - Deals with small amount of data.
  - Most of the time it handles bits.
  - The microcontrollers are optimized to be suitable for such control oriented application by:
    Having on chip the majority of the components of a microcomputer. Some of these components are: Timers, analogue to digital and digital to analogue converters, serial and parallel communication, etc.
  - Build-in data and programme memories.
  - Many bit-wise instructions.

Programmable pins (can be input, output or bidirectional).

Microcontroller capabilities are growing very fast, this is why we, sometimes put them in the classification under GPP. In such cases we can call the microprocessor as the proper GPP.

- **Digital signal processor (DSP):** This type of processors are optimized for signal processing applications that deals with:
- Large amounts of digitized data, often streaming.
- Real time processing: Many of the applications, e.g. cell phones, need real time processing.
- To meet such characteristics, DSP processors have the following features:
- Multiple execution units: This allows the system to execute more than one instruction at the same time.
- Multiple-accumulator: This allows the execution of the same instruction on more than one piece of data at the same time.
- Efficient vector operations: This allows, for example, the addition of two arrays
- Vector ALUs, loop buffers, etc.

ASIP is a compromise between general-purpose and single-purpose processors. This type gives some flexibility, good performance, smaller size than GPP and less power.

It is important here to mention that GPP, ASIP or ASIC can be used for the implementation of any device. This concept is shown graphically in Figure 2.27. The shaded triangle in Figure 2.27a symbolizes the functionality needed to achieve; to add 10 product terms represent the 10-tap FIR filter.

$$y(n) = \sum_{k=0}^{9} b_k x(n-k)$$

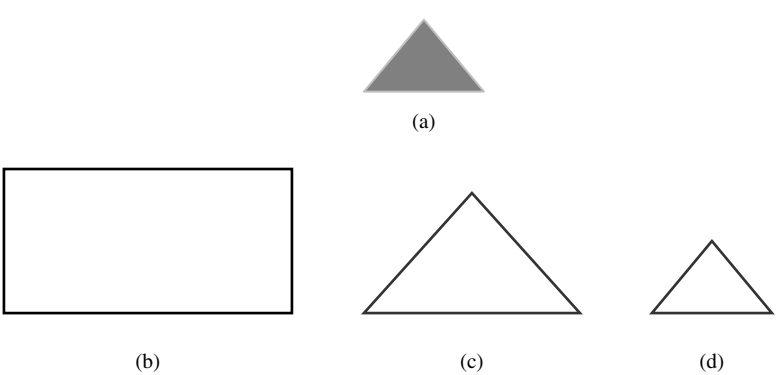


Figure 2.27 Possible implementation of a given functionality: (a) Required functionality, (b) use of GPP, (c) use of ASIP processor, (d) use of ASIC.

The three types of processors that can implement this functionality are shown by Figure 2.27b, Figure 2.27c, and Figure 2.27d consequently.

2.6.1 Use of General-Purpose Processor (GPP)

This is the case of using **microprocessor** to build a programmable system (we call it ***microprocessor-based system***, or ***microprocessor embedded system***). This approach can be used with many applications. To relate this definition with what we mentioned earlier, let use consider Example 2.1 again. The microprocessor can be used as a controller without using state machine design techniques, as there are certain concepts that can be transferred from the well-developed state machine theory. Using the microprocessor to achieve the function of the controller has many advantages over the use of the traditional state machine implementation.

The first advantage (strength) of using the microprocessor is the fact that it takes over the rule of the controller as well as the function of datapath (the execution part of the system). The ALU of the microprocessor is designed to be able to execute many logical and arithmetic operations. If the system under design needs addition or subtraction, for example, the internal ALU can do that easily.

The same ALU can be used if the circuit under design needs, for example, multiplication. As controller, an application calling for variable state times can be performed by the microprocessor's timing circuits and software. To achieve that using conventional digital systems, extra circuitry must be provided for these purposes. Thus, the microprocessor can consolidate the control function and other component functions of a digital system into a single unit. This is actually the main idea behind the microprocessor-based system; the microprocessor replaces the controller and the majority of the functional units. Figure 2.28 is the microprocessor-based system implementation of the block diagram of Figure 2.12. Figure 2.28 shows large saving in the chip-count compared with that of Figure 2.12.

2.6.1.1 Design metrics and GPP

Using a general-purpose processor in an embedded system may result in several design metric benefits and also may cause some drawbacks. In the following we discuss the effect on some metrics:

- ***Time-to-market***: This is short in case of using GPP. The designers does need to design any hardware, it is required only to write the programmes that achieves the required functionality from the system under design.

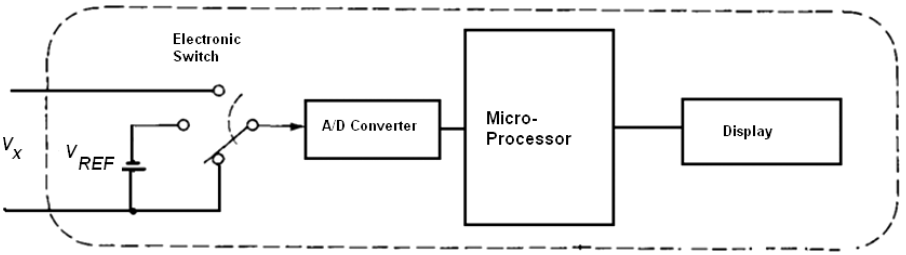


Figure 2.28 Implementing the circuit of Figure 2.9 using microprocessor.

- ***NRE costs***: NRE costs are low. It is very possible that the NRE needed to develop the GPP is very high, but the manufacturer is producing a large quantity. The manufacturer is distributing the NRE cost equally between all the units he produced. It is expected that the share of NRE per processor will be very small. From the other hand, the cost of the designer of the system to write the required programme is considered as NRE related to the system. But this cost is very small compared to the cost if he is going to design hardware.
- ***Flexibility***: Flexibility is high. It is very easy to change the functionality of the system by just writing a new programme without the need of any extra hardware.
- ***Unit cost***: If the customer who ordered the system under design will use one set of it only, then using PGG will reduce the cost: buying ready made processor is much cheaper than designing it. On the other hand, if the customer who ordered the system is planning to produce many of them, it is very possible that cost will be high. For large quantities it may be cheaper to design your processor than to use GPP.
- ***Size***: May be large. Using PGG means, normally, that we are using extra components not need by the application.
- ***Power consumption***: May be large. Again, the use of unnecessary components increases the power consumption.
- ***Performance***: GPP may improve the performance if we are using it in computational-intensive applications. In other applications, the use of PGG may degrade the performance. This is because using PGG means that the designer selecting the software option for solving his problem.

Performance may be slow for certain applications. Performance can put restrictions on the use of GPP as possible implementation. For example, implementing the 10-tap FIR filter using a standard general-purpose processor

would result in a processing time of approximately $1 < t_p < 5$ micro second, which translates to a maximum sampling frequency i.e. $f_s = 1/t_s \leq 1/t_p$ of around 200 KHz–1 MHz. This in turn implies a 100–500 KHz bandwidth of the system (using the Nyquist criteria to its limit). The 10-tap FIR filter used in our example is a very simple application. In many DSP applications, we need FIR with up to 1024-tap. When using such FIR algorithm, this kind of hardware approach is only useful for systems having quite low sampling frequencies. Typical applications could be low-frequency signal processing and systems used for temperature and/or humidity control, in other words, slow control applications.

Another problem that may arise when using such a system (GPP) would be the operating system. General purpose operating systems, for instance Windows™ (Microsoft) are not, due to their many unpredictable interrupt sources, well suited this type of applications, signal processing tasks. A specialized real-time operating system should preferably be used. In some other applications, no explicit operating system at all may be a good solution.

2.6.2 Single-Purpose Processor

A *single-purpose processor* is a digital circuit designed to execute exactly one programme. The processing algorithm is simply "hardwired" into the silicon. Hence, resulting circuit cannot perform any other function. This solution is pure hardware solution, in contrast to the case of using microprocessor that considered as software solution. In case of implementing the system of Figure 2.2, as example, the combinational and sequential logic design techniques are to be used to build the controller and the functional units.

Concerning design metrics: The use of single-purpose processor in implementing a digital system results in several design-metric benefits and drawbacks. It is expected that the benefits and drawbacks will be the reverse of using GPP:

- ***Performance***: The fastest. This is because this is a pure hardware implementation.
- ***Size***: Small. Normally the implementation needs only one chip.
- ***Power***: Small. Just the needed elements are included in the design, no unnecessarily component are used.
- ***NER***: High. This is especial design; the customer has to pay for it.
- ***Unit cost***: May be low for large quantities.
- ***Time-to-market***: Needs time. The designer has to design digital circuits that may be complex which needs time and effort.

For example, Figure 2.27(d) illustrates the use of a single-purpose processor in our digital system example. The processor is representing an exact fit of the desired functionality, nothing more, nothing less. The datapath contains only the essential components for this programme: register file, one multiplier and an accumulator. Concerning the controller, since the processor is executing only one programme, the designer is normally hardwired the instructions that form that programme directly to the logic circuits that form the controller. The designer uses also a state register to step through those instructions, so no programme memory is necessary.

There are mainly only two reasons for choosing the ASIC implementation method. Either we need maximum processing speed, or we need the final product to be manufactured in very large numbers. In the latter case, the development cost (the NRE) per manufactured unit will be lower than if standard chips would have been used.

In case of our example, an ASIC specially designed to run the 10-tap FIR filter is likely to reach a processing speed (today's technology) in the vicinity of $t_p = 2\,\text{ns}$, yielding a sampling rate of $f_s = 500\,\text{MHz}$ and a bandwidth of 250 MHz. This is approaching the speeds required by radar and advanced video processing systems.

2.6.3 Application Specific Processor (e.g. Use of Microcontroller and DSP)

An *application-specific instruction-set processor* (ASIP) can serve as a compromise between the other processor options. An ASIP is a programmable processor optimized for a particular class of applications having common characteristics, such as embedded control, digital-signal processing, or telecommunications. The designer of such a processor starts by optimizing the instruction set to include instructions commonly used in the field of application; he then used the optimized instruction set to optimize the datapath of the processor. The designer may add special functional unites and peripherals needed in the application and at the same time he can eliminate other infrequently used units. In Chapter 3 we are showing that the microcontroller, which is an ASIP, is a processor optimized to work in control-oriented applications.

The use of ASIP in embedded system implementation has benefits and drawbacks:

- ***Flexibility***: ASIP is a programmable device. This results in high flexibility for the system. This is correct for all the programmes that are related to the particular field for which the processor is optimized.

- ***Performance***: ASIP can be considered, to large extent, as a hardware implementation, accordingly it improves the performance of the system.
- ***Size***: Suitable for applications that need small size implementation. ASIP has on-chip, besides the processor, many components. For example, the microcontroller is a microcomputer on a single chip.
- ***NRE***: Higher than the case of using GPP, but less than when using ASIC. In many applications to use ASIP, the designer has to develop the processor itself and to build the related compiler. To reduce the NRE cost, many researches are currently in progress to develop techniques and tools that any designer to use to automatically generates the needed application specific processor and the associated compiler.

Microcontroller and DSP, which are in use since many years, belong to this category; ASIP category.

During design, the choice between the three technologies is a matter of choosing between software, hardware, or combination between them for implementing the system. The choice between them, especially between hardware and software, for implementing a particular function is simply a trade-off among various design metrics, like performance, power, size, NRE cost, and flexibility. Concerning functionality, any choice is capable to implement any functionality.

2.6.4 Summary of IC Technology and Processor Technology

It is very important to remind the reader that the IC technology and the processor technology are completely independent; each of the three processor technologies can be implemented in any of the three IC technologies. PLD, semicustom, or full custom IC can be used to implement many types of processors. An example of the concept is given in Figure 2.29.

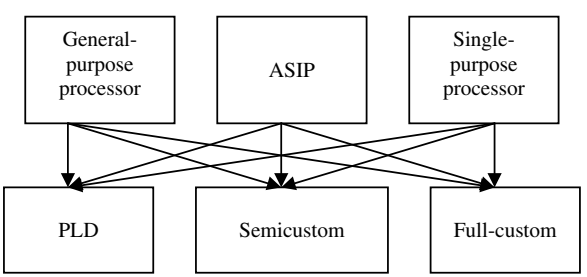


Figure 2.29 Use of available IC technologies to implement some processors.

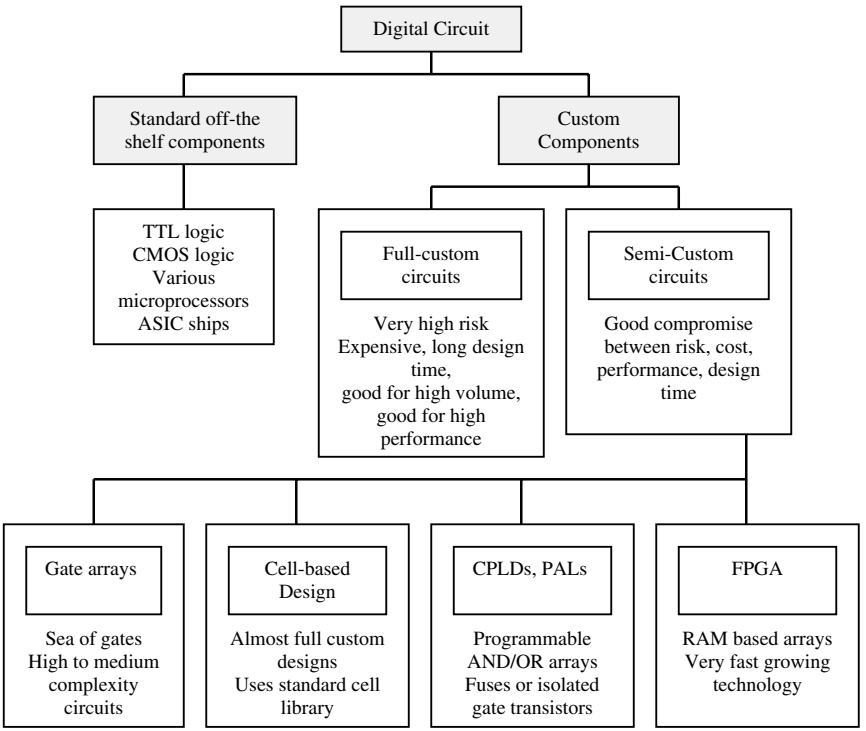


Figure 2.30 Implementation technologies: Processor and IC technologies.

Keeping this fact in mind, Figure 2.30 summarize the possible processor and IC technologies that may be used to implement any digital circuit. The figure gives the main features, advantages and limitation of each technology.

2.7 Summary of the Chapter

In this chapter we introduced the main technologies and tools that any designer has to be familiar with before starting designing any system. We started by introducing the design life cycle which is directly related to what is known as "design process technology". Structured design, use of top-down/bottom-up, and use of views are some of the techniques that can be used by the designer to convert requirements into a system.

Processor technology and IC technology are also discussed and the effect of each variant on the implementation of the design metrics (constraint) is discussed.

2.8 Review Questions

2.1 List and define the three main processor technologies. What are the benefits of using each of the three different processor technologies?
2.2 List and define the three main IC technologies. What are the benefits of using each?
2.3 List and define the three main design technologies. How are each of the three different design technologies helpful to designers?
2.4 Create a 3×3 grid with the three processor technologies along the x-axis, and the three IC technologies along the y-axis. For each axis, put the most programmable form closest to the origin, and the most customized form at the end of the axis. Explain features and possible occasions for using each of the combinations of the two technologies.
2.5 Using NAND gates as building block, (a) draw the circuit schematic for the function $F = xz + y\bar{z}$, and (b) draw the top-down view of the circuit on an IC.
2.6 Implement the function $F = xz + y\bar{z}$ using PAL.
2.7 Show behaviour and structure (at the same abstraction level) for a design that finds minimum of three input integers, by showing the following descriptions: a sequential programme behaviour, a processor/memory structure, a register-transfer behaviour, a register/MUX structure and a gate/flip-flop structure. Label each description and associated level with a point on the Y-chart.
2.8 Explain the terms IP core, FPGA, CPLD, PLA and PAL.
2.9 What do you mean by System-on-Chip (SoC)? How will the definition of embedded system change with System-on-Chip?
2.11 What are the advantages offered by an FPGA for designing an embedded system?
2.12 What are the advantages offered by an ASIC for designing an embedded system?
2.13 What are the advantages offered by an ASIP for designing an embedded system?
2.14 Real time video processing needs sophisticated embedded systems with specific real time constraints. Why? Explain it.
2.15 What are the techniques of power and energy management in a system?
2.16 What is the advantage of running a processor at reduced clock speed in certain sections of instructions and at full speed in other sections of instructions?

3

Introduction to Microprocessors and Microcontrollers

THINGS TO LOOK FOR…

- Processor Architecture
- The Microprocessor and its main blocks
- The Microcontroller and its main blocks
- The design using microprocessor-based systems and microcontroller-based systems.
- The design cycle of microcontroller based systems

3.1 Introduction

In Chapter-2 we classified processors into three classes: General Purpose Processor (GPP), Application Specific Instruction Set Processor (ASIP), and Single (Special) Purpose Processor (SPP). The microprocessor is designed to be a GPP; It is designed to be able to handle massive amount of data of any type (e.g. bytes, words, integers, real, complex, etc.) and it is possible to use it in any application, just programme it. This explains why the microprocessor is the brain of any general purpose computing system whether it is minicomputer, minicomputer, main frame or even super computer. The microcontroller, on the other hand, is one of the popular forms of ASIP; it is a processor optimized to work in control-oriented applications.

It is the goal of this chapter to introduce the microprocessor and microcontroller to the reader; the structure, the main building blocks, the differences and the capabilities. This chapter is necessary to give the reader an idea on the rest of the book; the microcontroller resources, capabilities and applications.

Figure 3.1 depicts the overall architecture of a typical general purpose embedded system that is traditionally known as microcomputer system. The components shown in the figure are: the Input/Output unit, the memory unit,

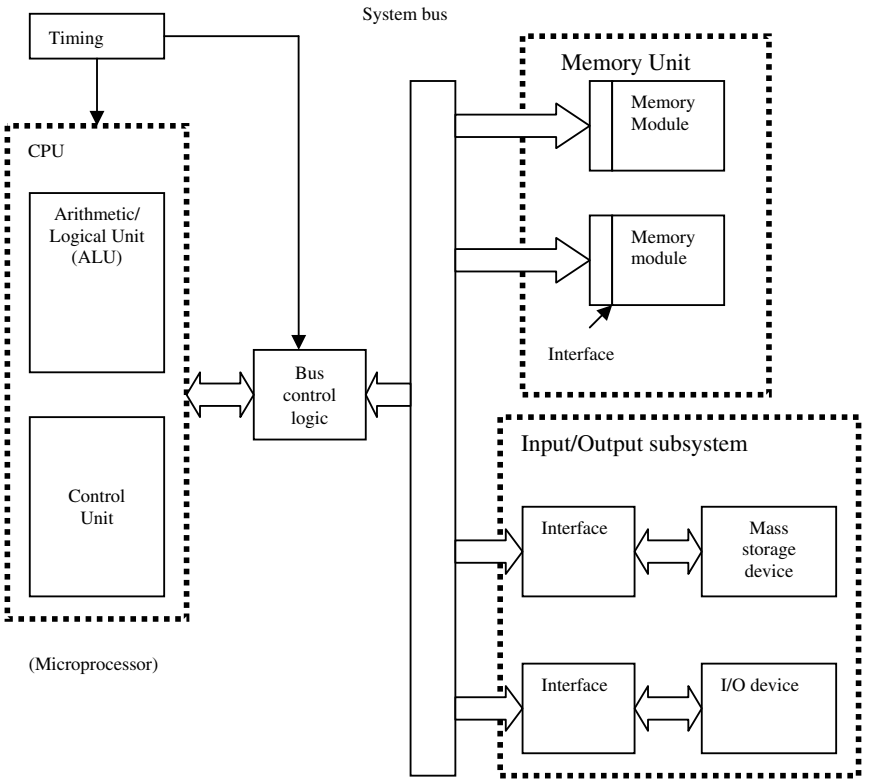


Figure 3.1 Organization of a typical microcomputer (general-purpose embedded system).

the arithmetic/Logic unit (ALU), the control unit, timing circuitry, bus control logic, and system bus. The arithmetic and logic unit together with the control unit form the central processing unit (CPU) of the microcomputer. When implementing the CPU as a single IC it is called "microprocessor" unit (MPU). The MPU is a general purpose processor its main purpose is to decode the instructions given to it and uses them to control the activity within the system. It also performs all arithmetic and logical operations.

The timing circuitry, or clock, generates one or more trains of evenly speed pulses needed to synchronize the activity within the microprocessor and the bus control logic. The pulse trains output by the timing circuitry normally have the same frequency but are offset in time, i.e., are different phases. Microprocessors require from one to four phases, with earlier processors requiring more than one phase. For many of the recent microprocessors, the timing circuitry, except for oscillator, is included in the same integrated circuit.

The memory is used to store both the data and programmes that are currently being used. It is normally broken into several modules, with each module containing several thousands (or several hundreds of thousands) locations. Each location may contain part or all of a datum or instruction and is associated with an identifier called a memory address (or simply address). The CPU does its work by successively inputting, or fetching, instructions from memory and carrying out the tasks dictated by them.

The I/O subsystem may consist of a variety of devices, e.g. mass storage units, analogue to digital converters (A/D), digital to analogue (D/A) converters, etc.

The system bus is a set of conductors used to connect the CPU to its memory and I/O devices. It is over these conductors, which may be wires in a cable or lines on a printed circuit (PC) board that all information must travel. The bus specifications determine exactly how information is transmitted over the bus. Normally, the bus conductors are separated into three groups:

- The data lines for transmitting the information.
- The address lines, which indicate where the information is to come from or is to be placed.
- The control lines, which regulate the activity on the bus.

The signals on the bus must be coordinated with the signals created by the various components connected to the bus. The circuitry needed to connect the bus to a device is called an interface, and the bus control logic is the interface to the CPU.

Memory interfaces consist primarily of the logic needed to decode the address of the memory location being accessed and to buffer the data onto or off of the bus, and of the circuitry to perform memory reads or writes. I/O interfaces may be quite simple or very complex. All I/O interfaces must be capable of buffering data onto and/or off the system bus, receiving commands from the CPU, and transmitting status information from their associated devices to the CPU. In addition, those connected to mass storage units must be capable of communicating directly with memory and this required them to have the ability to control the system bus. The communication between an I/O interface and the data bus is accomplished through registers which are referred to as I/O ports.

3.1.1 Processor Architecture and Microarchitecture

Computer architecture differs from computer microarchitecture. Computer architecture is related to the functions and the operations that the computer

can achieve, i.e. it is a specification for "what" the processor will do. It defines the functional capabilities of the computer. It is also related to the interface between hardware and software. This is directly related to the instruction set, the addressing modes and the types of data that the computer can handle. Computer architecture answers the question of "what" the processor will do, but does not answer the question of "how" to do it. Computer "Microarchitecture" answers the "how" question. While architecture is directly related to design for functionality, the answer of the "how" is related to implementing design constraints, e.g. performance, power consumption, etc. The designer, for example, may use pipelining, super-pipelining, cache memory with different levels, fast adders, etc. to get better performance. It is possible to have more than one microarchitecture all of them are answering the same question of how to implement a given architecture. The microarchitecture is not visible for the user but he can feel its effect; a user may execute his programme on two processors with different microarchitecture, the two will produce the same result but one of them will execute the programme faster than the other.

The Intel ×86 families is the best example that shows the difference between architecture (functionality) and microarchitecture (how to implement the functionality). Intel 8086, ×286, ×386, ×486, Pentium I up to Pentium IV represent different microarchitecture for the same architecture. The improvement that happened in the performance of the ×86 family members is entirely due to advancement in the internal organization (the microarchitecture). The internal organization of the processor has, generally, a large impact on its performance; how fast the processor executes a programme (or instruction).

Many users and many authors are using the two terminologies "architecture" and "microarchitecture" alternatively. To avoid any misunderstanding, we prefer the use of terminology "organization" also "structure" to replace the word "microarchitecture.

The internal organization (or microarchitecture) of any processor is broken down into different functional units. Each unit implements specific function. The flow of the data between the different blocks takes place under the control of the control unit to guarantee that the instructions are executed correctly. The way of executing the instructions may change from machine to machine but the organization of all of them contains the same basic functional units.

To make use of the functional capabilities of the processor, it must be supported by a minimum amount of hardware that facilitates its communication with the external words. The processor together with the supporting hardware forms together what we call the computer. Figure 3.1 represents the general organization of many computers. In Figure 3.1, the CPU administers

all activity in the system and performs all operations on the data as described in the instruction set.

3.2 The Microprocessor

The single chip IC implementation of the CPU is called microprocessor. The microprocessor internal structure, as a CPU, must be able to implement all the functional capabilities as defined by the instruction set of a GPP. The instruction set of any high-level language is seen that a CPU must facilitate working with (See Chapter 4):

1. Assignment and arithmetic expressions.
2. Unconditional branches.
3. Conditional branches and relational and logical expressions.
4. Looping.
5. Arrays and other data structures.
6. Subroutines.
7. I/O.

A typical CPU architecture that is designed to accommodate these features is given in Figure 3.2. It consists of:

- **Datapath or Arithmetic Unit:** This includes a set of working registers for helping with addressing and computational tasks, an arithmetic/logic unit (ALU) for executing the arithmetic and logical operations, and in modern microprocessors, dedicated hardware for multiplication/division and other special arithmetical operations as square root and floating-point units.
- **Controller:** It includes a control unit for decoding and carrying out the instructions and an I/O control section for handling I/O.
- **Internal bus system:** to allow the different blocks to communicate.

The microprocessor creates a set of buses called "external bus system" to allow the communication between the CPU and the external world.

The next subsections describe the function and operation of the different blocks forming a typical microprocessor internal structure.

3.2.1 General-Purpose Registers

Some of the general-purpose registers are available for general use by the programmer and some are dedicated to a specific task. We can recognize the

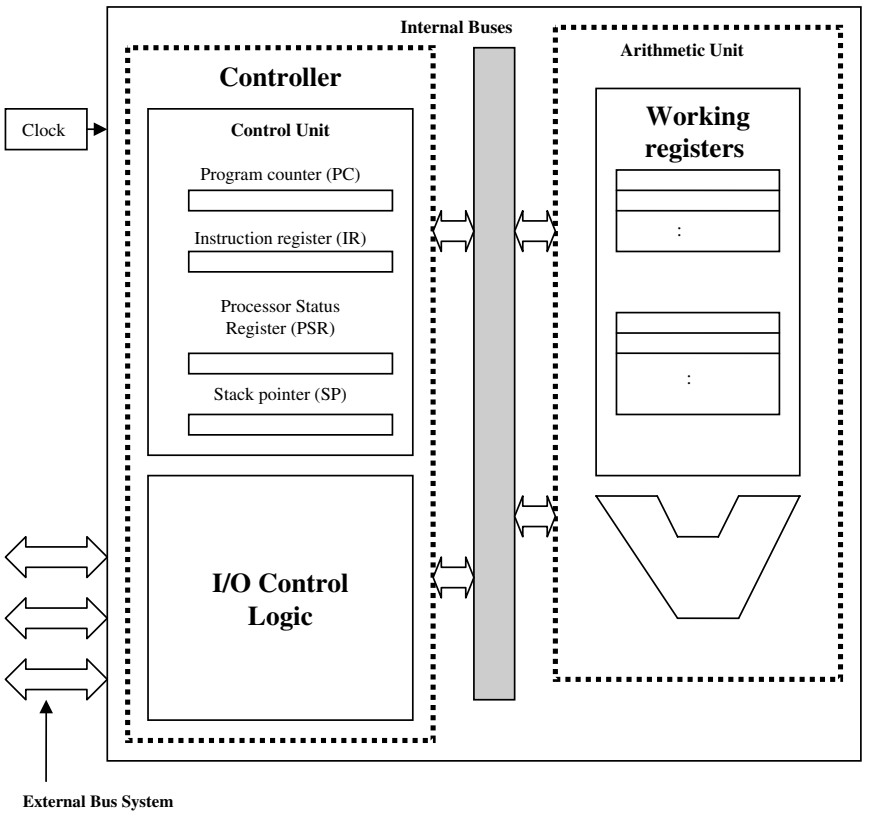


Figure 3.2 Typical CPU architecture.

following groups; working register group, address register group and special register group.

3.2.1.1 Working Registers Group

The working registers represent the top of the memory hierarchy pyramid. They are the fastest storage area available for use by the central processor at the same time they are limited in number; every processor normally has between 8 and 32 registers. The integrated technology used to implement the registers is that used to implement the ALU at the same time they are hard-wired right into ALU. The speed of the registers, accordingly, is comparable to that of the ALU. The processor uses them to hold data that is being worked on by the processor. The working registers are available for general use by programmers.

3.2.1.2 Arithmetic Registers group

Arithmetic registers are for temporarily holding the operands and results of the arithmetic operations. Because transfers over the system bus are the primary limiting factor with regard to speed, accessing a register is much faster than accessing memory. Therefore, if several operations must be performed on a set of data, it is better to input the data to the arithmetic registers, do the necessary calculations, and return the result to memory than it is to work directly from memory. The tendency is that the more arithmetic registers a CPU contains, the faster execute its computations.

The number and the function of the general-purpose registers depend on the structure of the processor. Concerning the number of registers there is two polices; the use of first level cache (primary cache) together with small number of working registers: and the use of large register-file only. Complex Instruction Set Computer (CISC) processors are using the first policy while "Reduced Instruction Set Computer" (RISC) processors are adopting the second policy. Each policy has its own advantage and disadvantages.

The function of the general-purpose (working) registers differs according to whether it is part of CISC or RISC structure. The working registers in CISC structures are storage registers where data can be modified only by writing a new data value to the register. Arithmetic and logical operations, in such a case, are taking place in a special register — the accumulator. In case of RISC structure all the working registers are operational registers, i.e. they have built-in adder and logic circuits such that they can modify their stored data by performing some arithmetic or logical data transformation.

The width (in bits) of the working registers determines how much data can be compute with at a time. 32-bit processor means that the width of the working registers is 32 bits and at the same time means that it is possible to operate on operands (add them for example) each of width 32 bits.

3.2.1.3 Address Registers Group

The address group is used for making the addressing of data more flexible. In Chapter 4 we are discussing the topic of "addressing modes"; the ways of getting the actual address of the location in the memory that contain the required information (programme instruction or data). In its simplest form, the required data is included as part of the instruction currently under consideration (this is called immediate addressing mode". The actual address (called effective address) of the required data can be given as part of the current instruction (this part is called address field). This mode is called "direct addressing" mode. More complex modes are included in the instruction set

of many microprocessors especially what is called "Complex Instruction Set Computer" (CISC). Many of these complex addressing modes need some calculations to get the effective address of the needed information. For example, the effective address may be the sum of the contents of the address field of the instruction and the contents of one or more other registers. Another example is when accessing elements of an array. In such a case the address of an element is comprised of two parts, a base address, which is the address of the first element in the array, and an offset. Because one often needs index through an array, it is helpful if the offset can easily incremented. Therefore, the address of an array element is frequently computed by adding two registers together, one which contains the base address and is called a **base register** and one which contains the offset and is called **index register**. A two dimensional array offers a slightly more complex situation which requires the addition of a base, a column offset, and a displacement. This normally involves the sum of part of the instruction (the displacement), a base register, and an index register. Base registers are also used to relocate programmes and blocks of data within memory.

We can conclude here that while getting the effective address of the location where the needed information is stored, some registers are needed. Having a separate address group of working registers improves the processors performance specially those using complex addressing modes. Programme counter (PC) and stack pointer (SP) belong also to the address group of registers.

3.2.1.4 Special Registers Group

This group includes the primary cache, scratchpad, and the stack.

(a) Primary (Level 1) Cache

Till recently, it was the CISC processors that use first level cache together with traditionally small number of working registers. Now, all the recent processors, including RISC processors, incorporate a small, high-speed cache right on the chip, to hold recently-used data and instructions from memory. The cache memory has extremely fast access that result from the technology used (it is SRAM), the way of accessing, and the short propagation time since it is right on the chip. The accessing takes place using "search" operation in which we are searching if the information needed is stored in it or not. If it is stored, we call it "hit", and because it is SRAM it put it on the bus very fast. If the search operation failed, we call it "miss", the main memory will respond by sending the required information. The cache memory uses a principle called "locality" of reference. The locality of reference states that "if the processor

recently referred to a location in memory, it is likely that it will refer to it again in the near future". Using this principle, the cache stores not only the most recent used piece of information but the block (normally eight words) of data that includes the recent word. This saves significantly the time needed to access data and improves the system performance.

The cache that is build-on the chip of the microprocessor is called level 1 cache and it is possible to consider it as extension to the register file. Some microprocessors have up to 64 K, even more, primary cache (level 1). The cache may contain data and instruction or it is possible to split it into two; data cache, and instruction cache.

The reader can find a lot of information about cache memory and their effect on the performance of the system in many computer textbooks.

(b) Scratchpad

It is important mentioning here that the level 1 cache is doing the same job as what was called "scratchpad". The scratchpad is simply a set of general-purpose internal registers that takes the form of an internal RAM. An address must be specified before the contents of the registers may be used. The name scratchpad was given to these internal general-purpose registers because they are essentially used to store information in a fast memory. Normally a scratchpad is connected to both the internal data and address buses. Scratchpad registers differ from usual registers by the way in which they are addressed. Normally registers are addressed by specialized instructions (the number of the register is contained within the instruction itself), while the scratchpad registers require the use of memory-type addressing instructions.

(c) The Stack

The stack is one of the specialized register structures (exactly as the level 1 cache or the scratchpad) that may also be available on the microprocessor chip. A stack is a "last-in-first-out" (LIFO) structure. It is a chronological structure that accumulates events (or symbols) in the order in which they are deposited. The oldest symbol is located at the bottom of the stack and the newest is located at the top. A stack works in the same way as a stack of plates in a restaurant (see Figure 3.3). In such a stack, plates are piled in a circular hole equipped with a spring at the bottom. New plates are deposited on the top and a plate is always removed from the top, (i.e., access to the stack is via the top). In other words, the last element in is the first element out: this is a LIFO structure.

As shown in Figure 3.4, a stack is manipulated by two instructions PUSH and POP. A PUSH operation deposits the contents of a register on the stack.

A POP operation removes the top element of the stack and deposits it into a register (normally the accumulator). Stacks are necessary to provide interrupt and subroutine levels. They can be implemented in two essential ways- through hardware and through software.

A hardware stack is implemented directly on the microprocessor chip by a set of internal registers. If *N* registers are dedicated to a stack operation, we call *N* as the depth of the stack. The advantage of a hardware stack is the high speed inherent to the use of internal registers, while the disadvantage is the limitation imposed on the depth of the stack. Whenever the *N* registers are full, the internal stack is full. So that the stack can continue to be used after it is full, all of the *N* registers must be copied into the memory. This process gives rise to another problem: one must be able to tell when the stack is empty or full. Most of the manufacturers of the early microprocessors simply forgot to incorporate these details into their designs, and most of the processors did not provide a "**stack full**" or a "**stack-empty**" flag (e.g. Z80). Unfortunately, in these early microprocessors one can be merrily pushing items into stack only to have them fall through the last word, without giving any indication to the programmer that something might be wrong. Naturally, this is a programming error; the programmer should know better. In practice, a flag would solve this problem. Conversely, it is possible for the programmer to keep pulling elements out of a hardware stack forever. For this reason, it is advisable to have a "stack-empty" indicator.

The alternative to a hardware stack is a software stack. To provide for unlimited growth, the stack is implemented in the read/write memory of the system, i.e. the RAM. The base of the stack is arbitrarily selected by the programmer. The top of the stack is contained and updated automatically within the stack pointer (SP) register. Each time a PUSH operation is executed, the SP is incremented or decremented, depending on the convention used (i.e., depending on whether the memory "grows" or "shrinks" from bottom to top). Similarly, each time a POP is done, the stack pointer is immediately updated. In practice, the SP usually points to the word above the last element of the stack. The goal is to provide the fastest possible PUSH operation. With this convention, the stack pointer can be used directly, without having to wait till it is incremented when saving a word quickly on the stack.

In case of microcontrollers, e.g. Intel 8031/8051 and Atmel AVR, the system uses an area within the internal SRAM as the system stack. In case of Intel 8051, the stack is kept in the internal RAM and is limited to addresses accessible by indirect addressing. These are the first 128 bytes on the 8031/8051 or the full 256 bytes of on-chip RAM on the 8032/8052. The programmer can reinitialize the SP with the beginning of the stack or he can let it retain its

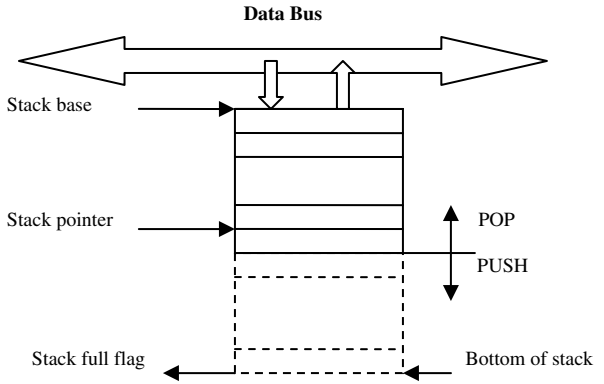


Figure 3.3 A stack is always accessed from the top.

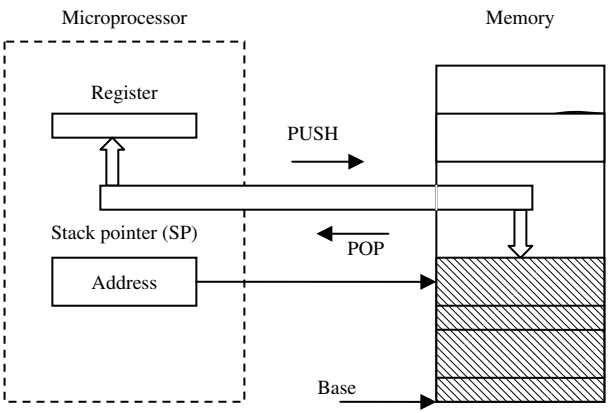


Figure 3.4 Software implementation of the stack.

default value upon system reset. The rest value of 07 H maintains compatibility with the 8051's predecessor, the 8048, and results in the first stack write storing data in location 08 H.

In other cases, as Code VisionAVR C language compiler, the compiler implements two stacks: The system stack starts at the top of the SRAM area and is used to store return addresses and processor control information. The data stack starts below the system stack and works its way down through memory, in a manner similar to the system stack, and is used to store temporary data, such as the local variables used in a function.

In microcontrollers, the stack is accessed, as any microprocessor, explicitly by the PUSH and POP instructions, and implicitly during operations such as subroutines calls, function calls and interrupt service action. In the first

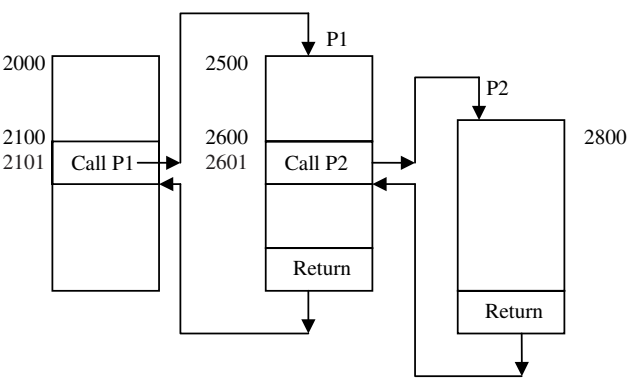
case, the stack can be used in the evaluation of arithmetic expressions (See example 3.1).

Use of stack

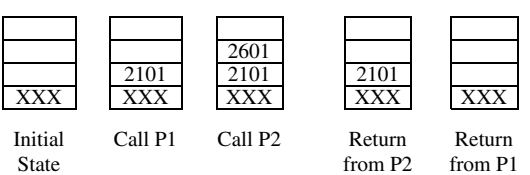
Some of the applications of the stack in computer are:

(a) To store the return address — managing procedure calls and returns.

One of the important developments of programming languages is the use of procedure. The procedure is a complete programme incorporated within the main programme. It is used to achieve economy and more important to achieve regularity. A procedure allows the programmer to use the same piece of code many times the matter that achieves effort and minimizes the storage area needed to store the programme. Modularity is achieved by dividing the complex programme into smaller programmes. Calling a procedure lets the programme branches from the current location to the procedure, executes the procedure, and then returns back from the procedure to the place from which it was called. Figure 3.5 illustrates the use of the procedure. Actually, as shown in Figure 3.5, it is possible to call a procedure within a procedure;



(a) Nested Procedures



(b) Stack contents

Figure 3.5 Use of stack to store return addresses.

this is called nesting of procedures. In such cases each procedure has its own return. For proper operation, the return address of each procedure must be stored safely to use it when the execution if the procedure completes. From Figure 3.5, it is clear that the order of returning from the nested procedures is following the opposite order of calling them. In other words they follow the last–in-first-out mechanism. This can be accomplished by using a stack; pushing the return addresses in the order of calling the procedures and popping them back in the reverse order after completing the procedures.

(b) To execute arithmetic operations.

Example 3.2 explains such application. In general, to use the stack to perform arithmetic operation we have to convert the usual "infix" notation that we are familiar with into what is called "reverse Polish" or "postfix" notation. For example, the expression (A + B) ∗ C, in "infix" notation, takes the form AB + C*, in "reverse Polish" notation. This subject is out of the scope of this book, this is why in Example 3.1 we showed how to use the stack to execute an expression written in the normal notation, i.e. written in infix notation.

Example 3.1 PUSH and POP operations

This example describes a PUSH A and a POP A instructions, where the contents of the accumulator will be transferred to or read from the top of the stack (TOS). In the PUSH operation illustrated in Figure 3.6a, the contents of the accumulator are transferred to the top of the stack. The contents of

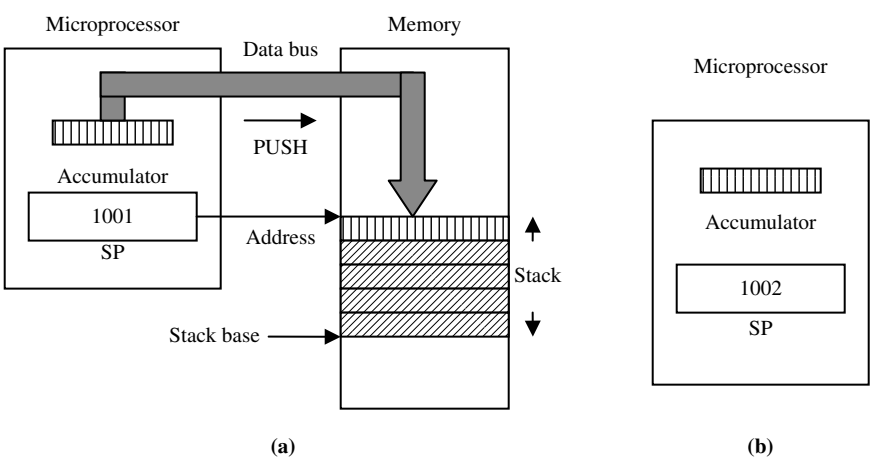


Figure 3.6 PUSH a instruction.

the stack pointer (SP), in our case 1001, points to the address of the first available location in the stack. After the PUSH is executed, SP is automatically incremented to the value 1002 (see Figure 3.6b) and point to the new "first available location".

Conversely, POP A fetches the top element of the stack (see Figure 3.7) and loads it into the accumulator. The initial value of SP was 1002. It is automatically decremented to 1001 prior to the memory fetch. The final contents of the stack appear in Figure 3.7b.

Example 3.2 Use of stack for Expression Evaluation

Consider the following programme fragment. Show the contents of the stack at the end of executing each step in the programme and give an expression describes the value of the result X (Use TOS as top of stack).

```
PUSH V          ; TOS ← V
PUSH W          ; TOS ← W
ADD             ; TOS ←(W + V)
PUSH V          ; TOS ← V
PUSH X          ; TOS ← X
ADD             ; TOS ← (X + V)
MUL             ; TOS ← (X + V) * (W + V)
PUSH Y          ; TOS ← Y
ADD             ; TOS ← Y + (X + V) * (W + V)
POP Z
```

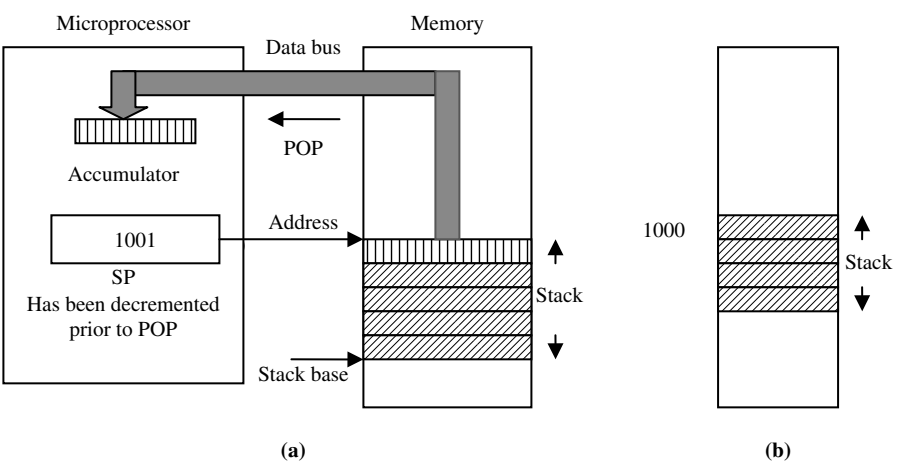


Figure 3.7 POP a instruction.

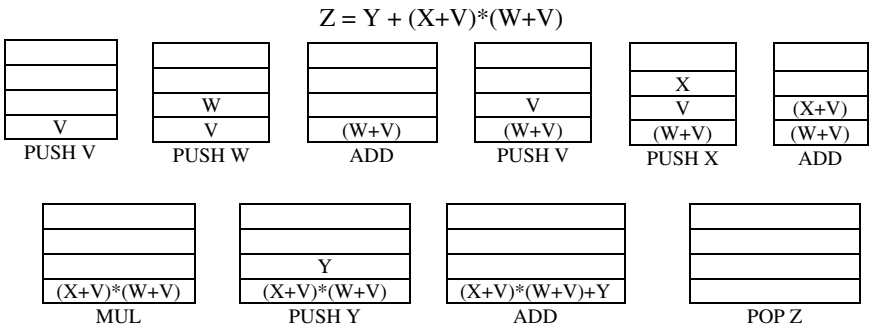


Figure 3.8 Use of the stack.

Answer: The contents of the stack during the execution of each instruction of this programme are shown in Figure 3.8

The programme is calculating the value of the expression:

$$Z = Y + (X + V) * (W + V)$$

3.2.2 Arithmetic and Logic Unit (ALU)

The main function of any processor system is to process data, for example, to add two numbers together. The arithmetic and logic unit (ALU) therefore is an essential feature of any microprocessor or microcontroller. The ALU takes, in general, two data words as inputs and combines them together by adding, subtracting, comparing and carrying out logical operations such as AND, OR, NOT, XOR. To achieve that, the arithmetic unit, Figure 3.9, uses a number of general-purpose registers to store temporarily, the information (normally, the operands defined by the instruction currently executed) besides the arithmetic and logic unit (ALU) for performing operations on this information. Normally, the ALU uses an accumulator that allows several operations to be performed on its contents, and a flag register (processor status register) that stores information relative to the results of each arithmetic or logic operations. The accumulator width in many cases equals to the data bus width. Although this relationship is still holds for many microprocessors, some devices now apply accumulators wider than the data bus width. A wider accumulator can speed up several operations.

The interface of the ALU, as shown in Figure 3.9, consists of; lines for the input operands terminated at source #1 and source #2 registers; lines (control, select, operation in the figure) for the input commands that define and initiate

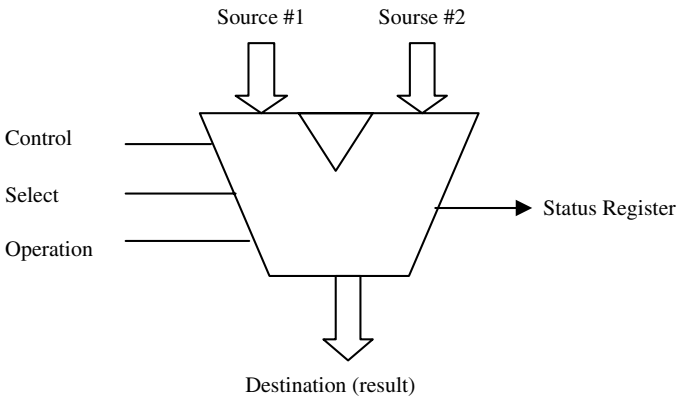


Figure 3.9 Arithmetic logic unit block diagram.

the operation; lines for the output of status indications (originate in the logic circuitry associated with the ALU and terminate at the status register). One of the differences between the different available microprocessors is the source of the two operands required for execution and the destination of the result. In accumulator–based processor (mainly CISC structure) the accumulator represents one source with another word. The accumulator in such structure is the destination of the result. In RISC processors, all the general-purpose registers are operational registers doing the function of the accumulator. The sources and the destination in RISC processors are working registers. In case if one (or the two) operands are in the memory, RISC structures use special instruction to bring that operand from the memory to one of the registers at first before operating on it. Other processors allow the user to define the two different sources and the distention.

To improve the performance of the CPU, it is normal to divide the ALU into two units: an arithmetic unit (AU) and a logic unit (LU). For achieving further improvement in the performance, some processors use:

— Different units to handle integer numbers and floating numbers.
— Dedicated hardware for multiplication/division
— Use of branch prediction unit: This is used by some processors to speed up the (next) instruction prefetch. It predicts whether a conditional branch will be taken or not and accordingly prefetch the next instruction from the correct location. This unit is normally used in RISC processors to keep the pipeline full all the time.

3.2.3 Control Unit

The control unit directs the operations carried by the processor. The control unit receives information from the other units and in turn transmits information back to each of them. When it is time for some input information to be read to the memory (this is defined by the current instruction in execution), the control unit stimulates the input device appropriately and sets up a path from input device to the proper place in the arithmetic unit (normally the accumulator) or memory unit. Similarly, when an arithmetic operation is to be performed, the control unit must arrange for the appropriate operands to be transferred from their source locations to the arithmetic unit, for the proper operation to be executed and for the result to go back to the defined destination.

The functions performed by the control unit are defined mainly by the instruction set of the processor. The internal structure (the organization) of the CPU has also great influence on such functions.

In its simplest form, Figure 3.2, the control unit contains:

- An operation control and **timing circuitry** (it is also called instruction decode and control unit) that determines the operation to perform and sets in motion the necessary actions to perform it,
- An **instruction Register** (IR) that holds the binary code for the current instruction while it is being decoded and executed,
- **Programme counter** (PC) holds the memory address of the next instruction to be executed. At the end of the execution of the present instruction, the address in the PC is placed on the address bus, the memory places the new instruction on the data bus, and the CPU inputs the instruction to IR. While this instruction is decoded its length in terms of memory width is

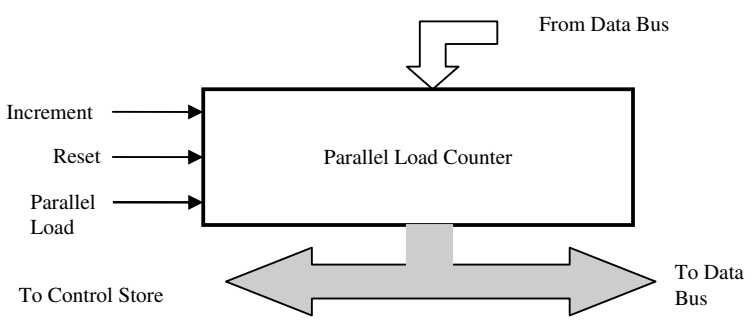


Figure 3.10 Programme counter.

determined and the PC is incremented by the length so that the PC will point to the next instruction.

- **Processor status word** (PSW) or **Processor status register** (PSR) (also called Flags status register): It stores information relative to the results of each arithmetic or logic operation.
- **Stack pointer** (SP) holds the address of the data item currently on the top of the stack. The control unit fetches each instruction in sequence, decodes and synchronizes it before executing it by sending control signals to other parts of the computer.

3.2.4 I/O Control Section (Bus Interface Unit)

The I/O control section or the bus interface unit is that part of the processor that contains the CPU's logic that is related to the I/O operations, i.e. that interfaces with the rest of the PC. It deals with moving information over the processor data bus, the primary agent for the transfer of information to and from the CPU. The bus interface unit is responsible for responding to all signals that go to the processor, and generating all signals that go from the processor to other parts of the system.

The control unit communicates with the external world via input and output lines that make up the control bus. The control bus is normally managed by the control unit.

3.2.5 Internal Buses

Buses in computer are used as highways; used by the different units to communicate. The communication between the different units within the CPU is provided by "internal buses". The microprocessor creates another set of buses "external buses" to communicate with the external world. The bus is a collection of conductors (wires). In some cases the width of the internal buses are wider than the external ones (usually twice). This has direct effect on the overall performance of the system. The width of the internal buses has direct effect on the speed of executing all the instructions at the same time it has less effect on the overall system cost.

3.2.6 System Clocks

A timer circuit suitably configured is the ***system clock***, also called ***real time clock (RTC)***. An RTC is used by the schedulers and for real time programming. It is designed as follows: Assume a processor generates a clock output every

0.5 μs. When a system timer is configured by a software instruction to issue timeout after 200 inputs from the processor clock outputs, then there are 10,000 interrupts (ticks) each second. The RTC ticking rate is then 10 kHz and it interrupts every 100 μs. The RTC is also used to obtain software-controlled delays and time-outs.

More than one timer using the system clock (RTC) may be needed for the various timing and counting needs in a system.

3.2.7 Basic Microprocessor Organization

Figure 3.11 shows the basic microprocessor organization.

The External Buses: Address, Data, and Control System Bus (External Bus System)

As mentioned before, the various blocks forming the CPU (e.g. ALU, the general purpose registers, the control unit, etc) must communicate with each other. Similarly, the processor must communicate with the external world represented by the different I/O peripherals. The buses that are used to implement

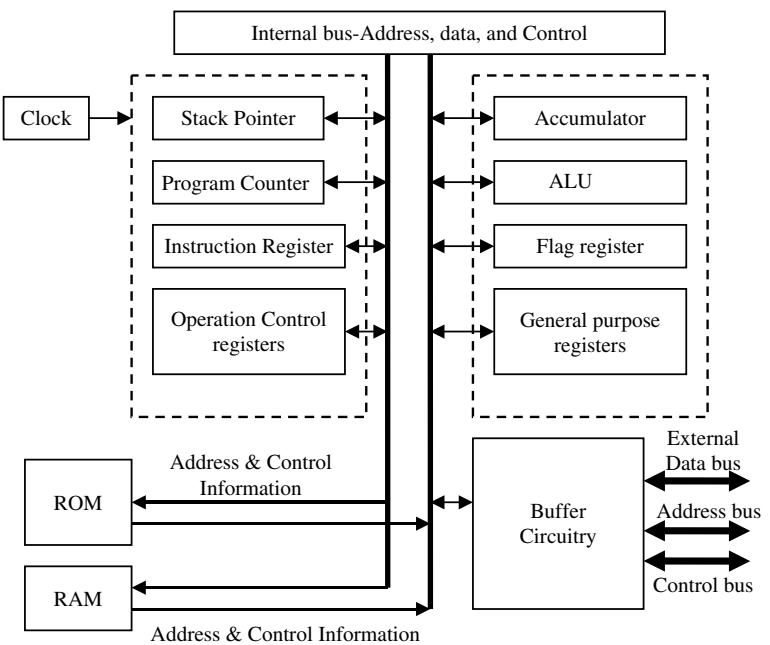


Figure 3.11 Basic microprocessor organization (CPU).

the interface between the internal blocks of the CPU are called "internal buses" and those used to interface the processor to the I/O devices are called "external buses"; data bus, address bus and control bus. The microprocessor creates the "external bus system".

The use of bus systems allows the user to connect new devices easily. In case of computer systems that use a common bus, it is easy to move the peripherals between them. The bus system has another advantage; it is a low cost system since it is a single set of wires that is shared in multiple ways. On the other hand, the limited bandwidth of the bus creates a communication bottleneck. For internal bus, such limitations affect directly the performance of the system, while the bandwidth limitations of the external bus limit the maximum I/O throughput. In many processors, to improve the overall performance of the system, they use internal bus wider than the external bus (normally twice).

System buses can be classified into:

- CPU-memory buses: This class is normally short, high speed and matched to the memory system to maximize memory-CPU bandwidth.
- I/O buses: This class of buses is characterized by high length, many types of I/O devices can be connected to it, have a wide range in the data bandwidth of the devices connected to them, and normally follow bus standards.

Traditionally the buses are classified into three function groups:

Data bus:

This is a set of wires that carry data being transferred between the CPU and the external devices connected to it, e.g. memory units and different types of I/O devices. The number of lines that form the data bus is referred to as the width of the data bus. The data bus width has a direct affect on the overall performance of the system. This is because the majority of the move operations are between the CPU and external memory (RAM or ROM). The width of the data bus defines the amount of data that the bus can support per second. In some applications the limitation in the amount of data that the bus can support causes a form of bottleneck. If the system has a CPU with tremendous computational power (can handle huge amount of data per second) at the same time it has memory system that has a huge capacity, then it is expected that the amount of data movement between the CPU and the memory per second to be huge. The bottleneck happens when the bandwidth of the data bus cannot support such amount of data. The data bus, in such cases, puts limitation on the number of

devices that can be connected to the bus system; adding more devices means more data per second has to move between the CPU and the devices.

It is important to remember here that when we say "32-bit processor" we do not mean that the data bus width is 32 bits. 32-bit processor means that the working registers are of width 32 bits each. As mentioned before, 32-bit processor means that the processor operates on operands of width 32-bit.

The data bus is bidirectional; data may travel in either direction depending on whether a read or write operation is intended.

Note: We used the term "data" in its general sense: the "information" that travels on the data bus may be the instructions of the programme, an address appended to an instruction, or data used by the programme.

Address bus:

The signals on the address bus represent the address of a location in the memory that represents a source of an operand or a destination of a result. The address can be also of an instruction or the address of an I/O port. The address bus width defines the memory space; the maximum number of memory locations that can be addresses. An n-bit address bus means that the memory space contains 2^n locations. When we say that our computer has a memory of 1 gaga byte (10^{30} bytes) this does not necessarily mean that the address bus is 30-bit width. It means that the internal RAM occupies an area of 1 gaga byte in the memory map (See Chapter 6). The address bus is unidirectional; address information is always supplied by the CPU (or sometimes by direct memory access DMA circuitry in addition to CPU) and the bus carries them to the different memory storage areas.

Control bus:

The control bus is used to control the access to and the use of the data and address lines. Because the data and address lines are shared by all components, there must be a means of controlling their use. Each combination of the control signals has a specific role in the orderly control of system activity. As a rule, control signals are timing signals supplied by the CPU to synchronize the movement of information on the address and data buses. The control bus provides, at least, four functions:

- Memory synchronization
- Input-output synchronization
- CPU (microprocessor) scheduling — interrupt and direct memory access (DMA)
- Utilities, such as clock and reset.

3.3 Microcontrollers

The microcontroller is an Application-Specific Instruction-set Processor (ASIP), that can serve as a compromise between the general-purpose processor (the microprocessor) and the single-purpose processor (application specific integrated circuit ASIC). An ASIP is a programmable processor optimized for a particular class of applications having common characteristics, such as embedded control, digital-signal processing, or telecommunications. The designer of such a processor can optimize the datapath for the application class, perhaps adding special functional units for common operations and eliminating other infrequently used units. The microcontroller is a processor optimized for embedded control applications.

Microcontrollers used in a wide number of electronic systems such as:

- Engine management systems in automobiles
- Keyboard of a PC
- Electronic measurement instruments (e.g., digital multimeters, frequency synthesisers, and oscilloscopes)
- Printers
- Mobile and satellite phones, video phones
- Televisions, radios, stereo systems, CD players, tape recording equipment
- Hearing aids, medical testing systems
- Security alarm systems, fire alarm systems, and building services systems
- Smart ovens/dishwashers, washers and dryers
- Factory control
- Many, many more

To understand how the architecture of the microcontroller is optimized for all the above mentioned application, we are going to start by considering the general block diagram of the microcomputer given in Figure 3.1. This general block diagram can be simplified to take that given in Figure 3.12. The microprocessor cannot work alone; it works only with minimum supporting hardware. The CPU and the supporting hardware are connected together with the external bus system to form a general purpose microprocessor-based-system which we call it normally a "microcomputer". The minimum supporting hardware includes: external clock, RAM, ROM, support for external I/O peripherals, besides the external bus system. Beside this minimum supporting hardware, the microprocessor-based system may include interrupt control circuit, serial interface, parallel interface, and timers.

The microcontroller is a microcomputer on-chip. It integrates many of the components of a microprocessor-based system (the components inside

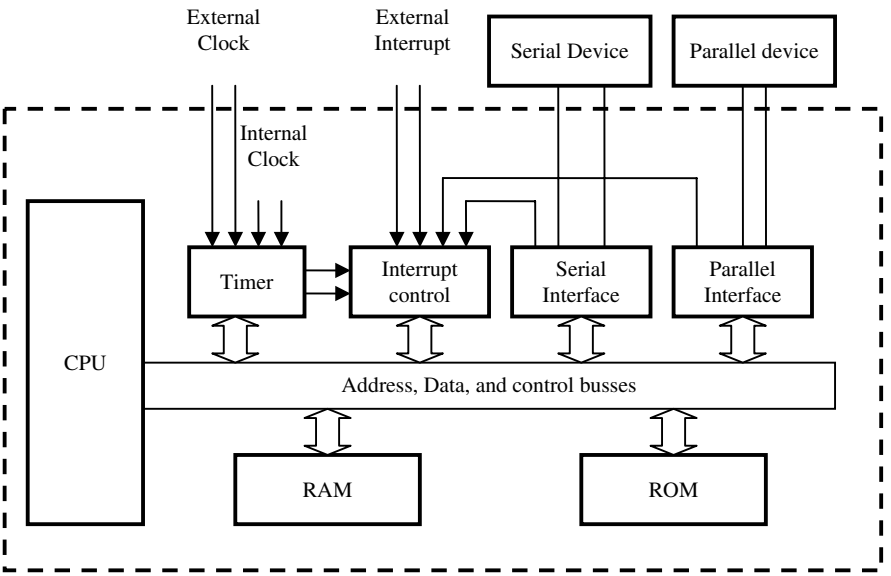


Figure 3.12 Microcomputer system.

the dotted line of Figure 3.12 onto a single chip. It is a complete computer system optimized for hardware control that encapsulates the entire processor, memory, and all of the I/O peripherals on a single chip. It needs only to be supplied power and clocking to start work. Being on the same chip enhances the processing speed because internal I/O peripherals take less time to read or write than external devices do.

Most microcontrollers, beside memory and I/O peripherals, combine on the same chip:

- A Timer module to allow the microcontroller to perform tasks for certain time periods.
- Serial I/O to allow data to flow (synchronously and/or asynchronously) between the microcontroller and other devices such as a PC or another microcontroller.
- Analogue input and output (e.g., to receive data from sensors or control motors). This helps in applications as data acquisition and control.
- Interrupt capability (from a variety of sources).
- Bus/external memory interfaces (for RAM or ROM).
- Built-in monitor/debugger programme.

- Support for external peripherals (e.g., I/O and bus extenders).
- Support for microcontroller area networks (CAN).
- Support Digital Signal Processing (DSP).

Integrating the components forming a microcomputer on a single chip represents the first step towards optimizing the microcontroller to be suitable for applications in the field of real time control. The second important aspect is that the CPU architecture of the microcontroller is also optimized. This is achieved by optimizing the instruction set of the microcontroller.

Optimizing the Instruction set: An instruction set architecture deals and specifies three wide areas: operations, data types, and addresses (or address modes). Then, saying that the architecture of the microcontroller is optimized for applications in the field of control (and other applications as mentioned before), it means accordingly that the instruction set of the microcontroller is optimized to contain instructions, uses data types and addressing modes that are suitable for control oriented applications. Microprocessor instruction sets are "processing intensive," implying they have many addressing modes with instructions that can handle large volume of data at one time. The addressing modes allow the system to access large arrays of data by using offsets and address pointers. The instructions can handle different types of data with different width; nibble, byte, 2 bytes, 4 bytes, and 8 bytes. Auto-increment and auto-decrement modes simplify stepping through arrays on byte, word, or double-word boundaries. Instructions that support multitasking, memory management, floating point calculation, and multimedia processing are part of the instruction set of many of the microprocessors available in the market. Microcontrollers, on the other hand, have instruction sets catering to the control of inputs and outputs. The interface to many inputs and outputs uses a single bit. For example, a motor may be turned on and off by a solenoid energized by a 1-bit output port. Microcontrollers have instructions to set and clear individual bits and perform other bit-oriented operations such as logically ANDing, ORing, or EXORing bits, jumping if a bit is set or clear, and so on. This powerful feature is rarely present in microprocessors, which are usually designed to operate on bytes or larger units of data. This gives the microcontroller the advantage of manipulating the single bit operations in one cycle compared with many cycles needed by the microprocessor instructions to handle such situation.

As a microcomputer on single-chip, the microcontroller is a microprocessor which has additional parts that allow it to control in real time external

devices. It can be used for:

- Collecting data from various sensors.
- Process the collected data to get a result that takes normally the form of a set of actions.
- Use the output devices to implement the tasks needed from the system.

If the designer considered the fact that the microcontroller has a large software and hardware capabilities he can use it to build a variety of efficient embedded systems. The software flexibility and the hardware resources available within the microcontroller allow the designer to create practically unlimited number of embedded systems by changing the software.

3.3.1 Microcontroller Internal Structure

For any microcontroller to be able to do its work, it must have some unavoidable sections (resources) on the chip, irrespective to the type or the manufacturer of the microcontroller. These resources include: CPU, I/O unit (ports), memory unit, serial communication, timer, watchdog timer, reset and brownout detector, oscillator and, in many cases, analogue-to-digital converter. Besides these hardware resources any microcontroller needs software (programme) resources. In the following we are briefly introducing these resources. The rest of the book is discussing in details these resources and how to use them.

3.3.1.1 CPU

CPU is the 'computer' part of the Microcontroller. Its main function is to run the programmes supplied by the designer. It does this by using memory, some registers, and the programme memory. Each manufacturer uses one microprocessor to be the base of his microcontroller devices. The ALU, in each case, is set up according to the requirements of the instruction set being executed by the timing and control block. For example Intel uses basic 8085 microprocessor core in Intel 8051 microcontroller. Similarly Motorola uses basic 6800 microprocessor core in M68HC11 microcontroller devices.

3.3.1.2 Memory unit

All microcontrollers have on-chip memory units (internal memory units) as part of its structure. The microcontroller uses the memory to store data and programmes. In some applications, when the capacity of the internal memory

units is not enough, additional external memory units are to be used with the microcontroller. The memory devices directly accessible by the CPU consist of semiconductor ICs called RAM and ROM.

The architecture of the microcontroller defines whether the data and the programmes are going to be stored in the same memory or in two separate memory units. In CISC structures one memory (with one memory map) is used to store the data and the programmes. To allow faster access and increased capacity, RISC structures use Harvard model. In RISC structures, the data and the programmes are separated from each other and two memory units, each has its own memory map, are required. The two memory units are:

- **Programme Memory:** The programme memory stores the instructions that form the programme. To accommodate large programmes, the programme memory may be partitioned as internal programme memory and external programme memory in some controllers.
- **Data Memory:** The data memory is used by the controller to store data. The CPU uses RAM to store variables as well as stack. The stack is used by the CPU to store return addresses from where to resume execution after it has completed a subroutine or an interrupt call.

Microcontrollers come with several different types of memory. The amount of memory on a single chip varies quite a bit by manufacturer.

Memory unit is the subject of Chapter 6.

3.3.1.3 Input-output unit

Having CPU, internal memory and buses on one chip, gives a unit that is capable, to some extent, of doing some functions. As far as functionality, the unit can do something, if it has the capabilities to communicate with the outside world. The busses are used to achieve this task in case of microprocessor. The microcontroller has an additional block contains several memory locations that act as interface between the internal data bus and the external devices. In other words one end of these special memory locations is connected to the internal data bus and the other end is connected to a number of pins on the chip. The above mentioned locations are called “ports” and the outside world units are called I/O devices or “computer peripherals”. Each port is identified by an address to which the read and write operation take place. The microcontroller gets the inputs by read operations at the port address. It sends the output by a write operation to the port address. Generally, connecting the microcontroller to the external words takes place in three phases or levels: ports, interfacing, and peripherals.

Ports

Ports for microcontroller are the same as the external buses of the microprocessor. The main difference between ports and buses is that many devices can be connected to the same bus but only one device can be connected to each port. In other words, the port is a bus that we can connect to it only one device (input or output device). Bus is a group of wires; similarly the port is a group of bins on a microcontroller. The busses start at registers inside the microprocessor; the port is a data register inside the microcontroller. The busses are interfaces between CPU and external world, similarly, the ports represented by its register represents the interface (the connection) between the CPU and the outside world. Ports can be either input ports, output ports or bidirectional ports. To define the direction, each port has a "data direction" register associated with its data register. This allows each pin to be set individually as an input or output before the data is read from or written to the port data register. When working with ports, first of all it is necessary to choose, based on the type of the peripheral, which port we need to work with, and then to send data to, or take it from the port. As far as the programme interaction with port is concerned, the port acts like a memory location. If a binary code is presented to the input pins of the microcontroller by external device (for instance, a set of switches), data is latched into the register allocated to that port when it is read. This input data can then be moved (copied), using the proper instruction, into another register for processing. If a port register is initialized for output, the data moved to that register is immediately available at the pins of the chip. It can then be displayed, for example, on a set of LEDs.

Ports can also be "Parallel" or "Serial", "synchronous or asynchronous" and "duplex or half duplex".

The interface circuitry

Many of the input/output devices are analogue. This means that they generate and send to the microcontroller analogue signals (input devices) and accordingly they are expecting, in response, to receive (output devices) analogue signals from the microcontroller. Keeping in mind that the ports are digital, it is the job of the interface circuitry to convert the voltage or current to binary data, or vice versa, and through software an orderly relationship between inputs and outputs is established.

Sometimes, the interface circuitry is also needed in spite of using digital peripherals. This happens, if the way of representing the logic "1" and logic "0" by the peripheral differs from that acceptable by the port. In this case the job of the interface circuitry is to match between them.

Input/output devices (computer peripherals)

I/O devices, or “computer peripherals," provide the path for communication between the computer system and the “real world." Without these, computer systems would be rather introverted machines, of little use to the people who use them. Three classes of I/O devices are mass storage, human interface, and control/monitor.

3.3.1.4 Serial communication

The I/O devices mentioned before provide means to transfer the data in the form of words (or bytes) between the computer system and outside world. This way is called “parallel communication” and it has some drawbacks; the number of lines (or channels) needed to transfer the data. The number of lines needed equals to the word length (number of bits per word). In case of long distance communication, the cost of the wire lines may affect the economy of the project. To reduce the number of lines, without affecting the functionality, serial communication can be used.

Serial communication, via port, is used to communicate with external devices on a serial data basis. The serial port takes data word from the computer system and shifts out the data one bit a time to the output (parallel to serial conversion). Similarly, it accepts external data a bit at a time, makes a word, and presents this to the computer system (i.e. serial to parallel conversion). The serial port can operate at any required data transfer speed.

Serial communication is the subject of Chapter 9.

3.3.1.5 Clock oscillator

The processor executes the programme out of the memory at a certain rate. This rate is determined by the frequency of the clock oscillator; the clock controls the time for executing an instruction. The clock controls the various clocking requirements of the CPU, the system timer and the CPU machine cycle. The clock oscillator could be an internal RC-oscillator or an oscillator with an external timing element, such as a quartz crystal, an LC resonant circuit, or even an RC circuit. As soon as the power is applied to the controller, the oscillator starts operating.

3.3.1.6 Timer unit

In case of microprocessor, timers are used normally for providing a constant delay. In microcontroller, timers are used for a lot more applications. The timer is used by the microcontroller system to time events, count events (external

as well as internal), and many other applications. The timer is the subject of Chapter 7.

3.3.1.7 Watchdog timer

In any computer system if something went wrong during execution, the computer will stop execution. The user can reset it and let it continuing working. This matter is facilitated by the availability of special reset button. This is not the case of microcontroller; there is no reset button. The microcontroller uses another mechanism to achieve the same important task; to guarantee the error-free operations of the microcontroller. The watchdog timer is used to check the error-free running of the microcontroller. The watchdog timer is a free-run counter. Instead of generating a tick as in the case of timers, the watchdog timer waits a tick generated by the programme to arrive indicating that the programme is running correctly. If such a tick did not arrive to the watchdog before the end of a pre-defined period of time, the watchdog timer will take action by resting the microcontroller. The watchdog timer is considered in Chapter 7.

3.3.1.8 A/D and D/A converters

The peripheral signals in the majority of control applications are analogue, which is substantially different from the ones that microcontroller can understand (zero and one). These signals have to be converted into a pattern which can be comprehended by a microcontroller, i.e. into a digital form. This task is performed using an analogue to digital converter (ADC or A/D converter). The ADC block is responsible under software control for converting an information about some analogue value to a binary number and for follow it through to a CPU block so that CPU block can further process it.

Analogue output is performed using a digital-to-analogue converter (DAC). Most controllers are equipped with pulse-width modulators that can be used to get analogue voltage with a suitable external RC filter. DACs are used to derive motors, for visual displays, to generate sound or music, etc. In Chapter 8 we are introducing the A/D and D/A converters.

3.3.1.9 Reset and brownout detector

The reset circuit in the controller ensures that at startup all the components and control circuits in the controller start at a predefined initial state and all the required registers are initialized properly. Three resources can normally cause reset to occur: as a part of the power-on sequence of the microcontroller, by a timeout of the watchdog timer, and by the brownout detector. The brownout

detector is a circuit that monitors the power supply voltage, and if there is a momentary drop in voltage, resets the processor so that the drop in voltage does not corrupt register and memory contents, which could lead to faulty operation of the controller. The reset can be caused also by a clock monitor detector; if the clock slows down below certain threshold frequencies due to a fault.

3.3.1.10 Programme

The above sections have focused on computer systems hardware with only a passing mention of the programmes, or software that makes them work. The relative emphasis placed on hardware versus software has shifted dramatically in recent years. Whereas the early days of computing witnessed the materials, manufacturing, and maintenance costs of computer hardware far surpassing the software costs, today, with mass-produced LSI (large scale integrated) chips, hardware costs are less dominant. It is the labor-intensive job of writing, documenting, maintaining, updating, and distributing software that constitutes the bulk of the expense in automating a process using computers.

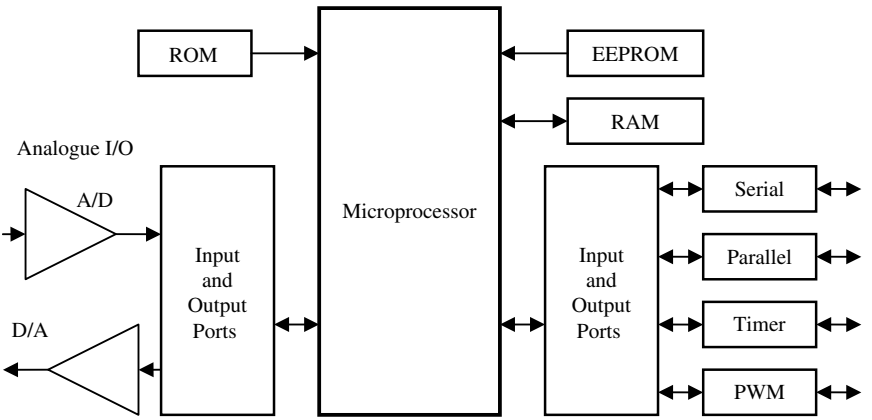
Assembly language and programming using assembly language are the subject of Chapters 4 and 5.

3.4 Microprocessor-Based and Microcontroller-Based Systems

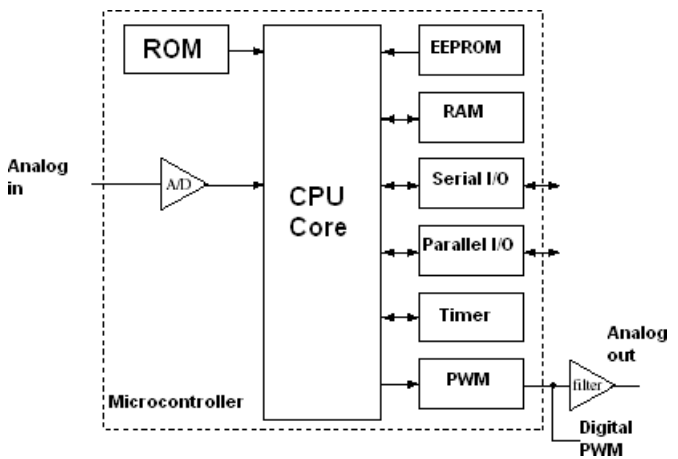
The microprocessor, as mentioned before, cannot work alone. It has to be supported by some external hardware units and software modules to form a working system. In general, the microprocessor, the supporting units and the software form what is called "microprocessor-based system" (See Figure 3.13a). Microprocessor-based systems are general purpose systems that can be programmed and reprogrammed depends on the task or the application and normally we call them computers. (The personal computer stands out as one of the most important and useful applications of microprocessor-based systems). There is minimum needed amount for the hardware as well as for the software that can form a functional microprocessor-based system. The minimum hardware configuration is called the "hardware kernel" and that of the software is called "software kernel".

Hardware Kernel:

- CPU
- System Clock
- System bus



(a) Microprocessor-Based System



(b) Microcontroller-Based System

Figure 3.13 Microprocessor- and microcontroller-based Systems.

Software Kernel:

This term is applied to the set of programmes in an operating system which implement the most primitive of the functions of the system. Typical kernel contains programmes for four types of functions:

- Process management-Routines for switching processors among processes; for scheduling; for sending messages or timing signals among processes; and for creating and removing processes.

- Memory management: Routines for placing, fetching, and removing pages or segments in, or from, main memory.
- Basic I/O control: Routines for allocating and releasing buffers; for starting I/O requests on particular channels or devices; and for checking the integrity of individual data transmission.
- Security: Routines for enforcing the access and information-flow control policies of the system; and for encapsulating programmes.

In some systems, the kernel is larger and provides for more than these classes of functions, in others, it is smaller.

The actual types, number and size of the supporting hardware and the software define the capabilities of the system (we call it easily computer system) and they are used to classify the system as microcomputers, minicomputers, mainframe computers, or supercomputers. All microprocessor-based system have a high RAM-to ROM ratio, with user programmes executing in a relatively large RAM space and hardware interfacing routines executing in a small ROM space.

The microcontroller, on the other hand, is the heart of any microcontroller–based system (See Figure 3.13b) which we can easily call an "embedded-system" (This does not mean that microprocessors or single-purpose processors cannot be used to build embedded systems). Such systems are single-function (executes a specific programme repeatedly), have tightly constrained on the design metrics (for example, small size, consumes low power, etc.), and they have real time reaction response. In such cases, the supporting units are peripheral devices (e.g., stepper motor, display unit, sensors, actuators, etc.). The type and the number of the peripherals depend on the application. External RAM and/or EPROM may be needed in some applications. Users of such products are quite often unaware of the existence of microcontrollers: to them, the internal components are but an inconsequential detail of design. Consider as examples microwave ovens, programmable thermostats, electronic scales, digital cameras, and even cars. The electronics within each of these products typically incorporates a microcontroller interfacing to push buttons, switches, lights, and alarms on a front panel; yet user operation mimics that of the electromechanical predecessors, with the exception of some added features. The microcontroller is invisible to the user.

Unlike computer systems, Microcontroller — based systems have a high ROM-to-RAM ratio. The control programme, perhaps relatively large, is stored in ROM, while RAM is used only for temporary storage.

3.4.1 Microprocessor-based and Microcontroller-based Digital Systems Design Using Top-Down Technique

The basic idea of microprocessor (microcontroller)-based digital system design is identical with that of conventional state machine design discussed before (Section 2.2.4). In this procedure, a digital system, characterized by a set of specifications, is first broken down into modules. These modules are functional modules that are as independent as possible of other modules. One of these modules is the controller. Each module except the controller is now decomposed into operational units, which often consist of MSI circuits. These operational units are circuits designed to perform specific functions. Binary counters, shift registers, and ALUs are examples of operational units.

Once the system is decomposed into operational units, the designer starts to study the design metrics and constraints (e.g. performance, cost, size and power consumption) that the final product has to fulfill in order to define the type of processor he is going to use for implementation. As mentioned before, the designer has to select between:

- Implementing the system as a single-purpose processor, i.e. pure hardware option. In this case conventional state machine techniques are going to be used to design the controller and all the operational units are going to be implemented as hardware, or
- Implementing the system using general purpose microprocessor. In this case he is implementing the system as a microprocessor-based system. The microprocessor takes over, in this case, the function of the controller and many other operational units. Or,
- Implementing the system as microcontroller-based system. The microcontroller takes over the function of the controller and, normally, more operational units.

The design metrics and constraints are the deciding factors at this point and not the functionality, since all the three options will implement the functions of the system.

The above discussions show that the procedure introduced in Section 2.4.4 to design digital system using top-down approach can be extended to cover the case of implementing the system as a microprocessor-based system or as a microcontroller-based system. The complete procedure to design a microprocessor-based or microcontroller-based digital system is as follows:

- Determine the system's goals from a thorough study of the system's specifications.

- Decompose the system into the functional modules required to implement these goals. One module will be the controlling state machine.
- Determine the next lower level of tasks that must be accomplished by each module.
- Decompose all modules except the controlling state machine into the operational units designed to accomplish these tasks. Often these subsections will correspond to MSI circuits.
- Decompose the operational units until the entire system is reduced to components that can be realized with basic circuits.
- Determine the number of states required for the controller.
- Determine whether the speed of one of the available, in the market, microprocessors (MPU)/microcontrollers (MCU) is sufficient to meet the specifications. Also determine whether the selected MPU/MCU satisfies the other design metrics and constraints.

If the answer of the above step is yes, determine the number of operational units or basic circuits that can be implemented by the MPU/MCU. In addition to being able to act as a system controller, the MPU/MCU can also perform many operations, including timing, counting, arithmetic, and logical operations. These capabilities can lead to the elimination of MSI circuits in the overall design.

Consider any enhancements of the features of the system that can result from the presence of a MPU/MCU.

In general, the top-down method is applicable to conventional, microprocessor-based and microcontroller-based digital system designs. Furthermore, this approach allows the three designs to be compared in order to choose the best one.

3.5 Practical Microcontrollers

This section introduces the reader to some practical microcontrollers showing how the discussions given in the above sections are used in practice. AVR ATmega8515 and Intel 8051 microcontrollers are considered. The features and the hardware architectures of each are given. The first one belongs to RISC structure and the second, Intel, belongs to the CISC structure. We start by giving the main features of the two architectures:

Definition:

CISC: A processor where each instruction can perform several low-level operations such as memory access, arithmetic operations or address

calculations. The term was coined in contrast to Reduced Instruction Set Computer (RISC).

Definition:

RISC, or Reduced Instruction Set Computer, is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions, rather than a more specialized set of instructions often found in other types of architectures.

3.5.1 AVR ATmega8515 Microcontroller

The current AVRs offer, in general, the following wide range of features, "*Taken from the data sheets of AVR*":

- RISC Core Running Many Single Cycle Instructions
- Multifunction, Bi-directional I/O Ports with Internal, Configurable Pull-up Resistors
- Multiple Internal Oscillators
- Internal, Self-Programmable Instruction Flash Memory up to 256 K
- In-System Programmable using ISP, JTAG, or High Voltage methods
- Optional Boot Code Section with Independent Lock Bits for Protection
- On chip debugging (OCD) support through JTAG or debugWIRE on most devices
- Internal Data EEPROM up to 4 KB
- Internal SRAM up to 8 K
- 8-Bit and 16-Bit Timers
- PWM output
- Input capture
- Analogue Comparators
- 10-Bit A/D Converters, with multiplex of up to 16 channels
- Dedicated I2C Compatible Two-Wire Interface (TWI)
- Synchronous/Asynchronous Serial Peripherals (UART/USART) (As used with RS-232, RS-485, and more)
- Serial Peripheral Interface Bus (SPI)
- Universal Serial Interface (USI) for Two or Three-Wire Synchronous Data Transfer
- Brownout Detection
- Watchdog Timer (WDT)
- Multiple Power-Saving Sleep Modes

- Lighting and motor control (PWM Specific) Controller models
- CAN Controller Support
- USB Controller Support
- Proper High-speed hardware & Hub controller with embedded AVR.
- Also freely available low-speed (HID) software emulation
- Ethernet Controller Support
- LCD Controller Support
- Low-voltage Devices Operating Down to 1.8 v
- picoPower Devices

Programme Execution

Atmel's AVRs have a single level pipeline design. The next machine instruction is fetched as the current one is executing. Most instructions take just one or two clock cycles, making AVRs relatively fast among the eight-bit microcontrollers. The AVR families of processors were designed for the efficient execution of compiled C code.

The AVR instruction set is more orthogonal than most eight-bit microcontrollers, however, it is not completely regular:

- Pointer registers X, Y, and Z have addressing capabilities that are different from each other.
- Register locations R0 to R15 have different addressing capabilities than register locations R16 to R31.
- I/O ports 0 to 31 have different addressing capabilities than I/O ports 32 to 63.
- CLR affects flags, while SER does not, even though they are complementary instructions.
  CLR set all bits to zero and SER sets them to one. (Note though, that neither CLR nor SER are native instructions. Instead CLR is syntactic sugar for [produces the same machine code as] EOR R,R while SER is syntactic sugar for LDI R,$FF. Math operations such as EOR modify flags while moves/loads/stores/branches such as LDI do not.)

Speed: The AVR line can normally support clock speeds from 0-16MHz, with some devices reaching 20MHz. Lower powered operation usually requires a reduced clock speed. All AVRs feature an on-chip oscillator, removing the need for external clocks or resonator circuitry. Because many operations on the AVR are single cycle, the AVR can achieve up to 1MIPS per MHz.

AVR Groups: AVRs are generally divided into three broad groups:

- *tinyAVRs*
  - *1–8 kB programme memory*
  - *8–20 pin package*
  - *Limited peripheral set*
- *megaAVRs*
  - *4–256 kB programme memory*
  - *28–100 pin package*
  - *Extended instruction set (Multiply instructions and instructions for handling larger programme memories)*
  - *Extensive peripheral set*
- *Application specific AVRs*
  - *megaAVRs with special features not found on the other members of the AVR family, such as LCD controller, USB controller, advanced PWM etc."*

3.5.1.1 ATmega8515/AT90S851 an Overview

The ATmega8515 is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture. By executing powerful instructions in a single clock cycle, the ATmega8515 achieves throughputs approaching 1 MIPS per MHz allowing the system designer to optimize power consumption versus processing speed.

Figure 3.14 shows the AVR RISC microcontroller architecture. The figure reflects the fact mentioned before that the AVR family as any other microcontroller follows the general architecture of a microcomputer. It is organized into a CPU, a memory, and an I/O section. The CPU is hidden in the block diagram, but the memory and I/O sections are shown and need to be understood by the application designer.

The ATmega8515 provides the following sections:

- 8 K bytes of In-System Programmable Flash with Read-While-Write capabilities,
- 512 bytes EEPROM,
- 512 bytes SRAM,
- An External memory interface,
- 35 general purpose I/O lines (ports A, B, C, D, and E),
- Register file of 32 general purpose working registers,

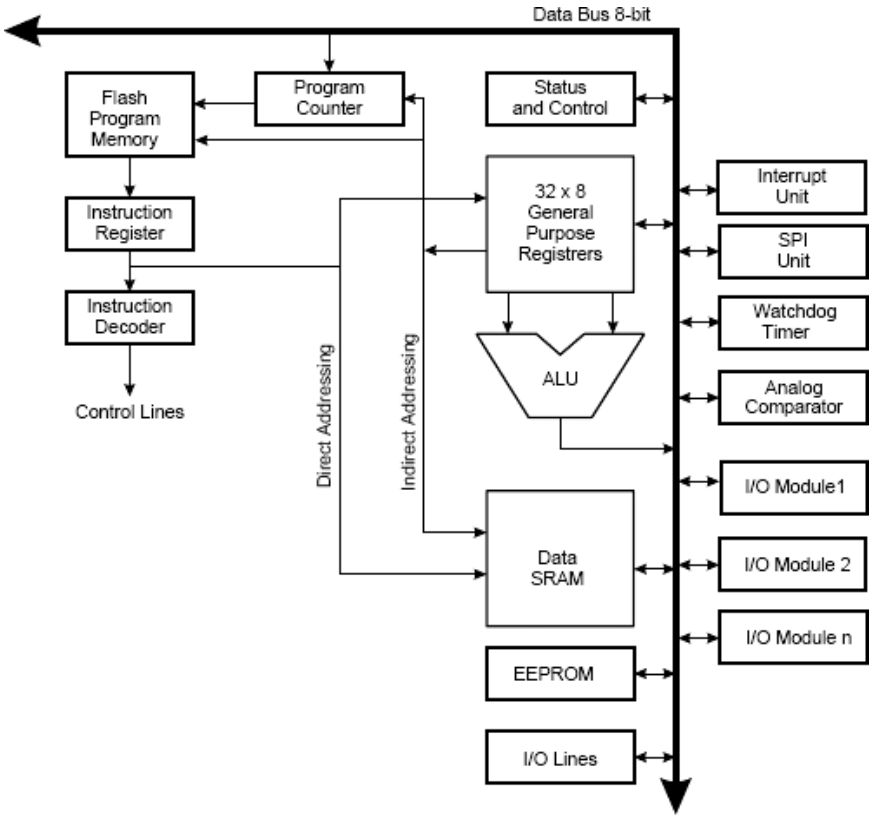


Figure 3.14 AVR RISC microcontrollers architecture.

- two Timer/Counters with compare modes,
- Internal and External interrupts,
- a Serial Programmable USART,
- a programmable Watchdog Timer with internal Oscillator,
- a SPI serial port, and

ATmega8515 has three software selectable power saving modes:

- ***Idle mode:*** The Idle mode stops the CPU while allowing the SRAM, Timer/Counters, SPI port, and Interrupt system to continue functioning.
- ***Power-down mode:*** This mode saves the Register contents but freezes the Oscillator, disabling all other chip functions until the next interrupt or hardware reset.

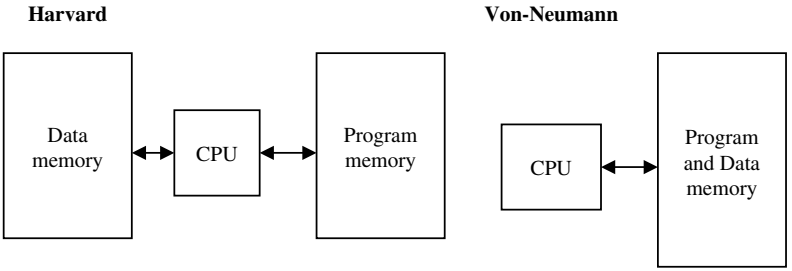


Figure 3.15 Harvard and Von Neumann architectures.

- ***Standby mode:*** In this mode, the crystal/resonator Oscillator is running while the rest of the device is sleeping. This allows very fast start-up combined with low-power consumption."

In order to maximize performance and parallelism, the AVR uses Harvard architecture — with separate memories and buses for programme and data (See Figure 3.15 for Harvard and Van Neumann architecture). Instructions in the Programme memory are executed with a single level pipelining. While one instruction is being executed, the next instruction is pre-fetched from the Programme memory. This concept enables instructions to be executed in every clock cycle.

The Programme memory is In-System re programmable Flash memory. The fast-access Register File contains 32 × 8-bit general purpose working registers with a single clock cycle access time. This allows single-cycle Arithmetic Logic Unit (ALU) operation. In a typical ALU operation, two operands are output from the Register File, the operation is executed, and the result is stored back in the Register File — in one clock cycle."

Six of the 32 registers can be used as three 16-bit indirect address register pointers for Data Space addressing — enabling efficient address calculations. One of these address pointers can also be used as an address pointer for look up tables in Flash Programme memory. These added function registers are the 16-bit X-, Y-, and Z-register, described later in this section.

3.5.2 Intel 8051 Microcontroller

Intel 8051 first introduced in 1980, the 8051 is the most well-established and widely used microcontroller. The original design of 8051 was a mid-range device with multiple parallel ports, timers and interrupts and a serial port. The 8051 can be used as conventional processor, as well as a microcontroller. It can

access external memory using Port 0 and Port 2, which act as multiplexed data and address lines. Some of Port 1 and Port 3 pins also have a dual purpose, providing connections to timers, serial port and interrupts. The 8031 was a version of 8051 without internal ROM, the application programme being stored in a separate EPROM.

The 8051 architecture has a very nice Boolean processing unit that allows the user to address certain ranges of memory as bits as well as bytes. It is a unique and often times useful feature.

Another great plus for the 8051 family is the large number of chips available. Many different manufacturers have 8051 based parts, and it isn't too difficult to find information on the web. There are chips with almost every conceivable set of peripherals onboard. Some are readily available, some are not. The user can get, for example, chips such as the 8052BASIC chip, which have an onboard BASIC interpreter. 8051 has the biggest variety of micros on earth.

The Intel 8051 is packaged in a 40-pin chip. It has 17 address lines, including the PSEN line (programme store enable), an 8-bit data bus, and an 8-bit arithmetic unit.

In addition to the 8-bit data bus, there are also three other 8-bit ports for data exchange with external peripherals. It has Harvard architecture in which the programme memory address space is separate from the data memory space. Figure 3.16 shows the block diagram of the 8051. Port 0 functions as the normal I/O bus, with ports 1 to 3 available for data transfers to other components of the system in I/O-intensive applications.

The lower 4 kbytes of programme ROM are implemented on the processor chip with the lower 128 bytes of data RAM. In addition, 21 special function registers in data memory space also appear on the chip.

Although we shall not explore timer chips or chips that convert between parallel and serial words until a later chapter, we shall mention here that these functions are also implemented on the processor chip. Normally, a separate timer chip or parallel / serial converter chip (universal asynchronous receiver/transmitter or UART) must be used. The two timers on the 8051 can be programmed to count the number of machine cycles beginning at some specified point in a programme or to count external events represented by the assertion of either of two input pins. When counting machine cycles, accurate time periods can be established, as these cycles are referenced to the master clock.

The 21 special function registers (SFRs) relate to several different functions of the 8051. Several registers are used in connection with the

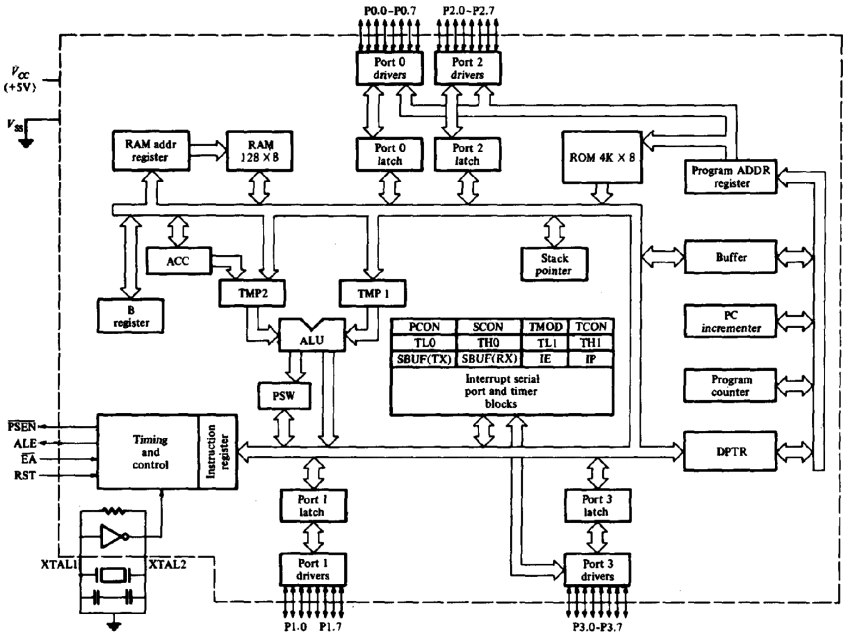


Figure 3.16 Block diagram of the 8051.

timer/counters and the serial data port. An accumulator and a B register are available for data manipulation. A stack pointer, a data pointer, and a programme status word register are also included in this group of special function registers. Interrupt enable, interrupt disable, and priority information are also controlled from registers in this group. Another important set of registers is associated with the ports, which contain information relating to the definition of function and the direction of the port pins.

Many instrumentation applications use the 8051 as a stand-alone chip. With sufficient on-chip programme memory, minimum data memory systems can be implemented without peripheral chips. The internal timer and serial port allow this device to solve a wide variety of instrumentation problems.

The accumulator, or A register, is used for the microcontroller's normal arithmetic and logic operations. Only unsigned, binary, integer arithmetic is performed by the arithmetic unit. In two-operand operations, such as add, add-with-carry, and subtract-with-borrow, the A register holds the first operand and receives the result of the operation. The multiply operation causes the 1-byte contents of register A to be multiplied by the 1-byte contents of register B. The result is a 16-bit number with the most significant byte in register B and the

least significant byte in register A. The divide operation divides the contents of register A by the contents of register B. The integer quotient is contained by register A, with a 1-byte remainder contained in register B.

3.5.2.1 8051 memory space

There are three memory spaces associated with the 8051: the 64-kbyte programme memory space, the 64-kbyte external data memory space, and the internal data memory space. The lower 4 kbytes of programme memory space are used by the on-chip or internal ROM. The internal RAM space uses 128 addresses for data RAM and 21 addresses for the special function registers (SFRs).

The programme counter (PC) is a 16-bit register that can address 64 kbytes. An external address pin, /EA, is provided to direct programme memory accesses to internal or external memory. If/EA is held high (not asserted), the internal ROM is accessed for PC addresses ranging from 0000(H) through 0FFF(H). External programme memory is accessed for PC addresses ranging from 1000(H) through FFFF(H). If it is appropriate to place all 64 kbytes of memory space external to the chip, the/EA pin will be held low. For this condition, all programme memory accesses are external. Figure 3.17 shows a memory map of programme and data memory. It is possible to have a total programme memory of 68 kbytes by placing external memory at locations

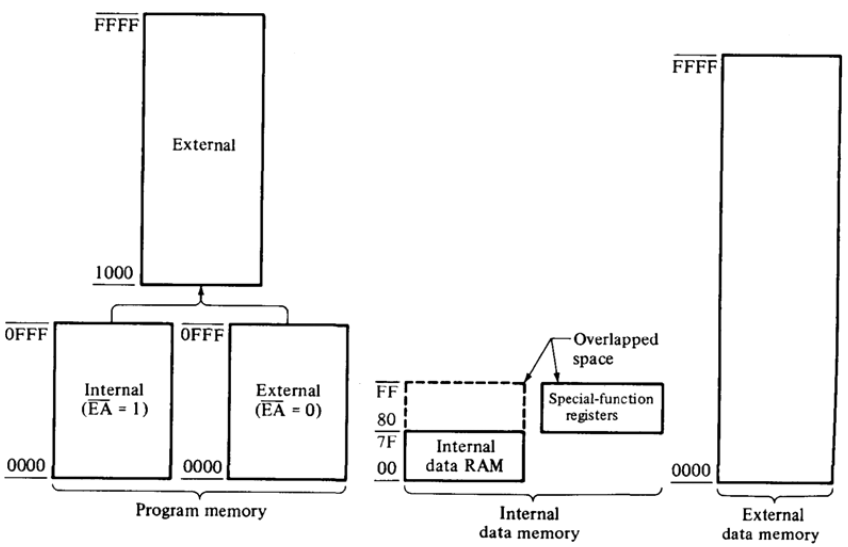


Figure 3.17 Memory map of 8051.

0000(H) to FFFF(H) and then using/EA to distinguish between internal and external accesses for addresses from 0000(H) to 0FFF(H). This requires additional information about the time that the EA must be asserted.

The programme memory space is separated from the data memory space by means of the/PSEN line. If the programme memory fetch is to come from internal memory, the address will fall below 1000(H), and the/PSEN will not be asserted. Programme fetches having an address of 1000(H) or greater will automatically lead to the assertion of the/PSEN. This signal must then be used to activate programme memory in the same manner in which a read strobe activates a memory. Thus, the/PSEN can be considered to be part of the address. Along with the 16 address bits, this gives 17 total address lines to access $2^{17} = 128$ kbytes of memory.

When data accesses take place, the/PSEN is not asserted, but either the/RD or /WR strobe is asserted at the proper time to access data memory. Neither of these strobes is asserted when programme memory is accessed.

It is possible to place both programme memory and data memory in the same physical address space. In order to do this, the /RD and /PSEN lines are ORed to enable the memory for programme or data fetches.

Certain locations in programme memory are reserved for specific functions. Following reset, the CPU begins execution at location 0000(H), and thus locations 0000(H) through 0002(H) are used to send the programme to the proper starting point. Locations 0003(H) to 002A(H) are reserved for purposes relating to the various interrupt service routines. This subject will be discussed later in this section.

The programme memory uses 16-bit addresses for fetching internal instructions and adds /PSEN for external programme fetches. The external data memory can use either 8-bit or 16-bit addressing, whereas the internal data memory uses only 8-bit addressing.

Although there is a 256-byte internal data memory address space, not all of it can be used. Locations 0 through 127 contain data RAM. The special function registers (SFRs) occupy 21 bytes of space between addresses 128 and 255. The maps of internal data memory and SFRs are shown in Figure 3.18. Any byte of internal data RAM may be accessed by direct addressing. In this type of addressing, an 8-bit number between 0 and 127 makes up 1 byte of the instruction word.

In order to decrease the number of bits required for addressing internal RAM locations, the lower 32 bytes are divided into 4 banks of 8 bytes. The 8 bytes are numbered register 0 (R0) through register 7 (R7). Two bits of the programme status word (PSW) select the active bank. Any register within the

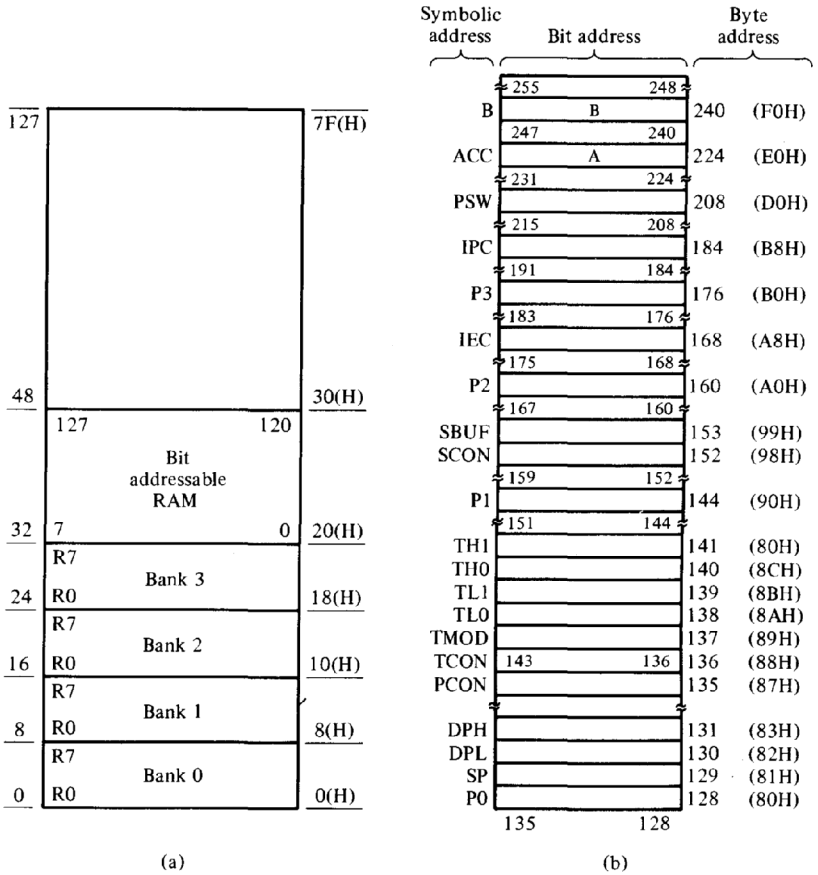


Figure 3.18 Memory maps: (a) internal data RAM registers and (b) special-function registers.

bank can then be addressed with only 3 bits. This approach minimizes the number of bits to the extent, in some cases, that an 8-bit instruction word can carry both the op code and the register address. A single instruction can be used to set the proper bits of the PSW, thereby selecting the desired bank. All register references now apply to the selected bank, until a new bank is selected by modifying the PSW.

The stack can be located anywhere within the internal data RAM and is thus limited in depth to the number of consecutive unused locations of internal data RAM.

In addition to allowing any of the 128 byte locations of RAM to be accessed, the internal data RAM allows certain locations to be addressed as bits.

The 16 byte locations from 32 through 47 contain a total of 128 bits. These bits are numbered from 0, starting with the LSB of location 32 and proceeding to number 127, which is the MSB of location 47. Whenever a bit instruction is used, the bits direct address is specified in an 8-bit code.

The 20 SFRs occupy 21 locations of the remaining 128 address spaces. Figure 3.19(b) shows the locations of the SFRs. If any other locations between 128 and 255 — except those of the SFRs — are read from, unknown data will be accessed. The SFRs can be accessed directly, but in many cases, the instruction word implies a specific register. For example, add instructions imply the use of the accumulator (register A), and the multiply instruction implies the use of registers A and B.

Every SFR located at an address that is a multiple of 8 is bit addressable, which means that bytes 128, 136, 144, 152, 160, 168, 176, 184, 208, 224, and 240 are bit addressable. These bits are numbered from 128 to 255, beginning with the LSB of location 128 and proceeding through the MSB of location 240.

The SFRs are given in the list given in Figure 3.20.

The registers that are both byte- and bit- addressable are A, B, PSW, P0, P1, P2, P3, IP, IE, TCON, and SCON.

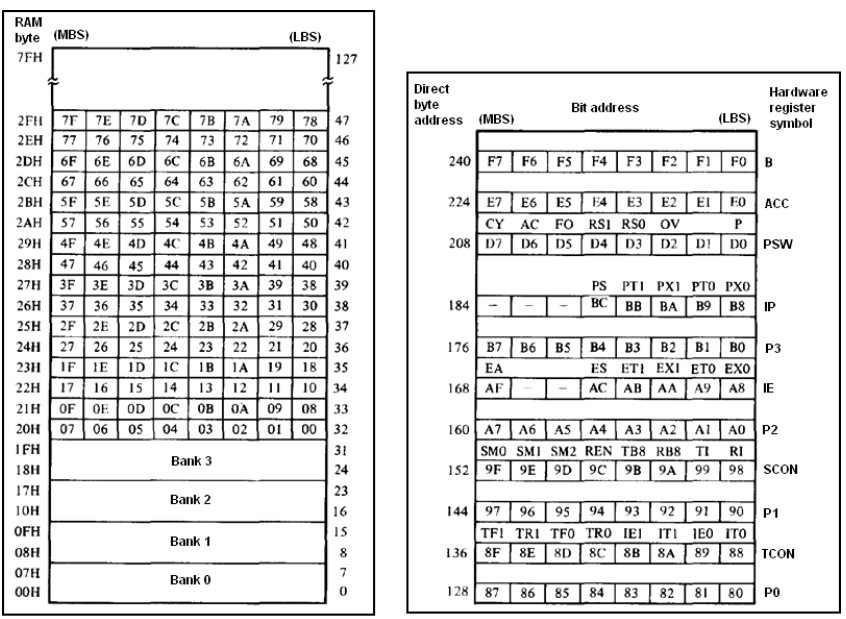


Figure 3.19 Bit addresses: (a) internal data RAM and (b) SFR group.

The memory map of Figure 3.18 shows an external data memory space consisting of 64 kbytes. For accesses to addresses under 256 of external RAM, the RAM address can be contained in R0 or R1 of the internal RAM bank. For addresses requiring 16 bits, this value is placed in the DPTR register. The MOVX instruction will either drive the address bus with the 8-bit address contained in R0 or R1 or the 16-bit address contained in DPTR, depending on the instruction word.

3.5.2.2 Addressing modes

There are five methods of addressing source operands in the 8051: register addressing, direct addressing, register-indirect addressing, immediate addressing, and base plus index register-indirect addressing. Destination operands can be expressed by register addressing, direct addressing, and register-indirect addressing. The addressing modes will be discussed in Chapter 4.

3.6 Summary of the Chapter

In this chapter we discussed two main topics; microprocessors and microcontrollers. The internal structure of each is given. The functions of the blocks forming each internal structure are given. A comparison between the two is given. The comparison showed that the microcontroller is a complete

A or ACC	Accumulator
B	B register
PSw	Programme status word
SP	Stack pointer
DPTR	Data pointer (2 bytes)
P0	Port 0
P1	Port 1
P2	Port 2
P3	Port 3
IP	Interrupt priority
IE	Interrupt enable
TMOD	Timer/counter mode
TCON	Timer/counter control
TH0	Timer/counter 0 (high byte)
TL0	Timer/counter 0 (low byte)
TH 1	Timer/counter 1 (high byte)
TL 1	Timer/counter 1 (low byte)
SCON	Serial control
SBUF	Serial data buffer
PCON	Power control

Figure 3.20 The SRF registers.

microcomputer that is optimized to suit the control oriented applications. Two practical microcontrollers are introduced; AVR and Intel. The capabilities and the main features of each of them are presented.

3.7 Review Questions

3.1 What is the role of processor reset and system reset?
3.2 Explain the need of watchdog timer and reset after the watched time.
3.3 What is the role of RAM in an embedded system?
3.4 Classify the embedded systems into small-scale, medium-scale and sophisticated systems. Now, reclassify these embedded systems with and without real-time (response time constrained) systems and give 10 examples of each.

4

Instructions And Instruction Set

THINGS TO LOOK FOR…

- Meaning of instruction and instruction set
- Instruction cycle
- Register Transfer Language (RTL)
- Addressing modes
- Stack machines
- AVR Instruction set

4.1 Introduction

Assembly language of any processor compromises of a set of (single-line) instructions. Each instruction represents a single operation that can be performed by the processor (can be microprocessor, microcontroller, or special purpose). In a broader sense, an “instruction" may be any representation of an element of an executable programme, such as a bytecode. The collection of the instructions that can be performed by a given processor is called “instruction set”. Assembly language programme is a sequence of instructions selected to achieve a required task. The assembly programme must be converted to object code in the form of machine code (machine language) by means of an assembler programme. The system stores (or the programmer download) the machine code in the memory of the system for execution.

Most instruction sets includes operations related to:

- **Data Movement (Data Transfer)**
  - Set a register (a temporary “scratchpad" location in the CPU itself) to a fixed constant value.
  - Move data from a memory location to a register, or vice versa. This is done to obtain the data to perform a computation on it later, or to store the result of a computation.
  - Read and write data from hardware devices.

- **Computing- Arithmetic and Logical**

  – Add, subtract, multiply, or divide the values of two registers, placing the result in a register.
  – Perform bitwise operations, taking the conjunction/disjunction (and/or) of corresponding bits in a pair of registers, or the negation (not) of each bit in a register.
  – Compare two values in registers (for example, to see if one is less, or if they are equal).

- **Transfer of control**

  This group of instructions affects the programme flow, i.e., changing the programme execution sequence. Three types of these operations are identifiable:
  – Jump to another location in the programme and execute instructions there.
  – Jump to another location if a certain condition holds (Conditional branches).
  – Jump to another location, but save the location of the next instruction as a point to return to (a call and Return).

Some processors include "complex" instructions in their instruction set. A single "complex" instruction does something that may take many instructions on other computers. Such instructions are typified by instructions that take multiple steps, control multiple functional units, or otherwise appear on a larger scale than the bulk of simple instructions implemented by the given processor. Some examples of "complex" instructions include:

- **Floating Point:**

This operation is optimized for floating points handling. This group involves instructions that perform floating point arithmetic. This group also includes complex instructions as sine, cosine, square root, etc.

- **String:**

These are special instructions optimized for handling ASCII character strings.

- **Conversion Instruction:**

These instructions change the format or operate on the format of data.

- **Input Output:**

This operation issues command to I/O modules. If memory-mapped I/O, it determines the memory mapped address.

- **Graphics, multimedia and DSP:**

These are special instructions optimized to speed up the processing of the data in the field of graphics, multimedia and DSP. For example, multimedia instructions are used to accelerate the processing of different forms of multimedia data. These instructions implement the ISA concept of *subword parallelism*, also called *packed parallelism* or *microSIMD parallelism*.

- **Saving many registers on the stack at once.**
- **Moving large blocks of memory.**
- **Performing an atomic test-and-set instruction.**

Table 4.1 lists of common types of instruction in each category. Depending on the nature of a particular instruction, it may assemble to between one to three (or more) bytes of machine code.

4.2 Instruction Format

Figure 4.1 represents the traditional architecture of an instruction. It includes, usually, two fields: an *op-code* field and an operand *specifiers field* (also called "*address field*"). The **op-code** specifies the operation to be performed, such as "add contents of memory to register". The second field contains zero or more **operand** specifiers. Operand specifiers are used to specify registers, memory locations, or literal data. The operand specifiers may have addressing modes determining their meaning or may be in fixed fields. In many cases, the number of bits in op-code field is $\geq \log_2$ (number of operations). For example an op-code field of 8 bits width can uniquely specify up to 256 operations or operation/data types, depending on how the operations are bound to the data types.

Concerning the address field, the number of addresses and their length are functions of how operands are addressed in either memory and/or registers (*addressing modes*). The instructions can be classified, as discussed latter, based on the number of operands (number of addresses) included in this field.

As an example of the above, the following instruction moves the data stored at address 26 h to address 4 Bh:

```
MOV 4Bh, 26h
```

Table 4.1 Common instruction set operations.

Type	Operation	Description
Data Transfer	Move (transfer)	Transfer word or block from source to destination.
	Store	Transfer word from processor to memory
	Load (fetch)	Transfer word from memory to processor
	Exchange	Swap contents of source and destination
	Clear (reset)	Transfer word of 0s to destination
	Set	Transfer word of 1s to destination
	Push	Transfer word from source to top of stack
	Pop	Transfer word from top of stack to destination
Arithmetic	Add	Compute sum of two operands
	Subtract	Compute difference of two operands
	Multiply	Compute product of two operands
	Divide	Compute quotient of two operands
	Absolute	Replace operand by its absolute value
	Negate	Change sign of operand
	Increment	Add 1 to operand
Arithmetic	Decrement	Subtract 1 from operand
Logical	AND, OR, NOT, XOR	Perform the specified logical operation
	Test	Test specified condition; set flag(s) based on outcome.
	Compare	Make logical or arithmetic comparison of two or more operands; set flag(s) based on outcome.
	Set control variables	Class of instructions to set controls for protection purposes, interrupt handling, timer control, etc.
	Shift	Left (right) shift operand, introducing constants at end
	Rotate	Left (right) shift operand, with wraparound end
Transfer of Control	Jump (branch)	Unconditional transfer; load PC with specified address
	Jump conditional	Test specified condition; either load PC with specified address or do nothing, based on condition.
	Jump to subroutine	Place current programme control information in known location; jump to specified address.
	Return	Replace contents of PC and other register from known location
	Execute	Fetch operand from specified location and execute as instruction; do not modify PC.
	Skip	Increment PC to skip next instruction.
	Skip conditional	Test specified condition; either skip or do nothing based on condition.
	Halt	Stop programme execution
	Wait (hold)	Stop programme execution; test specified condition repeatedly; resume execution when condition is satisfied.
	No operation	No operation is performed, but programme execution is continued
Input/Output	Input (read)	Transfer data from specified I/O port to destination (e.g., memory or processor register)
	Output (write)	Transfer data from specified source to I/O port or device
	Start I/O	Transfer instructions to I/O processor to initiate I/O operation.
	Test I/O	Transfer status information from I/O system to specified destination

(*Continued.*)

Table 4.1 (*Continued.*)

Type	Operation	Description
Conversion	Translate	Translate values in a section of memory based on a table of correspondences.
	Convert	Convert the contents of a word from one form to another (e.g., packed decimal to binary)
Interrupt and Exceptional	Interrupt	Events that are requested by the programme. Operating system calls and programme trace traps are examples.
	Traps	Events from inside the processor. Arithmetic exceptions and page faults are examples.
	Synchronization A synchronization	Events usually from outside the processor. I/O requests and time out signals are examples
Graphics, Multimedia, DSP	Parallel Arithmetic	Parallel subword operations (4 16 bit add). For example: Parallel multiply of four signed 16-bit words,
	Parallel comparison	Parallel compare for equality
	Pack and Unpack words	Pack words into bytes, or doublewords into words, Parallel unpack (interleave merge) high-order bytes, words, or doublewords.
	Etc	Pixel and vertex operations, compression/decomposition operations

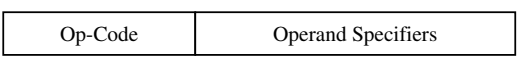


Figure 4.1 Instruction format.

In this instruction MOV represents the op-code part and indicates move operation, i.e. it is used to move data from one location (source) to another location (destination). The rest of the instruction represents the specifier part; in this case it represents addresses of two operands one stored at address 26 h (source address) and the other at address 4 Bh (destination address). We note here that the op-code is separated from the first operand by a space, and the operands are separated by commas.

Operand: In assembly language, an operand is a value (an argument) on which the instruction, named by mnemonic, operates. The operand may be a processor register, a memory address, a literal constant, or a label. A simple example (in the ×86 architecture) is

```
MOV  DS, AX
```

where the value in register operand 'AX' is to be moved into register 'DS'. Depending on the instruction, there may be zero, one, two, or more operands.

4.2.1 Expressing Numbers

Normally the operand in immediate addressing represents a number. In general the number can be represented as decimal, binary, octal, or hexadecimal. A suffix is used to indicate the base.

- **Binary representation:** This is indicated by "%" or "B" (also "b") suffix either following or preceding the number.
- **Hexadecimal representation:** This is indicated by "$" or "H" also "h".
- **Octal representation:** The suffix O or Q is used.
- **Decimal:** Has no letter following or preceding the number.

When no suffix is used, the default decimal base is assumed.

The symbols $ (H, h) and % (B, b) are necessary, because most computers cannot deal with subscripts and therefore the conventional way of indicating the base by a subscript is impossible. For example, the three instructions:

```
MOV     A,  00011001b,
MOV     A,  19h, and
MOV     A,  25
```

have identical effects because $25_{10} = 19_{16} = 00011001_2$.

The number representation letters ($, %, B, H, h, b) are not part of the instruction. It is a message to the assembler telling it to select that code for "move data" that uses the immediate addressing mode and this data are represented in using certain radix.

Representing Signed Numbers

In many microcontrollers, immediate addressing mode allows specification of an 8- or 16-bit constant value. This value can be specified as an unsigned or signed numbers. The assembler sign-extends immediate values to fill all 8 or 16 bits in the instruction. For example in case of using the instruction `LDI r16,37`, the assembler sign extends 0100101b (+37) to provide the 8-bit value 00100101b. If the constant value specified in the instruction were −37, then the assembler would sign-extend the two's complement representation of −37, 1011011b, to 11011011b to provide 8-bits.

4.2.2 Basic Instruction Cycle; Execution Path of an Instruction

The basic function performed by any computing system is the execution of a programme, which consists of a set of instructions stored in memory. The processor does the actual work by executing instructions specified in the programme. The sequence of operations, which each instruction must go

through in the processor, is called the ***execution path*** of the instruction or the ***instruction cycle***. The number of stages in the execution path is an architectural feature which can be changed according to the intended exploitation of instruction level parallelism. In its simplest form, the execution path consists of two phases:

- ***Fetch*:** In this phase the processor reads (fetch) instructions from memory, one at a time, and copied it into the control unit.
- ***Execute*:** This is the phase in which the system *executes* the instruction.

Normally, the instruction cycle is called fetch-encode-execute. In our simple form, the encoding is part of the execute cycle. Programme execution consists of repeating the process of instruction fetch and instruction execution. The instruction fetch as well as the instruction execution may involve several operations and depends on the nature of the instruction (e.g. depends on the addressing mode, operation code etc.).

The processing required for a single instruction is called an instruction cycle. Using the simplified two-step description given previously, the instruction cycle is shown in Figure 4.2. The two steps are referred to as the fetch cycle and the execute cycle. Programme execution halts only if the machine is turned off, when some sort of error occurs or when a programme instruction, which halts the computer, is encountered.

At the beginning of each instruction cycle, the processor fetches an instruction from memory. The programme counter (PC) normally holds the address of the instruction to be fetched next. Unless told otherwise, the processor always increments the PC after each instruction fetch so that it will fetch the next instruction in sequence. The fetched instruction is loaded into a register in the processor known as the instruction register (IR). The operation code part of the instruction specifies the action the processor is to take. The processor interprets the instruction (by using instruction decoder) and performs the required action. The action, as mentioned before, can involve a combination of data transfer,

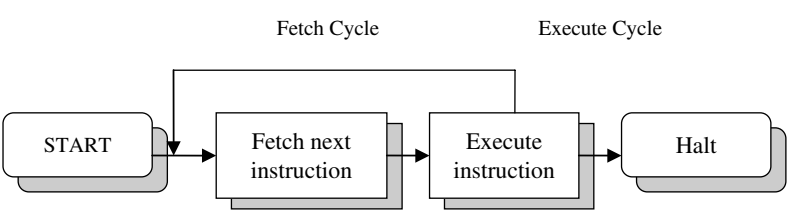


Figure 4.2 Basic instruction cycle.

data transform or some form of control. Control actions can be explained simply as follows: an instruction may specify that the sequence of execution be altered (e.g. by branching). For example, the processor may fetch an instruction from location 150, which specifies that the next instruction be from location 200. The processor will remember this fact by setting the PC to 200. On the next fetch cycle, the instruction will thus be fetched from location 202 rather than 152 (assuming here the instruction has a width of two words).

The instruction fetch as well as the instruction execution may involve several operations and depends on the nature of the instruction. The various operations that must be carried out to fetch and execute each instruction, is best described using what is called "*Register Transfer Language* (RTL)" or sometimes "*Register Transfer Notation* (RTN)". RTL and its use to describe the instruction cycle is the subject of Section 4.3.

4.2.3 Clock Cycle and Instruction Cycle

It is clear for the reader now the difference between clock and instruction cycle. The computer (or the microcontroller) clock produces "clock cycles", which can take any value from few hundreds of kilo hertz to hundreds of gaga hertz according to the field of application. The hardware then uses a number of these cycles (e.g. 12 clock cycles) to carry out one machine cycle operation. An instruction may use between one and four (sometimes more) of these machine cycles.

4.2.4 Labels

Many of the jump and call instructions take an operand which is actually a number representing a destination address. However, in most programming situations it is more convenient to replace this by a label, which is a name given to an address. During the assembly process the appropriate number is calculated and put into the object code. The use of labels enables changes to be made to the programme without recalculating the numerical values.

4.3 Describing the Instruction Cycle: Use of Register Transfer Language (RTL)

4.3.1 Register Transfer Language (RTL)

There are three basic operations that take place on data in any processing system:

- Data storage,

- Data transfer, and
- Data transform (manipulation).

All of these operations involve registers. Data is stored in registers (microprocessor registers, memory registers, and I/O registers), transferred between registers, and transformed in operational registers. The data transferred or transformed may be interpreted as data to be processed, instructions, or address values. Thus, the structure and operation of a system can be visualized in terms of its registers and the paths between them. The register view of a system is not only useful for developing a conceptual understanding of the system, but also for performing the system's hardware and software analysis and design. Figure 4.3 shows, as an example, the register view of a self-contained computer system, called RISC8v1 MCU. The various blocks of the system are controlled

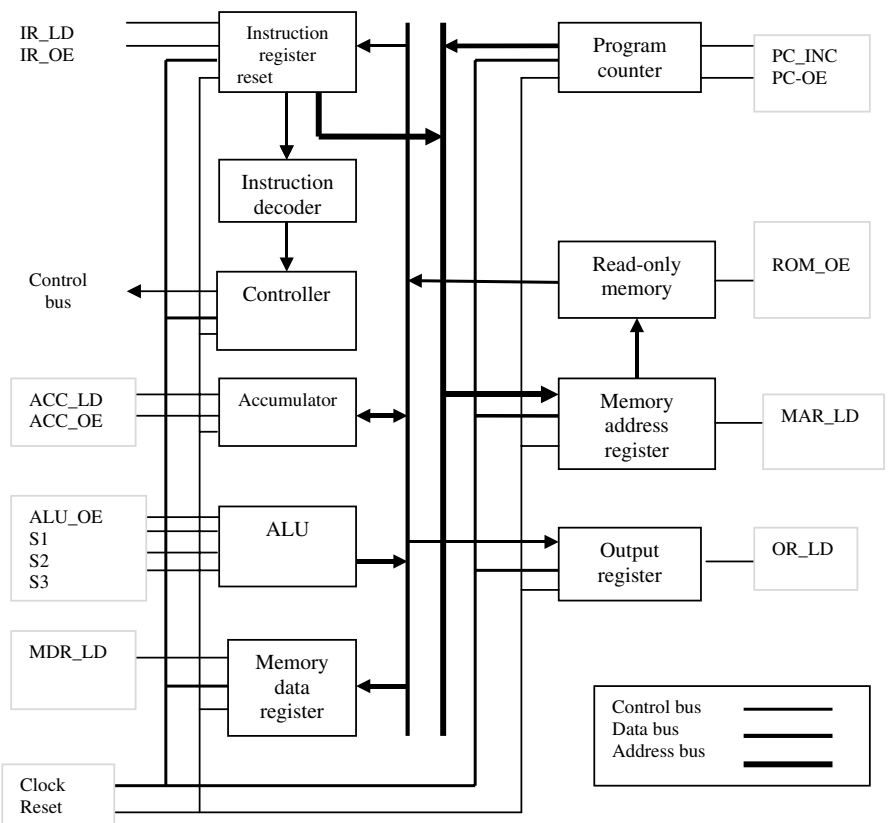


Figure 4.3 RTL view of RISC8v1.

by Output Enable (OE) and Load (LD) signals generated by the control unit (the controller).

A processor's instruction set specifies what transfers between registers (working registers, memory registers, and I/O registers) and what transformations on data in the processor's registers are possible with the processor's architecture.

The action of an instruction can be described by identifying the data transfers needed to do the requested work. The specification of this work is done by a ***register transfer language*** (RTL) (also called ***register transfer notation*** RTN). RTL descriptions specify the order of register transfers and arithmetic action required to carry out the work of an instruction. This information can then be utilized by the designer of the control system to identify the order of activation of control lines to actually cause the desired transfers. This goes out with the strategy of splitting any processor into two sections: ***datapath*** and ***controller***. The RTL specifications can be used to design the datapath section which includes identifying the data storage spaces (registers and memory), the arithmetic/logic capabilities and the interconnections required to guarantee the flow of the data between the different parts. The design of the control section is then carried out so that the timing requirements of the system are met.

A register transfer language can become as simple or as complex as needed to specify the transfers required in the system. Since we will be using in this chapter an RTL to describe the actions of the instructions, we will describe here a few of the basic operations of RTL.

The basic operation is the transfer of the contents of one register, to another:

$$[Rd] \leftarrow [Rs]$$

This specifies that the contents of register Rs (source register) are transferred to register Rd (destination register). For example, [MAR] ← [PC] means that the contents of the programme counter (PC) are transferred to the memory address register (MAR). If the data paths of the system are rich enough to allow multiple operations in the same time period, these can be represented by specifically linking the transfer together:

$$\begin{array}{l} [PC] \leftarrow [PC] + 1 \\ [IR] \leftarrow [MAR] \end{array}$$

It identifies that in the same time period the value of the PC is incremented and the contents of the MBR are transferred to the IR. Normally, all of the information is involved in the transfer. However, if a subset of the information

is to be transferred, then the specific bits are identified by the use of pointed brackets:

$$\texttt{ALU} \leftarrow \texttt{IR}\ \langle 3:0 \rangle$$

specifies that bits 3 to 0 of the instruction register are directed to the ALU. As another example, MAR ← IR <*adr*> means the address part of the instruction is directed to the MAR.

Similarly, locations of memory or a set of registers are specified with square brackets:

$$\texttt{M[(MAR)]} \leftarrow \texttt{REG[Rs]}$$

indicating that the contents of register Rs in a general register set (REG) is transferred to location in memory identified by the memory address register. The same action may be described as:

$$\texttt{M[(MAR)]} \leftarrow \texttt{[Rs]}$$

For operations that are conditional in nature, we include normally an "If" facility patterned after the C language if construct:

```
If (carry == 1)   [PC] ← [PC] + a
           Else   [PC] ← [PC] + 1
```

Identifying that if the carry is equal to 1, the programme counter is adjusted by a factor "a"; otherwise the programme counter is incremented.

Table 4.2 gives summary of the RTL notations that are used in explaining the instruction cycle.

4.3.2 Use of RTL to Describe the Instruction Cycle

Figure 4.2 shows the basic instruction cycle. The RTL can be used to describe the different steps included in the fetch cycle as well as in the execute cycle. For

Table 4.2 RTN summary.

Notation	Meaning
R	Register R
[R]	The contents of register R
M(01101101)	Memory location 01101101
[M(01101101)]	The contents of memory location 01101101
<ai ai+1>	Bits number i and (i+1) of A
IR <Op>	The contents of the op code part of IR
IR <adr>	The contents of the address part of IR
[R] ← [R] + k	The contents of register R is read, k is added, and the result is copied into R.

```
Fetch: The following register transfers get the instruction from the memory.
    1.     [MAR] ← [PC]          ; Transfer the Instruction location
                                 ; to MAR.
    2.     [MBR] ← M[(MAR)]      ; Retrieve the instruction in MBR
    3.     [IR] ← [MBR]          ; and then put it in IR.
    4.     [PC] ← [PC] + Inc     ; Increment the PC to
                                 ; prepare for the next instruction.
```

Decode: The decode process identifies the instruction.

Execute**:** and the execute portion performs the needed work.

```
    5.     [MAR] ← IR < adr >    ; Transfer the address of operand
                                 ; to MAR.
    6.     [MBR] ← M[(MAR)]      ; This is the value to add to
                                 ; accumulator ACC.
    7.      ACC ← ACC+[MBR]      ; Do the actual work of the
                                 ; instruction.
```

Figure 4.4 RTL implementation of ADD X.

example, consider the add instruction ADD X. Addition needs two operands. As we shall see later, the instruction ADD X is a single-address instruction. It gives the address of one of the two operands in the memory and considers implicitly that the second operand is the contents of the accumulator. Accordingly, the instruction ADD X means, add the contents of memory location X to the value stored currently in the accumulator and store the result in the accumulator. We will further assume that the address X is adequately contained in the instruction itself (direct address mode), thus no additional information beyond the instruction will be required. With those assumptions, a set of data transfers that will perform the work of the addition instruction follows (Figure 4.4).

The fetch cycle of this instruction is identical to most of the fetch cycles of other instructions: get the instruction and bring it into the instruction register (IR), then increment the programme counter to prepare for the next instruction in the programme. Figure 4.5 provides the block diagram of that part of the CPU which is involved in the fetch part of the instruction cycle, i.e. it shows the address paths and the paths needed to read an instruction from memory.

The real work begins in step 5, where the address of the operand is transferred to the MAR. The intended operand of the instruction, the value stored at location X, is then transferred (step 6) to the MBR. Since the address is contained in the instruction, the value of X needed for step 5 can come from either the instruction register or the MBR. Finally the value is added (step 7) to the value currently in the accumulator, and the result left there. Figure 4.6 shows the address and data paths during the fetch and execute phases of the instruction.

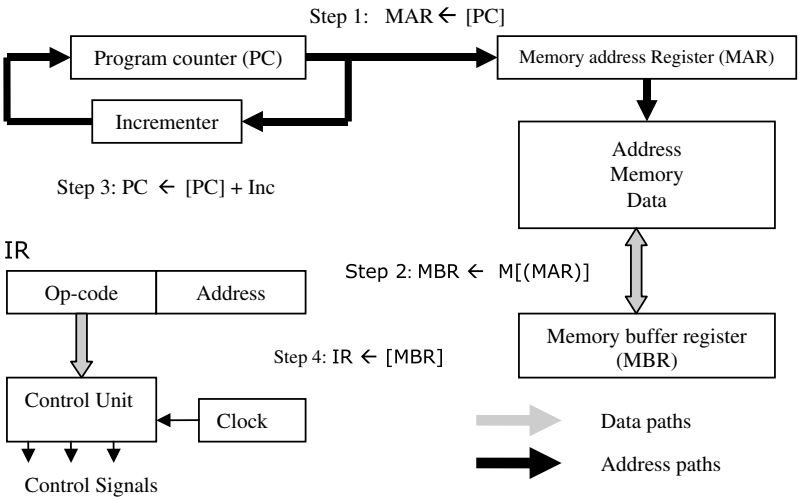


Figure 4.5 Address and data paths during the fetch phase.

The above example shows that the fetch cycle shown in Figure 4.2, consists of more than one step (four steps in the example) and similarly the execute cycle (three steps in the example). Figure 4.7, accordingly, gives a more realistic description for the instruction cycle. The fetch cycle is approximately the same for most of the instructions, while the steps included in the execute cycle change from one instruction to the other. The changes depend on the number of addresses included in the instruction and also on the address mode (see later for addressing modes).

Figure 4.7 is given in the form of a state diagram. The states included, represents all the steps given in Figure 4.6, but in a more general form. For any given instruction cycle, some states may be 0 and other may be visited more than once. For example, if the instruction contains more than one operand, the path between operand address calculation and operand fetch must be visited a number of times equal to the number of operands required by the instruction. The states can be described as follows:

- ***Instruction address calculation:*** determine the address of the next instruction to be executed. Usually, this involves adding a fixed number to the address of the previous instruction. For example, if each instruction is 16-bits long and memory is organized into 16-bit words, then add 1 to the previous address. If, instead, memory is organized as individually addressable 8-bit bytes, then add 2 to the previous address. Considering Figure 4.4, this means that *Inc* of step 4 is 1 or 2 respectively.

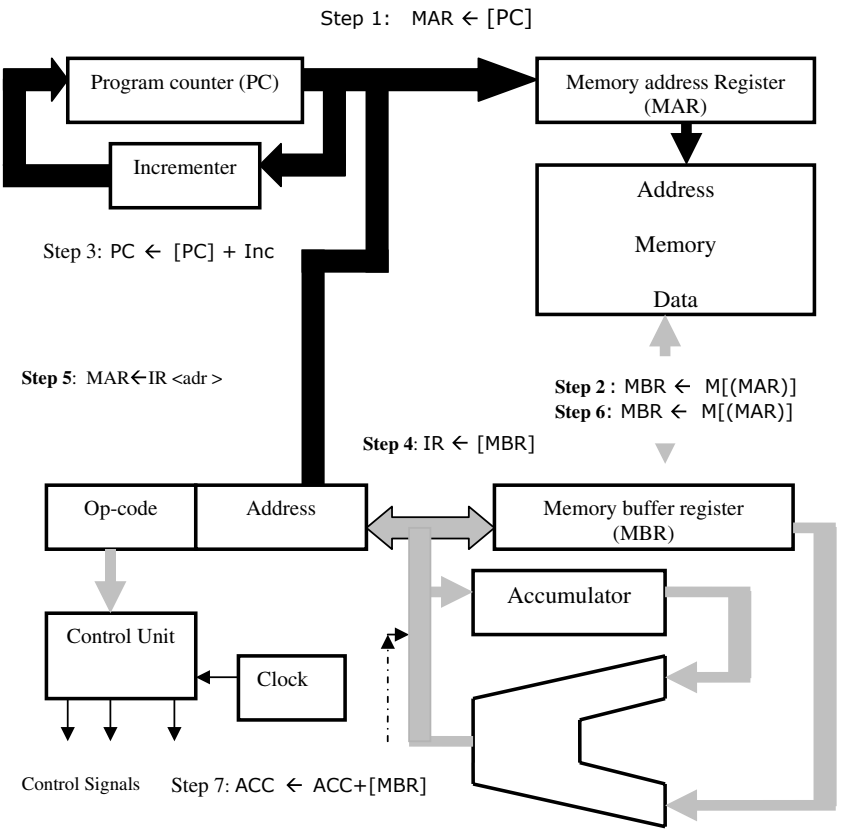


Figure 4.6 The CPU address and data paths during the fetch and execute phases.

- ***Instruction fetch*:** read instruction from its memory location into the processor.
- ***Instruction operation decoding*:** analyze instruction to determine type of operation to be performed and operand(s) to be used.
- ***Operand address calculation:*** if the operation involves reference to an operand in memory or variable via I/O, then determine the address of the operand.
- ***Operand fetch*:** fetch the operand from memory or read it from I/O.
- ***Data operation*:** perform the operation indicated in the instruction.
- ***Operand store*:** write the result into memory or out to I/O.

The instruction cycle of Figure 4.7 deals with instructions required to work in machines where work is defined as arithmetic or logic operations.

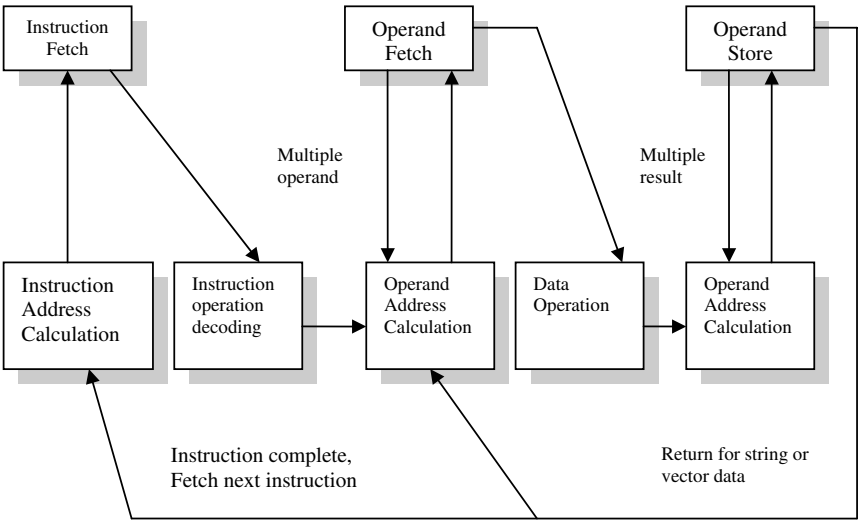


Figure 4.7 Instruction cycle state diagram.

Such actions, i.e. calculation of values, cover only one type of operation that computers must provide. In addition to computing, a machine must be able to make decisions, transfer information, and control devices. The instruction cycle in this case, a case where instructions are oriented toward input/output operations and used to control the programme flow, will be considered at a later stage. Table 4.3 summarizes the instruction cycle steps of some instructions.

4.4 Instruction Classifications According to Number of Operands

The instructions of the processor can be categorized in many different ways, based on how the instructions access data and operate upon it. One of the ways is based on the contents of the address field (operand specifier field) of the instruction, as well as the number and locations of the operands. The location of the operand is usually described by its address in the memory (if it is stored in the memory) or in the register file (if it is stored in a register).

This field (the address field) may provide anywhere from 0 to 4 or more operands and/or addresses of operands. Fewer operands make for shorter, but less flexible instructions, since some of the operands are implicit. The number of addresses contained in each instruction, is one of the traditional ways of describing processor architecture.

Table 4.3 Micro-operations to carry out some instructions.

Clock	Add	Subtract	Shift right	Load	Store
T1	[PC] → MAR	[PC] → MAR	[PC] → MAR	[PC] → MAR	[PC] →MAR
T2	[M(MAR)] → MBR	[M(MAR)] → MBR	[M(MAR)] → MBR	[M(MAR)] → MBR	[M(MAR)] → MBR
T3	[MBR] → IR	[MBR] → IR	[MBR] → IR	[MBR] → IR	[MBR] → IR
T4	[PC]+1 → PC	[PC]+1 → PC	[PC]+1 → PC	[PC]+1 → PC	[PC]+1 → PC
T5	IR<adr> → MAR	IR<adr> → MAR	SR[A] → A	IR<adr> → MAR	IR<adr> → MAR
T6	[M(MAR)] → MBR	[M(MAR)] → MBR		[M(MAR)] → MBR	[A] → MBR
T7	[A]+[MBR] → A	[A] – [MBR] → A		[MBR] → A	[MBR] → M(MAR)

	Branch	Branch if [A]=0		Halt
T1	[PC] → MAR	[PC] → MAR		[PC] → MAR
T2	[M(MAR)] → MBR	[M(MAR)] → MBR		[M(MAR)] → MBR
T3	[MBR] → IR	[MBR] → IR		[MBR] → IR
T4	IR<adr> → PC	If[A] ≠ 0 nothing	If[A] = 0 IR<adr> → PC	Reset state machine and disable
T5		[PC]+1 → PC		
T6	—	—	—	—
T7	—	—	—	—

Considering the number of operand (addresses), we can recognize the following:

(Note: In the next explanation we use *scr* to represent source operand, *dst* is the destination operand an OP to represent the operation specified in the op-code field).

0-operand ("zero address machines")

These are also called "***stack machines***", and all operations take place using the top one or two positions on the stack. In other words, 0-address instructions would reference the top two stack elements as discussed before in Section 2.1. The instruction contains only an op-code:

```
OP
```

In RTL, the operation performed is:

```
TOS  ←  (TOS) OP (TOS − 1)
```

Where, TOS is the top of stack and (TOS − 1) and the top of stack-1 (also known as the next top of stack), Figure 4.8.

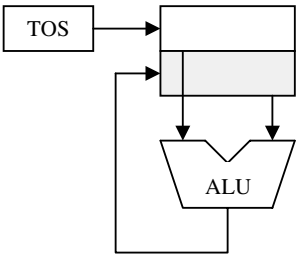


Figure 4.8 Stack machine (0-address).

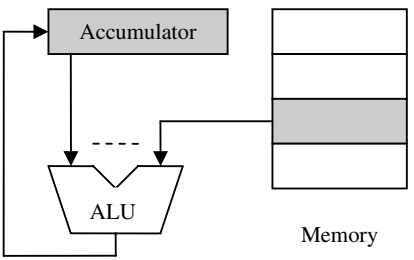


Figure 4.9 1-address accumulator.

Example 4.1

The instruction ADD will add the top of stack to the next top of stack and stores the result at the top of stack.

1-operand ("one address machines") — often called "accumulator machine":

This class includes most early computers, with the implied address being a CPU register known as the accumulator (AC). The accumulator contains one of the operand and is used to store the result, see Figure 4.9. The 1-address instruction takes the general form:

```
Op dst
```

In RTL means:

```
[dst] ← [Acc] OP [dst]
```

Example 4.2 The following instructions are 1-operand

```
load a, add b, store c
```

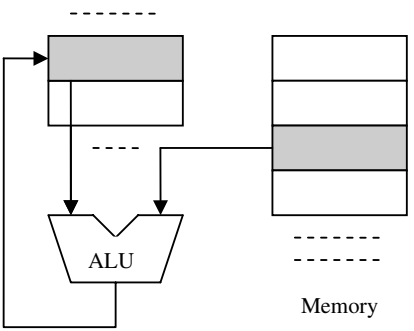


Figure 4.10 2-operand machine (Register-memory).

2-operand — one operand is both source and destination.

Many RISC machines fall into this category, though many CISC machines also fall here as well (See Figure 4.10). The instruction takes the general form:

OP dst, src

In RTL:

[dst] ← [dst] OP [src]

Example 4.3 For a RISC machine (requiring explicit memory loads), the instructions would be:

```
load a, reg1; load b, reg2;
    add reg1,reg2; store reg2,c
```

3-operand

In this case we have two source operands and one destination operand. Some CISC machines fall into this category. In such a case (case of CISC) the source and destination locations can be at the memory or any register. The instruction takes the general form:

```
OP dst, src1, src2
```

In RTL:

```
[dst] ← [src1] OP [src2]
```

Three-address instruction formats are not common, because they require a relatively long instruction format to hold the three address references.

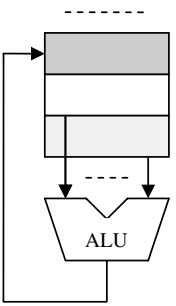


Figure 4.11 3-address RISC machine operation.

3-operand RISC

Most RISC machines fall into this category, because it allows “better reuse of data". In a typical three-operand RISC machines, all three operands must be registers, so explicit load/store instructions are needed (See Figure 4.11). An instruction set with 32 registers requires 15 bits to encode three register operands, so this scheme is typically limited to instructions sets with 32-bit instructions or longer.

4- or more operands

Some CISC machines permit a variety of addressing modes that allow more than 3 operands (registers or memory accesses), such as the VAX “POLY” polynomial evaluation instruction. In case of 4-Address (as an example), the fourth operand is either address of next instruction (obsolete technique) or for a certain reason, the instruction requires more than three operands (e.g. integer division produces quotient and remainder).

Many of the modern processors (especially CISC processors) have many instruction formats. They can handle 3-, 2- and 1-address format. Table 4.4 summarizes the interpretations to be placed on instructions with zero, one, two and three addresses. In each case in the table, it is assumed that the address of the next instruction is implicit and that one operation with two source operands and one result operand is to be performed. In Table 4.4 we used AC to represent accumulator, TOS to represent the top of the stack, (TOS-1) the contents of second element of stack, and OP as the operation to be performed.

Example 4.4

This example helps the reader to compare the different types of instructions and to understand the use of RTL. Consider the case of using 3-, 2-, 1-, and

Table 4.4 Utilization of instructions addresses (non-branching instructions).

Number of addresses	Symbolic representation	RTL
3	OP A, B, C	A ← B OP C
2	OP A, B	A ← A OP B
1	OP A	A ← AC OP A
0	OP	TOS ← (TOS − 1) OP TOS

0- address instructions to compute the expression:

$$X = (A + B) * (C + D)$$

where A, B, C, D and X are five main memory locations representing five variables.

Case 1: Use of 3-address format:

With three addresses, each instruction specifies two operand locations and a result location. Assume variables A, B, C, D, and X are stored in main memory locations labeled by their names, the calculation of X needs three instructions:

Instruction in Assembly Language	RTL	;Comment
`ADD R1, A, B`	`[R1]←[A] + [B]`	; add the contents of memory ; location A to that at memory ; location B and store the result at ; register R1
`ADD R2, C, D`	`[R2]←[C] + [D]`	; add the contents of memory ; location C to that at memory ; location D and store the result at ; register R2.
`MUL X, R1, R2`	`[X]←[R1] x [R2]`	; multiply the contents of register R1 ; to that of register R2 and store ; the result at memory location X.

Note: Intel MCS 51 family instructions are used.

Case 2: Use of 2-address format:

The calculation of X in this case needs the following sample programme:

Instruction in Assembly Language	RTL	;Comment
`MOV R1, A`	`[R1]←[A]`	; Move the contents of memory ; location A to register R1
`ADD R1, B`	`[R1]←[B]+[R1]`	; Add the contents of memory ; location A to the contents register ; R1 and store the result at R1
`MOV R2, C`	`[R2]←[C]`	; Move the contents of memory ; location C to register R2
`ADD R2, D`	`[R2]←[D]+[R2]`	; Add the contents of memory ; location D to the contents register R1 ; and store the result at R2
`MUL R2, R1`	`[R2]←[R1]×[R2]`	; Multiply the contents of R1 by the ; of R2 and store the result at R2
`MOV X, R2`	`[X]←[R2]`	; Move the contents of register R2 ; to memory location X.

Note: Intel MCS 51 family instructions are used.

From this example, it is clear that with 2-address instructions, one address must do double duty as both an operand and a result. The instruction ADD R1,B thus carries out the calculation A + B and stores the result in R1. The 2-address format reduces the space requirement, but also introduces some awkwardness. To avoid altering the value of an operand, a MOV instruction is used to move one of the values (C) to a temporary location before performing the next operation. The sample programme expanded into six instructions.

The following are some of the AVR instructions that use 2-operand format.

Instruction in Assembly Language	RTL	;Comment
`ADD Rd, Rr`	`[Rd]←[Rd]+[Rr]`	; add the contents of register Rd ; to that of register Rr and store ; the result at register Rd
`AND Rd, Rr`	`[Rd]←[Rd] AND [Rr]`	; AND the contents of register Rd ; and the contents o register Rs ; and store the result at Rd.
`SUB Rd, Rs`	`[Rd]←[Rd]−[Rs]`	; subtract the contents of register Rs ; from that of register Rd and ; store the result at register Rd.

Case 3: Use of 1-address format:

For 1-address to work, as mentioned above, the accumulator represents the second operand (implied address). The accumulator is used to store the result.

Instruction in Assembly Language	RTL	;Comment
`LOAD A`	`AC←[A]`	; Load the contents of memory location A ; into the accumulator
`ADD B`	`AC←[AC]+[B]`	; Add the contents of location B to that of the ; accumulator. The result stores at the ; accumulator
`STORE R`	`[R]←[AC]`	; Store the contents of the accumulator into R
`LOAD C`	`AC←[C]`	; Load the contents of memory location C ; into the accumulator
`ADD D`	`AC←[AC]+[D]`	; Add the contents of location D to that of the ; accumulator. The result stores at the ; accumulator
`MUL R`	`AC←[AC]×[R]`	; Multiplier the contents of R by that at the ; accumulator. The result stores at the ; accumulator.
`STORE X`	`[X]←[AC]`	; Store the contents of the accumulator ; into location X

Note: AVR family instructions are used.

Case 4: Use of 0-address format:

The use of stack to evaluate the given expression is given in Example 2.2 in Chapter 2. One of the main applications of the stack is the evaluation of arithmetic expressions. The system, in general, converts the expression from its given form (called infix notation form) into "reversed Polish notation (RPN)". The above equation in the Polish Notion (this notation is after the Polish philosopher and mathematician Jan Luckasiewicz- is as follows:

$$(A + B) \times (C + D) \Rightarrow AB + CD + \times$$

The RPN expression results directly in the programme given in Example 2.2, i.e.

```
PUSH        A
PUSH        B
ADD
```

```
PUSH        C
PUSH        D
ADD
MUL
POP         X
```

The contents of the stack during the execution of this programme are shown before in Example 2.2.

4.5 Addressing Modes

Each instruction tells the processor to perform an operation which may involve one or two pieces of data (operands). Addressing modes are concerned with how an operand is located by the CPU. A computer's addressing modes are almost entirely analogous to human addressing modes, as the following examples demonstrate.

- Here's $1000 (literal value)
- Get the cash from 12 North Street (absolute or direct address)
- Go to 15 East Street and they'll tell you where to get the cash (indirect address)
- Go to 20 North Street and get the cash from the fifth house to the right (relative address)

These four "addressing modes" show that we can provide data itself (i.e. the $1000 cash), say exactly where it is, or explain how you go about finding where it is. Now if we apply the same principles to the computer, addressing modes will be defined as the ways one can compute an effective address (EA) where the effective address is the address of operand used by instruction.

Definition: The effective address (EA) of an operand is the address of (or the pointer to) the main memory (MM) or register file (RF) location in which the operand is contained:

$$\text{Operand} = [\text{EA}];$$

In case of computer the need for different addressing modes arises from the fact that the address field or fields in a typical instruction format are relatively small for most computers. Any programmer would like to be able to reference a large range of locations in main memory or, for some systems, virtual memory. To achieve this objective, a variety of addressing techniques has been

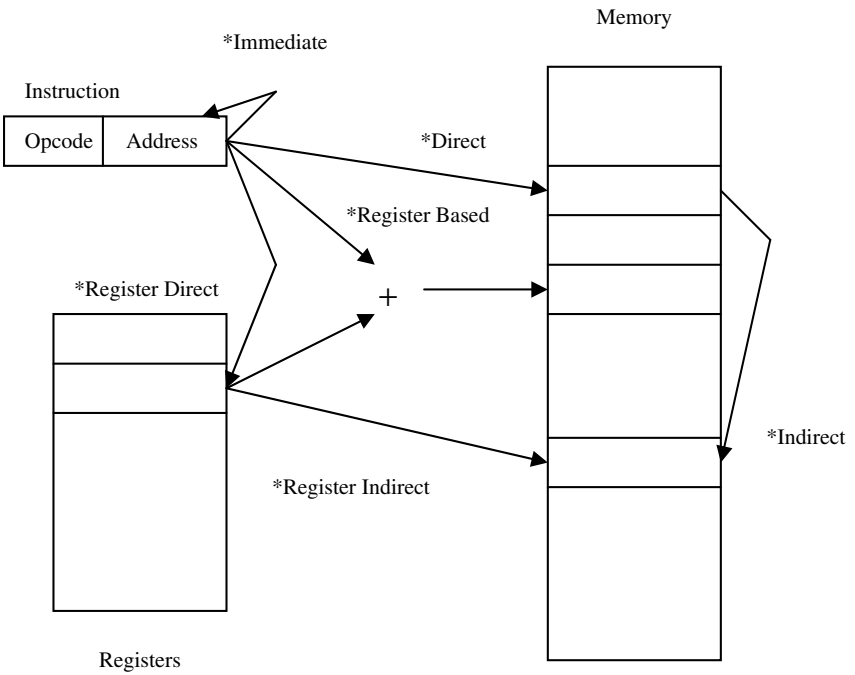


Figure 4.12 Various addressing modes.

employed. They all involve some trade-off between address range and/or addressing flexibility, on the one hand, and number of memory references and/or complexity of address calculation, on the other. Effective-address calculations establish the paths to operands of instructions. Operands can be accessed by immediate, direct, indirect, or register-based addressing, and by combinations of these. The different forms of addressing are illustrated in Figure 4.12. In this section, we examine the most common addressing techniques:

- Immediate
- Direct or Absolute
- Memory Indirect
- Register indirect
- Displacement
- Stack

While explaining the addressing modes we are going to take, in the majority of the cases, our examples from the instruction set of AVR microcontroller. The AVR Enhanced RISC microcontroller supports powerful

and efficient addressing modes for access to the Programme memory (Flash) and Data memory (SRAM, Register file, I/O Memory, and Extended I/O Memory).

Before beginning the discussion, we have to mention that all processors (microprocessors/microcontrollers) architectures provide more than one of these addressing modes. Several ways are used to let the control unite determines which address mode is being used in a particular instruction. One approach is to allocate one or more bits in the instruction format to be used as a "*mode field*". The value of the mode field determines which addressing mode is to be used.

In our text, as many other texts, we are using the word "*Effective address*". In a system without virtual memory, the effective address (EA) means either a main memory address or a register. In a virtual memory system, the effective address is a virtual address or a register. The actual mapping to a physical address is a function of the paging mechanism and is invisible to the programmer. In all microcontroller based systems we do not use the concept of virtual memory.

In our discussions we use the following notations:

A = Contents of an address field in the instruction.

R = Contents of an address field in the instruction that refers to a register.

[X] = Contents of memory location X or register X.

EA = Effective address.

OP = the word OP means the operation code part of the instruction word.

To simplify, not all figures show the exact location of the addressing bits. To generalize, the abstract terms RAMEND and FLASHEND have been used to represent the highest location in data and programme space, respectively.

4.6 Immediate Addressing Mode

Immediate addressing, which is also known as *literal* addressing, is provided by all microprocessors/microcontrollers and allows the programmer to specify a constant as an operand. Immediate addressing does not access any memory locations or any register in a register file, but uses the value following the op-code in an instruction. In other words the address field is not a reference to the address of an operand, but is the actual operand itself (Figure 4.13), i.e. the constant value is the source in a transfer.

```
Operand = A
```

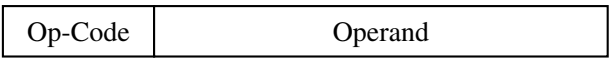


Figure 4.13 Immediate addressing mode.

Assembly language form	RTL form	Addressing mode
ADD 1234,D0	[D0] ← [D0]+[1234]	absolute addressing
ADD #1234,D0	[D0] ← [D0]+1234	immediate addressing

(a) Motorola 68K immediate and absolute addressing modes

Assembly language form	RTL form	Addressing mode
LD Rd,Y	Rd ← [Y]	absolute addressing
LDI Rd,K	Rd ← K	immediate addressing

(b) AVR immediate and absolute addressing modes.

Assembly language form	RTL form	Addressing mode
MOV R0,#12	R0 ← 12	immediate addressing
ADD A,#15	[A] ← [A] + 15	immediate addressing

(c) Intel 8051 immediate addressing mode

Figure 4.14 Immediate addressing mode.

The sizes allowed for immediate data vary by processor and often by instruction (with some instructions having specific implied sizes). In some microprocessors assembly language (e.g. Motorola 68 K series, Intel 8085, etc.), the symbol # precedes the operand to indicate immediate addressing. In other microprocessors/microcontrollers the instruction contains the letter "I" to indicate the immediate mode. For example in ATMEL AT90S8515 series, the instruction "AND" is a logical AND, while ANDI is logical AND with immediate. The four instructions given in Figure 4.14(a) demonstrate how absolute and immediate addressing modes are represented in Motorola 68 K assembly language and in RTL, respectively. Figure 4.14(b) demonstrates the case of Atmel AVR microcontroller and Figure 4.14c is the case of Intel 8051.

All the numbers in Figure 4.14 are represented in base 10 (decimal). Any base can be used to represent the numbers. The symbol #, in case of using it, is not part of the instruction. It is a message to the assembler telling it to select that code for "move data" that uses the immediate addressing mode. This is exactly as when we use the symbols $, H, % or B to indicate the base used to represent the numbers.

4.6.1 Advantages of Immediate Addressing

The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand, thus saving one memory or cache cycle in the instruction cycle. The main disadvantage of the immediate addressing mode is the limited operand magnitude. It is actually restricted to the size of the address field, which, in most instruction sets, is small compared with the word length.

4.6.2 AVR Instructions with Immediate Addressing

In the following we introduce some of the AVR instructions with immediate addressing mode.

Example 4.5: CPI — Compare with Immediate (CPI Rd,K)

```
      cpi r19,3        ; Compare r19 with 3
      brne error       ; Branch if r19<>3
       ...
error: nop             ; Branch destination
                       ;(do nothing)
```

Note: AVR can deal with full word of 16 bits in the immediate mode. As an example consider the following:

```
adiw r25:24,1          ; Add 1 to r25:r24
adiw ZH:ZL,63          ; Add 63 to the Z-pointer
                       ;  (r31:r30)
```

Example 4.6: SBCI — Subtract Immediate with Carry (SBCI Rd,K)

```
                   ; Subtract $4F23 from
                   ;  r17:r16
subi r16,$23       ; Subtract low byte
sbci r17,$4F       ; Subtract with carry high
                   ;  byte
```

Example 4.7: ANDI — Logical AND with Immediate (ANDI Rd,K)

```
andi r17,$0F       ; Clear upper nibble of r17
andi r18,$10       ; Isolate bit 4 in r18
andi r19,$AA       ; Clear odd bits of r19
```

Some AVR microcontrollers can handle in the immediate addressing mode 16-bit words. Add immediate to word (ADIW) and subtract immediate from word (SBIW) instructions are used for that goal. We note here that ADIW and SBIW are not available in all devices.

Example 4.8: ADIW — Add Immediate to Word (ADIW Rd+1:Rd,K)

```
adiw r25:24,1        ; Add 1 to r25:r24
adiw ZH:ZL,63        ; Add 63 to the Z-pointer
                     ; (r31:r30)
```

Example 4.9

The instruction SBIW

```
sbiw r25:r24,1       ; Subtract 1 from r25:r24
sbiw YH:YL,63        ; Subtract 63 from the
                     ; Y-pointer(r29:r28)
```

4.7 Direct (Absolute) Addressing Mode

In direct or absolute addressing the operand field of the instruction provides the location of the operand. The location can refer to:

- Register (Register Direct Addressing)
- Location in the memory (Memory Direct Addressing)

Thus for direct addressing:

$$EA = A$$

(A is the contents of the operand field)

4.7.1 Register Direct Addressing

1. Register Direct: Single Register Rd

This is a form of direct addressing in which the address field of the instruction refers to a register. In Figure 4.15 the operand is contained in register d (Rd).

$$EA = d$$

In the figure shown, the address field is given 5 bits, which means that it can access any location within a register file of maximum size of 32 registers.

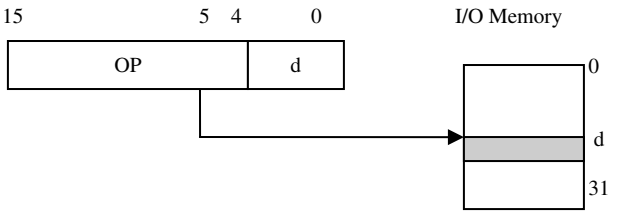


Figure 4.15 Direct single register addressing.

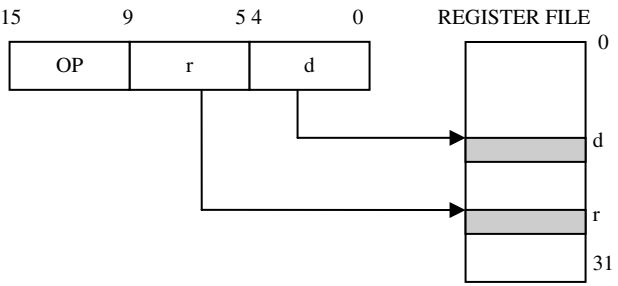


Figure 4.16 Direct register addressing: case of two registers.

AVR microcontroller is an example of such a case. The length of d can be any value depends on the processor structure.

2. Register Direct: Two Registers Rd and Rr

Many instructions contain the addresses of both operands in their instruction word. Figure 4.16 shows this and d (Rd). The result is stored in register d (Rd). The figure assumes that the register file contains 32 registers, consequently the length of each of the two fields d and r is 5 bits each. AVR microcontroller is an example of such a case.

An example of direct addressing of two registers is the instruction:

```
Add r0, r1
```

The contents of the registers r0 and r1 will be added and the result will be stored in register r0. This can be expressed as:

$$\text{REG[(R0)]} \leftarrow \text{REG[R0]} + \text{REG[R1]}$$

The advantages of register addressing are that:

1. Only a small address field is needed in the instruction, and
2. No memory references are required.

The disadvantage of register addressing is that the address space is very limited.

Some AVR Register Direct Instructions:

Example 4.10: ADD — Add without Carry (ADD Rd,Rr)

```
add r1,r2       ; Add r2 to r1 (r1=r1+r2)
add r28,r28     ; Add r28 to itself (r28=r28+r28)
```

Example 4.11: AND — Logical AND (AND Rd,Rr)

```
and r2,r3       ; Bitwise and r2 and r3,
                ; result in r2
ldi r16,1       ; Set bitmask 0000 0001 in r16
and r2,r16      ; Isolate bit 0 in r2
```

4.7.2 Memory Direct Addressing

4.7.2.1 I/O Direct

Instructions that handle the I/O memory contain both the port address (6 bit) and either the source or destination register (5 bit) of that operation, as shown in Figure 4.17. Operand address (port address) is contained in 6 bits (P in the figure) of the instruction word. *n* is the destination or source register address.

AVR I/O Direct Addressing Instructions

Example 4.12: IN — Load an I/O Location to Register

```
in r25,$16      ; Read Port B
cpi r25,4       ; Compare read value to constant
breq exit       ; Branch if r25=4
...
```

exit: nop ; Branch destination (do nothing)

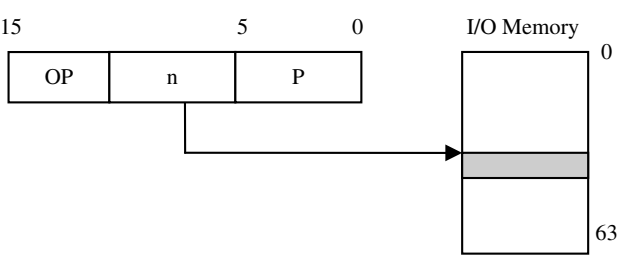


Figure 4.17 I/O direct addressing.

Example 4.13: OUT — Store Register to I/O Location

```
clr r16          ; Clear r16
ser r17          ; Set r17
out $18,r16      ; Write zeros to Port B
nop              ; Wait (do nothing)
out $18,r17      ; Write ones to Port B
```

4.7.2.2 Data direct (SRAM direct addressing)

Instructions in the SRAM direct addressing mode needs two 16-bit words. The 16-bit SRAM Data Address is contained in the lower word, i.e. in the 16 LSBs of the two-word instruction. The source or destination register (5 bits) of that operation is contained in the higher word. Figure 4.18 shows this addressing mode.

Example 4.14

An example of SRAM direct register addressing is the instruction:

```
Lds r0, $1000
```

Register r0 will be loaded with the contents of SRAM location $1000.

AVR Load Direct Instruction

In case of AVR, the direct addressing mode is used with two instructions LDS (Load Direct SRAM) and STS (Store Direct to SRAM). LDS loads one byte from the SRAM to a register, while STS stores one byte from a register to the SRAM. In the two instructions, memory access is limited to the current SRAM page of 64 K bytes. The two instructions use the RAMPZ register to access memory above 64 K bytes. The following sequence of instructions

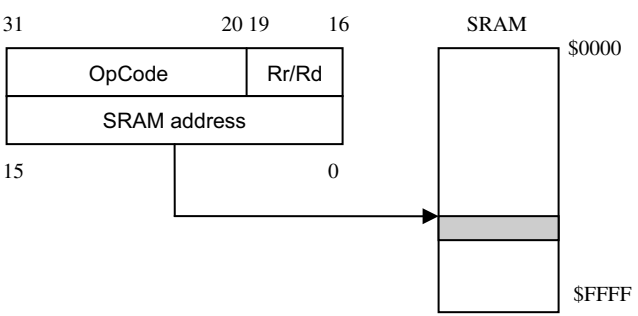


Figure 4.18 Direct data addressing.

represents an example of this mode:

Example 4.15: LDS — Load Direct from Data Space

```
lds r2,$FF00 ; Load r2 with the contents of
             ; data space location
             ; $FF00
add r2,r1    ; add r1 to r2
sts $FF00,r2 ; Write back
```

Example 4.16: STS — Store Direct to Data Space

```
Lds r2, $FF00      ; Load r2 with the contents
                   ; of SRAM
                   ; location $FF00
Add r2, r1         ; add r1 to r2
Sts $FF00, r2      ; write back.
```

4.8 Indirect Addressing Mode

The need for indirect addressing mode: In immediate as well as direct addressing modes the addresses don't change during the execution of a programme. In many situations it is difficult to access data structures such as tables, lists or arrays unless, the computer has a means of changing addresses during the execution of a programme.

To demonstrate that, let us consider the case of adding together several numbers. At first assume that we want to add together 200 numbers stored in consecutive locations using the immediate and/or direct addressing modes only. Because immediate and direct addressing modes require operand addresses to be known at assemble-time, the programme will take the following form:

```
MOVE NUM1, D0    ; Put first number in D0.
ADD  NUM2, D0    ; Add second number to D0.
ADD  NUM3, D0    ; Add third number to running
                 ; total.
:
ADD  NUM199, D0 ; Add the 199th number.
ADD  NUM200, D0 ; Add the last number.
```

Clearly, there has to be a better solution to this problem. What we want is an instruction that adds in number 1 when we execute it the first time, number 2 when we execute it the second time and number i when we execute it the i^{th} time. Indirect addressing allows you to generate plenty of different addresses once a programme is executed.

In indirect addressing mode, the address of an operand is obtained indirectly. The effective address (EA) of the operand is in the register, or memory location, whose address is given in the instruction. This means that, in indirect addressing, instead of telling the computer where the operand is, the pointer tells it where the address of the operand can be found. (Some times the location holding the real address is known as a ***vector.***)

If EA is in one of the address registers, then we have **"register indirect addressing".** If EA is in memory location, then we have "**memory indirect addressing"** or simply **"Indirect addressing".** In register indirect addressing, the contents of a designated register points to the memory location, contains the operand (Figure 4.19a). This means that in register indirect:

$$EA = [R]$$

In memory indirect addressing, the effective address of the operand is given by the contents of the memory location pointed at by the address following the op-code (Figure 4.19b). In other words, the instruction provides the address of the address of the data. In memory indirect:

$$EA = [A]$$

The register indirect addressing mode is supported by all processors, while the memory indirect addressing mode is supported by very few processors.

Next we consider the indirect addressing modes used by AVR.

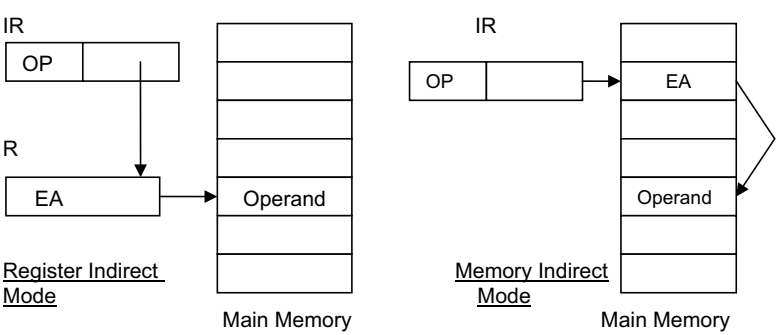


Figure 4.19 Indirect addressing.

4.8.1 AVR Indirect Addressing

The indirect data addressing modes are very powerful feature of the AVR architecture. The overhead to access large data arrays stored in SRAM can be minimized with these data modes. AVR indirect addressing uses "register indirect addressing" in which the contents of designated registers are used as pointer to memory.

AT908515 manufacturers call the X-, Y-, and Z-registers, "index registers". They are using them as designated registers to store the location of the operand in the memory in case of address register indirect addressing (Figure 4.20). All the instructions that use indirect addressing, accordingly, one of the index registers appears in the syntax of the instruction. For example, the instruction LD Rd, X is a load indirect from SRAM to register using index X. Some instructions are only allowed to use the Z register as index. Accordingly, the index register does not appear in the syntax, instead the letter "I" appears in the syntax. For example, the instructions ICALL (indirect call to subroutine) and IJMP (indirect jump) of AT90S8515. As a matter of fact ATMEL call the two cases of Call and Jump as "Indirect Programme Addressing".

The following AT90S8515 instructions illustrate address register indirect addressing and provide RTL definitions for the action to be carried out, together with a plain language description.

Assembly Language	RTL	Description
LD Rd, X	`[Rd]←M[(X)]`	Loads one byte indirect from SRAM to register. The SRAM location is pointed to by the contents of the X pointer register.
I CALL	`PC<15-0>←Z<15-0>`	Indirect call of a subroutine pointed to by the Z pointer register
IJMP	`PC<15-0>←Z<15-0>`	Indirect jump to the address pointed to by the Z pointer register

In case of AVR the operand address is the contents of the X, Y or the Z-register.

As Figure 4.21 shows, for indirect data addressing the operand address is the contents of the X-, Y-, or Z- register. The source or destination register (needs 5 bits) of that operation is contained in the instruction word. An example of indirect SRAM addressing is the instruction:

```
Ld r4, Z
```

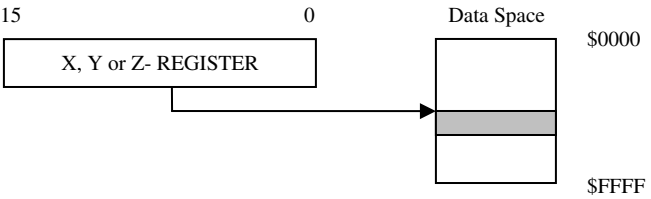


Figure 4.20 Data Indirect addressing.

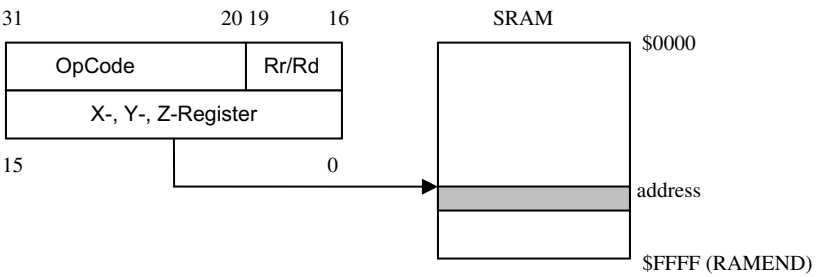


Figure 4.21 Indirect addressing mode using index register.

When executing this instruction, register r4 will be loaded with the contents of the SRAM location defined by the contents of the Z-register.

4.8.2 Variation on the Theme

Many processors supports more than one variation on address register indirect addressing. The possible variations are:

- Address register indirect with postincrement
- Address register indirect with predecrement
- Address register indirect with preincrement
- Address register indirect with postdecrement
- Address register indirect with displacement

ATMEL AT90S8515 series, ATMega series, for example, support the first two modes and the last mode. In the following we consider these variations on the indirect register addressing. We are going to consider that address register indirect with displacement as one of the possible forms of displacement addressing.

4.8.2.1 Data Indirect with Pre-decrement

For indirect data addressing with pre-decrement, the X, Y or the Z-register is decremented before the operation. Operand address is the decremented contents of the X, Y or the Z-register.

$$EA = [Y] - 1$$

The original contents of the X-, Y-, or Z-register are replaced by the decremented value after that operation. The source or destination register (5 bits) of that operation is contained in the instruction word. Figure 4.22 shows this addressing mode.

This mode is useful for a loop where the same or similar operations are performed on consecutive locations in memory. This address mode can be combined with a complimentary postincrement mode for stack and queue operations.

An example of indirect SRAM addressing with pre-decrement is the instruction:

```
Ldd r4, Y-
```

When executing this instruction, register r4 will be loaded with the contents of the SRAM location defined by the contents of Y-register decremented by 1. After that operation, the Y-register contains the decremented value.

4.8.2.2 Data Indirect with Post-increment

For indirect data addressing with post- increment, the operand address is the contents of X, Y-, or Z-register. The X, Y or the Z-register is incremented by 1 after the operation. The source or destination register of that operation is contained in the instruction word. Figure 4.23 shows this addressing mode.

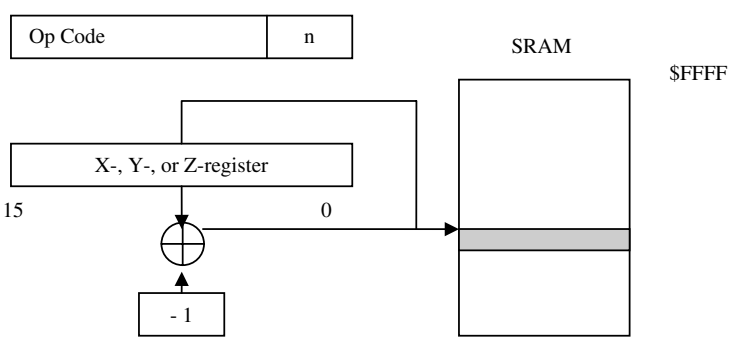


Figure 4.22 Data indirect addressing with pre-decrement.

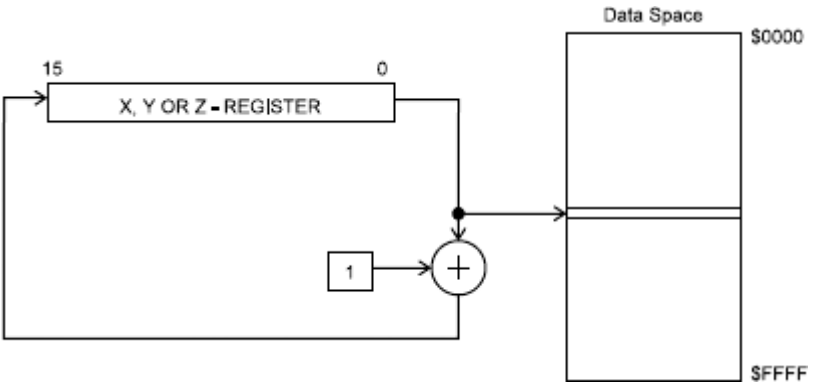


Figure 4.23 Data indirect addressing with post-decrement.

An example of indirect SRAM addressing with post-increment is the instruction:

```
Ldd r4, X+
```

Register r4 will be loaded with the contents of the SRAM location addressed by the contents of the X-register. After that operation, the X-register contains the incremented address value for the next operation.

4.8.2.3 Examples of AVR Instruction with Indirect Addressing

As mentioned earlier, AVR manufacturers call the X-, Y-, and Z-registers, "index registers". They are using them as designated registers to store the location of the operand in the memory in case of address register indirect addressing. All the instructions that use indirect addressing, accordingly, one of the index registers appears in the syntax of the instruction. As an example of the AVR instructions that use indirect addressing mode, we introduce here the store indirect, and the load indirect instructions;

ST — Store Indirect from Register to Data Space using Index X

Description: This instruction stores one byte indirect from a register to data space.

For AVR AT90S8515/ATMega 8515 and all other microcontrollers that have SRAM, the data space consists of the Register File, I/O memory and internal SRAM and external SRAM (if applicable).

The data location is pointed to by the X (16 bits) Pointer Register in the Register File. Memory access is limited to the current data segment of 64K bytes.

The X-pointer Register can either be left unchanged by the operation, or it can be post-incremented or pre-decremented. As mentioned before, these features are especially suited for accessing arrays, tables, and Stack Pointer usage of the X-pointer Register. Note that only the low byte of the X-pointer is updated in devices with no more than 256 bytes data space. For such devices, the high byte of the pointer is not used by this instruction and can be used for other purposes.

Using the X-pointer:

Syntax	Operation		Comment
ST X, Rr	(X) ← Rr		X: Unchanged
ST X+, Rr	(X) ← Rr	X ← X+1	X: Post Incremented
ST - X, Rr	(X) ← Rr	X ← X-1	X: Pre decremented

Note:
1. In all cases: $0 \leq r \leq 31$ and PC ← PC + 1
2. Since r26 and r27 represent the low byte and the high byte of X pointer, the result of the following combinations is undefined:

```
ST X+, r26
ST X+, r27
ST -X, r26
ST -X, r27
```

Example 4.17

```
clr r27         ; Clear X high byte
ldi r26,$60     ; Set X low byte to $60
st X+,r0        ; Store r0 in data space loc.
                ; $60(X post inc)
st X,r1         ; Store r1 in data space loc.
                ; $61
ldi r26,$63     ; Set X low byte to $63
st X,r2         ; Store r2 in data space loc.
                ; $63
st -X,r3        ; Store r3 in data space loc.
                ; $62(X pre dec)
```

Example 4.18: ST uses Index Y

```
clr r29          ; Clear Y high byte
ldi r28,$60      ; Set Y low byte to $60
st Y+,r0         ; Store r0 in data space loc.
                 ; $60 (Y post inc)
st Y,r1          ; Store r1 in data space loc.
                 ; $61
ldi r28,$63      ; Set Y low byte to $63
st Y,r2          ; Store r2 in data space loc.
                 ; $63
st -Y,r3         ; Store r3 in data space loc.
                 ; $62 (Y pre dec)
std Y+2,r4       ; Store r4 in data space loc.
                 ; $64
```

4.9 Displacement Addressing

This is a very powerful mode of addressing that combine the capabilities of direct addressing and register indirect addressing. Depending on the context of its use, it is known by a variety of names, but the basic mechanism of calculating the effective address is the same. The effective address is calculated as:

```
EA = A + [R]
```

Displacement addressing requires that the instruction have two address fields, at least one of which is explicit. The value contained in one address field (value = A) is used directly. The other address field, or an implicit reference based on opcode, refers to a register whose contents are added to A to produce the effective address. We will describe three of the most common uses of displacement addressing:

- Base-register addressing and also called address register indirect with displacement (e.g. in case of AVR microcontrollers and Motorola).
- Relative addressing
- Indexing

4.9.1 Address Register Indirect with Displacement (also called "Base-Register Addressing")

In **address register indirect with displacement** operations, the contents of the designated register are modified by adding or subtracting a displacement

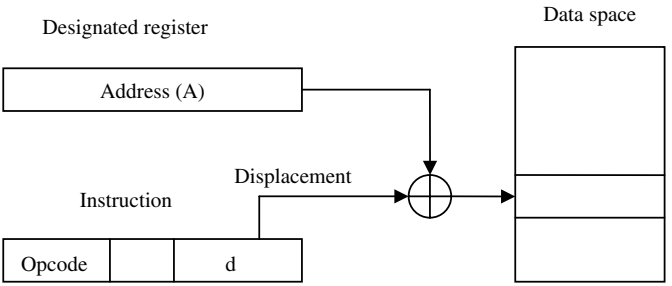


Figure 4.24 Register indirect with displacement.

integer and then used as a pointer to memory. The displacement integer is stored in the instruction (Figure 4.24) and if shorter than the length of the processor's address space (the normal case), sign-extended before addition (or subtraction).

AVR Base — Register Addressing

AVR uses the registers X, Y and Z as pointers i.e. as designated registers. The value of the displacement is limited by giving six bits in the instruction. Figure 4.24 illustrates how the effective address is calculated for the AT90S8515 instruction LD Rd, $Z + q$, which will load register Rd with the contents of the SRAM location defined by the contents of the Z-register incremented by *q, where* $0 \leq q \leq 63$.

4.9.2 Data Indirect with Displacement

Operand address is the result of the Y or Z-register contents added to a displacement value represented by the address contained in 6 bits of the instruction word. The source or destination register (5 bits) of that operation is also contained in the instruction word. Figure 4.25 shows the required calculation before addressing SRAM.

$$\text{EA} = [\text{Y- or Z-register}] + q$$

An example of indirect SRAM addressing is the instruction:

```
Ldd r4, Y+2
```

Register r4 will be loaded with the contents of the SRAM location defined by the contents of the Y-register incremented by 2.

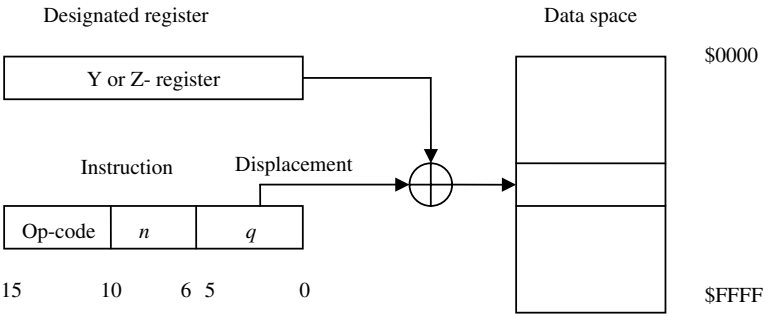


Figure 4.25 AVR register indirect with displacement: The referenced register contains a memory address and the address field contains displacement.

4.10 Relative Addressing Mode

Relative addressing or, more properly, programme counter relative addressing is similar to address register indirect addressing because the effective address of an operand is given by the contents of a register plus a displacement. However, the implicitly referenced register is the programme counter (PC). In other words relative addressing uses the programme counter (PC) to calculate the effective address, rather than an address register; that is, the location of the operand is specified relative to the current instruction.

The relative addressing mode is used normally, as we shall see, with certain jump instructions.

AVR Relative Addressing

In case of AVR relative addressing is used in the two instructions rjmp and rcall that are effective for jumps or calls in a reduced memory area. The jumps or calls work relative to the programme counter. Figure 4.26 shows the memory addressing for the rjmp and rcall instructions.

For AVR microcontrollers with programme memory not exceeding 4 K words (8 K bytes), this instruction can address the entire memory from every address location.

4.11 Programme Memory Addressing: Constant Addressing using LPM Instruction

Access to tables stored in programme memory can be handled very easy by the LPM instruction. The LPM instruction loads 1 byte indirectly from programme memory to a register. The programme memory location is pointed to by the Z

```
RCALL   k       ;[PC] ← [PC] + k + 1. Calls a subroutine within ± 2K
                ;words (4K bytes). This means that −2K≤ k≤ 2K )
                ;The return address (the instruction after the CALL) is
                ;stored onto the stack.
RJMP    k       ;[PC] ← [PC] + k + 1.
                ;Relative jump to an address within
                ; PC -2K and PC +2K words).
```

Figure 4.26 Use of r jump.

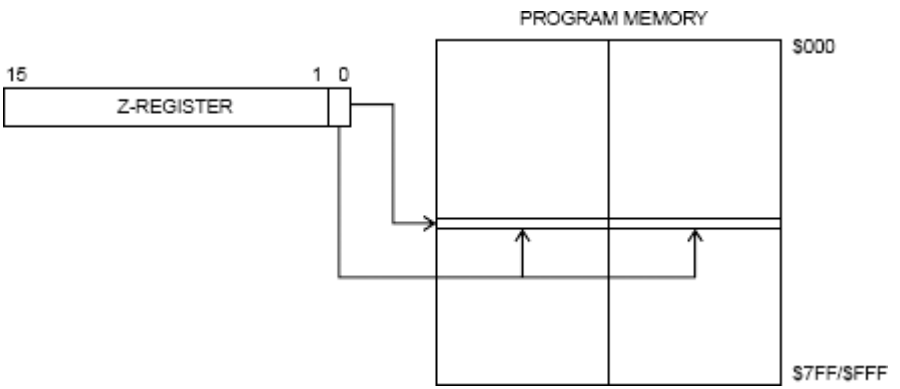


Figure 4.27 Code memory constant addressing.

(16 bits) pointer register in the register file. Memory is limited to 4 K words. The 15 MSBs of the Z-register select word address (0 − 2 K/4 K), the LSB selects low byte if cleared (LSB = 0) or high byte if set (LSB = 1). Figure 4.27 shows addressing programme memory with instruction lpm. The Branch/Jump and Call instructions are also accessing the programme memory. This group of instructions will be discussed while discussing Branch and Call.

LPM — Load Programme Memory

Description: This instruction loads one byte pointed to by the Z-register into the destination register Rd. It features a 100% space effective constant initialization or constant data fetch. The Programme memory is organized in 16-bit words while the Z-pointer is a byte address. Thus, the least significant bit of the Z-pointer selects either low byte (ZLSB = 0) or high byte (ZLSB = 1). This instruction can address the first 64 K bytes (32 K words) of Programme memory. The Z-pointer Register can either be left unchanged by the operation, or it can be incremented. The incrementation does not apply to the RAMPZ Register.

Not all variants of the LPM instruction are available in all devices. Refer to the device specific instruction set summary. The LPM instruction is not implemented at all in the AT90S1200 device.

The result of these combinations is undefined:

```
LPM r30,    Z+
LPM r31,    Z+
```

Example 4.20

```
ldi ZH, high(Table_1<<1)   ; Initialize Z-pointer
ldi ZL, low(Table_1<<1)
lpm r16, Z                 ; Load constant from
                           ; programme
                           ; Memory pointed to by Z
                           ; (r31:r30)
...
Table_1:
.dw 0x5876                 ; 0x76 is addresses when
                           ; Z_LSB = 0
                           ; 0x58 is addresses when
                           ; Z_LSB = 1
...
```

4.12 Stack Addressing

As mentioned before, a stack is a linear array of locations. It is sometimes referred to as a pushdown list or last-in-first-out (LIFO) queue. The stack is reserved block of locations. Items are appended to the top of the stack so that, at any given time, the block is partially filled. Associated with the stack is a pointer whose value is the address of the top of the stack. The stack pointer is maintained in a register called stack register. Thus, references to stack locations in memory are in fact register indirect addresses.

The stack addressing mode is a form of implied addressing. The machine instructions need not include a memory reference but implicitly operate on the top of the stack. The stack can be handled by the use of the two instructions POP and PUSH.

POP — Pop Register from Stack (POP Rd)

Description: This instruction loads register Rd with a byte from the STACK. The Stack Pointer is pre-incremented by 1 before the POP; [Rd] $\leftarrow$ STACK. Also it results in the operations: SP $\leftarrow$ SP $+1$ and PC $\leftarrow$ PC $+ 1$

Example 4.21

```
            call routine    ; Call subroutine
            ...
routine:    push r14        ; Save r14 on the Stack
            push r13        ; Save r13 on the Stack
            ...
            pop r13         ; Restore r13
            pop r14         ; Restore r14
            ret             ; Return from subroutine
```

PUSH — Push Register on Stack (PUSH Rr)

Description: This instruction stores the contents of register Rr on the STACK. The Stack Pointer is post-decremented by 1 after the PUSH: STACK $\leftarrow$ Rr. Also : SP $\leftarrow$ SP + 1 and PC $\leftarrow$ PC + 1

Example 4.22

```
            call routine    ; Call subroutine
            ...
routine:    push r14        ; Save r14 on the Stack
            push r13        ; Save r13 on the Stack
            ...
            pop r13         ; Restore r13
            pop r14         ; Restore r14
            ret             ; Return from subroutine
```

4.13 Programme Control Instructions

The instructions dealt with so far are instructions required to do work in machines, where work is defined as arithmetic or logic operations. Calculation of values covers only one type of operation that computers must provide. In addition to computing, a machine must be able to make decisions, Input and Output information, and control devices. In other words the instruction set must contain instructions cover:

- Input/output operations, and
- Control the programme flow

In this section we are discussing the programme flow instructions. The area of programme flow instructions can be divided into two general groups: instructions that change the flow of the programme without side effects, and instructions that modify the programme counter and also cause additional operations to occur. Examples of the first type of instruction are conditional and unconditional branches, while the second type of instruction is exemplified by a subroutine call.

a. Case of Jump/Branch

The simplest instructions to deal with are those that change the programme flow without any side effects. As we have indicated before, the assumed address for the next instruction to execute identities the location immediately following the current instruction. That is, normal programme behavior calls for the programme counter to be incremented from one instruction to the next. When the next instruction to execute is not the next one in the memory, then the programme counter must be modified accordingly. The programme counter must be changed to identify the appropriate instruction to be fetched next. The terminology used by many manufacturers that a programme counter change that uses direct addressing mechanisms is called a "jump". and a programme counter change that identities its target address as an offset from the current location (PC relative) is a "branch".

The jump/branch instruction is very straight forward: the target address is identified, and the programme counter is changed accordingly. The target address can be specified by combinations of the various addressing modes that we have already identified. The system operation changes somewhat when the branch is made conditional. In this situation, the contents of the PC at the completion of the branch instruction are dependent upon some system status condition or on some comparison identified by the instruction. The conditions may include the status bits contained within the status register of the machine.

In all the cases, if the proper conditions are satisfied, the PC contents are modified to allow the programme to continue at an address identified by the instruction. If the conditions are not satisfied for modifying the programme flow, then the programme counter is incremented in the normal fashion and execution of the programme continues with the next instruction in the normal order of execution. These programme counter modification mechanisms are demonstrated by the following example.

Normally jumps can use any of the appropriate addressing modes to identify the target address. The target address is identified, by whatever

combinations of addressing modes are available, and the specified address is placed in the programme counter. In the case of conditional execution, the necessary condition is tested, and then the appropriate action is taken. Some machines uses more complicated mechanisms for branching. For example, one minicomputer used a three-way branch for its arithmetic tests: three target addresses followed an arithmetic conditional branch instruction. A different target address was used for the greater than, equal to, and less than arithmetic conditions,

The conditional mechanisms provided in instruction sets reflect the intended use of the systems. For example, some instruction sets will contain a dedicated CASE instruction that facilitates decisions requiring a multiway branch capability. Other systems will use combinations of instructions to perform this function. Another example is the use of a special LOOP instruction, such as used in the 80×86 system, to simplify implementation of loops. Such instruction decrements a register and branches to a target address unless the result of the decrement is zero.

b. Case of Calling Subroutine

While the PC modification instructions jump/branch are relatively simple, the subroutine instruction brings additional complications. The basic requirement is that the programme flow is changed in such a way that control transfers to another routine, a subroutine, in such a way that programme flow can return to the point of departure having accomplished some useful function. The machine then executes the code that follows the subroutine call. The method of accomplishing the subroutine linkage can be very simple or quite complicated. To transfer control in a reasonable fashion, we must create a mechanism that will cause the programme counter to change so that instructions are fetched from the subroutine. At the same time, the linkage mechanism must provide a way to return to the calling routine. There are a number of methods which are used to provide this facility; in the following we will describe the most extensively utilized method in which s subroutine stack is used.

Systems that use this method will have a register designated as the stack pointer, which will control a stack in the memory of the machine. A subroutine call will push the return address onto this stack. The return reverses the process, popping the address from the stack to the programme counter. The stack is built in memory, which need not be shared with the programme, so the programme can be in ROM while the stack is in RAM. When multiple users are executing programmes on a single computer, then each user will have a private stack space and can share a single copy of the code. Since each call to a routine will

push a new return address onto the stack, recursive routines can be utilized as well. For these reasons, the stack method for establishing subroutine linkage is used by many systems.

4.13.1 Jumps, Branch and Call in AVR Architecture

As mentioned before, jumps can use any of the appropriate addressing modes to identify the target address. The target address is identified, by whatever combinations of addressing modes are available, and the specified address is placed in the programme counter. AVR uses three addressing modes for jump/branch and call.

a. Direct Programme Addressing: JMP and Call

Figure 4.28 shows how to access the programme memory directly.

Figure 4.29 shows **direct** memory addressing in `jmp` and `call` instructions, named long jump or long call, respectively.

b. Indirect Programme Addressing, IJMP and ICALL

For the **indirect** addressing used in ijmp and icall instructions, the programme execution continues at address contained by the Z-register (i.e. the PC is loaded with the contents of the Z-register). Figure 4.30 shows this addressing mode.

c. Relative Programme Addressing, RJMP and RCALL

The relative addressing used in `rjmp` and `rcall` instructions is effective for jumps or calls in a reduced memory area. The jumps or calls work relative to the programme counter (PC).

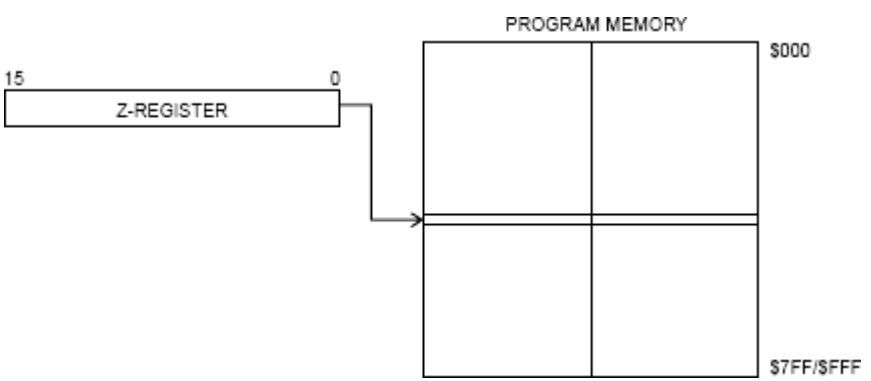


Figure 4.28 Accessing programme memory directly.

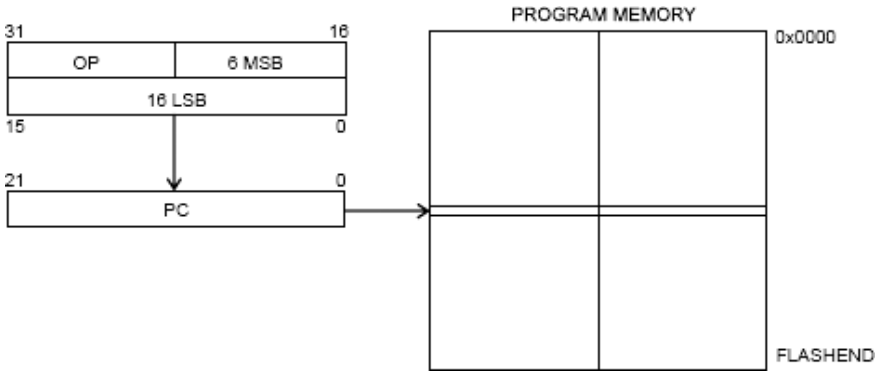


Figure 4.29 Direct programme memory addressing.

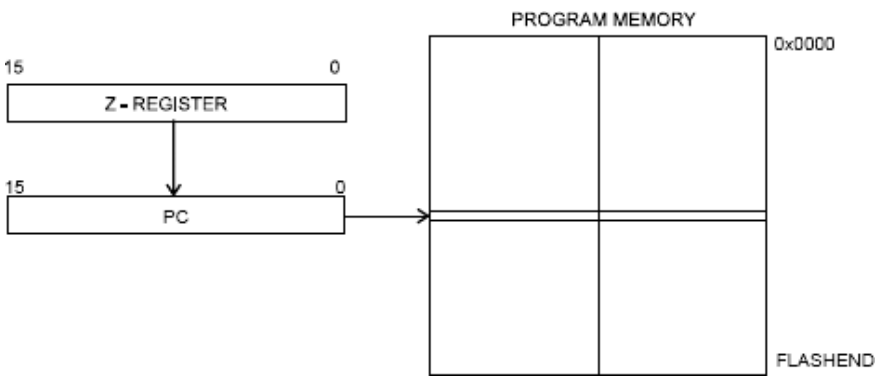


Figure 4.30 Indirect programme memory addressing.

Programme execution continues at address PC + k + 1. The relative address k is −2048 to 2047. Figure 4.31 shows this addressing mode.

Example 4.23: The instruction: JMP- Jump

```
mov r1,r0       ; Copy r0 to r1
jmp farplc      ; Unconditional jump
...
farplc: nop     ; Jump destination (do nothing)
```

Example 4.24: The instruction: CALL- Long Call to a Subroutine

```
mov r16,r0        ; Copy r0 to r16
call check        ; Call subroutine
```

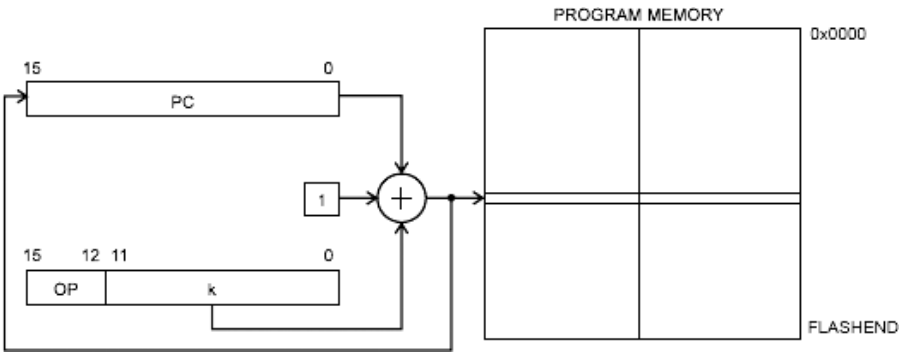


Figure 4.31 Relative programme memory addressing.

```
        nop                 ; Continue (do nothing)
        ...
check:  cpi r16, $42        ; Check if r16 has a
                              special
                            ; value
        breq error          ; Branch if equal
        ret                 ; Return from subroutine
        ...
        error: rjmp error   ; Infinite loop
```

4.14 I/O and Interrupts

The instructions discussed thus far have included mechanisms for doing data transformations (arithmetic and logical instructions), mechanisms for passing information (moves, etc.), and mechanisms for controlling the actions (programme control instructions). One of the areas not covered yet is the transfer of information to and from an external device (peripheral).This is generally called input/output processing (I/O) and normally involves more than transfer of data. Additional requirements include such things as testing of conditions and initiating action in an external device. Some of the I/O programming is in response to an external event signaling the processor that a device needs to be serviced. This signaling process is called an interrupt, and the processor responds to the interrupt in a predetermined fashion. If the signaling result from conditions detected internal to the processor (e.g. some form of overflow signals), sometime we call them traps instead of interrupts.

I/O processing has evolved from the very simple capabilities of the first machines to sophisticated mechanisms used in some machines available today. In its simplest form, I/O transfers data to or from an addressed device under the direct control of the processor. This concept is called programmed I/O. This concept uses specific instructions for input and output. Some of the microprocessors that use this concept include an additional control signal to indicate that the address appearing on the address lines is to identify an I/O device address, rather than a memory address. However, another technique, called memory mapped I/O, is perhaps more widely utilized. With this method, I/O devices are assigned specific locations in the address space of the processor, and any access to that address actually results in an I/O transfer of some kind. The memory mapped I/O scheme has the advantage that no special I/O instructions are required in the instruction set, and this reduces the complexity of the instruction decode mechanism. In addition, devices attached to the processor need not decode special signals to differentiate between memory and I/O requests. However, the fact that I/O instructions are included in an instruction set does not prevent the use of memory mapped I/O techniques in a system. The user of the system can decide which technique would be most appropriate for the goals of that particular implementation.

Concerning the responsibility signaling the processor that some peripheral device needs service and also the method by which information can be transferred between a processor and a peripheral device, we can recognize three techniques:

- Polling
- Interrupt, and
- Direct Memory Access (DMA).

In polling the machine will poll the I/O device until either the device has information for the system, or the I/O device can accept data from the system. The polling is done by reading the status register of the I/O device. When the status register indicates that transfers can occur, they are performed by writing/reading the appropriate memory location.

The polling mechanism is extremely inefficient in many circumstances. For example, if the processor issued a command to a tape drive to seek a particular file on a tape, a very long time will pass between issuing the request and having the device respond with the desired results. With the polling technique, the capabilities of the system are not available for anything else during the seek time. Therefore, it is more efficient to have the I/O device send a signal to the system when the action of a command (in this case the seek action) has been

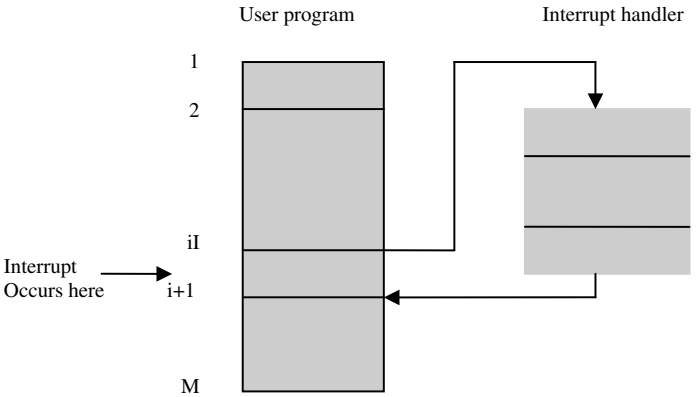


Figure 4.32 Transfer of control via interrupt.

completed. This signal is an interrupt, and it signals the processor to interrupt its current action and do something. What the system should do when it responds to an interrupt is defined in a routine called an interrupt service routine (ISR) or interrupt handler. The behavior of the system when responding to an interrupt is identical in many respects with the action of calling a subroutine. For the user programme, an interrupt is just that: an interruption of the normal sequence of execution. When the interrupt processing is completed, execution of the original programme resumes (Figure 4.32). Thus the user programme does not have to contain any special code to accommodate interrupts; the processor and the operating system are responsible for suspending the user programme and then resuming it at the same point. The instructions dealing with interrupts mimic the instructions involved in subroutine linkage.

To accommodate interrupts, the basic instruction cycle shown in Figure 4.4 has to be modified to accommodate an interrupt cycle (Figure 4.33). In the interrupt cycle, the processor checks to see if any interrupts have occurred, indicated by the presence of an interrupt signal. If no interrupts are pending, the processor proceeds to the fetch cycle and fetches the next instruction of the current programme. If an interrupt is pending, the processor does the following:

It suspends execution of current programme being executed and saves its context. This means saving the address of the next instruction to be executed (current contents of programme counter) and any other data relevant to the processor's current activity.

It sets the programme counter to the starting address of an interrupt handler routine.

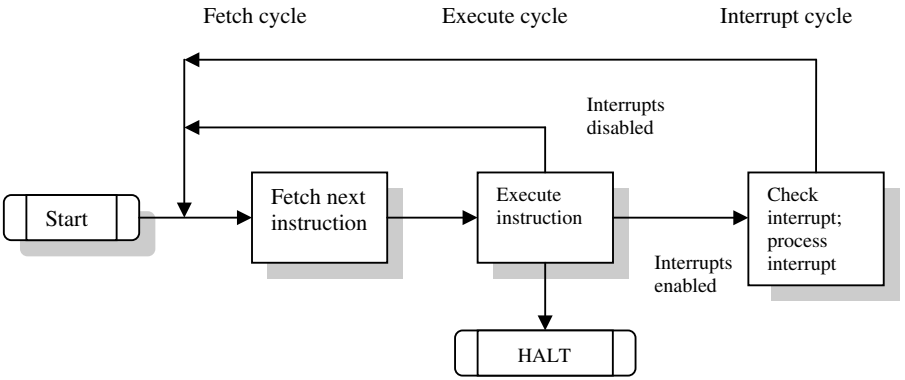


Figure 4.33 Instruction cycle with interrupts.

The processor now proceeds to the fetch cycle and fetches the first instruction in the interrupt handler programme, which will service the interrupt (this is why it is normally called interrupt service routine ISR). The interrupt handler programme is generally part of the operating system (in case of microcontroller-based systems, the ISR are part of the contents of the programme memory). Typically, this programme determines the nature of the interrupt and performs whatever actions are needed.

With polling (or programmed I/O) and interrupt I/O, the processor is responsible for extracting data from main memory for output and storing data in main memory for input. In some cases, it is desirable to allow I/O exchanges to occur directly with memory. In such a case, the processor grants to an I/O module the authority to read from or write to memory, so that the I/O-memory transfer can occur without tying up the processor. During such transfer, the I/O module issues read or write commands to memory, relieving the processor of responsibility for the exchange. This operation is known as direct memory access (DMA).

AVR Return from Interrupt Instruction

RETI — Return from Interrupt

Description: Returns from interrupt. The return address is loaded from the STACK and the Global Interrupt Flag is set.

Note that the Status Register is not automatically stored when entering an interrupt routine, and it is not restored when returning from an interrupt routine.

This must be handled by the application programme. The Stack Pointer uses a pre-increment scheme during RETI.

Example 4.25

```
            ...
extint:     push r0          ; Save r0 on the Stack
            ...
            pop r0           ; Restore r0
            reti             ; Return and enable
                             ; interrupts
```

4.15 Summary of Addressing Modes

Table 4.5 indicates the address calculation performed for each addressing mode and the limitations of each.

Addressing Modes of some Popular Processors

AVR AT90S8515	Motorola 68 K Family	Pentium
— Data register direct	— Data register direct	— Immediate mode
— Address register direct	— Address register direct	— Register operand mode
— Absolute	— Absolute short	— Displacement mode
— Register indirect	— Absolute long	— Base mode
— Post-increment register indirect	— Register indirect	— Base with displacement
— Pre-decrement register indirect	— Post-increment register indirect	— Scaled index with displacement
— Register indirect with offset	— Pre-decrement register indirect	— Base index with displacement
— PC-relative with offset	— Register indirect with offset	— Based scaled index with displacement
— Immediate	— Register indirect with index and offset	— Relative addressing
— Implied register	— PC-relative with offset	
	— PC-relative with index and offset	
	— Immediate	
	— Immediate quick	
	— Implied register	

Table 4.5 Addressing modes limitations.

Mode	Algorithm	Principal Advantage	Principal disadvantage
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory reference
Register	EA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference
Displacement	EA = A + (R)	Flexibility	Complexity
Stack	EA = top of stack	No memory reference	Limited applicability

4.16 Review Questions

4.1 What are the relative advantages and disadvantages of one-address, two-address, and three-address instruction format?
4.2 What is a register-to-register architecture and why is such architecture also called a load and store computer?
4.3 What are the three fundamental addressing modes? Are they all necessary? What is the minimum number of addressing modes required?
4.4 What does the RTL expression $[100] \leftarrow [50] + 2$ mean?
4.5 What does the RTL expression $[100] \leftarrow [50 + 2] + 2$ mean?
4.6 What is the operand?
4.7 In the context of an instruction register, what is the meaning of a *field*?
4.8 What is a literal operand?
4.9 What is the difference between a dedicated and a general-purpose register?
4.10 Why is the programme counter a pointer and not a counter?
4.11 Some machines have a one-address format, some a two-address format, and some a three-address format. What are the relative merits of each of these instruction formats?
4.12 Describe the action of the following assembly language instructions in RTL. That is, translate the assembly language syntax of the 8051 processor instruction into the RTL notation that defines the action of the

instruction:
(i) MOVE 3000,4000 (ii) MOVE #12,(A0)
(iii) MOVE #4000,5000 (iv) ADD (A3), 1234

4.13 Give examples (if available) of valid Intel Motorola 68K and AVR instructions that use:
(i) Register-to-register addressing
(ii) Register-to-memory addressing
(iii) Memory-to-register addressing
(iv) Memory-to-memory.

4.14 An address field in an instruction contains decimal value 14. Where is the corresponding operand located for:
(i) Immediate addressing? (ii) Direct addressing?
(iii) Indirect addressing? (iv) Register addressing?
(v) Register indirect addressing?

4.15 Let the address stored in the programme counter be designated by the symbol X1. The instruction stored in X1 has an address part (operand reference) X2. The operand needed to execute the instruction is stored in the memory word with address X3. An index register contains the value X4. What is the relationship between these various quantities if the addressing mode of the instruction is (a) direct; (b) indirect; (c) PC relative; (d) indexed?

5

Machine Language and Assembly Language

THINGS TO LOOK FOR…

- Machine language and Assembly language
- Directive and Micros
- Design of Assembley language Programme
- Program Template
- Data manipulation
- Generating time delays

5.1 Introduction

Before we can explore the process of writing computer programmes, we have to go back to the basics and learn exactly what a computer is and how it works. Every computer, no matter how simple or complex has at its heart exactly two things: a CPU and some memory. Together, these two things are what make it possible for your computer to run programmes. CPU and memories uses only binary numbers; CPU processes data given in binary form, and memory stores data and information in binary form.

On the most basic level, a computer programme is nothing more than a collection of numbers stored in memory. Different numbers tell the CPU to do different things. The CPU reads the numbers one at a time, decodes them, and does what the numbers say. For example, if the CPU reads the number 64 as part of a programme, it will add 1 to the number stored in a special location called AX. If the CPU reads the number 146, it will swap the number stored in AX with the number stored in another location called BX. By combining many simple operations such these into a programme, a programmer can make the computer perform many incredible things.

As an example, here are the numbers of a simple computer programme: 184, 0, 184, 142, 216, 198, 6, 158, 15, 36, 205, 32. If you were to enter these

numbers into your computer's memory and run them under MS-DOS, you would see a dollar sign placed in the lower right hand corner of your screen, since that is what these numbers tell the computer to do.

The numbers given above, accordingly, represent a language that is understandable by the computer (but it is difficult for the human to remember). It is actually one form of what we call "Low-level Programming Language".

In computer science, a low-level programming language is a language that provides little or no abstraction from a computer's instruction set architecture. The word "low" refers to the small or nonexistent amount of abstraction between the language and machine language; because of this, low-level languages are sometimes described as being "close to the hardware." It deals directly with registers, memory addresses, stacks and other components that are directly related to the hardware of the computer.

The low-level languages which made up of numbers are normally called the "object code" or "machine language" programme. On the other hand the programme which is written using alpha-numeric characters (more understandable for the programmer) is called the "source code".

Low-level languages are directly understandable by computer and, accordingly do not need a compiler or interpreter to run; the processor for which the language was written is able to run the code without using either of these.

High-Level Language

High-level programming language isolates the execution semantics of computer architecture from the specification of the programme, making the process of developing a programme simpler and more understandable. This makes high-level languages independent from the computer organization and architecture.

The term "high-level language" does not imply that the language is superior to low-level programming languages — in fact, in terms of the depth of knowledge of how computers work required to productively programme in a given language, the inverse may be true. "High-level language" refers to the higher level of abstraction from machine language. Rather than dealing with registers, memory addresses and call stacks, high-level languages deal with usability, threads, locks, objects, variables, arrays and complex arithmetic or Boolean expressions. In addition, they have no opcodes that can directly compile the language into machine code, unlike low-level assembly language. Other features such as string handling routines, object-oriented language features and file input/output may also be present.

Table 5.1 ATMEL AVR Directives.

Directive	Description
BYTE	Reserve byte to a variable
CSEG	Code segment
DB	Define constant byte(s)
DEF	Define a symbolic name on a register
DEVICE	Define which device to assemble for
DSEG	Data segment
DW	Define constant word(s)
ENDMACRO	End macro
EQU	Set a symbol equal to an expression
ESEG	EEPROM segment
EXIT	Exit from file
INCLUDE	Read source from another file
LIST	Turn list file generation on
LISTMAC	Turn macro expansion on
MACRO	Begin macro
NOLIST	Turn list file generation off
ORG	Set programme origin
SET	Set a symbol to an expression.

5.2 Directives: Pseudo-Instructions

The source code consists of assembler instructions, which has been explained in Chapter 4, and "**directives**". Directives are *pseudo-operations* that most modern assemblers also support, and represent directives obeyed by the assembler at assembly time instead of the CPU at run time. (For example, pseudo-ops would be used to reserve storage areas and optionally set their initial contents.) The names of pseudo-ops often start with a dot to distinguish them from machine instructions.

Table 5.1 shows a summary of all directives available for the ATMEL AVR assembler. All the directives listed in the table control the operations of the Assembler. There are directives for different purposes. We can identify the following groups of directives:

- The first group handles some memory and naming aspects and includes:
  - Space reservation: BYTE, DB, DW
  - Segmentation and origin: CSEG, DSEG, ESEG, ORG
  - Variables, constants and names: DEF, EQU, SET.
- The second group controls the list file generation and includes: LIST, NOLIST.
- The third group consists of more specialized directives such as:
  - Macro related directives: MACRO, ENDMACRO, LISTMAC
  - Miscellaneous: DEVICE, INCLUDE and EXIT.

To demonstrate the use of directives, a documented example programme is shown in List 5.1. The List 5.1 follows the programme template that will be discussed latter in the chapter.

Listing 5.1 Use of Directives

```
;***************************************************
; Directive Test
;***************************************************
.DEVICE        AT90S8515  ;Prohibits use of non-
                          ;implemented
                          ;instructions
.NOLIST                   ;Disable listfile
                          ;generation
.INCLUDE  "8515def.inc"   ;The included files will
                          ;not be shown
                          ;in the listfile
.LIST                     ;listfile generation
      rjmp    RESET       ;Reset Handle
;***************************************************
.EQU   tab_size = 10      ; Set tab_ize to 10
.DEF   temp = R16         ; Names R16 as temp
.SET   io_offset =0x23    ; Set io_offset t0 0x23
;.SET  porta=io_offset+2  ; Set porta to io_offset
                          ; +2 (commented
                          ; because defined in
                          ; 8515def.inc)
.DSEG                     ; Start data segment
Lable:.BYTE tab_size      ; reserve tab_size bytes
                          ; in SRAM
.ESEG                     ; Start EEPROM segment
Eeconst: .DB 0xAA, 0x55   ; Defines constants
.CSEG
RESET:  ser  temp         ; Initializes temp (R16)
                          ; with $FF
        Out  porta, temp  ; Write constants of
                          ; temp to Port A
        Ldi  temp,  0x00  ; Load address to EEPROM
                          ; address register
        Out  EEAR,  temp  ;
```

```
        Ldi   temp,  0x01  ; Set EEPROM read enable
        Out   EECR,  temp  :
        In    temp,  EEDR  ; Read EEPROM data
                           ; register
        Clr   r27          ; Clear X high byte
        Ldi   r26,0x20     ; Set X low byte by $20
        St    X, temp      ; Store temp in SRAM
Forever: rjmp forever      ; Loop forever
;***********************************************
```

5.2.1 Macros

Macros enable the user to build a virtual instruction set from normal Assembler instructions. You can understand a macro as a procedure on the Assembler instruction level.

The MACRO directive tells the Assembler that this is the start of a macro and takes the macro name as its parameter. When the name of the macro is written later in the programme, the macro definition is expanded at the place it was used. A macro can pond up to 10 parameters. These parameters are referred to as @0–@9 within the macro definition. When a macro call is issued, the parameters are given as a comma-separated list. The macro definition is terminated by an **.ENDMACRO** directive.

A simple example will show the definition and use of a macro:

```
.MACRO SUBI16                   ; Start macro
                                ; definition
       subi @1,low(@2)          ; Subtract low byte
       sbci @0,high(@2)         ; Subtract high byte
.ENDMACRO                       ; End macro
                                ; definition
.CSEG                           ; Start code segment
        SUBI16 r17, r16, 0x1234
```

A macro to subtract immediate a word from a double register named SUBI16 is defined. In two steps, Hi-byte and Lo-byte are subtracted from this double register. Subtracting the Hi-byte takes the carry flag into consideration.

When the macro SUBI16 is called, the symbolic parameters (@0, @1, @2) are replaced by registers or immediate data.

By default, only the call to the macro is shown on the list file generated by the Assembler. In order to include the macro expansion in the list file, a

LISTMAC directive must he used. A macro is marked with a + in the opcode held of the listfile.

The following extract from the listfile of the example programme shows this detail:

```
...........
        .MACRO SUBI16                   ; Start macro
                                        ; definition
               subi @1,1ow(@2)          ; Subtract low
                                        ; byte
               sbci @0, high(@2)        ; Subtract high
                                        ; byte
        .ENIDMACRO                      ; End micro
                                        ; definition
000001 e112    RESET: ldi r17, 0x12
000002 e304    ldi r16, 0x34
000003   +     SUBI16 r17,              ; Sub.0x1234
               r16,0x1234               ; from r17 r16
...........
```

The macro definition consists of two subtraction instructions. Before a call of this defined micro is possible, the double register must he loaded. These operations will be done by the two ldi instructions. After that, the macro call calculates the chosen subtraction.

5.2.2 ATMEL AVR Studio

AVR Studio enables the user to fully control execution of programmes on the AVR In-Circuit Emulator or on the built-in AVR Instruction Set Simulator. AVR Studio supports source-level execution of Assembly programmes assembled with Atmel's AVR Assembler and C programmes compiled with IAR's ICCA90 C Compiler for the AVR microcontrollers. AVR Studio runs under Microsoft Windows95/NT.

AVR Studio can be targeted toward an AVR In-Circuit Emulator or the built-in AVR Simulator. When the user opens a file, AVR Studio automatically detects whether an Emulator is present and available on one of the system's serial ports.

If an Emulator is found, it is selected as the execution target. If no Emulator is found, executions will be done on the built-in AVR Simulator instead. The

Status bar will indicate whether execution is targeted at the AVP In-Circuit Emulator or the built — in AVR Simulator

The user has full control of the status of each part of the microcontroller using many Separate windows:

- Register window displays the contents of the register file
- Watch window displays the values of defined symbols (in C programmes)
- Message window displays messages to the user
- Processor window displays information such as Programme Counter, Stack Pointer Status Register, and Cycle Counter
- Memory windows show programme, data, I/O and EEPROM
- Peripheral windows show 8-bit timer, I/O ports, and EEPROM registers

AVR Studio supports all available types of AVR microcontroller. The simulation with AVR Studio itself is more comfortable than simulation with the AVR Simulator, although the handling is quite similar.

5.3 Design of an Assembly Language Programme

The process of developing a programme goes through a sequence of steps (or phases). This sequence is shown by the flowchart given in Figure 5.1.

The structure of assembly language programmes follows the style known as linear programming. This is different from block-structured programming (Pascal, C) or object-oriented programming (C++, Java). The basic flow control of a linear programme can be expressed in the form of a flowchart, and will normally include sequential operations, decisions and jumps. To reduce the length and complexity of programmes the use of subroutines enables separation of the sets of operations and permits re-use of sections of the programme.

The programming model of any processor (microprocessor/microcontroller) may be seen, as explained while discussing RTL, as a collection of registers and memory locations. The memory locations can be located on the processor chip (internal memory) or outside the processor (external memory). The assembly language instructions are used to manipulate the contents of the different storage areas (registers, on-chip memory and external memory) to perform the data transfer operations (moving the data from one location to another) and the data transform operations (operating on the data by different arithmetic and logic operations). The system uses the different digital data paths and processing blocks that are physically contained on chip to achieve such transfer and transform operations.

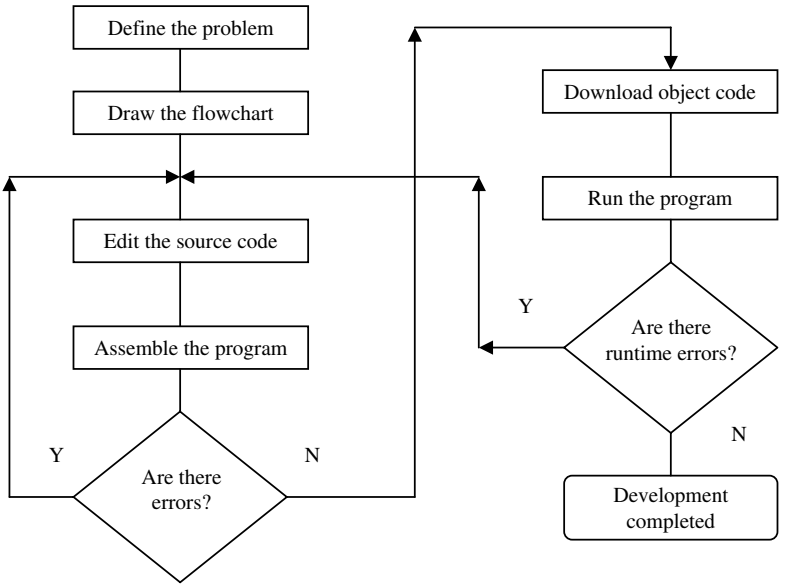


Figure 5.1 The programme design procedure.

Table 5.2 Elements of an assembly programme line.

Label	Op-Code	Operands	;Comment
Delay:	MOV	A, #Fh	;Put initial value (3Fh) in A
	INC	A	; Increment A by 1
	CLR	C	; Carry = 0
	DJNZ	Acc, Delay	

For more than trivial projects, it is advisable to commence the software design by drawing a flowchart. This enables visual checking of the overall logic and the flow through the actions of the programme. From this flowchart the assembly language programme can be developed.

5.3.1 The Basic Programming Method

5.3.1.1 Programme line format

Each line of an assembly programme can contain four elements: Lable, Op-code, Operands, and Comments. Table 5.2 shows some examples. The first line represents the case where all elements are present. The rest of the table shows cases where the label and/or comment have been omitted.

These elements are normally separated by spaces, and often the Tab key is used to align the columns vertically. The presence of a semicolon (;) indicates

that the remainder of the line is a comment, and will therefore be ignored by the assembler.

The label identifies a particular stage in the programme, and as mentioned before is a more convenient identifier than a memory address which may change as the programme is developed. A label at the beginning of a line must be followed by a colon (:), but when used as the operand of a **jump** or **call** instruction there is no colon.

Example 5.1

```
DELAY:          MOV A, 3Fh
REPEAT:         DEC A
                CJNE A, #21h, REPEAT
```

The comment is added to make clear the nature of the actions being programmed. Comments must begin with a semicolon (;). Entire lines may be comment lines by beginning them with the semicolon. Except in very short programmes, all the major stages and subroutines of a programme should be commented and any complex operations explained. Although the assembler does not regard the layout of an instruction on a line, it improves readability if the four elements are indented in a consistent manner.

5.3.1.2 Programme template

Most programmes will have a certain overall structure, and there are certain common elements needed for all programmes to work. Besides having a programme that is work, it is important to make it understandable for any reader and especially for the programmer in case if he wants to modify it or extend it. To achieve this target it is better to have a programme template that covers such needs and it is possible for the programmer to save it, and then load it every time he wants to start writing a programme. One of the recommended templates is shown in Figure 5.2.

- **The box made up of asterisks at the top** of the template is the programme header (the asterisks are there purely for decorative purposes). The contents of the box have no bearing on the actual functioning of the programme, as all the lines are preceded by semicolons. The contents help the main goal of the programme.
- The **'clock frequency:'** line refers to the frequency of the oscillator (e.g., crystal) that designer used in designing his system.
- In some applications (e.g. PWM and the calculation of the baud rate in case of serial communication) the programmer has to calculate the initial

```
;*************************************************************
; File Name:                                                 *
; Title:                                                     *
; Author:                                                    *
; Date:                                                      *
; Version:                                                   *
; File saved as:                                             *
; Target MCU:                                                *
; Clock frequency:                                           *
;*************************************************************
; Programme Function:_____________________________
;*****************************************************************************
; ********* Directives:
.device        at90s1200             ; Device is AT90S1200
.nolist
.include       "at90s1200.inc"
.list
;*****************************************************************************
; Declarations:
.def    temp    = r16
;*****************************************************************************
; Start of Programme
       rjmp Init                     ; first line executed
;*****************************************************************************
Init:  idi     temp, 0bxxxxxxxx      ; Sets up inputs and outputs on PortB
       out     DDRB, temp            ;
       Idi     temp, 0bxxxxxxxx      ; Sets up inputs and outputs on PortD
       out     DDRD, temp            ;
       Idi     temp, 0bxxxxxxxx      ; Sets pulls ups for inputs of PortB
       out     PortB, temp           ; and the initial states for
                                     ; the outputs
       Idi     temp, 0bxxxxxxxx      ; Sets pulls ups for inputs of PortD
       out     PortD, temp           ; and the initial states for
                                     ; the outputs
;*****************************************************************************
; Main body of programme:
Start:
       <Write your programme here>
       rjmp Start                    ;loops back to Start
;*****************************************************************************
```

Figure 5.2 Recommended templates.

values of some control registers that will let the system works as required by the specifications given to the designer. In the majority of the cases the frequency used is one of the parameters needed to calculate the initial values.

- The "**version**" and "**date**" are needed especially if the programme has more than one version.
- '**Target MCU**:' refers to which particular microcontroller the programme is written for. We used here AVR as example but it represents in general the microcontroller used.

- The second section is "**;Directives**". It starts by the directive "**.device**" which represents the beginning of the lines which actually do something. "**.device"** is a directive which tells the assembler which device the programmer is using. For example, if you were writing this for the 1200 chip, the complete line would be:

.device at90sl200

- Another important directive is "**.include**", which enables the assembler to load what is known as a *look-up file*. This is like a translator dictionary for the assembler. The assembler will understand most of the terms you write, but it may need to *look up* the translations of others.
- The path included is just an example assuming that 1200 is the microcontroller used.

.include "at90sl200def.inc"

- **".nolist"** and **".list"** — As the assembler reads the programme code, it can produce what is known as a *list file*, which includes a copy of the programme complete with the assembler's comments on it. If the programmer does not want this list file also to include the lengthy look-up file, the programmer therefore writes "**.nolist**" before the "**.include**" directive, which tells the assembler to stop copying things to the list file, and then you write "**.list"** after the "**.include"** line to tell the assembler to resume copying things to the list file. In summary, therefore, the "**.nolist**" and "**.list**" lines don't actually change the working of the programme, but they will make your list file tidier.
- "**;Decleration:"** — After the general headings, there is a space to specify some declarations. These are the programmer's own additions to the assembler's translator dictionary — this is an opportunities to give more useful names to the registers you will be using. For example, many programmers are always using a working register called temp for menial tasks. They can assign this name to R16 (for example). The programmer can define the names of the working registers using the "**.def"** directive, as shown in the template. Another type of declaration that can be used to generally give a numerical value to a word is "**.equ"**. This can be used to give your own names to I/O registers. For example, if the programmer has connected a seven segment display to all of Port B, and decided that he wishes to be able to write DisplayPort when referring to PortB. PortB is I/O register number 0x18, so he might write DisplayPort in the

programme and the assembler will interpret it as PortB:

.equ DisplayPort = PortB or
.equ DisplayPort = 0x18

- After the declarations, we have the first line executed by the chip on power- up or reset. In this line I suggest jumping to a section called **Init** which sets up all the initial settings of the AVR. This uses the **rjmp** instruction:

  rjmp Init ;

  This stands for relative jump. In other words it makes the chip jump to a section of the programme which you have labeled Init.
- The first part of the **Init** section defines the input and output pins, i.e. it sets which pins are going to act as inputs, and which as outputs. This is done using the Data Direction I/O registers: **DDRB** and **DDRD**. Each bit in these registers corresponds to a pin on the chip. For example, bit 4 of **DDRB** corresponds to pin PB4, and bit 2 of DDRD corresponds to pin PD2. As we shall see latter, setting the relative DDRX bit high makes the pin an output, and making the bit low makes the pin an input.

If we configure a pin as an input, we then have the option of selecting whether the input has a built-in pull-up resistor or not. This may save us the trouble of having to include an external resistor. In order to enable the pull-ups make the relevant bit in PORTx high; however, if you do not want them make sure you disable them by making the relevant bit in PORTx low. For the outputs, we want to begin with the outputs in some sort of start state (e.g. all off), and so for the output pins, make the relevant bits in PORTx high or low depending on how you wish them to start. An example should clear things up.

5.4 Use of Template: Examples

In this section we are showing how to use the template, and the instruction set, to write some simple programmes. More programmes are given in Section 5.7.

Example 5.2

In this example it is required to write a simple programme that can be used to turn on an LED and keeps it on.

Solution:

The required programme is given as Programme A. In this case we need only one output to which we are going to connect the LED. The programme assumes that the LED is connected to PB0.

Programme A — LED on

```
;***************************************************
; File Name:                                       *
; Date:                                            *
; Author:                                          *
; Version: 1.0                                     *
; File saved as: LEDon.asm                         *
; Target MCU: AT90S1200                            *
; Frequency: 4MHz                                  *
;***************************************************
; Programme Function: Turns an LED on
;***************************************************
; Directives
.device       at90s1200
.nolist
.include      "1200def.inc"
.list
;***************************************************
; Declarations:
.def temp     = r16
;***************************************************
; Start of Programme
       rjmp   Init          ; first line executed
;***************************************************
Init:  ser    temp          ; PB0 - output. This
                            ; instruction
                            ; moves 0b11111111
                            ; to r16
       out    DDRB, temp    ;
       out    DDRD, temp    ; PD0-7 all N/C
       clr    temp          ; all Port B outputs off
       out    PortB, temp   ;
       out    PortD, temp   ; all Port D N/C
;***************************************************
```

```
Start:
       sbi    PortB, 0    ; turns on LED
       rjmp   Start       ; loops back to Start
;***************************************************
```

Example 5.3

Write a programme which represents the following: An input pin is connected to a button and an output pin is connected to a LED. It is required to write a programme that turns the LED when the button is pressed, and turns off when it is released.

Solution:

The required programme is given: Programme B. In this programme we assumed AT90S1200 microcontroller is used. The same programme can be used with AT90S8515 after changing the corresponding directives.

Programme B — Push Button

```
;***************************************************
; File Name:                                       *
; Date:                                            *
; Author:                                          *
; Version: 1.0                                     *
; File saved as: PushA.asm                         *
; Target MCU: AT90S1200                            *
; Clock frequency: 4MHz                            *
;***************************************************
; Programme Function: Turns an LED on when a button
; is pressed
;***************************************************
; Directives
.device     at90s1200
.nolist
.include    "1200def.inc"
.list
;***************************************************
; Declarations:
.def temp   = r16
;***************************************************
```

```
;Start of Programme
      rjmp Init              ; first line executed
;************************************************
Init: ser  temp              ; P80 - output, rest
                             ; N/C
      out  DDRB, temp        ;
      ldi  temp, 0b11111110  ; PD0 - input, rest
                             ; N/C
      out  DDRD, temp        ;
      clr  temp              ; all Port B outputs
                             ; off
      out  PortB,temp
      ldi  temp, 0b00000001  ; PD0 - pull-up, rest
                             ; N/C
      out  PortD,temp        ;
;************************************************
Start:
      sbis PinD, 0           ; tests push button
      rjmp LEDoff            ; goes to LEDoff
      sbi  PortB, 0          ; turns on LED
      rjmp Start             ; loops back to Start
LEDoff:
      cbi  PortB, 0          ; turns off LED
      rjmp Start             ; loops back to start
;************************************************
```

Example 5.4: Extending Programmes B and C

In this example we are assuming that the AVR is connected as in Figure 5.3. The circuitry used is part of Atmel's development board for AVR microcontrollers.

LEDs in series with a current limiting resistor are connected to each pin of PortB (i.e. eight LEDs are connected to the Port). Pin2 of PortD serves as input where the simple push button with a pull-up resistor is connected.

In this case, after some initialization, the programme queries the push button. The programme waits until the push button is pressed. After the pressed key is detected, the bit pattern initialized after reset is changed and output to the LED afterwards. Now the programme waits until the key is released, and repeats the whole process endlessly.

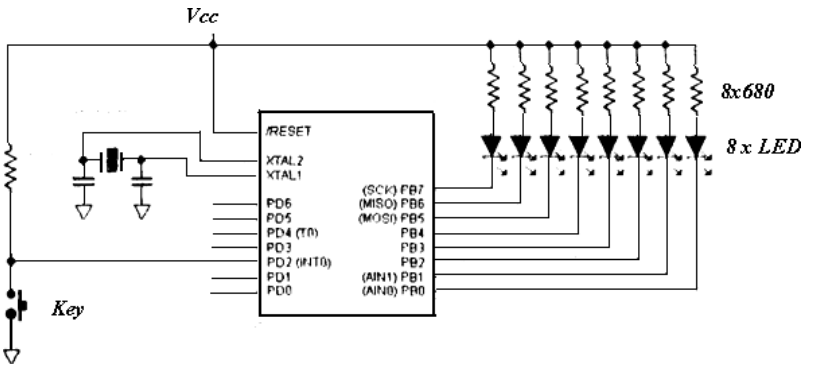


Figure 5.3 Configuration of Example 5.5.

The programme is shown in Programme C. After a descriptive header, the programme source starts with some directives. The first directive defines which device to assemble for. We choose the AT90S1200 (can be used also for AT90S8515. To use all I/O register names and I/O register bit names appearing in the data book and this description the file "1200def.inc" is included. The list and notlist at directives are described before. In the subsequent directive, the variable temp is defined and assigned to register R16.

Programme C: Push Button and LED display

```
;***************************************************
; File Name:        rnc_test0.asm
; Title:            Test and Display
; Date:
; Version:          1.0
; File saved as: PushDDisplayB.asm
; Target MCU: AT90S1200       ;Device is AT90S1200
; DESCRIPTION
;* Test an input Pin AND DISPLAY
;***************************************************
;**** Directives
.device     at90s1200
.nolist
.include "1200def.inc"
.list
.def        temp = r16
;***** Interrupt vector table****************
```

```
            rjmp    RESET       ;Reset handle
            reti                ;External Interrupt0
                                ;handle
            reti                ;Overfiow0 Interrupt
                                ;handle
            reti                ;Analogue Comparator
                                ;interrupt handle
;***** Main **********************************
RESET: ser  temp
            out     DDRB, temp  ;PORTB = all outputs
            sec
            ldi     temp, $FF
            out     PORTB,temp  ;Set bit pattern
loop:       sbic    PIND, 2     ;Wait tint key is
                                ;pressed
            rjmp    loop
            rol     temp        ;Rotate bit pattern
            out     PORTB,temp  ;Output rotated bit
                                ;pattern
wait:       sbis    PIND, 2     ;Wait until key is
                                ;unpressed
            rjmp    wait
            rjmp    loop        ;Repeat forever
;************************************************
```

5.5 Data Manipulations: Examples

As mentioned at the beginning of Chapter 4, the instructions of any processor (microprocessor/microcontroller) can be divided into groups; data manipulation instruction (data movement instruction), arithmetic and logic instructions group, branch instructions, etc. In this section and the following sections, we are going to give some programmes related to each group. In some cases we are going to give the programme in two versions; one version using AVR instruction set and the second version uses Intel 8051 instruction set. In many of the cases we are going to use the template to write the AVR programmes.

5.5.1 Copying Block of Data

We can use this example to show possible differences between different microcontrollers, e.g., AVR and Intel 8051. Let us consider the following

two examples:

Example 5.5

Write a programme that copies the contents of an area within the on-chip memory (between addresses 25h and 45h) to another area within the same memory (addresses 55h to 75h). The on-chip memory is called the internal RAM and in case of AVR is called internal SRAM.

Solution:

The programme in this case is as follows:

Possible Solution 1: Case of Intel 8051

```
START:  MOV     R0, #25h        ;
        MOV     R1, #55h        ;
LOOP:   MOV     A, @R0          ;
        MOV     @R1, A          ;
        INC     R0              ;
        INC     R1              ;
        CJNE    R0, #46,LOOP    ;
        SJMP    $               ;
```

Possible Solution 2: Case of AVR.

```
START LDI   R28,$55 ; Set Y low byte to the value
                    ; $25
      LDI   R26,$25 ; Set X low byte to the value
                    ; $55
LOOP  LD    R0, X+  ; Load R0 with the contents of
                    ; SRAM location $25
                    ; X post incremented, referring
                    ; to next location.
      ST    Y+,R0   ; Store R0 at SRAM location
                    ; $55. Y post
                    ; incremented referring to the
                    ; next location.
      CPI   R26,$46 ; Compare X low with $46
      BRNE  LOOP    ; Branch if R26 is not $46
      NOP           ;
```

Example 5.6

In this example it is required to copy the block of data in the on-chip memory in the range 25 h to 45 h to the external memory (RAM/ SRAM) addresses starting at 6000 h.

In case of AVR an access to external SRAM occurs with the same instructions as for the internal data SRAM access. When internal data SRAM is accessed, the read and write strobe pins (/RD and /WR) are inactive during the whole access cycle. The external data SRAM physical address locations corresponding to the internal data SRAM addresses cannot be reached by the CPU. External SRAM is enabled by setting the SRE bit in the MCUCR control register. More details are given in Chapter 6 on accessing external SRAM.

The programme in this case will be as that given in the previous example with one change at the first line. The first line:

```
START LDI R28,$55 ; Set Y low byte to the value $25
                  ; is replaced now with:
START LDI YH, $60h
      LDI YL, $00h
```

It is also possible to replace it (in this case) with:

```
  START   LDI YH, $60h
          CLR YL
```

5.5.2 Arithmetic Calculations

5.5.2.1 Using the stack for arithmetic expression evaluation

Example 5.7

The task of this example is to use the stack as temporary storage to evaluate the following expression for Z:

```
Z = (V + W) + (V + X) − (W + Y)
```

where the values of the variables are stored in the following addresses:

```
V: 50h W: 51h X: 52h Y: 53h
```

Put the final value of Z into location 60 h.

Solution

We are using this example to clarify some of the differences between CISC and RISC architectures by solving it using Intel 8051 and then by using AVR.

Intel is a CISC structure microcontroller. The memory is accessible by the majority of the instructions and at the same time the accumulator represents in most cases one of the operand. As an example to move data from memory location V (for example) the following instruction moves it directly to the accumulator:

MOV A, V

This is not the case with AVR which is a RISC structure microcontroller. All the general purpose registers (R0 to R31) are working registers, i.e. besides storing they are acting as accumulator. Accessing the memory is possible only by certain instructions.

Possible solution 1: Use of Intel 8051 Instruction set

The bracketed components of the expression are evaluated and pushed to the stack in the following order: (V+W), (V+X), (W+Y). They are then pulled from the stack in the reverse order to build up the complete expression to give the value of Z. The programme takes the following form:

```
V        EQU 50h
W        EQU 51h
X        EQU 52h
Y        EQU 53h
Z        EQU 60h
START:   MOV     A,V
         ADD     A,W
         PUSH    ACC  ; (V+W) is now on the stack
         MOV     A,V
         ADD     A,X
         PUSH    ACC  ; (V+X) is now on the stack
         MOV     A,W
         ADD     A,Y
         PUSH    ACC  ; (W+Y) is now on the stack
         POP     Z    ; Z contains (W+Y)
         POP     ACC  ; A contains (V+X)
         CLR     C
         SUBB    A,Z
         MOV     Z,A  ; Z contains (V+X) - (W+Y)
         POP     ACC  ; A contains (V+W)
         ADD     A,Z
         MOV     Z,A  ; Z contains (V+W)+(V+X)-(W+Y)
         SJMP    $
```

Possible solution 2: Use of AVR Instruction set

In this case we initialize the stack pointer as part of the general initialization following reset. This process will be considered in the next chapter while discussing the memory resources of the AVR microcontroller.

We are going to assume here that the variables V, W, X and Y are in registers r10, r11, r12 and r13 respectively.

```
START: push r10 ; V is now on the top of the stack
Push   r11      ; W is on the top of the stack
       ADD      ; (V+W) is now in the stack.
                ; V and W are no more in the stack
Push   r10      ; V is now in the stack
Push   r12      ; X is the top of the stack
       ADD      ; (V+X) are on the TOS. We must
                ; remember that (V+W) are at TOS-1
       ADD      : Add TOS and TOS-1.
                ; [V+X) + (V+W)] is now at TOS
       Push r11 ; W at TOS
       Push r12 ; Y at TOS
       ADD      ; Add TOS and TOS -1, i.e. get
                ; (W+Y)
                ; the result goes to the TOS. At
                ; the end of
                ; this instruction, the TOS
                ; contains (W+Y) and
                ; TOS-1 contains [(V+X+ + (V+W)]
       ADD      ; Add TOS and TOS+1. This gives the
                ; final
                ; result of the expression. It will
                ; stay at the TOS
```

5.5.2.2 Using a Look-up Table for Arithmetic Calculation

Example 4.8

Write a programme to give the value of any of the first 16 prime numbers using a look-up table with values 1, 2, 3, 5, 7, 11,…. If a number (between 1 and 16) is put into the accumulator A, then the programme should put the prime number at that position on the list into A. (For example, 6 would return the sixth prime number, which is 11). Two possible methods of programming

this operation are given below,

Possible solution 1: Using Intel 8051

Put the list of prime numbers into code memory using the DB pseudo-instruction. Then use MOVC and DPTR to access the table and find the answer.

The programme can take the following form:

```
        ORG     0000h
MAIN:   MOV     DPTR, #0200h
        MOVC    A,@A+DPTR
        SJMP    $
        ORG     0200h
TABLE:
DB      0,1,2,3,5,7,11,13,17,19
DB      23,29,31,37,41,43,47
```

5.5.2.3 Signed arithmetic calculations

The programme-designer can decide whether the data values in the programme should represent unsigned or signed numbers (or other data types, such as ASCII values). If signed representation is used, then the 8-bit numbers can only have the range −128 to +127. The MSB (bit 7) shows the sign of the number, with zero meaning positive, and one indicating negative. If negative, then the 7-bit value must be presented in two's-complement form. The following examples illustrate this concept:

```
01101100 = +1101100b = +6Ah = 106
10101101b = −1010011b = −53h = −83
```

The ADD, ADDC and SUBB instructions can be used with signed numbers in much the same manner as with unsigned integers. The carry flag (C) will indicate a carry in the case of addition and a borrow when using the SUBB instruction.

However, there is a possibility that the result of an arithmetic operation will exceed the limits of the signed number. Therefore the overflow flag (OV) will be set if the value in the accumulator lies outside the −128 to +127 range (−80 h to +7 Fh). The following programme illustrates the addition and subtraction of both positive and negative signed numbers, with comments to indicate the values:

```
MOV     A, #24h         ; A = 24h
ADD     A, #24h         ; A = 48h
```

```
    CLR     C
    SUBB    A, #30h          ; A=18h
    CLR     C
    ADD     A, #10101011b    ; Add -55h; A = -3Dh
    CLR     C
    SUBB    A, #10011011b    ; Subtract -65h A= +28h
```

5.5.2.4 Multi-byte unsigned calculations

The limitations of the 8-bit data size on the MCS-51 devices can be overcome by spreading the data value of a number over several bytes. The most common is to use a 16-bit number stored in two successive memory locations, allowing values up to 65535. Addition and subtraction can then be performed, making use of the carry flag.

For addition, the ADDC instruction will include the value of the carry in the result, thus making multi-byte addition possible. For subtraction, the value of the carry is automatically subtracted from the result. The following examples show how two 16-bit numbers can be added and subtracted.

Example 5.9

The number 7E04h is stored in locations 30h (low byte) and 31h (high byte), while the number 50EDh is stored in locations 40h (low byte) and 41h (high byte). Write a programme to:

1. Add these two numbers, and put the result in locations 50h (low) and 51h (high).
2. Subtract these two numbers and put the result in 60h (low) and 61h (high).

Possible solution:

```
; Addition of 16-bit numbers
    MOV     A, 30h
    ADD     A, 40h  ; Add low bytes
    MOV     50h, A  ; Store the low-byte result
    MOV     A,31h
    ADDC    A, 41h  ; Add high bytes + carry
    MOV     51h, A  ; Store the high-byte result
; Subtraction of 16-bit numbers
    MOV     A, 30h
    CLR     C
    SUBB    A, 40h  ; Subtract low bytes
```

```
MOV     60h, A  ; Store the low-byte result
MOV     A,31h
SUBB    A, 41h  ; Subtract high bytes-carry
MOV     61h, A  ; Store the high-byte result
```

5.5.2.5 Logical expression evaluation

In the following section we are showing how to use the microcontroller to implement a combinational logic. Two examples are given; one uses AVR and the second uses Intel microcontroller.

Example 5.10

Assume that it is required to implement the following function:

$$Z = \overline{A}.B + \overline{B}.A + C.\overline{D}$$

We are going to use this example to show the following:

1. How to use the microcontroller to implement combinational logic, and
2. To illustrate again the idea of selecting the proper processor for implementing the required device, we are going to introduce here three possible solutions:
   i. Use of dedicated digital circuit (equivalent to use of single purpose processor),
   ii. Use of Programmable Logic Device (PLD),
   iii. Use of microcontroller.

Use of Dedicated Circuit:

Here the dedicated circuit is using discrete ICs for implementing the various logic operations: AND, OR, etc. Figure 5.4 represents the required circuit.

Use of PLD

A PLD, as mentioned in Chapter 2, contains an array of logic function blocks. The designer selects the required functionality and the interconnections between these functional blocks. The PLD implementation of the given logic expression is shown in Figure 5.5.

Use of Microcontroller

The microcontroller can be used to implement the logic function. Figure 5.6 shows that the four variables A, B, C and D are connected to the PB0, PB1, PB2, and PB3 pins of a microcontroller. Pin PB5 gives the result Z.

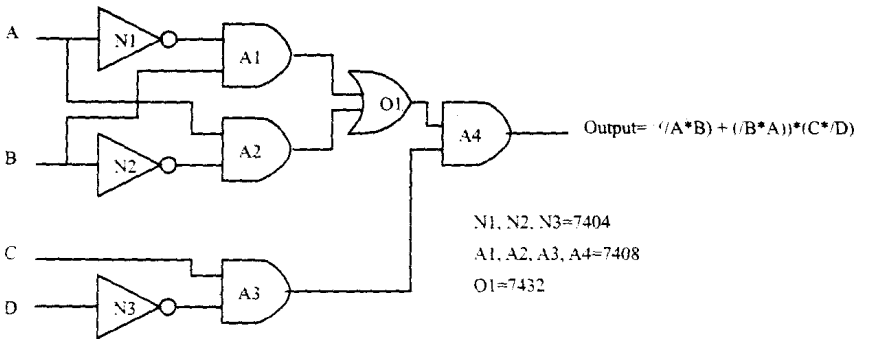


Figure 5.4 Use of ICs for implementation.

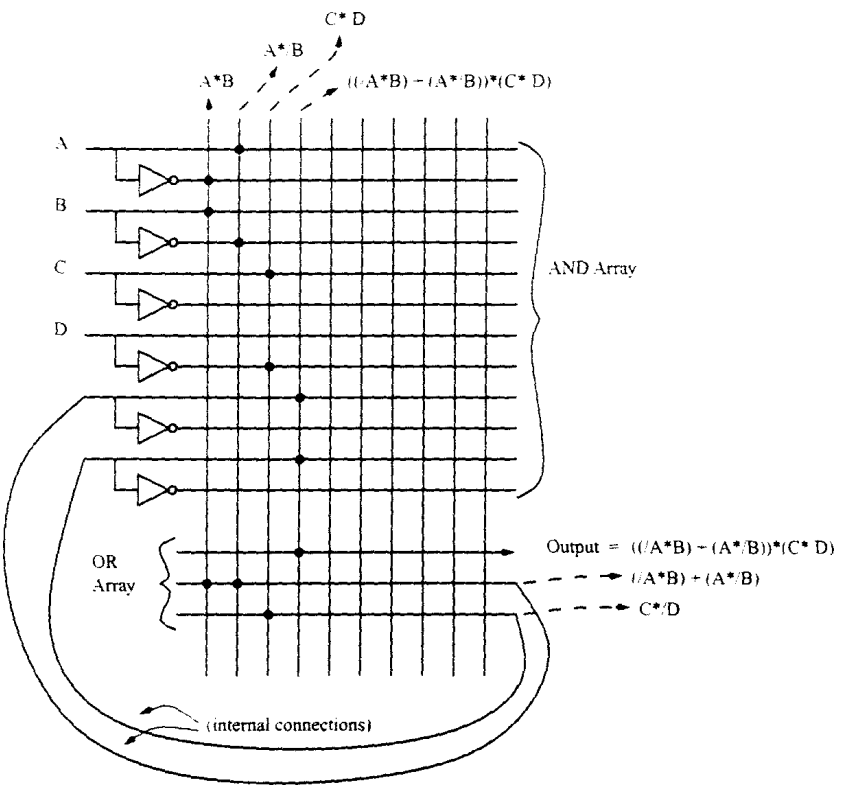


Figure 5.5 Use of PLD.

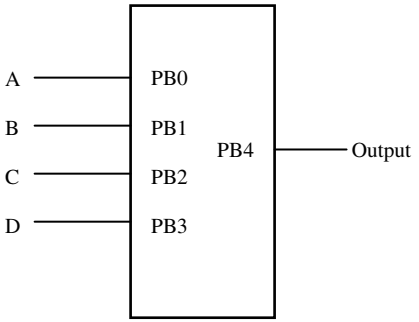


Figure 5.6 Use a Microcontroller for implementing logic function.

The programme that implements the equation using AVR instruction set is the subject of question 5.4.

5.5.3 Software-generation of Time Delays

5.5.3.1 Generating short time delays

In some cases no internal timer can he used to generate time delays. Software loops with included nop instructions or without any instructions can help to kill time.

The time delay must be calculated by the number of instruction cycles times the execution time for one instruction or clock cycle. If the delay loop is interrupted, then the execution time for the interrupt handler must he added. In interrupt-controlled applications, therefore, no exact time prediction is possible.

Because a simple loop allows only 256 runs through the loop, in general combined loops are used. Figure 5.7 shows a flowchart of two combined loops.

If the inner loop is initialized with the value n and the outer loop with m, then $m * n$ loop cycles are possible in all with total delay time depends on the instructions within each loop and the number of clocks needed to execute each instruction.

In the following we are starting by writing a subroutine that will run for a given period of time before returning and then we give another programme that uses the subroutine to generate a longer delay time.

Example 5.11

If the oscillator clock frequency is 12 MHz, design a subroutine that will run for exactly 100 ms before returning. Use Intel 8051 Instruction set.

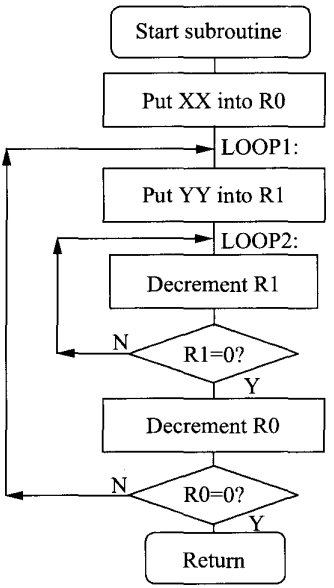


Figure 5.7 Flowchart for time delay subroutine.

Solution

The method used here is to design a programme with a set of nested loops. Each loop will decrement a register once each time it is executed. Initially the values of data required in the registers are not known. Therefore values called XX, YY,… are used in writing the programme, and a time budget is done to calculate the total time in terms of the unknown variables. This is done using the "cycles" column in the Instruction Set sheets of the microcontroller. Then the time is adjusted to the required length by putting suitable values into the registers. In this case two loops will be used, but if these are insufficient more can be added. The flowchart for the decrementing the loops is shown in Figure 5.7.

The assembly language programme corresponding to the flowchart is as follows.

```
START:      MOV     R0, #XX
LOOP1:      MOV     R1, #YY
LOOP2       DJNZ    R1, $
            DJNZ    R0, LOOP1
            RET
```

Table 5.3 Time budget for the subroutine.

Source code			Instruction cycles #	No of times executed	Total clock cycles
START	MOV	R0,#XXh	12	1	12
LOOP1:	MOV	R1,#YYh	12	XX	12*XX
LOOP2:	DJNZ	R1,LOOP2	24	XX*YY	24*XX*YY
	DJNZ	R0,LOOP1	24	XX	24*XX
		RET	24	1	24
			Total number of cycles		$12 + 12 * XX + 24 * XX * YY + 24 * XX + 24$

In practice, the label LOOP2 is not required, since the $ means jump to the same address. But it is included here to help clarify the operation of the programme.

The next step is to calculate the values of XX and YY representing the number of repeating the outer and inner loops correspondingly. This can be achieved by preparing from the programme a time budget is prepared. The time budget takes into consideration (1) the number of cycles per instruction, (2) the number of times each instruction is executed and (3) the total cycles used by that instruction. This is shown in Table 5.3.

Ignoring the first and last instructions, which are only executed once, the total number of cycles taken by the subroutine is

$$12 * XX + 24 * XX * YY + 24 * XX$$

The required time for the subroutine is 100 ms, which for a clock frequency of 12 MHz requires 12×10^5 cycles.

Therefore

$$12 * XX + 24 * XX * YY + 24 * XX = 12 \times 10^5$$
$$3 * XX + 2 * XX * YY = 10^5$$

Choosing a value for XX, say 255 (FFh)

$$3 * 255 + 2 * 255 * YY = 10^5$$
$$YY = [10^5 - 3 \times 255]/[2 \times 255] = 194.57$$

Obviously the fractional part of the number will not be stored in the microprocessor memory and the value of YY should be treated as 195, which in hexadecimal is C3h. The complete subroutine can now be written as follows:

```
START:      MOV R0, #0FFh
LOOP1:      MOV Al, #0C3h
```

```
DJNZ        R1, $
DJNZ        R0, LOOP1
RET
```

If the value of YY had come out to be very small, the rounding error would have been significant, and it would have been necessary to reduce XX accordingly. In the present case both XX and YY are large enough for the rounding error to be ignored.

If the value of YY is larger than FFh, it means that the required delay cannot be achieved with two loops, and the method of the next section will have to be employed.

5.5.3.2 Generating long time delays

From the previous section it can be seen that using two loops imposes an upper limit on the length of time delay that can be generated. If XX and YY are both made to be 255, then the above programme can produce a maximum delay of:

$$\begin{aligned} & 12 * 255 + 24 * 255 * 255 + 24 * 255 \\ & \quad = 36 * 255 + 24 * 255 * 255 \text{ clock cycles} \\ & \quad = 3 * 255 + 2 * 255 * 255 \text{machine cycles} \\ & \quad = 131835 \text{ machine cycles} \end{aligned}$$

If the oscillator frequency is 12 MHz, then one machine cycle will be 11 micro s, resulting in a maximum delay of 132 ms.

The way to increase the delay time is to add another nested loop, using initial values XX, YY, ZZ. For such a programme with three loops, the innermost loop will be executed XX $\times$ YY $\times$ ZZ times. Longer times can be achieved with additional loops. Minor adjustments can be made by placing NOP operations in the programme.

Example 5.12

Write a complete programme for time delay generation using AVR Instruction Set and the standard template.

Solution

The next programme shows the implementation of a software-based time delay in the subroutine part. At the beginning of this subroutine, the variable delcnt is initialized with zero fix. The variable *dly* must be set before the function call. In programme delay.asm the variable dly is set to FFh before the function call. In general *dly* depends on the instructions included in

the programme, the required delay time and the clock frequency. The following formula can be used in our case.

$$\text{Delay} = [1026 * dly + 4]/f_{clock}$$

Under the setup conditions in the given programme, we can calculate a delay of 65.4085 ms for an AVR microcontroller with a clock frequency of 4 MHz. For longer delays, one or more loops must be connected together.

```
;************************************************
; File Name:    delay.asm
; Title:        Software delay
; Date:
; Version:      1.0
; Target MCU    AT90S1200
; Description:
; Software delay with counted loops
;************************************************
;********Directives ******************************
.device at90S1200           ; Device is AT90S1200
.notlist
.include "1200def.inc"
.list
.def            temp = r16
.def            dly = r17
.def            delent = r18 ;Loop counter
;********Interrupt vector table

                rjmp RESET   ;Reset handle
                reti         ;External Interrupt0
                             ;handle
                reti         ;Overflow0 Interrupt
                             ;handle
                reti         ;Analogue Comparator
                             ;Interrupt handle
;************Subroutines ************************

DELAY: clr      delent       ;Init Loop Counter
LOOP1: dec      delent
```

```
                nop
                brne    loop1
                dec     dly
                brne    loop1
;********main ************************************
RESET:          er      temp
                Out     DDRB, temp ;PORTB = all
                                   ;outputs
                Out     PORTB, temp
                Ldi     dly, $FF   ;Initialize delay
                                   ;of about 65 ms
Loop:           bi      PORTB, PB0
                Ldi     dly, $FF   ;Initialize delay
                                   ;of about 65ms
                rcall   DELAY      ;Number of cycles
                                   ;= 1026*dly+4
                Sbi     PORTB, PB0
                Ldi     dly, $FF   ;Initialize delay
                                   ;of about 65ms
                rcall   DELAY      ;Number of cycles
                                   ;= 1026*dly+4
                rjmp    loop       ;Repeat forever
;************************************************
```

5.5.3.3 Disadvantages of software-generated time delays

The above approaches to producing time delays all make use of the concept of "busy waiting", which means that the CPU is fully occupied throughout the time period. Often microcontroller applications require time measurement or delay to take place in the background, while the CPU carries out more urgent tasks. For this reason most microcontrollers have in-built hardware timer circuits, which allow the programme to start a timer and continue with other tasks until interrupted when the time is complete. The MCS-5 1 devices have two timers, known as Timer 0 and Timer 1. These are started and stopped by short instructions in the main programme.

5.5.3.4 Use of timers to generate time delays

In Chapter 9 we are discussing the use of timers to generate a long delay.

5.6 Summary of the Chapter

In this chapter we have looked at how the reader can use a microcontroller/microprocessor instruction set to write simple programmes using assembly language. We have examined some of the AVR and Intel instructions and demonstrated how to write a programme. The use of the stack is also introduced.

5.7 Review Questions

5.1 Write a programme that copies the block of data in external memory at addresses 8000 h ~807 Fh to internal memory addresses 80 h~FFh

5.2 By means of a look-up table in code memory, evaluate the cube of any number in the range 1 to 6.

5.3 Two 32-bit numbers are each stored as four bytes in internal SRAM, with the MSB having the highest address. The numbers are 35A74F56h (locations 30 h~33 h) and 6A12D5CCh (locations 40 h~43 h). Write a programme to add the numbers and store them in address range 60 h~60 h.

5.4 For example 5.12 write the programme that implements logic expression using Figure 5.6.

5.5 Write a programme that takes the numbers 1 to 9 and exclusive-OR each one with the data at address 5 Ch. The programme puts the results in the same order in locations 71 h to 79 h.

5.6 Use stack to evaluate the following logic expression (in which the operators are Boolean such as AND, OR, EOR), and put the result in one of the general registers.

$$G = ((J + K)X(L + M)) \oplus (J + M)$$

5.7 Write a programme that counts the number of times a push button is pressed.

6

System Memory

THINGS TO LOOK FOR…

- The memory as the centre of the computing system model
- The different types and classes of memories
- Semiconductor memories- SRAM and DRAM
- Memory interfacing
- Timing diagrams
- AVR and Intel microcontroller memory systems

6.1 Introduction

Any computing system (microprocessor or microcontroller- based or general purpose computer) can be defined as a state machine that combines three components (Figure 6.1); memory, processor, and an I/O system. The memory is the centre of this model. It is possible, because of this centrality, to describe the computer as a memory–centered system. The memory is the space that holds information consists of programmes and data. The programme space stores the instructions of the programme in execution, the operating system, compiler, and other system software. In this model, the information in memory at any instant represents the process state at that instant. The information (data and instructions) flow to the processor (the logic in the figure) where it is processed and modified according to the instructions, and then the new modified data (and in some designs the instructions) returns back to the memory. Storing the new data in the memory updates the system state. This flow is shown in Figure 6.1 as state machine. The data arriving to the computer system from its inputs have to be stored at first at the memory to become part of its contents. Any information needed from the computer must come from the information stored in the memory and the system uses the output devices to provide the external world with the needed information.

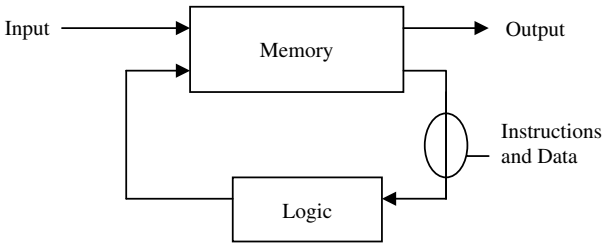


Figure 6.1 Computer as State machine.

As the centre of the computer system, the memory has a significant effect on many of the design metrics discussed in Chapter 1:

- Performance: The use of slow-memory (i.e. large access time) degrades the overall performance of the system. Slow-memory may create "memory-bottlenecks", which causes the processor to work below its performance capabilities.
- Software Support: The capacity of the memory limits the use of the computer; less memory capacity limits the programmes and the applications that can run on it.
- Reliability and Stability: The design of the memory defines the probability of finding one bit or more in error. Bad memory means high probability of bit in error, which means less reliable system. It is very important to use high-quality, error-free memories to insure smooth running for the system.
- Upgradability: Because of the importance of the memory capacity, as mentioned before, it is important during the system design to use some form of universal memory. This allows the user, when the needs arise, to upgrade the current memory if the motherboard allows, or to replace it with newer ones.

The selection of the memory unit(s) for a given application is a series of tradeoffs, since a number of different factors must be considered. These include: memory size, datapath width (bus width and word width), access time, the write ability and the organization method. The speed of the memory depends on many factors, including technology of implementation and organizational method.

This chapter familiarizes the reader with many aspects related to the memory and memory systems. The memory systems of AVR and Intel microcontrollers are used as case studies.

6.2 Memory Classification

The memories can be classified according to many parameters. Some of the parameters are:

1. **Memory Technologies:** Many materials can be used to build the memory.
    - Magnetic material: Used to build hard disk, magnetic tapes, etc.
    - Semiconductor flip-flops: Used to build static random access memory (SRAM),
    - Charges on a capacitor: Dynamic memory DRAM stores the data as a charge on capacitor.
    - Optical: CD ROM and DVDs
2. **Way of identifying the data**, or way of calling a needed piece of data (normally a word): The following main techniques are used:
    - Identification by address: Each word is stored in a location that is identified with a unique address as in case of RAM, ROM, etc.,
    - Identification by contents: The data rather than being identified by the address of its location, it is identified from properties of its own value. To retrieve a word from associative memory, a search key (or descriptor) must be presented which represents particular values of all or some of the bits of the word. Such type of memory is called content addressable memory or associative memory. Some cache memories are fully associative memories.
3. **Write ability**: Possibility of Read and Write. This term is used to indicate the manner and the speed that a particular memory can be written. Three wide classes can be recognized:
    - Read/write memory (normally called RAM): Any memory device whose contents can be modified by a simple write instruction is a read/write memory (we know that all memory devices are readable). The majority of the semiconductor and magnetic memories are read/write memories.
    - Permanent memory (also Non-erasable Memory): The contents of this class cannot alter or erased. The data stored in it during manufacturing and during its lifetime we read it only. This class retains its contents regardless of power supply (i.e. it is nonvolatile). Many forms of Read-only memory (ROM) are permanent. This type of memory is used in computers to store operating systems and the firmware.

- Read mostly memory: This is some form of ROM that can be reprogrammed, out-of system, from time to time to change its contents. Erasable PROM (EPROM) and Electrically EPROM (EEPROM) are some examples.

4. **Volatility**: Refers to the ability of memory to hold its stored bits after those bits have been written:
   - Nonvolatile Memory: Is the memory that retains its contents when the system is switched off or when there is a power failure. All currently used forms of magnetic storage and optical disk storage and also the ROMs are nonvolatile.
   - Nonvolatile memories may split into two types:
     - **Erasable**, e.g., EPROM, EEPROM and magnetic memories.
     - **Nonerasable**: this is the memory range that will essentially never lose its bits-as long as the memory chip is not damaged, e.g. ROM and PROM.
   - Volatile Memory: The data in such class of memory stays there only as long as the memory is connected to the power.
5. **Destructive or Non Destructive Read Out**: This parameter describes the effect of the READ operation on the information stored. In Ferrite Core-memories, the read operation destroys the stored information under consideration. The read operation must be followed by rewrite operation. Such memory is called Destructive Read Out (DRO) memory. Most of the Semiconductor memories, the magnetic drums, disks, and tapes are nondestructive. The read operation in this case will not affect the stored information. Dynamic RAM and semiconductor latches (buffers) accordingly is a destructive read storage
6. **Static or Dynamic**:
   - Static RAM (SRAM): is semiconductor memory which holds its contents so long as there is a power supply. Data can be changed by over-writing it with new data. The data is stored in the form of a state of a flip-flop.
   - Dynamic RAM (DRAM): DRAMs store data in the form of an electronic charge on the inter-electrode capacitance of a field effect transistor. This capacitor is not ideal, accordingly it starts to leak the charge stored on it. Because of this discharging effect, it is possible that the cell lose the data stored in it. To avoid that, dynamic memories are to be refreshed regularly. Refreshing is used

to restore the charge on the capacitors. This takes place by reading and rewriting the contents frequently. This operation is known as memory refreshing. DRAMs are more widely used than SRAMs because they are much cheaper and need less power compared with the same capacity SRAMs.
- Both Static and dynamic RAM are volatile. EEPROM, flash memory and all the memories that use magnetic material as storage media are non volatile.

7. **Memory Time response** (Memory Timing):

   - Random access, e.g. all semiconductor memories.
   - Sequential (or serial) access, e.g. magnetic tap
   - Direct access, e.g. hard disk.

   This will be discussed next when discussing memory access time.
8. **Memory Location**: The term location here refers to whether memory is internal or external to the computer. Internal memory is often called main memory or inner memory. But there are other forms of internal memory. The processor requires its own local memory, in the form of registers or register file. Further the control unit portion of the processor may also require its own internal memory. Internal memory covers also the cache memory. External memory consists of peripheral storage devices, such as disk, tape, and CD ROMs, that are accessible to the processor via I/O controller. Storage is the general term that covers all units of computer equipment used to store information (data and programmes).

Figure 6.2 illustrates some of the currently available memories classified according its location (external or internal), technology and some other physical characteristics. Figure 6.4 considers the various types of memories that use semiconductor as technology.

6.3 Memory Response Time

Accessing the memory to read or to write a specific procedure must be followed. The procedure that control memory accessing consisting of:

- *Location selection*: The memory controller supplies the system with the correct address of the location needed to be accessed.
- *Operation signal*: The memory controller generates another control signal defining the nature of operation needed: Read or Write.

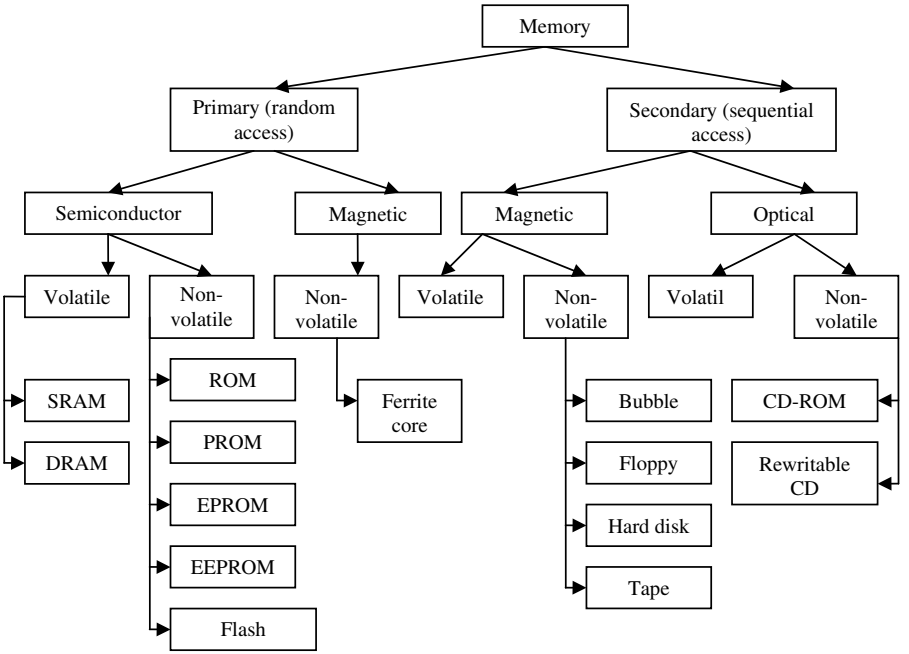


Figure 6.2 Classess of memory.

- *Data release* (retrieve) (case of read): The memory responds by providing the data stored in the defined location which will show up on the data bus. The data reaches the CPU or, in general, the device requested it.

This procedure must be followed each time the memory is accessed. The time needed to complete this procedure is called "***memory response time***". Two parameters can be used to measure the memory response time (Figure 6.3):

- **Access Time** (also called **response time**): This time defines how quickly the memory can respond to a read or write request. Access time consists of two parts:
  - **Latency:** represents the time taken from the instant the instruction decoder sends asking for certain memory location till this location is found without reading the contents.
  - **Data transfer time:** This represents the time between finding the required location till the data stored at this location be available for the CPU.

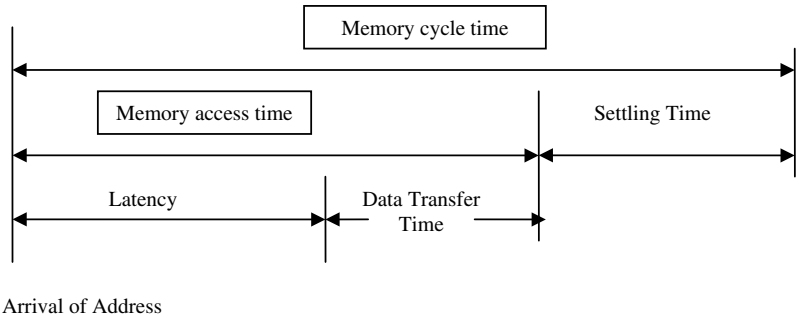


Figure 6.3 Memory response time.

In case of random-access memory, the access time is the sum of the two components, while in case of non-random-access (e.g. sequential access), the latency represents the access time.

- **Memory Cycle Time**: This is measured by the minimum period between two successive requests of the memory. Compared with access time, memory cycle time takes into consideration the time required for the signals to settle on the data bus; "settling time". Many memory types have identical access time and cycle time.

The "read access time" and "write access time" are almost the same for the majorities of the semiconductor memories but not the same for other memories.

Concerning the response time, normal memories (that use addresses for identifying the locations of the data stored within the memory) are classified into random access, sequential address and direct address.

6.3.1 Random Access (also, Immediate Access)

In this type of memory, each location is described by a unique address. Accessing the memory needs the controller to generate the address of the needed location. A memory address register (MAR) receives the address. This register, which has the same width n as the address bus, is connected to the inputs of an n inputs address decoder. The decoder has 2^n output wires representing all the locations available in the *memory space*. The decoder represents a hard wired mechanism mapping an address to unique location. The time needed for this mapping is fixed, i.e. completely independent on the location within the memory or what happened before calling this specific location. This means that any location in the memory space that selected at random is accessed in a fixed time.

Any memory with random read cycle, it is also random for write cycle. It is generally called *random access memory* (RAM). We must note here that the term RAM has no relation with the write-ability of the memory. It is related to the access time. Unfortunate this term is used wrongly to describe read/write ability of the memory. As a matter of fact ROM is also RAM if we consider the correct definition of the term "random access".

The semiconductor memories are, generally, random access memories.

6.3.1.1 Associative accessing

The content addressable memory (or associative memory) is random access. The word in such class of memories is retrieved based on its contents (or part of the contents) rather than its address through a search operation (a form of compare operation). In case of RAM we used the address decoder as a mechanism linking physically each location to a unique address. In associative memory a search mechanism is linked with each location the matter that makes the retrieve time constant for all the possible locations. It is correct to say, accordingly, that associative memory is a random access memory.

6.3.1.2 Burst mode access

Modern computers use what is known as "burst mode" access to improve the memory performance. Instead of accessing 64-bit unit, the system in one access reads four consecutive 64-bit units of memory. It is needed only to supply the address of the first unit. The great advantage of the burst mode is that most of the overhead of the first access does not have to be repeated for the other three. If the first 64-bit unit needs 5 clock cycles, the remaining three need 1 to 3 clocks each.

The timing of burst mode access is generally stated using the notation: "x-y-y-y". The first number ("x") represents the number of clock cycles to do the first 64-bit read/write. The second, third, and fourth numbers are used to describe how many clock cycles needed to read/write the remaining three 64-bit blocks. When we describe burst mode as "5-2-2-2" we mean that reading the four 64-bit strings needs 11 clock cycles. SDRAM is "5-1-1-1".

6.3.2 Sequential Access (also, Serial Access)

As mentioned before, the access time means the time from when the instruction decoder supplies the system with the address of the needed location at the memory until the desired information is found but not read. In sequential access memories data is arranged in the form of units, called records, each

with its unique address. The address of the record is stored as part of the contents of the record. The record contains normally more than one word. The addresses are used as a way to separate the records from each other at the same time the system uses them for retrieving the records. The sequential access memory has one read/write mechanism; in case of hard disk this mechanism is called *head assembly*. Accessing any record must follow a specific linear sequence; any time the system needs to access record n, the head assembly has to start from an initial address passing and rejecting any intermediate address till it reaches the required address. In some systems the initial address is that of the first record (barking place of the head assembly in case of hard disk), other systems starts from the current location. This makes the access time to be a function of the physical location of the data on the medium with reference to the default position from which the read/write mechanism starts its movement.

6.3.3 Direct Access

As in case of sequential access, this system uses one mechanism for read/write. In direct address, individual blocks or records have a unique address based on physical location. Access is accomplished by direct access to reach general vicinity plus sequential searching, counting, or waiting to reach the final location. Disk units are direct access.

6.4 Semiconductor Memory

Semiconductor random access memory is fabricated on silicon chips by the same process used to manufacture microprocessors. Ongoing developments in semiconductor memories result in continually increasing density of memory chips. Historically and in line with Moor's law, the number of bits per chip has quadrupled roughly every 3.1 (orπ) years. One-megabit dynamic RAM (DRAM) chips have been available since 1985, 64 Mbit DRAM chips since 1994, and 1 Gbit DRAM chips became available in 2000. For static RAM (SRAM) chips, the development follows the same exponential curve albeit with a lag of about one generation (which is about 3.1 years) due to the large number of devices required for SRAM cells.

The exponential increase in the number of bits per chip has caused the area per memory cell and the price to decrease exponentially as well. As a matter of fact, without the availability of such high density, low-cost semiconductor memories, the microprocessor revolution would have been seriously delayed

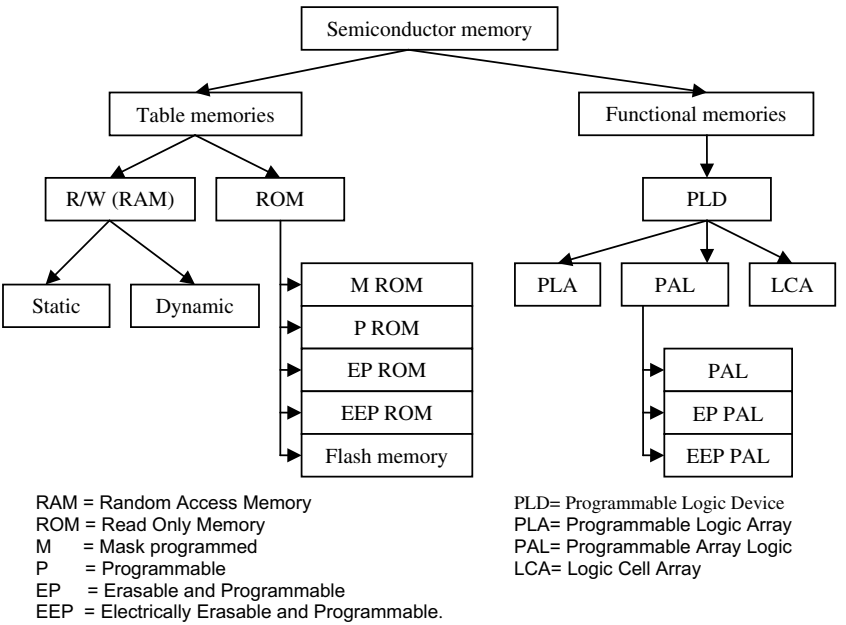


Figure 6.4 Categories of commonly used semiconductor memories.

and microprocessors been forced to use the slow, bulky, and expensive ferrite core memory of 1960s, and 1970s mainframes.

Semiconductor memories fall into two main categories, as shown in Figure 6.4: table memories and function memories. With table memories, an address A is defined in the range

$$0 \leq A \leq n = 2^N - 1$$

This expression defines the address space of a memory chip that has address width N bits. Data can be stored at each of the 2^N addresses. If the data word width of the memory chip is m bits, then it has storage capacity of $K = m.2^N$ bits. It is possible to extend the used address space and the word width by using several memory chips (see Section 6.6.1). The extension in the address space is limited by the width of address bus of the system. For address bus width N_b, the memory space available for the system is 2^{Nb}. This will allow any tables such as truth tables, computer programmes, results or numbers to be stored. In general the table memories are used to store information (data, instructions,…).

Function memories store logic functions instead of tables. This type of memories is not the subject of this book and the reader can find many textbox that cover them.

We shall limit our discussion to main classes of semiconductor memories: The read-only memory (ROM) and the read-write memory (RWM). The latter is inappropriately called random access memory (RAM). As mentioned before both semiconductor ROM and RWM are random-access devices. Thus, when a RAM is referred to, it is understood that a RWM is being discussed.

6.4.1 Read-Only Memory (ROM)

Read-only memory is a non-volatile memory that can be read from, but not written to, by a processor in an embedded or general–purpose system. The mechanism that is used for setting the bits in the memory, is called programming, not writing. For traditional types of ROM, such programming takes place off-line at the factory, when the memory is not actively serving as a memory in a system.

Read-only Memory (ROM) cells can be built with only one transistor per bit of storage. A ROM array is commonly implemented as a single-ended NOR array using any of the known NOR gate structures, including the pseudo-nMOS and the footless dynamic NOR gate. Figure 6.5 shows a 4-word by 6-bit ROM using pseudo-nMOS pull-ups with the following contents:

Word 0 :	010101
Word 1 :	011001
Word 2 :	100101
Word 3 :	101010

From an engineering point of view, semiconductor ROMs are very easy to use because interfacing ROM to a CPU is even easier than interfacing static RAM. Because the ROM is never written to, a ROM chip requires nothing more than the address of the location to be accessed together with a chip-select signal to operate the output circuits of the chip's data bus.

ROMs are used in many applications where it is needed to store a software that will stay fixed for long time. One of the applications is in dedicated microprocessor-based controllers. When a microcomputer is assigned to a specific task, such as the ignition control system in an automobile, the software is fixed for the lifetime of the device. A ROM provides the most cost- effective way of storing this type of software.

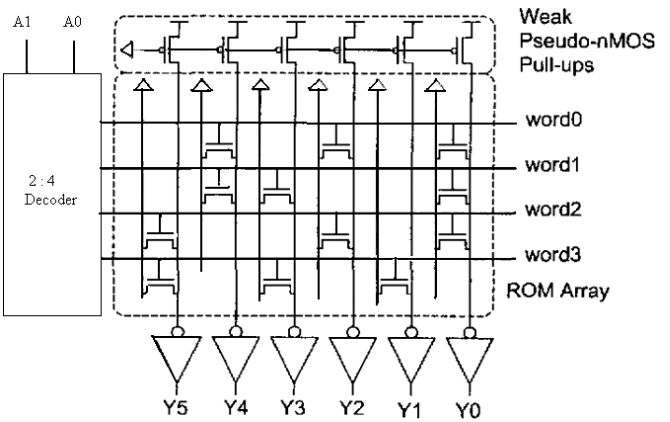


Figure 6.5 Pseudo-nMos ROM.

It is easy to see the two main reasons that read-only memory is used in such and similar applications to fulfill certain functions within the PC or the microcontroller:

- **Permanence**: The values stored in ROM are always there, whether the power is on or not. A ROM can be removed from the PC, stored for an indefinite period of time, and then replaced, and the data it contains will still be there. For this reason, it is called non-volatile storage. A hard disk is also non-volatile, for the same reason, but regular semiconductor RAM is not.
- **Security**: The fact that ROM cannot easily be modified provides a measure of security against accidental (or malicious) changes to its contents. You are not going to find viruses infecting true ROMs, for example; it's just not possible. (It's technically possible with erasable EPROMs, though in practice never seen.)

While the whole point of a ROM, as mentioned before, is supposed to be that the contents cannot be changed, there are times when being able to change the contents of a ROM can be very useful. There are several ROM variants that can be changed under certain circumstances; these can be thought of as "mostly read-only memory". PROM, EPROM, EEPROM, and flash memory are some of the modified forms of the ROM. In terms of write ability, the latter two have such a high degree of write ability that calling them read-only memory is not really accurate. In terms of storage permanence, all ROMs have high storage permanence, and in fact, all are nonvolatile.

We now describe the characteristics of these commonly used read-only memories.

6.4.1.1 Mask-programmable ROM

Mask-programmed ROM is constructed from hard-wired logic, encoded in the silicon itself, exactly the same technique as that used to implement a processor. It is designed to perform a specific task and this task cannot change after manufacturing it. They are programmed during their manufacture and are so called because a mask (i.e. stencil) projects the pattern of connections required to define the contents. Mask-programmed ROMs obviously has extremely low write-ability but has the highest storage permanence of any memory type, since the stored bits will never be altered unless the chip is damaged. Because of this inflexibility, regular ROMs are only used to store programmes that do not change often and with it needed in mass production because the cost of setting up the mask is very high. The other read-only memories we describe next are all programmable and some are reprogrammable.

6.4.1.2 Programmable ROM (PROM) (also, One-Time PROM)

This is a type of ROM that can be programmed using special equipment; it can be programmed only once. Programmable ROMs can be fabricated as ordinary ROMs fully populated with pull-down transistors in every position. Each transistor is placed in series with a fuse or a link made of polysilicon, nichrome, or some other conductor that can be burnt out by applying a high current. The transistor can be turned on or off to store a 1 or a 0. The transistor's state (ON or OFF) is determined by the condition of the metallic link that connects one of the transistor's inputs to a fixed voltage. When a user buys a PROM, it is filled with all 1s because each link forces the corresponding transistor into a 1 state. To programme a PROM, the user provides a file that indicates the desired ROM contents. Special equipment called a ROM programmer is then used to configure each programmable connection according to the file by passing such a large current through a link. That link melts and changes the state of the transistor from a 0 to a 1. For obvious reasons, these links are often referred to as fuses. A PROM is programmed once and once only, because if you fuse a link it stays that way. This is why PROMs are also called "*one-time programmable memories* (OTP ROM)".

PROMs have a low access time (5–50 ns) and are largely used as a logic element rather than as a means of storing programmes.

OTP ROMs have the lowest write-ability of all PROMs since they can only be written once, and they require a programmer device. However, they

have very high storage permanence, since their stored bits won't change unless someone reconnects the device to a programmer and blows more fuses. Because of their high storage permanence, OTP ROMs are commonly used in final products, versus other PROMs, which are more susceptible to having their contents inadvertently modified from radiation, maliciousness, or just the mere passage of many years.

Use of PROM is useful for companies that make their own ROMs from software they write. When the company changes its code, the company can create a new PROM with the new code. This can be achieved without the need of expensive equipment.

6.4.1.3 Erasable Programmable ROM (EPROM)

Programming EPROM needs a special machine costing from about a hundred dollars upwards (an EPROM programmer is different from a PROM programmer). Data is stored in an EPROM memory cell as an electrostatic charge on a highly insulated conductor. The charge can remain for periods in excess of ten years without leaking away. Essentially, an EPROM is a dynamic memory with a refresh period of tens of years.

To erase an EPROM IC, the entire IC chip has to be exposed to ultraviolet light (generally wavelengths shorter than 4000 angstrom) through a quartz window on the IC package. This results in erasing the entire cells in the EPROM simultaneously. Sunlight, fluorescent light, and incandescent light are all capable of erasing an EPROM, if the length of exposure is sufficiently long. Thus, use of EPROMs in production parts is limited because of the availability of electrical noise and radiation in the environment. If used in production, EPROMs should have their windows covered by a sticker to reduce the likelihood of undesired changes of the memory. EPROMs are suitable for small-scale projects and for development work in laboratories because they can be programmed, erased, and reprogrammed by the user.

Once programmed and placed in a system, an EPROM is only read. This is the mode of operation that is of concern to you when designing a memory subsystem.

EPROMs have a problem with the maximum number of times a bit can be written before failure. This number is typically less than few thousands.

6.4.1.4 Electrically Erasable PROM (EEPROM)

Electrically erasable PROM, or EEPROM (also E2PROM), is developed in the early 1980s to eliminate the problems related to reprogramming EPROMs. Reprogramming EPROM needs long time and sometimes has impossible

requirement of exposing an EPROM to UV light to erase the ROM. It is possible to erase and to reprogramme the EEPROM electronically by using higher than normal voltage. Erasing the EEPROM requires few seconds, rather than the many minutes required for EPROMs.

Furthermore, EEPROMs can have individual words erased and reprogrammed, whereas EPROMs can only be erased in their entirety. EEPROMs are more expensive than EPROMs, but they are more convenient to use.

Because EEPROMs can be erased and programmed electronically, we can build the circuit providing the higher-than-normal voltage levels for such electronic erasing and programming right into the embedded system in which the EEPROM is being used. Thus, we can treat this as a memory that can be both read and written. For EEPROM writing a particular word would consist of erasing that word followed by programming that word. Thus, an EEPROM is in-system programmable. We can use it to store data that an embedded system should save after power is shut off. For example, EEPROM is typically used in telephones that can store commonly dialed phone numbers in memory for speed-dialing. If you unplug the phone, thus shutting off power and then plug it back in, the numbers will still be in memory. EEPROMs can typically hold data for 10 years and can be erased and programmed tens of thousands of times before losing their ability to store data.

In-system programming of EEPROMs has become so common that many EEPROMs come with a built-in memory controller. A memory controller hides internal memory-access details from the memory user, and provides a simple memory interface to the user. In this case, the memory controller would contain the circuitry and single-purpose processor necessary for erasing the word at the user-specified address, and then programming the user-specified data into that word.

While read accesses may require only tens of nanoseconds, writes may require tens of microseconds or more, because of the necessary erasing and programming. Thus, EEPROMs with built-in memory controllers will typically latch the address and data, so that the writing processor can move on to other tasks. Furthermore, such an EEPROM would have an extra "busy" pin to indicate to the processor that the EEPROM is busy writing, meaning that a processor wanting to write to the EEPROM must check the value of this busy pin before attempting to write. Some EEPROMs support read accesses even while the memory is busy writing.

A common use of EEPROM is to serve as the programme memory for a microprocessor. In this case, we may want to ensure that the memory cannot be in-system programmed. Thus, EEPROM typically comes with a pin that

can be used to disable programming. The majority of the available microcontrollers have EEPROM on chip; it represents the programme memory of the microcontroller.

6.4.1.5 Flash memory

Flash memory is an extension of EEPROM that was developed in the late 1980s. While also using the floating-gate principle of EEPROM, flash memory is designed to handle large blocks of data at the same time. In flash memory it is possible to eras large blocks of memory at once not just one word at a time as in traditional EEPROM. A block is typically several thousand bytes large. This fast erase ability can vastly improve the performance of embedded systems where large data items must be stored in nonvolatile memory. Some examples are: digital cameras, TV set-top boxes, mobile phones, and medical monitoring equipment. It can also speed manufacturing throughput, since programming the complete contents of flash may be faster than programming a similar-sized EEPROM. Like EEPROM, each block in a flash memory can typically be erased and reprogrammed tens of thousands of times before the block loses its ability to store data, and can store its data for 10 years or more.

Flash memory's primary advantage is that, unlike EPROM, it can be electrically erased and reprogrammed in-circuit. This allows its use when it is desirable to be able to load an updated version of your program's object code into memory through a serial interface.

With this capability, you can update code without opening the system's case. You can even update code remotely over a telephone line. Flash memory is also useful for storing calibration and configuration data that must occasionally be changed by the system.

While EEPROMs are also in-circuit erasable and reprogrammable, they are less dense and typically more expensive than Flash.

The total number of erase/programme cycles of flash memory is limited. Typically, a Flash memory sustains 10,000 minimum to 100,000 typical erase/programme cycles.

Since EEPROMs and flash memories are in-system programmable and thus writable directly by a processor. Thus, the term “read-only-memory" for EEPROM or for flash is really a misnomer, since the processor can in fact write directly to them. Such writes are slow compared to reads and are limited in number, but nevertheless, EEPROMs and flash can and are commonly written by a processor during normal system operation. For this reason it is misleading now to consider them as types of the read only memories.

6.4.2 Read-Write Memory (RWM or RAM)

The RAM is used in systems that require the contents of the memory to be changed by the system. These devices allow a location to be accessed and written into or read from in fractions of a microsecond. Semiconductor RAM is a volatile memory. It loses its contents for any power failure or when we switch the power off.

There are two methods of storing data in semiconductor RAM. One method represents data as the state of a bistable flip-flop cell. The cell maintains its contents as long as dc power is applied to the device. This type of RAM is called the static RAM. The second type represents the data as charge on a capacitor and is called dynamic RAM.

6.4.2.1 Static RAM (SRAM)

Static RAM (SRAM) is the most widely used form of on-chip memory. Static semiconductor memory is created by fabricating an array of latches on a single silicon chip. This memory has a very low access time, but is about four times more expensive than dynamic memory. Furthermore, because of the number of transistors used to build the SRAM cell, a static RAM stores only one quarter (or less) as much data as a dynamic memory of the same silicon area. Static RAM is easy to use from the engineer's point of view and is found in small memories. Some memory systems use static memory devices because of their greater reliability than dynamic memory. Large memories are constructed with dynamic memory because of its lower cost. Some static memories use less power than DRAM and some use more power.

SRAM Cell Structures

A fully static RAM cell (SRAM cell) is a bistable circuit, capable of being driven into one of two states. The cell is activated by raising the wordline and is read or written through the bitline. After removing the driving stimulus, the circuit retains its state. In case of MOS memories, three memory cell configurations are in use; 12-transistor SRAM cell, 6-transistor SRAM cell and the SRAM cell with polysilicon load devices (4-transistor cell). Figure 6.6 shows a 6-transistor SRAM cell plus some read/write circuitry common to an entire column of cells. A typical static RAM will contain an array of these cells, for example, a 32-row by 32-column configuration.

Transistors T1 to T4 make up a bistable circuit. T1, and T2 are the amplifying elements, and T3 and T4 form active loads for T1 and T2. Less space is required to fabricate an MOS transistor than is required for a resistor; thus, the

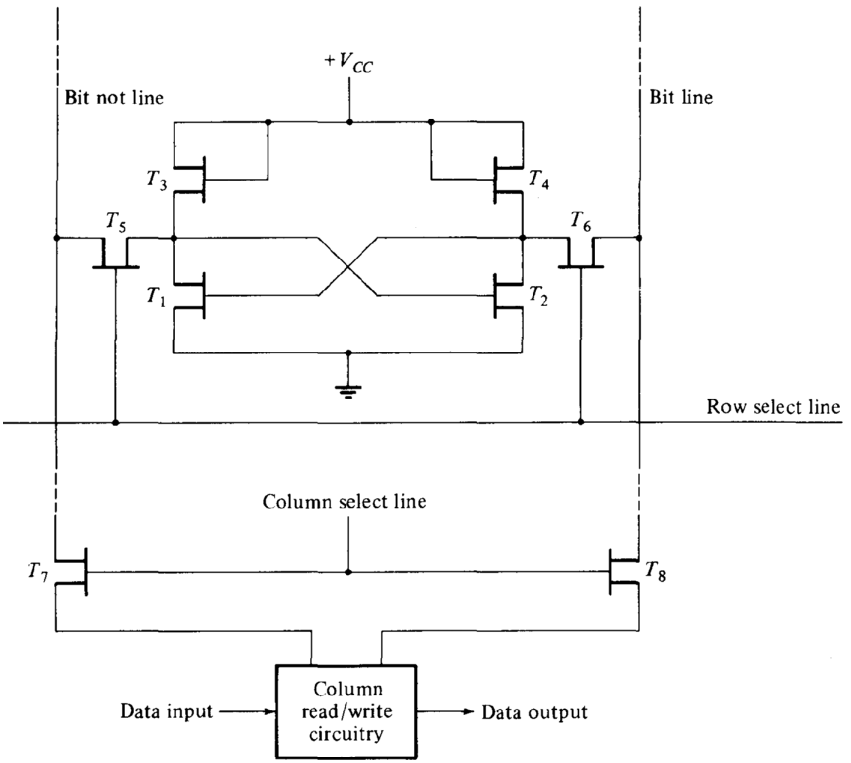


Figure 6.6 A static RAM cell.

active loads conserve chip space. Transistors T5 to T8 act as gate switches. When off, an open circuit is approximated between the transistor's source and drain. When on, the transistor approximates a short circuit.

When an input address is applied to a static RAM chip, part of this address is decoded to select a column, and the remainder is decoded to select a row. The column select line turns on T7 and T8. An entire column of cells are now connected to the read/write circuitry via the bit line and the bit not line. The bit not line always contains the complement of the data on the bit line. If the data are to be entered (written) into memory, these input data drive the bit and bit not lines through the read/write circuit. The row select line corresponding to the decoded row address determines which cell of the column will accept the input data. This line turns on T5 and T6 to allow the bit and bit not lines to drive the bistable cell of the active row. The bistable is driven to the appropriate state by the input data.

The read operation addresses the cell in exactly the same way, but in this case the data input is disabled and is not allowed to drive the bit lines. Information from the bistable cell is passed over the bit line through an amplifier to become the data output.

6.4.2.2 Dynamic RAM (DRAM)

Dynamic random access memory (DRAM) is the most common kind of random access memory (RAM) for personal computers and workstations. DRAMs store their contents as charge on a capacitor. DRAM is dynamic in that, unlike static RAM (SRAM), it needs to have its storage cells refreshed or given a new electronic charge every few milliseconds. Static RAM does not need refreshing because it operates on the principle of moving current that is switched in one of two directions rather than a storage cell that holds a charge in place. Thus, the DRAM basic cell is substantially smaller than SRAM, but the cell must be periodically read and refreshed so that its contents do not leak away. Commercial DRAMs are built in specialized processes optimized for dense capacitor structures. They offer an order of magnitude greater density (bits/cm^2) than high-performance SRAM built in a standard logic process, but they also have much higher latency.

Another advantage of dynamic RAM results from storing the bit as charge on the capacitor rather than as the state of a bistable, as much less power is required to store and maintain the capacitor charge than to maintain the state of the bistable.

DRAM Cell

A 1-transistor (1T) dynamic RAM cell consists of one transistor and one capacitor (see Figure 6.7(a)). Like SRAM, the cell is accessed by asserting the wordline to connect the capacitor to the bitline. On a read, the bitline is

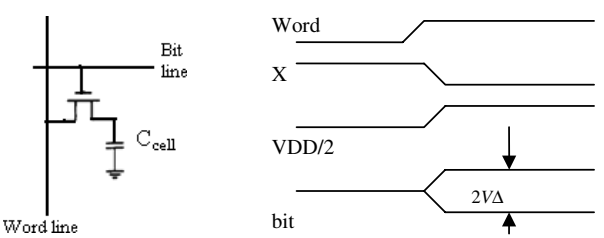


Figure 6.7 Dram cell (a) The cell, (b) Read operation.

first precharged to VDD/2. When the wordline rises, the capacitor shares its charge with the bitline, causing a voltage change Δ V that can be sensed (see Figure 6.7(b)). The read disturbs the cell contents at x, so the cell must be rewritten after each read. On a write, the bitline is driven high or low and the voltage is forced onto the capacitor. Some DRAMs drive the wordline to VDDP = VDD + V_t to avoid a degraded level when writing a '1'.

The DRAM capacitor C_{cell} must be as physically small as possible to achieve good density. However, the bitline is contacted to many DRAM cells and has a relatively large capacitance C_{bit}. Therefore, the cell capacitance is typically much smaller than the bitline capacitance. According to the charge-sharing equation, the voltage swing on the bitline during readout is

$$\Delta V = \frac{V_{DD}}{2} \cdot \frac{C_{cell}}{C_{cell} + C_{bit}}$$

This equation shows that a large cell capacitance is important to provide a reasonable voltage swing. This large capacitance is also necessary to retain the contents of the cell for an acceptably long time and to minimize soft errors. For designers, 30 fF (30×10^{-15}) is a typical target. In modern technology, the most compact way to build such a high capacitance is to use three-dimensional capacitor structures.

Dynamic RAM Refresh Cycle

In case of DRAM each cell needs only one transistor. The cell's transistor is nothing more than a gate. All cell gates in the entire row are opened by assertion of the row select line. Each row capacitor drives a bit-line sense amplifier that also includes a bistable storage cell. With this arrangement, the assertions of a row select line transfers and stores data from all cells of the row to the corresponding sense amplifiers. A particular sense amplifier can be gated to the output by assertion of the appropriate column select line.

There are two problems with the dynamic cell that are not encountered in the static cell. The first is that the sense operation draws current from the cell and modifies the capacitor's charge. The second problem is capacitor leakage. Even if the gating transistor remains off, the capacitor tends to lose charge. Assuming that the target value of 30 fF for the cell capacitance is achieved, we can see that even when using a very large leakage resistance to ground, the time constant will be low. In order for the cell to maintain enough charge to result in accurately sensed data, a refresh operation must be performed before significant charge is lost. A common refresh time for practical DRAMs is 2 ms. The refresh operation must restore all lost charge on the capacitors every 2 ms or less.

6.5 Interfacing Memory to Processor

In this section we discuss the interfacing of the different components that form the semiconductor memory subsystems. To make it simple, we will approach memory interfacing using both general and specific examples. The processor is connected to the memory and I/O devices via the bus. The bus contains timing, command, address, and data signals. The timing signals, as system clock signals, determine when strategic events will occur during the transfer. The command signals, like R/W line, specify the bus cycle type. During a read cycle data flow from the memory or input device to the processor, where the address bus specifies the memory or input device location and the data bus contains the information. During a write cycle data are sent from processor to the memory or output device, where the address bus specifies the memory or output device location and the data bus contains the information. During each particular cycle on most computers there is exactly one device sending data and exactly one device receiving the data. Some buses like the IEEE488 and SCSI allow for broadcasting, where data is sent from one source to multiple destinations.

The interfacing of a device (RAM, ROM, or I/O device) takes place in two steps:

- The first step is to design an address decoder (also called chip select decoder). Let SELECT be the positive logic output of the address decoder circuit. The command portion of our interface can be summarized by Table 6.1.
- The second step of the interface will be timing. Most of the microcomputers and microcontrollers use synchronous bus, thus all timing signals will be synchronized to the clock. It is important to differentiate timing signals from command signals within the interface. A timing signal is one that has rising and falling edges at guaranteed times during the memory access cycle. In contrast, command signals are generally valid during most of the cycle but do not have guaranteed times for their rising and falling edge.

In this section we are discussing the two steps of designing the interfacing.

Table 6.1 The Address Decoder and R/W determine the type of bus activity for each cycle.

Select	R/W	Function	Rationale
0	0	OFF	Because the address is incorrect
0	1	OFF	Because the address is incorrect
1	0	WRITE	Data flow from the processor to the external device (can be external memory)
1	1	READ	Data flow from the device to the processor

6.5.1 Memory Organization

In practice, the computer's memory can be based on several memory chips connected to produce the desired overall memory size. Expanding memory size by using several chips may take three forms:

- Expansion in the depth (total number of locations): In this case the word width of the available individual memory chips is the same as that of the desired overall memory. An enough number of chips is used to increase the number of memory locations.
- Expansion in width (increasing the number of bits per word): In this case the available chips have the same number of locations as the overall required memory, but with less word length (width) than that required for the final memory.
- Expansion in width and depth. Case when the available chips have a width and a depth less than that of the overall memory.

In the following we shall use some examples to demonstrate expanding memory size by using several chips.

Example 6.1: Expanding the width (number of bits per word)

The first example uses 1024×1 static RAM chips to construct a 1024×8 memory. Figure 6.8 indicates the appropriate organization for this chip.

In this figure, the chip select lines of all chips are connected in parallel, thus this line becomes the memory select input. Similarly, all $R/\overline{W}$ lines are paralleled and thus each individual address line is paralleled with the corresponding line on the other chips. With these connections, all chips are enabled simultaneously; the read/write control is applied to all chips simultaneously; and each input address accesses the same numbered location on each chip.

The data inputs and outputs are not connected in parallel. An 8-bit input word is stored with one bit of the word in each chip at the location accessed by the input address. An output word is made up of 1 bit from each chip, all coming from corresponding locations

Example 6.2 Expanding the depth of the memory

This example demonstrates the use of $1\,\text{K} \times 8$ chips to construct a $4\,\text{K} \times 8$ memory. Figure 6.9 shows the final block diagram.

Twelve input address lines are required to access 4 k locations, but each chip has only 10 address lines. The remaining 2 input address lines are decoded by AND gates (or a decoder) to drive the individual chip select inputs. Only when the memory select line enables the gates will one chip be selected.

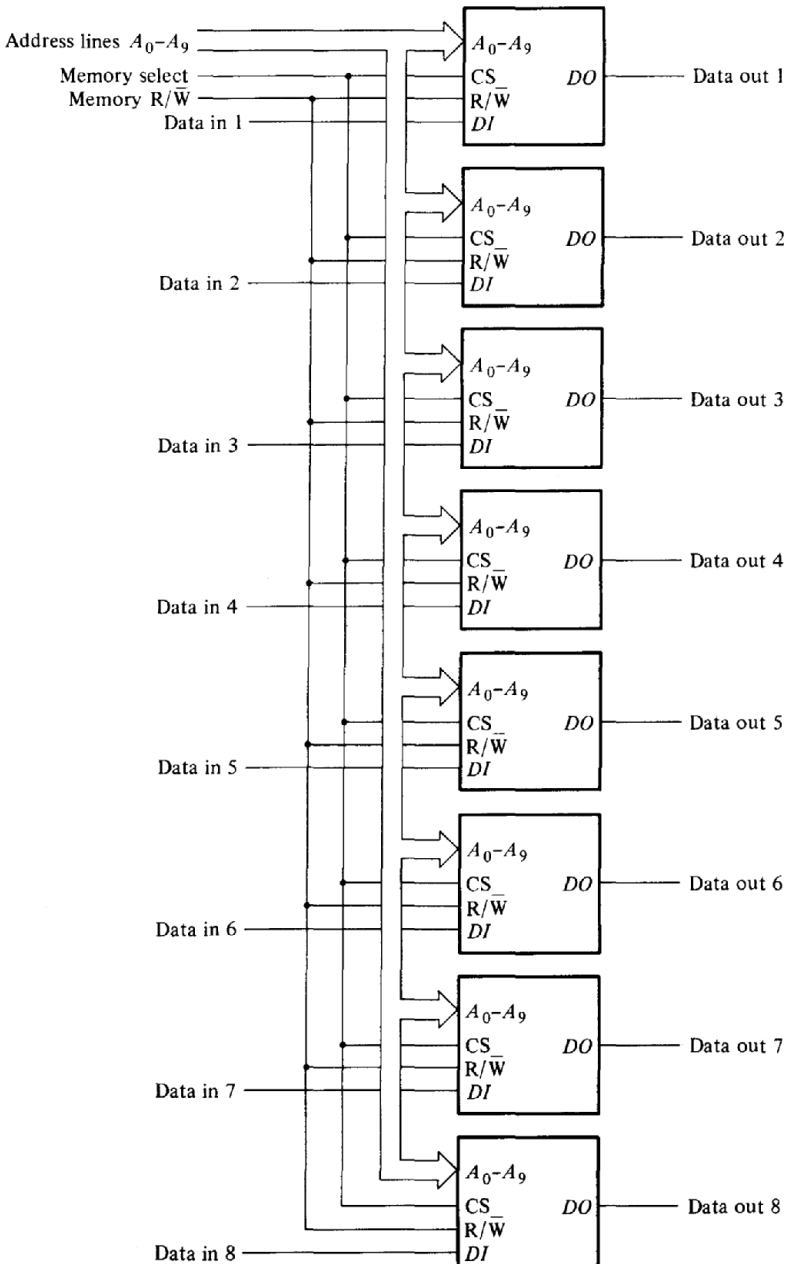


Figure 6.8 A 1 k × 8 memory system.

The ranges of addresses are:

Chip 1000000000000 to 001111111111(0 − 1023)
Chip 2010000000000 to 011111111111(1024 − 2047)
Chip 3100000000000 to 101111111111(2048 − 3071)
Chip 4110000000000 to 111111111111(3072 − 4095)

Example 6.3 Expanding the width and depth

In this example it is required to use 1 K × 4 chips to construct a 2 K × 8 memory. Such a circuit is shown in Figure 6.10. When the memory select line has enabled the AND gates, the input address line A10 is used to enable either the upper two chips or the lower two chips. The upper pair of chips are active for addresses 00000000000 to 01111111111 (0 − 1023), and the lower pair are active for addresses 10000000000 to 11111111111 (1024 − 2047). These particular chips have common I/O lines that connect to the I/O bus. Two 4-bit-wide chips are connected to create 8-bit-wide input and output words.

Calculation of the number of chips

The number of chips needed to implement a memory can be calculated from:

Number of chips = Bit capacity of memory/Bits per chip

This equation is appropriate only if the word size of the memory is an integer multiple of the width of the chip.

Example 6.4: How many chips are required to construct an 8K × 8 memory if the configuration of (a) Figure 6.8, (b) Figure 6.9, or Figure 6.10 is used?

Solution:

The total bit capacity of the overall memory is:

8 Kbytes × 8 bits/byte = 64 kBits

a. The number of 1 K × 1 chips required is:
   64 Kbits/1 K bits = 64 chips
b. The number of 1K x 8 chips required is:
   64 Kbits/1 K × 8 = 8 chips.
c. The number of 1K × 4 chips required is:
   64 Kbits/4 K = 16 chips

Often a memory is expanded in sections called pages. Although there are several definitions of a page of memory, for our purposes we are going to

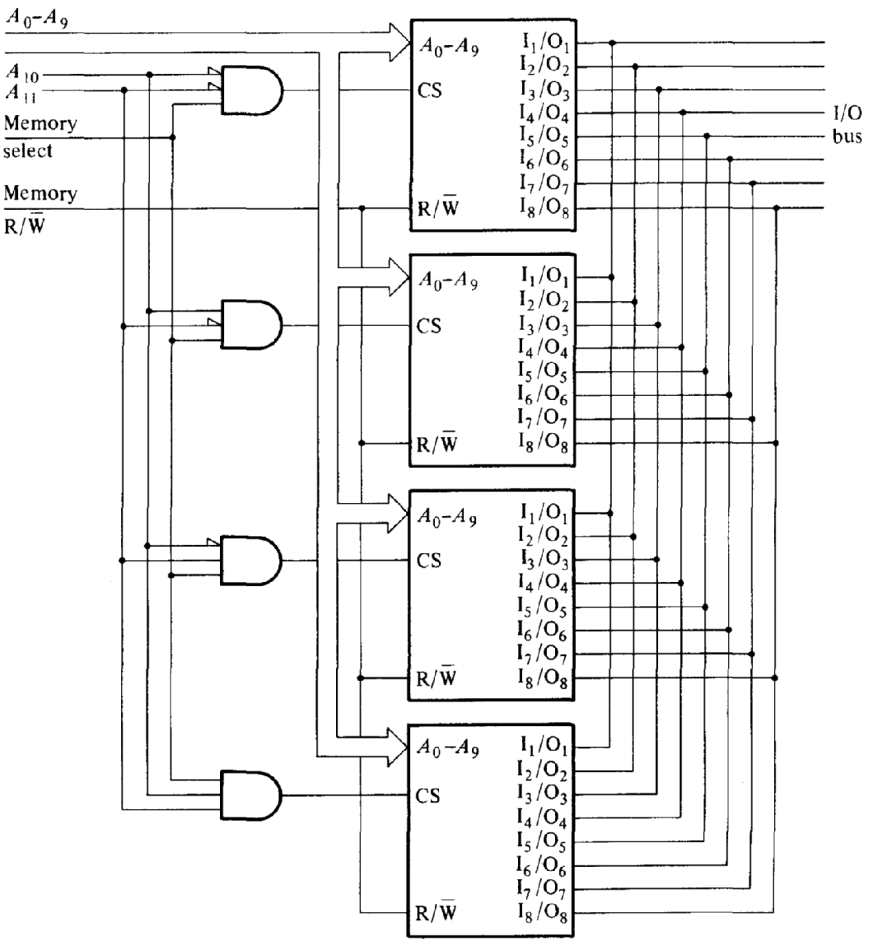


Figure 6.9 A 4k × 8 memory.

define the page as:

> "*A page of memory represents the maximum number of locations that can be accessed directly by the chip address lines*".

For example, if the memory is made up of chips having 10 address lines, a page contains 2^{10}, or 1 k, locations. If the entire memory contains 4 k locations, the memory is made up of 4 pages. Each page is activated by asserting the proper chip select lines.

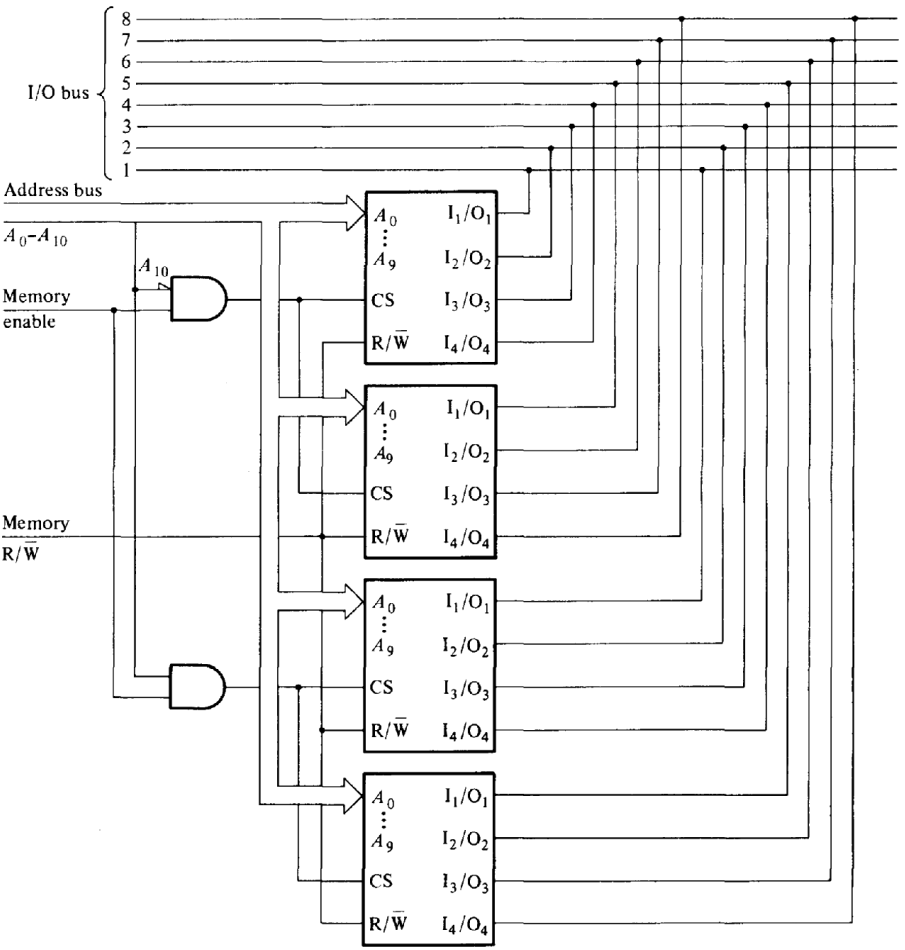


Figure 6.10 A 2 k × 8 memory.

It is possible to clarify this concept of memory page by returning to the organizations depicted in Figures 6.8, 6.9, and 6.10. The 1 k × 8 memory of Figure 6.8 has 10 address lines and thus consists of 1 page. Figure 6.9 consists of a 2-kbyte memory made up of chips with 10 address lines. Again each page consists of 1 kbyte, and 2 pages make up the entire memory. We note that 11 address lines are required to access the memory. Ten of these lines are connected directly to the chip address lines, and the eleventh is used to select the correct page by asserting the appropriate chip select lines. Figure 6.10 shows a 4-page memory, again with 1 kbyte per page.

Example 6.4 indicated the number of chips required to realize an 8-kbyte memory using the configurations of Figures 6.8, 6.9, and 6.10. The number of pages for each of these configurations would be 8, as each configuration has 10 address lines per chip.

Example 6.5: A 2 K × 4 chip is used to construct a 16 K × 8 memory. How many chips are required? How many pages of memory are used? Repeat these calculations for a 16K × 16 memory.

Solution:

The total bit capacity of the 16-Kbyte memory is:

$$16 \text{ Kbytes} \times 8 = 128 \text{ K bits}$$

The number of chips required is:

$$(128 \text{ Kbits})/(2\text{K} \times 8) = 16 \text{ chips}$$

A 2 K × 4 chip requires a number of address lines given by:

$$m = \log_2 2048 = 11 \text{ address lines}$$

The entire memory requires:

$$m = \log_2 16,384 = 14 \text{ address lines}$$

The number of pages required is:

$$2^{14}/2^{11} = 8 \text{ pages}$$

Each page consists of two chips, and the page decoder consists of a 3-line to 8-line decoder.

For the 16 K × 16 memory, the number of chips required is:

$$16\,\text{K} \times 16/2\,\text{K} \times 4 = 32 \text{ chips}$$

The number of total memory locations does not change, and therefore 8 pages are again required. In this case, each page consists of four chips rather than two.

6.5.2 Address Decoding

6.5.2.1 Memory space and memory map

The address bus width defines a ***memory space*** which is the maximum number of locations that can be addressed by the system. For example if the address bus

width is 32 bits (or lines) A0–A31, the memory space will be 2^{32} locations. Microprocessor systems (also microcontroller systems) often have memory components that are smaller than 2^{32}. Moreover, there normally use more than one type of memory at the same time to form the memory system: read/write, ROM, EEPROM and memory – mapped peripherals. A ***memory map*** is a graphical representation describes the way such different types of memories are organized in the memory space. The memory map shows the address allocation of each memory component and the address of any other devices connected to the address bus.

Figure 6.11 gives an example of a 16-bit width address bus. The 16-bit address bus gives 2^{16} address space, which means that a maximum of 64 K word different storage devices can be addressed when using this processor. The user can use all the available space or part of it and can use different types of storage devices, e.g. register files, RAM, and ROM. In case of Figure 6.11 four different memory chips each of 16 K words are used by the system. The RAM has been the first 48 K addresses (from 0000 to BFFF) and ROM given the highest 16 k (i.e. addresses C000 to FFFF). Each area of memory can also be subdivided into smaller pages of 256 addresses each. Page zero occupies addresses 0000 to 255 in decimal or 0000 to 00FF in hex.

In memory mapped I/O system, the peripherals are considered as memory locations.

Example 6.6

Assume a processor with 8-bit word and with a 16-bit address bus. Draw the memory space and the memory map, if the processor is connected to the

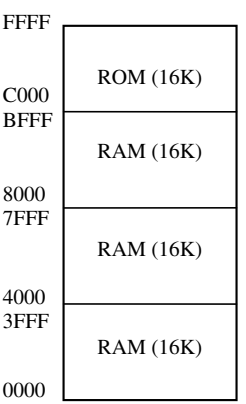


Figure 6.11 An example of a memory map.

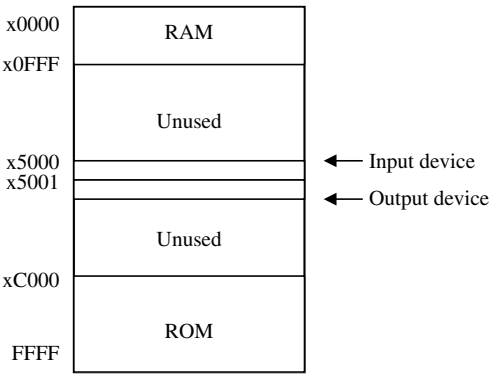


Figure 6.12 Memory map of Example 6.6.

following memory components and peripherals:

4 K RAM	\$0000 to \$0FFF
Input	\$5000
Output	\$5001
16 K ROM	\$*C*000 to \$FFFF

The memory map in this case is given in Figure 6.12.

6.5.2.2 RISC structures memory maps

In case of CISC architectures, the data and the programmes are stored in one memory (the main memory). This system has one address bus system to which the main memory is connected. Accordingly, the CISC architectures have normally one memory map that covers all the address space. On the other hand, and as mentioned before, the RISC structures separate completely between the programme memory and the data memory. Each of them is connected to a separate address bus, and accordingly each has its own memory map. Then it is expected that in case of RISC, the system has two different memory maps.

The AVR microcontrollers are RISC architecture, then the microcontroller has more than one memory map.

Example 6.7 Memory Maps of AVR processors

AVR is a RISC microcontroller that has two separate memories; data memory and programme memory (flash memory). The data memory is 8-bit word and has an address width of 16 bits. The programme memory is a 16-bit word with address width of 11 bits (or 12 bits). The microcontroller comes with the programme memory and the following components of the data

memory built-in:

32 working registers	$0000 to $001F
64 I/O registers	$0002 to $005F
256(512) Internal SRAM	$0060 to $015F($025F)

Show the memory maps of the AVR microcontroller.

Solution

The AVR as any RISC structure has two memory maps: programme memory map and data memory map. As a matter of fact the majority of the AVR microcontrollers has a third map; data EEPROM memory map. The data EEPROM is a 4 K × 8-bit word. Figure 6.13 shows the complete memory maps of ATmega8515/AT90S8515. The external SRAM space in the data memory map is normally unused unless the designer needs more storage space than that available for him by the internal SRAM. In such cases the designer can connect an external SRAM of capacity up to the 64 K words.

6.5.2.3 Chip-select signal and chip-select decoder

Consider now the normal terminology in which the processor represents the master, and all the components connected to the address bus are slaves. The

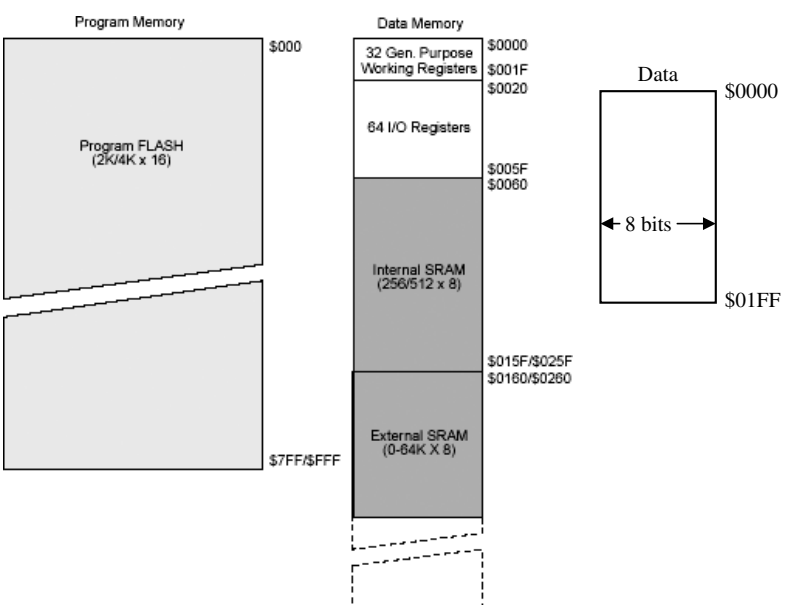


Figure 6.13 ATmega8515/AT90S8515 Memory.

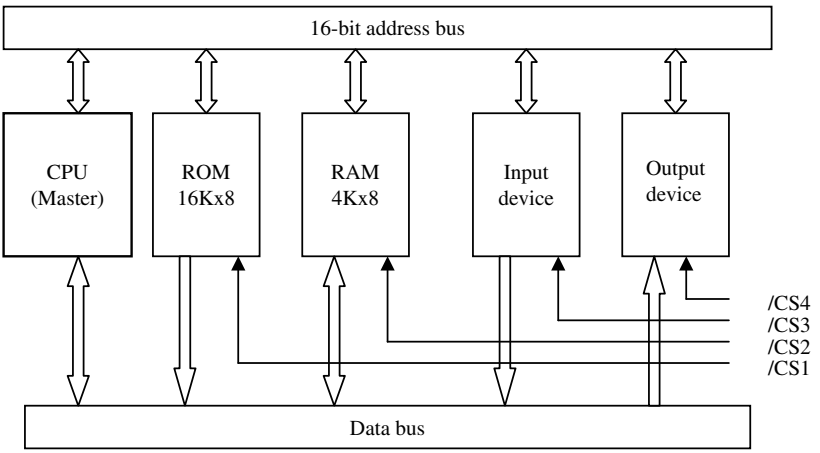


Figure 6.14 Chip-select signals.

main function of the address decoder is to monitor the N address lines from the bus master (the processor) and determine whether or not the slave has been selected to communicate in the current cycle. Each slave has its own address decoder, uniquely designed to select the addresses intended for the device. Care must be taken to avoid having two devices driving the data bus at the same time. It is required to select exactly one slave device during every cycle.

Consider the situation illustrated by Figure 6.14, which represents the connections of the components mentioned in Example 6.6 to the address bus of the processor. The ROM address input pins are connected to 14 of the address bus lines coming from the master, and the RAM is connected to 12 of these lines. This means that there are 2^{12} locations are common between the ROM and the RAM. Whenever one of these common locations is addressed in the RAM, the corresponding location in the ROM is addresses. The data outputs of the ROM and RAM (also the other I/O devices) are connected to the system bus. Since all the data outputs of the different devices are connected together, the data bus derives in the different components must have tri state outputs. That is, only one of the components may put data on the system data bus.

To achieve that, all the components connected to the buses have a chip-select signal (/CS1, /CS2, /CS3, and /CS4). Whenever the chip-select input of any component is active-low, that device takes part in memory access and puts (or receives) data on (from) the data bus if $R/W = 1$. When the chip-select is not active (i.e. in high state) the appropriate data bus drivers are turned off, stopping all the devices from putting any data on the data bus. Special circuit

called "*chip select decoder*" are used to generate the chip-select signals. The decoder is designed such that no conflict between the different devices; the address decoders should not select two devices simultaneously. In other words, no two select lines should be active simultaneously. In this case,

$$\begin{aligned}
&(\text{Select RAM}) \text{ AND } (\text{Select Input}) = 0\\
&(\text{Select RAM}) \text{ AND } (\text{Select Output}) = 0\\
&(\text{Select RAM}) \text{ AND } (\text{Select ROM}) = 0\\
&(\text{Select Input}) \text{ AND } (\text{Select ROM}) = 0\\
&(\text{Select Output}) \text{ AND } (\text{Select ROM}) = 0\\
&(\text{Select Input}) \text{ AND } (\text{Select Output}) = 0
\end{aligned}$$

In this way it is guaranteed that for each address location within the memory space, only the chip-select of the memory component that has the needed data is active and the rest are not active. By this way the memory map of our system now contains four disjoint memory components.

The address decoder is usually located in each slave interface. There are several different strategies for decoding. These strategies may be divided into groups: *full address decoding*, *partial address decoding*, *block address decoding* and *minimal cost address decoding.* Only the first strategy is discussed here. But before discussing the decoding strategies we have to introduce the use of address/data multiplexing.

Utilizing the limited number of pins on the microcontroller

Not all the microcontrollers/microprocessors have enough pins to let each pin has a single function. To utilize the limited number of pins on the microcontroller, use is made of address/data multiplexing in which some pins have dual functions. For example, the Intel 8051, the Motorola 6811, the AVR and MC68HC912B32 use a multiplexed address/data bus. The 8051, the AVR and 6811 (where the address bus width N =16 and word length of 8-bit) use the same eight wires AD7-AD0 for low address (A7-A0) in the first half of each cycle and for data (D7-D0) in the second half. The address strobe output (AS) in case of 6811 and the Address Latch Enable (ALE) in case of 8051/8085/8088/8086/AVR microcontrollers and microprocessors are used to capture first half of the address into an external latch, 74HC375 in Figure 6.15. In this way, the entire 16-bit address is available during the second half cycle.

In the MC68HC912B32, the same 16 wires, ADI5-AD0, are used for the address (A15-A0) during the first half of each cycle and for the data (D15-D0) during the second half of each cycle. We use the rising edge of the E clock to capture the address into an external latch (two 74HC374s), as shown in

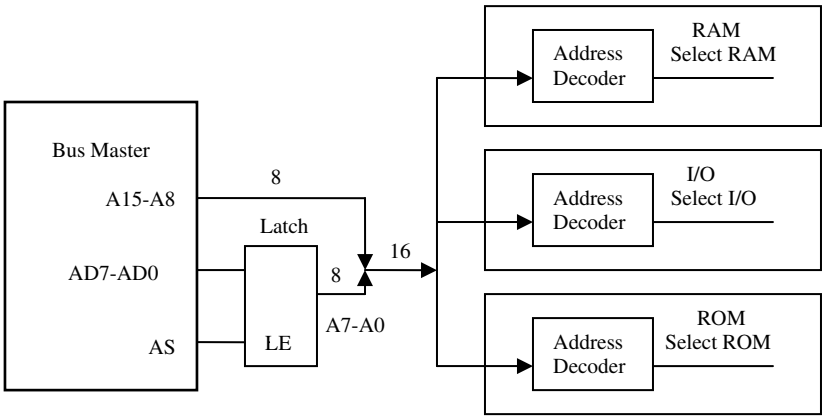


Figure 6.15 The 6811 (also 8051/8085/8088/8086) multiplexes the low address.

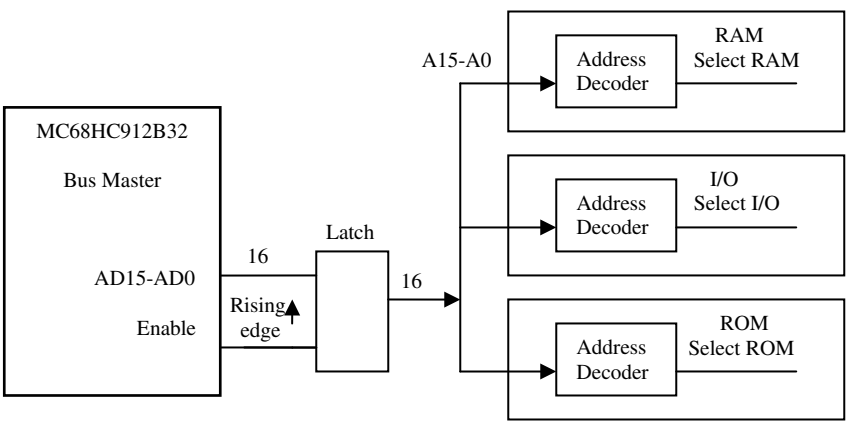


Figure 6.16 The MC68HC912B32 multiplexes the 16-bit address.

Figure 6.16. In this way, the entire 16-bit address is available during the second half of the cycle.

Some processors, e.g. MC6811C812A4, have enough number of pins to let the address pins completely separated from the data pins and accordingly run in non-multiplexed mode.

Whether multiplexing 8 pins or 16 pins or separating the address pins from data pins, the address decoders should not select two devices simultaneously. In other words, no two select lines should be active at the same time. In

examples given in Figure 6.14 this means:

$$\text{(Select RAM) AND (Select I/O)} = 0$$
$$\text{(Select RAM) AND (Select ROM)} = 0$$
$$\text{(Select I/O) AND (Select ROM)} = 0$$

6.5.2.4 Full address decoding

Full address decoding, also known as ***Exhaustive decoding***, is where the slave is selected if and only if the slave's address appears on the bus. In positive logic,

$$\begin{aligned} \text{Select} &= 1 \quad \text{if the slave address appears on the address bus;} \\ &= 0 \quad \text{if the slave address does not appear on the address bus.} \end{aligned}$$

This way guarantees that for each address appears on the address bus of the system only one location will respond. All the microprocessor's address lines are used to access each physical memory location, either by specifying a given memory device or by specifying an address within it.

Example 6.8: Design a fully decoded positive logic Select signal for a 1 K RAM at 4000H - 43FFH

Solution:

Step 1: Write in binary the start address and the last address of the space occupied by the chip. Compare the two and then write specified address using 0, 1, X, using the following rules:

a. There are 16 symbols, one for each of the address bits A15, A14, …, A0
b. 0 means the address bit must be 0 for this device
c. 1 means the address bit must be 1 for this device
d. X means the address bit can be 0 or 1 for this device
e. All the Xs (if any) are located on the right-hand side of the expression
f. With n represents the number of Xs, the size of the memory in bytes is 2^n
g. Let I be the unsigned binary integer formed from the $16 - n$ 0s and 1s, then

$$\text{I} = \text{(beginning address)/(memory size)}$$

In this example:
Beginning address \$4000 = 0100 0000 0000 0000
Last address \$43FF = 0100 0011 1111 1111
The needed address = 0100 00XX XXXX XXXX
From this result:
$n = 10$
size of the chip $= 2^{10} = 1024$ bytes
$I = 010000_2 = 16_{10} = (\$4000)/(\$0400)$

Step 2: Write the equation using all 0s and 1s. A 0 translates into the complement of the address bit, and a 1 translates directly into the address bit. For our example,

$$Select = \overline{A15}.A14.\overline{A13}.\overline{A12}.\overline{A11}.\overline{A10}$$

Step 3: Build circuit using real TTL gates. The result is given in Figure 6.17.

Example 6.9: Design a fully decoded select signal for an I/O device at \$5500 in negative logic

Solution:

Step 1: address = 0101 0101 0000 0000

Step 2: The negative logic output can be created simply by inverting the output of a positive logic design.

$$Select = \overline{\overline{A15}.A14.\overline{A13}.A12.\overline{A11}.A10.\overline{A9}.A8.\overline{A7}.\overline{A6}.\overline{A5}.\overline{A4}.\overline{A3}.\overline{A2}.\overline{A1}.\overline{A0}}$$

Step3: Figure 6.18.

Example 6.10: Build a fully decoded positive logic address decoder for a 20 K RAM with an address range from \$0000 to \$4FFF.

Start with

0000, 0000, 0000, 0000 Range \$0000 to \$0000

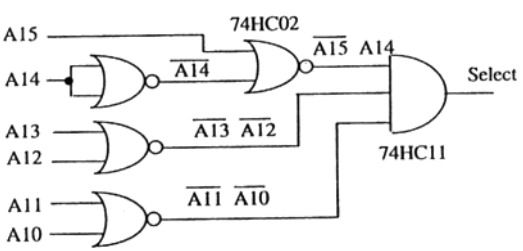


Figure 6.17 An Address Decoder identifies on which cycle to activate.

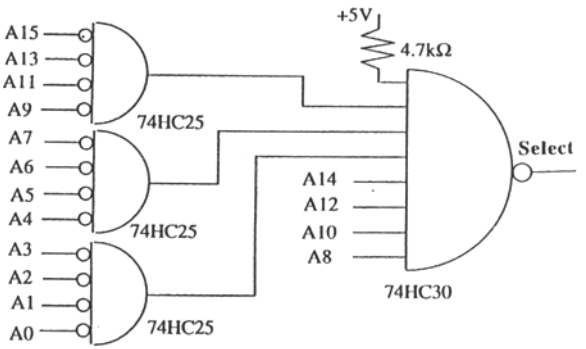


Figure 6.18 The required address decoder.

add Xs while the range is still within $0000 to $4FFF

0000, 0000, 0000, 000X	Range $0000 to $0001
0000, 0000, 0000, 00XX	Range $0000 to $0003
0000, 0000, 0000, 0XXX	Range $0000 to $0007
0000, 0000, 0000, XXXX	Range $0000 to $000F

Stop at

00XX,XXXX,XXXX,XXXX Range $0000 to $3FFF

Start over

0100, 0000, 0000, 0000 Range $4000 to $4000

add Xs while the range is still within $0000 to $4FFF

0100, 0000, 0000, 000X	Range $4000 to $4001
0100, 0000, 0000, 00XX	Range $4000 to $4003
0100, 0000, 0000, 0XXX	Range $4000 to $4007
0100, 0000, 0000, XXXX	Range $4000 to $400F

Stop at

0100, XXXX, XXXX, XXXX Range $4000 to $4FFF

$$RAMSelect = \overline{A15}.\overline{A14}. + \overline{A15}.A14.\overline{A13}.A12$$

Example 6.11: Build a fully decoded negative logic address decoder for a 32 K RAM with an address range of $2000 to $9FFF.

Even though the memory size is a power of 2, the size 32,768 does not evenly divide the starting address 8192. We break the $2000 to $9FFF irregular address range into three regular address ranges.

001X, XXXX, XXXX, XXXX	Range $2000 to $3FFF
01XX, XXXX, XXXX, XXXX	Range $4000 to $7FFF
100X, XXXX, XXXX, XXXX	Range $8000 to $9FFF

$$Select = \overline{\overline{A15}.\overline{A14}.A13 + \overline{A15}.A14 + A15.\overline{A14}.\overline{A13}}$$

6.5.3 Accessing Memory: Timing Diagram

If we consider the two examples given in Figure 6.19 where Figure 6.19a is a 1 K × 4 bits IC chip and Figure 6.19b is a 16 K × 8 bits IC. In the two cases the address bits shown are normally the lower-order, or least significant, address bits from the system bus and they are connected directly to the corresponding address pins of a memory IC. The higher-order address bits (those not connected directly to the memory IC) are normally used externally to generate the chip select signals that identify which of many possible memory ICs, is to be accessed. The chip select input is similar to an on/off switch. When this input is inactive, the chip is disabled, and all output lines are in the high-impedance state. This state approximates an open switch in series with each output line. When disabled, although physically connected, these output lines are effectively disconnected from any circuitry that follows. In order to use the RAM, the CS line must be active.

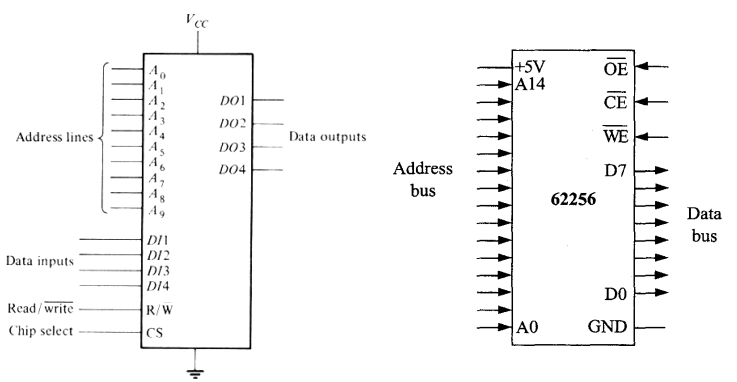


Figure 6.19 A 1 K × 4 RAM.

The CS (or chip enable CE) signal represents one of the control signals that control the operation of the memory. The number of control signals has its effect on the read/write timing diagram. RAM ICs typically have three control inputs. In addition to a chip select (/CS), the IC has a write enable (/WE) input, and an output enable (/OE) input. (The write enable signal is also known as read/write R/W enable). Less common are RAMs with only two control inputs, /CS and /WE.

When writing to the RAM, care must be taken that the correct address is present before assertion of the write enable. If the write enable is asserted before the desired address is present, the incoming data may write over existing data as the address changes.

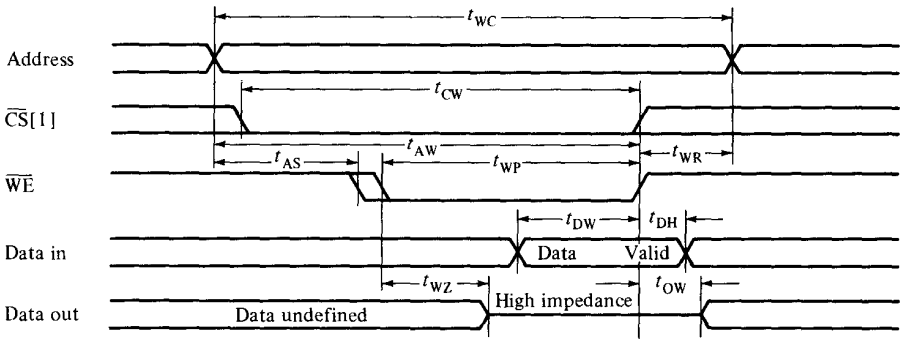
To write a RAM IC that has three control inputs involves the following steps:

1. The address of the location to be written is placed on the address inputs.
2. /CS is asserted, to select the memory IC.
3. The data to be written is placed on the data inputs.
4. /WE is asserted.
5. /WE is unasserted, causing the data to be latched.
6. The data is removed from the data inputs.
7. /CS is unasserted, to deselect the memory IC.
8. The address is removed (changed).

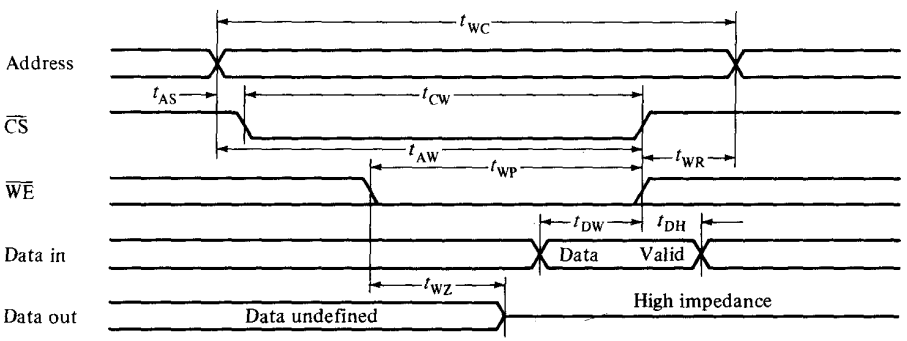
   Note that /OE remains unasserted throughout a write cycle.

As mentioned before there are two modes of writing: /WE control and /CS control. The waveforms of Figure 6.20 summarize the important timing specifications for these two modes of writing data to the RAM. This device has common input/output pins.

In mode 1, the chip select line is always asserted before assertion of the write enable and becomes deasserted at the same time or after WE is deasserted. Thus, the write enable controls the entry of data. This mode includes the case of a continuously selected chip. The /WE line can be asserted simultaneously with the application of a valid address because the address setup time, t_{AS}, is specified as 0 ns. The /WE line must remain asserted for 35 ns, as indicated by the write pulse width parameter t_{wp} Input data may be changed after /WE is asserted, but the valid input data to be written must be stable for a minimum of 20ns. This figure corresponds to the data valid to end of write time tow. Note that while /WE is low, the data output lines do not contain valid data. If the chip select /CS remains low after /WE goes high, the



(a) Waveforms Write cycle No. I (/WE controlled)



(b) Write cycle No. 2 (/CS controlled)

Figure 6.20 Memory write waveforms: (a) for device continuously selected, (b) for address valid before or coincident with chip select.

valid output data will become present immediately, as indicated by an output active to the end of the write time, tow, of 0.

Mode 2 allows the /CS input to control the write operation. The /WE line can be asserted before or after the /CS assertion, but remains asserted at least until the /CS is deasserted. In this mode the WE can be asserted before the address change as long as the /CS has not been asserted. This feature is not always available on RAM chips, especially those chips with separate data input and output lines. Only on chips having a /CS line that can disable input data lines can the address change after the WE line is asserted.

To read a RAM that has three control inputs involves the following steps:

1. Place the address of the location to be read on the address inputs.
2. /CS is asserted, to select the memory IC.

3. /OE is asserted, to enable the memory IC's output buffers.
4. The data driven on the data bus by the memory IC is latched by the processor.
5. /OE is unasserted; disabling the memory IC's output buffers.
6. /CS is unasserted, to deselect the memory 1C.
7. The address is removed (changed).

Note that /WE remains unasserted throughout a read cycle.

Figure 6.21 indicates the two modes of read-cycle waveforms. In the mode shown in Figure 6.21(a), the chip is selected before the address is changed. When the address is changed, the old output data remain valid for a minimum of 5 ns, as indicated by t_{OH}. There is a transition period during which the output data are invalid before reaching their final output. The time between the valid new address and the valid corresponding data is t_{AA}, which has a maximum value of 45 ns for this particular device. The read cycle, t_{RC}, cannot be shorter than t_{AA} and has a minimum value of 45 ns.

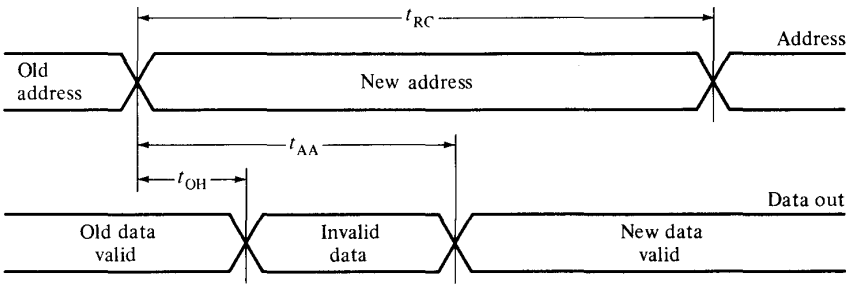
In the mode shown in Figure 6.21(b), the address is first selected and then the chip select becomes active. This device requires a low-asserted chip select signal. After the /CS is asserted, a minimum time of 20 ns is required to switch from the third- or high-impedance state to low-impedance outputs. This is given by t_{LZ}. Additional time is required before the correct data are present at the outputs. The total time from chip select to valid output is called t_{ACS} and in this instance has a maximum value of 45 ns. The maximum time taken for the outputs to enter the high-impedance state after chip deselection is 20 ns.

Read and write operations on a memory IC must be mutually exclusive, or bus contention results. Therefore, the memory IC's /WE and /OE inputs must never be asserted simultaneously.

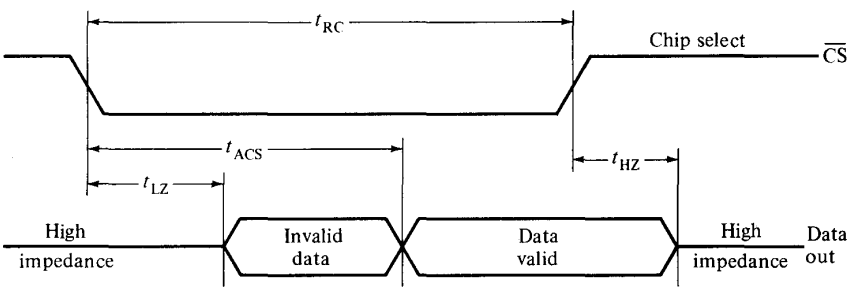
The timing relationship shown in Figure 6.20 and Figure 6.21 are called, in general, timing diagram. The timing information given by the timing diagrams must be carefully considered each time a RAM chip is to be used. The designer must determine which of the specifications are pertinent to the application at hand and design the controlling circuits to meet or exceed these specifications.

6.6 AVR Memory System

The memory section of the Atmel RISC AVR processors is based on the Harvard model, in which various portions of the memory are separated to allow faster access and increased capacity. The CPU has a separate interface



(a)



(b)

Symbol	Parameter	Time (ns) Min	Max
t_{RC}	Read-cycle time	45	
t_{AA}	Address-access time		45
t_{ACS}	Chip-select access time		45
t_{OH}	Output hold from address time	5	
t_{LZ}	Chip select to output in low Z	20	
t_{HZ}	Chip select to output in high Z		20

(c)

Figure 6.21 Memory read waveforms: (a) for device continuously selected, (b) for address valid before or coincident with chip select, and (c) typical times for high-speed IAM.

for the FLASH code (programme) memory section, the data memory section, and the EEPROM memory section, if one is present. The data memory of the AVR processor typically contains three separate areas of read/write (R/W) memory.

- **The register file**: This is the lowest section and contains 32 registers. All the 32 registers are general-purpose working registers.
- **The I/O registers**: This is the next 64 registers following the register file.

- The **internal and external SRAM**: This represents the memory area following the I/O registers.

Memory Maps of AVR processors

The memory maps of ATmega8515/AT90S8515 are given before in Figure 6.13. The following subsections deal with different parts of the microcontroller memory.

6.6.1 Flash Code Memory Map

The FLASH code memory section is a block of FLASH memory that starts at location 0×0000 and is of a size that is dependent on the particular Atmel AVR microcontroller in use. The ATmega8515/AT90S8515 has on-chip an 8 K bytes flash memory. The flash memory is in-system reprogrammable and used to store the programmes. All the instructions of AVR are 16 or 32 bits width. As a result, the $8\,\text{K} \times 8$ flash memory is organized as $4\,\text{K} \times 16$. The FLASH memory, as mentioned before, is non-volatile memory and it is used to store the executable code and constants, because they must remain in the memory even when power is removed from the device. The code memory space is 16 bits wide at each location to hold the machine-level instructions that are typically a single 16-bit word.

The Programme Counter (PC) is 12 bits wide, thus it can be used to address all the 4K Programme memory locations.

Although the FLASH memory can be programmed and reprogrammed with executable code, there is no provision to write to the FLASH by means of an executable programme; it must be programmed by external means. Consequently it is viewed as read-only memory from the programmer's perspective and is, therefore, used only to store constant type variables, along with the executable code. Constants are automatically promoted to `int` size variables when they are stored in FLASH code memory space because of the memory width.

6.6.2 Data Memory Map

The data memory of the Atmel AVR processor typically contains three separate areas of read/write (R/W) memory. The lowest section contains the thirty-two general-purpose working registers, followed by the sixty-four I/O registers, which are then followed by the internal SRAM.

The general-purpose working registers are just that: they are used for the storage of local variables and other temporary data used by the programme

while it is executing, and they can even be used for the storage of global variables. The sixty-four I/O registers are used as the interface to the I/O devices and peripherals on board the microcontroller. And the internal SRAM is used as a general variables storage area and also for the processor stack.

6.6.2.1 Working registers

All AVR processors have 32 general-purpose registers (working registers), Figure 6.22. The working registers occupy the lowest thirty-two cells in the data memory. These registers are used much like the storage locations in a calculator, in that they store temporary or intermediate results. Sometimes they are used to store local variables, sometimes global variables, and sometimes the pointers into memory that are used by the processor. In short, the processor uses these thirty-two working registers as it executes the programme. The programmer has control on the thirty two working registers only when he is using assembly language to write his programme. They are out of the programmer control in case of writing the programme in C where the registers are controlled by the C compiler. The registers are normally named R0 through R31. As mentioned in Chapter 4, the user can give names for the working registers using the assembler directives.

```
MyTable:
.DW 0x1248,0x2458,0x3344,0x4466,0x5666 ; The table values, organized as words
.DW 0x5789,0x679A,0x22AB,0x9AB8,0x33AC ; the rest of the table values
Read5:    LDI ZH, HIGH(MyTable*2)       ; Address of table to pointer Z
   LDI ZL,LOW(MyTable*2)                ; multiplied by 2 for bytewise access
   ADIW ZL,10                           ; Point to fifth value in table
   LPM                                 ; Read least significant byte from
                                        ; programme memory
   MOV R24,R0                           ; Copy LSB to 16-bit register
   ADIW ZL,1                            ; Point to MSB in programme memory
   LPM                                  ; Read MSB of table value
   MOV R25,R0                           ; Copy MSB to 16-bit register
```

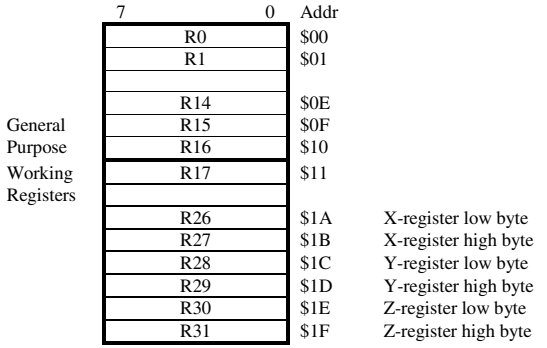


Figure 6.22 AVR CPU general-purpose (Register File).

The AVR register file is broken up into 2 parts with 16 registers each, R0 to R15 and R16 to R31. All instructions that operate on the registers have direct access to the registers and need one cycle for execution. The exception is the instructions that use Immediate Address Mode and the instructions that used a constant mask to set or clear the bits of a register. Such instructions allow the programmer to specify a constant as an operand and they must use registers between R16 and R31. These instructions are: LDI, ANDI, ORI, SBCI, SUBI, CPI, SBRI, SER, and CBR. The general SBC, SUB, CP, AND and OR and all other operations between two registers or on a single register apply to the entire register file.

Some of the general-purpose registers have additional special functions. Registers R0 and R26 through R31 have additional functions. R0 is used in the instruction LPM (load programme memory), while R26 through R31 are used as pointer registers. The instruction LPM loads the contents of programme memory location pointed out by the contents of register Z to register R0.

Pointer-register

The register pairs R26:R27, R28:R29 and R30:R31 have an additional special function. Because of the importance of this extra function, each pair has an extra name in the assembler; X, Y and Z respectively, Figure 6.23. Each pair forms a 16-bit register that can be used as a pointer pointing to a location within the address space of the SRAM. To point to a location in programme memory we must use the Z register. The lower register of each pair stores the lower byte of the address and the upper register stores the higher byte of the address. The Y pointer for example, the lower register (R28) is named YL and R29 is called YH. The names XL, XH, YL, YH, ZL and ZH are defined in the standard header file for the chip. To load a 16-bit address to Y pointer we use the following statements:

```
.EQU Address = RAMEND      ; RAMEND is the highest 16-bit address
                           ; in SRAM
LDI YH, HIGH(Address)      ; Load the higher byte of the address
LDI YL, LOW(Address)       ; Load the lower byte of the address
```

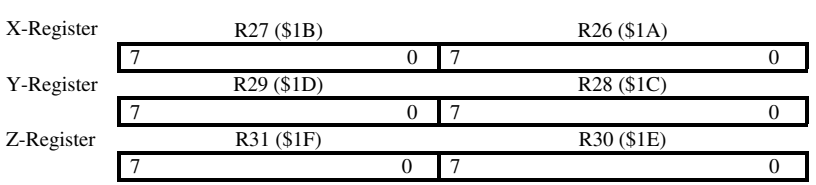


Figure 6.23 X-, Y-, and Z-registers.

Two special instructions are designed specially to access the SRAM using pointers: LD (LoaD) and ST (STore). LD is to read the contents of the memory location and ST is to write into the location.

There is only one command for the read access to the programme memory. It is defined for the pointer pair Z and it is named LPM (Load from Programme Memory), see Chapter 4.

It is very common also to use the pointers to access tables stored at the programme memory. The following is an example of a table with 10 different 16-bit values stored at the programme memory and the programme is written to read the fifth table value to R25:R24.

6.6.2.2 I/O Registers

The I/O registers occupy the next highest sixty-four bytes in the data memory space (refer to Figure 6.13). Each of these registers provides access to the control registers or the data registers of the I/O peripherals contained within the microcontroller. The programmer uses the I/O registers extensively to provide interface to the I/O peripherals of the microcontroller.

Each I/O register has a name as shown in Table 6.2. The registers can be accessed as I/O registers with addresses between $00 and $3F or as SRAM with addresses between $20 and $5F. A C language programmer will most commonly use the I/O register name. The two different numeric addresses are important when writing in assembly language because some of the instructions relate to the SRAM address and some relate to the I/O address.

However the C language compiler does not inherently know what these names are or know what addresses to associate with the names. Each programme contains a #include header file that defines the names and the associated addresses for the C language compiler. The C language programme in Figure 6.24 demonstrates these concepts.

```
#include <90s8535.h>
main ()
{
       DDRA = OxFF;   //all bits of Port A are output
       DDRB = 0x00;   //all bits are input
       while (1)
       {
          PORTA = PINB;  //read port B and write to A
       }
}
```

Figure 6.24 Port initialization using register names.

Table 6.2 Shows the I/O space of ATmega8515/AT90S8515.

Component	Register	Name	Address
Accumulator	Status Register	SREG	0×3F
Stack	Stack pointer High/Low	SPH/SPL	0×3E/0×3D
External SRAM/ External Interrupt	MCU General Control Register	MCUCR	0×35
External Interrupt	General Interrupt Mask Register	GIMSK	0×3B
	General Interrupt Flag Register	GIFR	0×3A
Timer Interrupts	Timer/counter Interrupt Mask Register	TIMSK	0×39
	Timer/counter Interrupt Flag Register	TIFR	0×38
Timer 0	Timer/Counter 0 Control Register	TCCR0	0×33
	Timer/Counter 0 (8 bits)	TCNT0	0×32
Timer 1	Timer/Counter 1 Control Register A	TCCR1A	0×2F
	Timer/Counter 1 Control Register B	TCCR1B	0×2E
	Timer/Counter 1 High/ Low byte	TCNT1	0×2D/0×2C
	T/C 1 Output Compare Register A High/Low byte	OCR1A	0×2B/0×2A
	T/C 1 Output Compare Register B High/Low byte	OCR1B	0×29/0×28
	T/C Input Capture Register High/Low byte	ICR1L/H	0×25/0×24
Watchdog Timer	Watchdog Timer Control Register	WDTCR	0×21
EEPROM	EEPROM Address Register High/Low byte	EEARH/L	0×1F/0×1E
	EEPROM Data Register	EEDR	0×1D
	EEPROM Control Register	EECR	0×1C
SPI	Serial Peripheral Control Register	SPCR	0×0D
	Serial Peripheral Status Register	SPSR	0×0E
	Serial Peripheral I/O Data Register	SPDR	0×0F
UART	UART I/O Data Register	UDR	0×0C
	UART Status Register	USR	0×0B
	UART Control Register	UCR	0×0A
	UART Baud Rate Register	UBRR	0×09
Analogue Comparator	Analogue Comparator Control and Status Register	ACSR	0×08
I/O-Ports	Data Register Port A	PORTA	0×1B
	Data Direction Register Port A	DDRA	0×1A
	Input Pins Port A	PINA	0×19
	Data Register Port B	PORTB	0×18
	Data Direction Register Port B	DDRB	0×17
	Input Pins Port B	PINB	0×16
	Data Register Port C	PORTC	0×15
	Data Direction Register port C	DDRC	0×14
	Input Pins Port C	PINC	0×13
	Data Register Port D	PORTD	0×12
	Data Direction Register port D	DDRD	0×11
	Input Pins Port D	PIND	0×10

This programme reads the pins of Port B using the PINB I/O register and writes the results to the output latches of Port A using the PORTA register. The C compiler gets the addresses to use with these register names from

the #include header file in the first line of the programme, in this example, 90s8535.h.

In summary, the C language programmer uses the I/O registers as the interface to the I/O hardware devices within the microcontroller. Subsequent sections of this chapter describe the use and function of each of the I/O peripherals in the microcontroller and their associated I/O registers.

Two instructions are to be used to access the different I/O registers; IN and OUT. The IN and OUT instructions are used to transfer data between the 32 working registers and the I/O registers.

I/O registers within the address range 0×00–$0\times1F$ are bit-accessible. The two instructions SBI and CBI can be used to access directly any bit in any I/O register.

In the following we are considering two of the I/O registers (the status register and the stack register), while the rest will be consider during discussing the hardware resources of the microcontroller.

6.6.2.3 The Status Register

The Status Register stores exceptional conditions occurring within the ALU. The width of the status register, the number of the flags it contains, the names of each flag and the state that each flag represents depend on the microcontroller or the microprocessors. The contents of the flags register may then be tested by specialized instructions or read on the internal data bus. For example, depending on the value of one of these bits, a conditional-branch instruction may be used to cause the execution of a new programme sequence.

Conditional branch instructions are used when a condition must be tested in a programme. The instruction normally tests the condition of one of the bits within the status register. If the condition is met (the bit is 1), the branch will occur; if it is not met (the bit is 0), the branch will not occur, and the programme will continue in its normal sequence.

The bits within the status register are normally set automatically by most instructions. Each manufacturer supplies a list of the status flags that are affected by each of the instructions. It is sometimes possible to set specific bits explicitly through specialized instructions. For example, the instruction CLC clears the Carry Flag C, and the instruction SEC sets the Carry flag C in the status register SREG of ATMEL AT90S8515 and ATmega8515.

Usually each bit of the status flags can be tested independently for the value 0 or for the value 1 and can result in a branch. The corresponding branch instructions will bear a different name, depending on the manufacturer. Sometimes more sophisticated instructions are available that will test combinations

of bits or combinations of conditions, such as a bit being "greater than or equal to", or "less than or equal to" another bit.

As an example of a simple conditional branch, the instruction "BBCC nocarry" will result in a branch to label nocarry if the carry flag C is cleared. "BRCS carry" will result in a branch to label carry if the C flag is 1.

In case of AT90S8515, the STATUS register at address $3F (memory address $5F) contains 8-flag bits indicate the current state of the processor as summarized in Table 6.3. All these bits are cleared (i.e., at logic "0") at reset and can be read or written to, individually, by the programme.

The STATUS register is not automatically stored by the machine during an interrupt operation and restored when returning from an interrupt routine. The instruction in an interrupt routine can modify the STATUS flag bits and so the user programme must store and retrieve the STATUS register during an interrupt.

The various flags of the STATUS register and their functions are summarized in Table 6.3. The different flags are examined below.

1. Global Interrupt Enable Bit (I)

Bit 7, I-flag, is a Global Interrupt Enable bit. Setting this bit enables all the interrupts. The individual interrupt enable control is then performed in separate control registers. If the global interrupt enable bit is cleared (zero), none of the interrupts are enabled independent of the individual interrupt enable settings.

Table 6.3 Summary of the various flags of the STATUS register.

Bit	Name	Meaning	Possible value	Command used
7	I	Global Interrupt flag	0: Disable all Interrupts	CLI
			1: Enable interrupts	SEI
6	T	Bit storage	0: Stored bit is 0	CLT
			1: Stored bit is 1	SET
5	H	Half carry-flag	0: No half carry generated	CLH
			1: Half carry generated	SHE
4	S	Sign-flag	0: The sign is positive	CLS
			1: the sign is negative	SES
3	V	2's Complement-flag	0: either the carry or overflow from bit 6 to bit 7	CLV
			1: either none of them occurred or both occurred	SEV
2	N	Negative-flag	0: Result was not negative/smaller	CLN
			1: Result was negative/smaller	SEN
1	Z	Zero-flag	0: The result has value	CLZ
			1: Result is zero	SEZ
0	C	Carry-flag	0: No carry	CLC
			1: Carry occurred	SEC

The 1-bit is cleared by hardware after an interrupt has occurred and is set by the REI instruction to enable subsequent interrupts.

2. Bit Copy Storage (T)

Bit Copy Storage (Bit 6 – T), used with instructions BLD (bit load) and BST (bit store) for loading and storing bits from one register to another. The two instructions BLD and BST use the T-bit as source and destination for the operated bit. A bit from a register file can be copied into T by the BST instruction and a bit in T can be copied into a bit in a register in the register file by the BLD instruction.

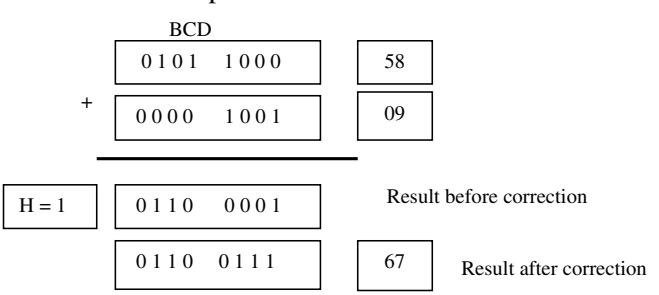
Example 6.12

```
bst   r1,   2    ; Store bit 2 of file register r1 in T-flag
bld   r0,   4    ; Load T flag in bit 4 of r0.
Clt              ; Clear T flag
```

The state of the processor, i.e. the results of the arithmetic and logical operations, has no automatic effect on the T-bit. Its value is defined explicitly by the instructions BCLR T, BSET T, BST, BLD, CLT and SET.

3. Half Carry Flag (H)

It indicates half carry (carry from bit 3 to bit 4) in some arithmetic instructions. This bit is used, in most of microcontrollers and microprocessors, during binary coded decimal (BCD) operations. BCD is a notation often used in business applications requiring exact results without the round-off errors caused by the usual two's complement notation. BCD uses a 4-bit code to represent each decimal digit. In order to provide data compaction, a standard 8-bit word will contain two BCD digits placed side by side. When performing arithmetic on bytes, an addition might generate a carry from bit 3 into bit 4, i.e., from the first BCD digit into the second BCD digit. This carry is normally undesirable and must be detected. The H-bit performs this role. The H-bit is the carry from bit 3 into bit 4. The disadvantages of BCD are that it is inefficient in its use of memory space and that it is somewhat slow in performing arithmetic calculation. Here is an example:



The carry from bit 3 to bit 4 in this example sets the H flag to1 indicating that if BCD is the notation used, a decimal adjust instruction must be used to let the final result takes the correct BCD value.

Example 6.13:

What are the value of the Half Carry flag H and the content of r21 after execution of the following instruction sequence?

```
LDI     r21, $07
LDI     r22, $09
ADD     r21, r22
```

Solution:

The binary addition that takes place in the third instruction is illustrated below:

	00000111	(r21 = 07H)
+	00001001	(r22 = 09H)
	00010000	(result in r21 = 10H)

The half carry bit H = 1, because there was a carry from bit 3 to bit 4.

In the case of the microcontrollers and microprocessors that uses BCD notation, the half carry flag H is also set if the result in the lower nibble is in the range \$0A − \$0F. In such cases the add instruction must be followed by decimal adjust instruction (e.g., DA A in case of Intel 8051) to bring results greater than 9 back into range.

Many instructions change automatically the value of the Half Carry Flag H. Besides that, it is possible to clear or to set the H-bit explicitly through the instructions CLH and SEH respectively. Also the instructions BCLR 5, can be used to clear the H-bit (bit 5 in the SREG).

Example 6.14:

```
      Add   r0, r1 ; Add the contents of r1 to that of r0 and store the result
                   ; at r1
      brhs  hset   ; Branch if half carry flag is set
      ......
hset: nop          ; Branch destination (Do nothing)
```

4. Sign Flag (S)

The S- flag (bit 4) is an exclusive OR between the negative flag N and the Overflow flag V, i.e. $S = N \oplus V$. Accordingly, this flag does not reflect the sign of the result (See N-flag).

5. Two's Complement Overflow Flag (V)

Overflow denotes that an arithmetic carry within a word has modified the value of the most significant bit. This results in a sign error when the two's complement notation is used. Bit 7 in two's complement indicates the sign: 1 means negative, 0 is positive. Whenever 2 two's complement numbers are added, the carry generated (from bit position 6) during an addition or subtraction might overflow into the sign bit (bit position 7). When this happens, it might change a negative number into a positive number. The overflow bit is used to indicate this occurrence. Mathematically, the overflow is the exclusive OR of the carry bit (out of bit 7) and the carry generated from bit 6 into bit 7. The overflow will normally be used only when performing two's complement arithmetic. When unsigned numbers are added, the overflow bit can be ignored. For example, the following addition causes an overflow and sets the V bit in the status register:

Hex	Binary	Decimal
0E	00001110	14
+7E	+01111110	+126
8C	10001100	140

As a signed number, \$8C represents −116, which is clearly not the correct result of 140; therefore, V bit is set. In this example, the carry flag C is zero, at the same time, there is carry from bit 6 into bit 7, as a result:

$$V = (C \oplus \text{carry from bit 6 into bit 7}) = 0 \oplus 1 = 1$$

Example 6.15:

What are the state of the overflow flag and the contents of r21 after the execution of the following instruction sequence?

```
Ldi     r21, $FF
Ldi     r20, $0F
Add     r21, r20
```

Answer:

V = 0, contents of r21 = \$0E

Discussion:

The register r21 is initialised with \$FF, which as signed number equals $(-1)_{10}$. The register r20 is initialised with \$0F, which equals 15_{10}. The result of the addition is $(15) + (-1) = 14 =$ \$0E. During adding the two binary

numbers, there is carry from bit 6 to bit 7 and also there is the normal carry (C = 1). As mentioned before, the overflow bit V is the exclusive OR of the carry bit C and the carry generated from bit 6 into bit 7. Accordingly, $V = 1 \oplus 1 = 0$.

Many instructions automatically affect the value of the V-bit. It is possible also to clear or set the V-bit explicitly by using the instructions CLV, SEV and BCLR 4 (V is bit 4 in the SREG).

6. Negative Flag (N)

The negative flag N (bit 2 in the status register) indicates a negative result after the different arithmetic and logic operations. The N-bit is directly connected to bit position 7 of the result. Recall that in two's complement notation, a 1 in bit position 7 indicates a negative number, hence the name of the status bit.

7. Zero Flag (Z)

It indicates zero result after an arithmetic or logical operation. The Z-bit is set to 1 whenever the result of an operation is 0. It is used by arithmetic instructions to determine whether or not a result is 0, and by logic operations such as COMPARE. The latter implements a logical XOR between the word being tested and the pattern to which it is being compared. Whenever the result of a comparison is successful, the Z-bit is set to 1.

8. Carry Flag (C)

It indicates a carry in arithmetic or logical operation. It is the overflow of the 8-bit result. However, the word "overflow" has a specific meaning that will be explained below. As an example, if the following two binary numbers are added:

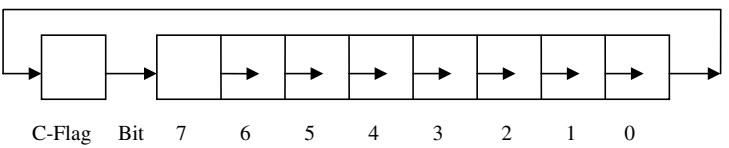
		Binary		HEX
		1 1 1 0 1 1 0 1		ED
	+	1 0 0 0 0 0 0 0		80
= 1 (Carry)		0 1 1 0 1 1 0 1	1	6D

the result generates a carry (i.e., a ninth bit). The 1 generated by this addition is stored by the ALU in the C-bit, where it can be tested. Special instructions, such as "ADD with carry" can be used to add the carry automatically to the result of the next addition. A test can also be performed by the programmer, using a conditional-branch instruction, to determine whether or not some action should be undertaken.

The carry bit performs another different and independent function than that mentioned above. It is used as a spill-out during the shift and rotate operations. When used as a spill-out, the carry behaves again as a ninth bit of the result, which justifies the merging of these two functions into the same bit.

Example 6.16:

```
ror     r15     ; Rotate right the contents of register r15
```



The execution of this instruction results in: shifting all the contents of r15 register one bit to the right, shifting the carry flag C into bit 7 of r15 and put 0 into the carry flag.

6.6.2.4 Stack pointer register

This register is a 1 byte wide for processors that have up to 256 bytes of SRAM and is 2 bytes wide (called SPH and SPL) for those processors that have more than 256 bytes SRAM. This register is used to point to the area in SRAM that is the top of the stack. The stack as mentioned in Chapter 2 has more than one application in computer, one of them is to use it to store the return addresses by the processor during an interrupt and subroutine call. Since the SP is initialized to $00 (or $0000 for a 2-byte SP) at reset, the user programme must initialize the SP appropriately, as the SRAM starting address is not $00. SRAM starting address is $60. The stack grows down in memory address — i.e., pushing a value on stack result in the SP getting decremented. Popping a value out of stack increments the SP.

6.6.3 SRAM Data Memory

The AVR memory map allocates two sections for the SRAM, one for the internal SRAM and the other for the external SRAM. The SRAM is used to store variables that do not fit into the registers and to store the processor stack. SRAM can be accessed by more than one addressing mode. The following addressing modes are in use with SRAM:

- Direct addressing: This mode reaches the entire memory space.
- Indirect addressing mode: This includes also pre-decrement and post-increment addressing modes.

- Displacement addressing: The address has two address fields one, at least, is explicit. The contents A of one of them is used directly, the contents of the other field refers to a register whose contents are to be added to A to produce the effective address of the required location.

The indirect and displacement addressing modes allow the user to build a ring buffers for interim storage of constant values or calculated tables. (Note: The reader is referred to Chapter 4 for the details of the addressing modes).

The CPU can access the contents of any of the file register but it cannot access directly the contents of any location of the SRAM. To operate on the contents of any of the locations of the SRAM, a register is usually used as interim storage. Special load instructions must be used to read a location of the SRAM and load it into the intermediate register (LDI, LD, LDD, LDS instructions). Similarly store instructions (ST, STD and STS instructions) are to be used to store the contents of the intermediate register into a given location at the SRAM. The Load and Store instructions contain the location of the SRAM and the name of the intermediate register. X, Y, and Z-registers are normally used as pointers to point to the SRAM location. The following example shows how to copy a value in SRAM to register R2, add it to the contents of register R3 keeping the result at R3, at the end it writs back the result to the SRAM.

```
LDS R2, 0x0075 ; Load register R2 with the
               ; contents of
               ; SRAM location 0x0075
ADD R3, R2     ; Add the contents of registers R2
               ; and R3
               ; and store the result at R3
STS 0x0075, R3 ; Copy the contents of register R3
               ; at SRAM
               ; location 0x0075.
```

This programme fragment shows that operating on values stored in SRAM are much slower than doing the same operations on values stored at registers.

Use of SRAM

We use the SRAM for one or more of the following purposes:

- As extra storage space
- Possibility of using more effective addressing modes
- As a stack

6.6.3.1 As extra storage

In many applications, the capacity of the internal SRAM is not enough. The AVR offer the opportunity of extending the storage capacity by connecting an external SRAM to the system. The assembler handles the external SRAM exactly in the same way as the internal SRAM without the need to any extra commands.

6.6.3.2 More addressing modes

As mentioned before the SRAM may be accessed by any of the following addressing modes:

- Direct,
- Indirect with Displacement,
- Indirect,
- Indirect with Pre-decrement, and
- Indirect with Post-increment.

6.6.3.3 Use of SRAM as stack

The most common use of SRAM is its use as stack. The stack mechanism is given before in Chapter-4 and it is called Last-In-First-Out (LIFO). The CodeVisionAVR C language compiler actually implements two stacks: The system stack starts at the top of the SRAM area and is used to store return addresses and processor control information. The data stack starts below the system stack and works its way down through memory, in a manner similar to the system stack, and is used to store temporary data, such as the local variables used in a function.

Stack Pointer:

The stack pointer is a 16-bit-pointer, accessible like a port. Two of the I/O registers are used to form the stack pointer (SP). These are SPH at memory location $3D and SPL at memory location $3E. Figure 6.25 shows the SP of ATmega8515/AT90S4414/8515.

The stack pointer keeps track for the address of the top of the stack (TOS), representing the next available location to store data onto the stack. It works

$3E($3E)	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	PL
$3D($5D)	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	PH
Read/Write	/W	/W	/W	/W	/W	/W	/W	/W	

Figure 6.25 Stack pointer register.

as follows:

1. When new data (or address) is ready to be placed on to the stack, the processor uses the SP to find where in memory to place the data or the address. For example, if the SP contains the value 0×300, the data will be placed in SRAM memory location 0×300.
2. The SP is then decremented by one (i.e. to location 0×2FF) when data is pushed onto the stack with the PUSH instruction or decremented by two (to location 0×2FE) when an address is pushed onto the stack with subroutine calls and interrupts. After that the processor begins to execute the next instruction. In this manner, the SP always contains the location of the next memory cell of the stack that is available to be used.

At some later point, the processor may want to retrieve the last piece of data (or address) pushed on to the stack. The retrieval process works as follows:

1. When the processor is ready to retrieve or pop data off the stack, it reads the SP and immediately increments the SP contents by one if we are popping data by POP instruction and increment it by two if we are popping an address with RET instruction from a subroutine or RETI instruction when returning from an interrupt. The new contents of SP point now to the top of the stack (TOS).
2. The processor uses the address in the SP to pop or retrieve the data from the stack. Since the data has been retrieved from this location, its address is left in the SP as being the next available location on the stack.

To construct the stack the programmer must initialize the stack pointer by loading it with the highest available SRAM address. This is shown in the following code where we are assuming that the stack grows downwards, i.e. towards lower addresses!).

```
.DEF FirstRegister = R16
LDI FirstRegister, HIGH(RAMEND)     ; Upper byte
OUT SPH, FirstRegister              ; to stack pointer
LDI FirstRegister, LOW(RAMEND)      ; Lower byte
OUT SPL, FirstRegister              ; to stack pointer
```

The first line is for the assembler to confirm the name selected for register R16, the next two lines are used to load the upper byte to stack pointer and the last two lines to load the lower byte of the address to the stack pointer. The programme starts the stack at the end of the memory space of the processor. The RAM end is normally referred to by RAMEND which is specific for each

processor type. RAMEND is defined in the INCLUDE file for the processor type. For example, in case of AVR8515, the file 8515def.inc has the line:

.equ RAMEND = $25F ; Last On-Chip SRAM Location

The file 8515def.inc is included with the assembler directive

.INCLUDE "C:\somewhere\8515def.inc"

at the beginning of the assembler source code.

After defining the initial address of the TOS, there is now need for the programmer to be warred or to give any attention to the contents of SP while executing the programme. To push the contents of any register he will just use the instruction PUSH FirstRegister. The process will push the contents of R16 to the TOS. Of course it is not important for us to know where this TOS is, it is important that when we want to get back the data we just pushed we get it. This is the case, if you need at any point in the programme to get the last data you pushed just write the instruction POP FirstRegister.

In case of subroutines, the CALL statement is doing the function of the PUSH instruction in case of pushing data. The CALL instruction pushes the return address (the current programme counter value) onto the stack. After pushing the address the programme jumps to the subroutine. Each subroutine ends with a return RET instruction. After the execution of the subroutine, the instruction RET pops the return address from the stack and loads it back into the programme counter. Programme execution is continued exactly one instruction behind the call instruction.

```
RCALL Somewhat              ; Jump to the label somewhat
    [the rest of the programme]  ; here we continue with the programme.
```

The RCALL statement lets the system jumps to the label *somewhat* which exists somewhere in the programme code.

```
Somewhat:                   ; this is the jump address
  [the reset of the subroutine] ; The processor starts executing
                              ; the subroutine.
                              ; When finishing execution we want to jump
                              ; back
                              ; to the calling location to continue the main
                              ; programme

  RET                         ; This takes us back to the main programme.
```

During execution of the RCALL instruction the already incremented programme counter, a 16-bit-address, is pushed onto the stack, using two pushes. By reaching the RET instruction the content of the previous programme counter is reloaded with two pops and execution continues there.

If the programme using nested subroutine, i.e. to call a subroutine within a subroutine, the system will do the same every time there is a CALL. The return addresses will be popped in the reverse order of pushing them. This means that we have to complete the most outer subroutine followed by the next one till it goes back to the main programme. This is how it should be.

Common mistakes with the stack operation

Some of the most common mistakes that may arise during using the stack are:

1. ***Stack pointer setting***: The first problem is the use of the stack without first setting (initializing) the stack pointer. When we start the system, the default value of SP is zero. This initial value means that the TOS is referring to register R0. If the programmer starts by pushing a byte to the stack, this byte will go to R0 overwriting its previous contents, which may be by itself a problem. At the same time the push instruction part of it is to decrement the SP to point to the new TOS. Decrementing SP results in 0×FFFF. Unless you are using an external RAM, 0×FFF is an undefined position. A RCALL and RET will return to a strange address in programme memory.
2. Another common mistake is to try to pop some data from an empty stack. You have to be sure that you pushed something before using POP in the programme.
3. Another mistake is to try to push data to a full stack (stack overflow). This happens in case of a never-ending recursive call.

The major concept relative to the stack is that as data is pushed onto the stack, the stack uses progressively more memory in the SRAM area, starting at the top of SRAM and working downward. As data is popped off the stack, the memory that has been previously used is released, and the available stack location moves up in memory. At the same time, the SRAM memory is being used for the storage of variables starting at the bottom of memory and working upwards. A microcontroller has limited SRAM space, so it is important to be sure that the stack does not come down or the data does not move up far enough to interfere with one other. Overwriting the stack with data or overwriting data with the stack will cause unpredictable results in your programmes.

Internal SRAM

This is available on most of the AVR processors except the baseline processors such as the ATM90S1200. The amount of SRAM varies between 128 bytes to 4K bytes. The SRAM is used for stack as well as storing variables.

Data is usually stored in the internal SRAM starting at the bottom of the SRAM, and the processor stack (or stacks) start at the top of memory and utilize memory from the top down. As data is pushed onto the stack, the stack uses progressively more memory in the SRAM area, starting at the top of SRAM and working downward. At the same time, the SRAM memory is being used for the storage of variable starting at the bottom of memory and working upward. A microcontroller has limited SRAM space, so it is important to be sure that the stack does not come down or the data does not move up far enough to interfere with one other. Overwriting the stack with data or overwriting data with the stack will cause unpredictable results in your programme.

Note: The CodeVisionAVR C language compiler actually implements two stacks: The system stack starts at the top of the SRAM area and is used to store return addresses and processor control information. The data stack starts below the system stack and works its way down through memory, in a manner similar to the system stack, and is used to store temporary data, such as local variables used in a function.

External SRAM

This is possible only on the larger processors of the AVR family. Those processors that have external data and memory access ports (as AT90S8515 and ATmega8515), can use any available external SRAM the user may decide to implement.

To access the external SRAM we use the same instructions as for accessing the internal SRAM access. When internal data SRAM is accessed, the read and write strobe pins (/RD and /WR) are inactive during the whole access cycle. The external data SRAM physical address locations corresponding to the internal data SRAM addresses cannot be reached by the CPU. To enable the use of the external SRAM we must set the SRE bit in the memory control register- MCUCR.

Because of the limited number of the available pins of the chip many pins has more than one function; alternative functions. The Port A and Port C pins have alternative functions related to the operational external data SRAM. The alternative functions of Port A and Port C pins are activated when we enable the signal XMEM (eXternal MEMory). Enabling XMEM allows the use of all the address space outside the internal SRAM by using Port A and Port C. In

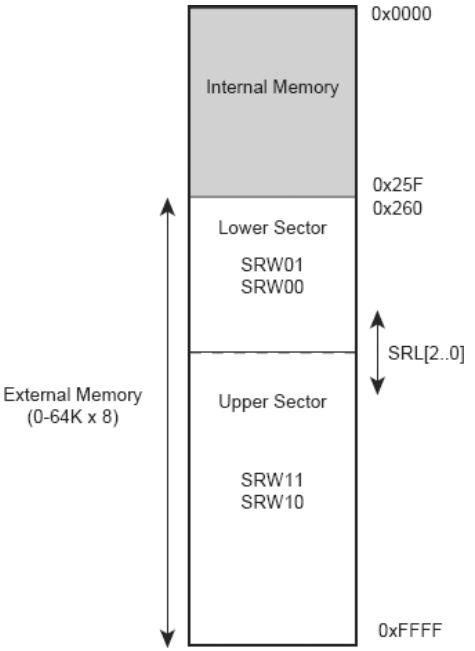


Figure 6.26 External memory with sector select.

this case, Port A is automatically configured to be the multiplexed low-order address/data bus during access to the external data memory, and Port C is automatically configured to be the high-order address byte.

Figure 6.26 shows the memory configuration when the external memory is enabled.

6.6.3.4 Interface to External SRAM

In case of AVR the interface to the SRAM consists of:

- Port A: Multiplexed low-order address bus and data bus (AD7 …AD0)
- Port C: High-order address bus (A15…A8)
- The ALE-pin: Address latch enable
- The /RD and /WR-pin: Read and write strobes.

The control bits for the External Memory Interface are located in three registers:

- MCUCR: The MicroController Unit Control Register,
- EMCUCR: The Extended MicroContrller Control Register, and
- SFIOR: The Special Function IO Register — SFIOR.

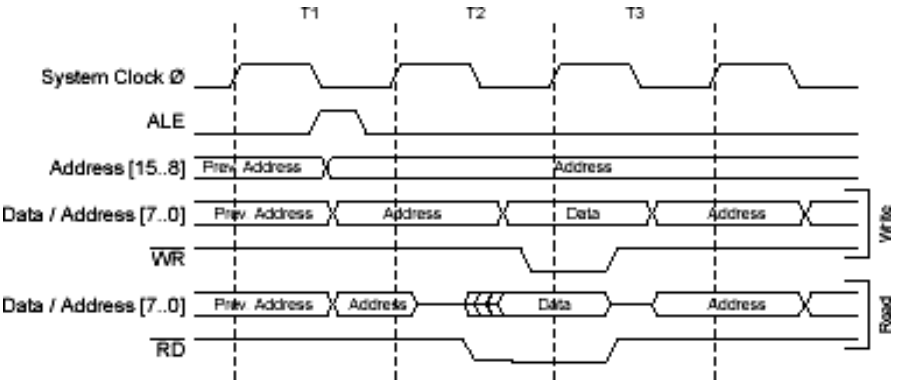


Figure 6.27 External data SRAM memory cycle without wait state.

The external data SRAM is enabled by setting the External SRAM enable bit SRE located at MCUCR control register. Setting the external memory (XMEM) interface results in overriding the setting of the data direction registers of Port A and Port C (The ports dedicated to the interface in case of ATmega). The XMEM interface will auto-detect whether an access is internal or external. If the access is external, the XMEM interface will output address, data, and the control signals on the ports according to Figure 6.27 (this figure shows the wave forms without wait states). At the falling edge of ALE goes there will be a valid address on port A (AD7:0). ALE is low during a data transfer. When the XMEM interface is enabled, also an internal access will cause activity on address-, data-, and ALE ports, but the /RD and /WR strobes will not toggle during internal access.

When the SRE bit is cleared (zero), the external data SRAM is disabled, and the normal pin and data direction settings are used.

Figure 6.28 sketches how to connect an external SRAM to the AVR using 8 latches (an octal latch) which are transparent when G is high.

Default, the external SRAM access is a three-cycle scheme as depicted in Figure 6.27. When one extra wait state is needed in the access cycle, set the SRW bit (one) in the MCUCR register. The resulting access scheme is shown in Figure 6.28. In both cases, note that PORTA is data bus in one cycle only. As soon as the data access finishes, PORTA becomes a low order address bus again.

Summary of "How to access External SRAM"

- The same instructions are used to access the internal and the external SRAM. The /RD and /WR signals are only active when we access the

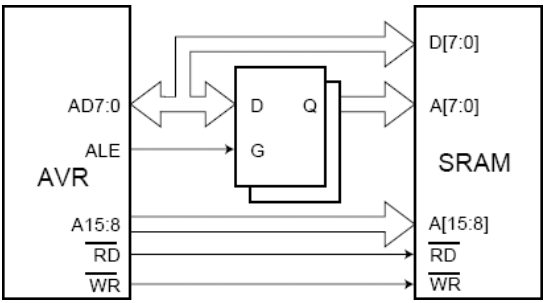


Figure 6.28 External data SRAM connected to the AVR.

external SRAM. The external data SRAM physical address locations corresponding to the internal data SRAM addresses cannot be reached by the CPU. To enable the external SRAM, we must set the SRE bit (bit 7) in the MCUCR register.

- The Port A and Port C pins work in the alternative functions mode. Port A in the alternative function mode works as multiplexer to multiplex low-order address (A0 to A7)/data (D0 to D7) bus during accesses to the external data memory. Port C in the alternative function mode configured to be the high-order address byte.
- The output ALE signalizes the state of the multiplexer.
- When ALE goes from high to low, the valid low-order addresses on Port A are saved in a byte-wide latch. The outputs of this latch are connected to the low-order address pins of the external memory. The high-order addresses are stable on Port C from the beginning of the memory access, before the read strobe (/RD) or write strobe (/WR) occurs, so all address lines are stable for addressing external memory.
- Reading data from external memory, the rising edge of the read strobe transfers the data to Port A.
- The rising edge of the write strobe writes the data to the external memory.
- The default access time for an external SRAM access is three clock cycles. This can be increased to four clock cycles by setting the SRW bit (bit 6) in the MCUC register.
- In both cases, Port A is data bus in one cycle only. As soon as the data access finishes, Port A becomes a low-order address bus again.
- If we use part of the external SRAM as a stack, interrupts, subroutine calls and returns take four clock cycles; two cycles for pushing/popping the higher byte and two cycles to push/pop the lower byte of the address.

External Memory (XMEM) Register Description

MCU (Micro Controller Unit) Control Register — MCUCR

- Bit 7- SRE: External SRAM/XMEM Enable
  Writing SRE to one enables the External Memory Interface. The pin functions AD7:0, A15:8, ALE, /WR, and /RD are activated as the alternate pin functions. The SRE bit overrides any pin direction settings in the respective Data Direction Registers. Writing SRE to zero, disables the External Memory Interface and the normal pin and data direction settings are used. (See Figure 6.29)
- Bit 6 — SRW10: Wait State Select Bit

Memory Access and Instruction Execution

The Atmel microcontrollers are driven by the system clock, which can be sourced from outside or, if available and enabled, an internal clock can be used. This system clock without any division is used directly for all accesses inside the processor. The processor has a two-stage pipeline, and instruction fetch/decode is performed concurrently with the instruction execution. This is illustrated in Figure 6.30.

Once the instruction is fetched, if it is an ALU-related instruction, it can be executed by the ALU, as illustrated in Figure 6.31, in a single cycle.

On the other hand, and as mentioned before, the SRAM memory access takes two cycles. This is because the SRAM access uses a pointer register for

Bit	7	6	5	4	3	2	1	0
	SRE	SRW10	SE	SM1	ISC11	ISC10	ISC01	ISC00
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Figure 6.29 MCUCR register.

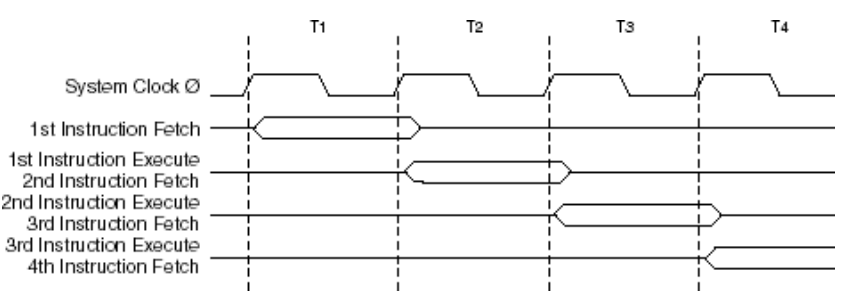


Figure 6.30 Instruction fetch.decode and instruction execution.

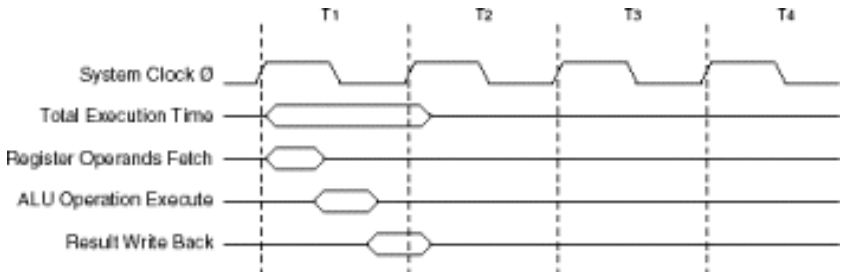


Figure 6.31 ALU execution consisting of register fetch, execute, and write back.

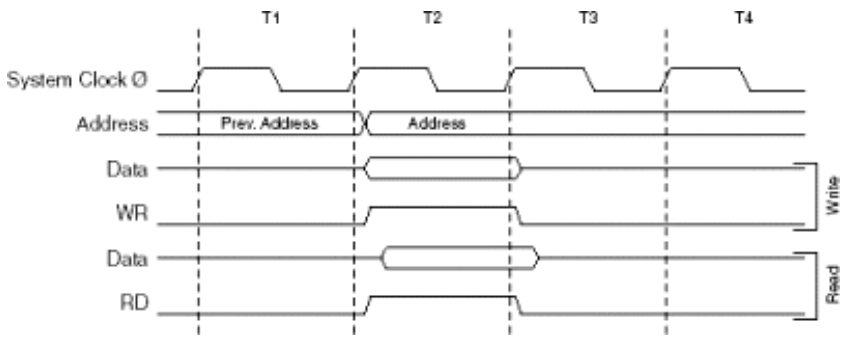


Figure 6.32 On-chip SRAM data access cycles.

the SRAM address. The pointer register is one of the pointer registers (X, Y, or Z register pairs).

- The first clock cycle is needed to access the register file and to operate upon the pointer register (the SRAM access instructions allow pre/post-address increment operation on the pointer register).
- At the end of the first cycle, the ALU performs this calculation, and then this address is used to access the SRAM location and to write into it (or read from it into the destination register), as shown in the Figure 6.32.

6.6.4 EEPROM Memory

The EEPROM section of memory is an area of read/write memory that is non-volatile. It is typically used to store data that must not be lost when power is removed and reapplied to the microcontroller. The EEPROM is available on almost all AVR processors and is accessed in a separate memory map. The starting address of the EEPROM is always at 0x000 and goes up to a

maximum value that is dependent on the specific microcontroller in use. The ATmega8515/AT90S8515 contains 512 bytes of data EEPROM memory.

Although the EEPROM section of memory may be both read and written, it is seldom used for general variable storage. The reason is that EEPROM memory is very slow to write, and it can take up to one millisecond to complete a write operation on one byte of memory. The extensive use of this memory for variable storage would slow the processor down appreciably. Also, EEPROM memory can withstand only a limited number of write cycles, as was mentioned before.

6.7 Intel Memory System

6.7.1 Internal Code Memory of 8751/8951

Both EPROM and Flash memories are available for internal storage of programmes in MCS-51 microcontrollers. Table 6.4 shows some of the devices that can be used as standalone microcontrollers.

When downloading a programme into the microcontroller, the two types of memory are programmed in very similar ways. The main difference between the types lies in the method of erasing the memory prior to reprogramming. The EPROM is erased by exposure to ultra-violet light for a period up to 30 minutes. The Flash memory is electrically-erased by a combination of programming signals.

The basic circuit for programming the 4 kbyte versions of both types of device is shown in Figure 6.33.

To write a programme into the EPROM/Flash, the programming-unit puts the addresses of the programme bytes onto Port 1 and onto lines 0 to 3 of Port 2. The bytes of the programme code are put onto Port 0. The data is transferred according to the bit settings shown in Tables 6.5 (EPROM) and 6.6 (Flash). The 8052 has a larger memory and therefore also uses pin P2.4 as an address line.

Table 6.4 Specifications of MCS-51 family.

Device	Memory type	Internal programme memory size (Kbytes)	Internal data RAM size (bytes)
8751 AH	EPROM	4K	128
8751BH	EPROM	8K	128
8752BH	EPROM	8K	256
8951	Flash	4K	128

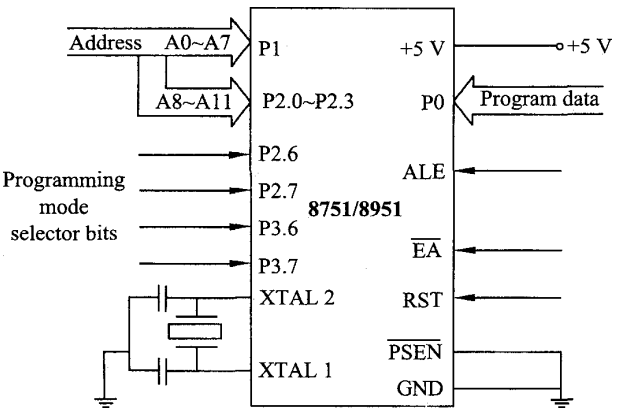


Figure 6.33 Circuit connections for programming the 8751AH and 8951.

Table 6.5 Bit settings for programming the 8751 EPROM device.

Action	RST	/PSEN	ALE	/EA	P2.6	P2.7	P3.6	P3.7
Programme code	1	0	50 ms low pulse	Vpp	0	1	X	X
Verify code	1	0	1	1	0	0	X	X
Security lock	1	0	50 ms low pulse	Vpp	1	1	X	X

Table 6.6 Bit settings for programming the 8951 Flash device.

Action	RST	/PSEN	ALE	/EA	P2.6	P2.7	P3.6	P3.7
Programme code	1	0	50 ms low pulse	Vpp	0	1	1	1
Verify code	1	0	1	1	0	0	1	1
Security lock	1	0	50 ms low pulse	Vpp	1	1	1	1
Erase chip	1	0	10 ms low pulse	Vpp	1	0	0	0

The required values of Vpp on the /EA pin vary considerably. The EPROMs required 21 volts, while the Flash memory can be programmed with either 5 or 12 volts depending on the version of the chip. The security lock can be set to prevent unauthorized people from reading or changing the code in the programme memory.

6.7.2 Adding External Code Memory Chip

If the programme memory on the chip is insufficient for a particular application, the programme can be placed partially or completely in external memory using EPROM chips. Table 6.7 shows the range of memory chips that can be used.

Figure 6.34 shows the connection pins on the 64 kbyte EPROM 27512. In addition to data and address lines, there is a /CE (Chip Enable) line. When

Table 6.7 EPROM chips for external code memory.

EPROM device	Memory size (bytes)	Number of address lines	Package type
2716	2048	11 lines	24-pin DIL
2732	4096	12 lines	24-pin DIL
2764	8192	13 lines	28-pin DIL
27128	16384	14 lines	28-pin DIL
27256	32768	15 lines	32-pin DIL
27512	65536	16 lines	28-pin DIL

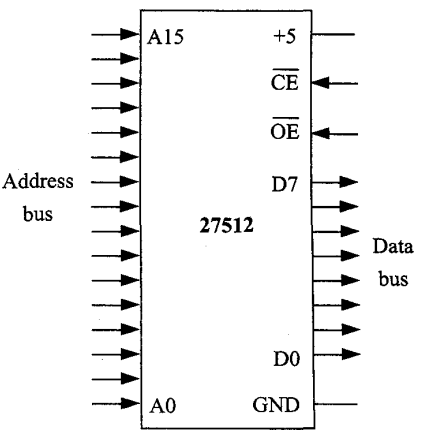


Figure 6.34 Connections of the 64 kbyte EPROM.

/CE is low the device is turned on, and when it is high the data lines change to high-impedance mode. The /OE (Output Enable) toggles the device between programming and operating mode. In normal use it is connected to Ground.

When the programme is stored entirely in an external EPROM, the /EA pin on the microcontroller must be grounded. However, when the /EA pin is connected to +5 V, the addresses in the programme between 0000 h and OFFFh (the lowest 4096 addresses) will be fetched from the internal EPROM, and addresses above OFFFh will be loaded from an external EPROM. (In the case of the BH versions of the chip, this limit is 1FFFh.)

Expansion of code memory

An external EPROM can be added to the 8751 circuit by using the 16 pins of ports P0 and P2 as the data and address buses. The circuit configuration is shown in Figure 6.35.

The 74LS373 data latch has the connections shown in Figure 6.36.

The EN (Enable) effectively closes all the switches when it is high, and transfers the binary values D0 to D7 to the memory units. This is called the

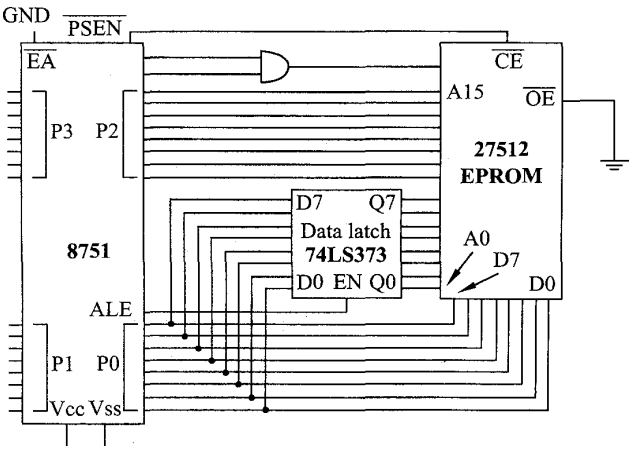


Figure 6.35 8751 chip with 64 kbyte external EPROM.

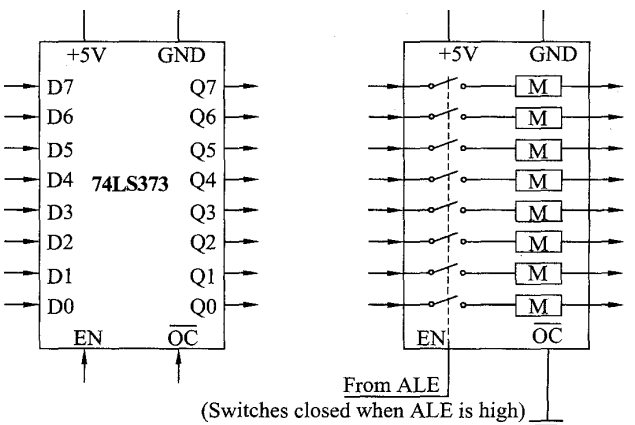


Figure 6.36 The 74LS373 data latch and its equivalent switching circuit.

"transparent" state, in which the outputs are the same as the inputs. When EN goes low, the outputs Q0 to Q7 retain the values in the memory units until new values are received. The /OC (Output Control) pin is usually grounded.

The timing diagram for expanded code memory

The timing diagram for the 8051 connected to an external EPROM is shown in Figure 6.37. Because of using address/data multiplexing, every machine cycle is divided into six state cycles, each comprising two clock cycles. For one and a half state cycles (3 clock cycles) the data outputted on P0 is the low byte of the address. During the next three clock cycles, the instruction is

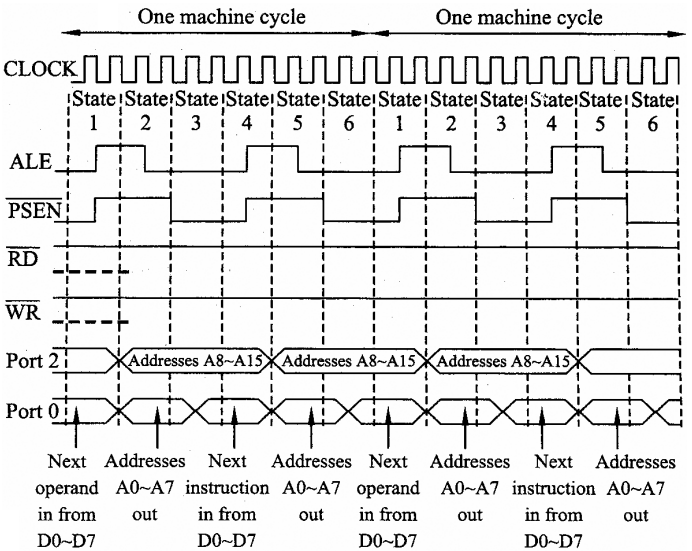


Figure 6.37 Address/data multiplexing of data during fetch from EPROM.

read from the EPROM pins D0–D7 into P0. The ALE and /PSEN lines are low twice in each machine cycle, which allows two fetches from the EPROM during that period. Thus it is possible to fetch an opcode and its operand from the EPROM in one machine cycle.

The microcontroller reads the data from the EPROM at the mid-point of State 1 and State 4, just before /PSEN disables reading.

6.7.3 Adding Extra RAM

Figure 6.38 shows the arrangement for adding an external RAM chip of 32 kbytes to a microcontroller.

The configuration for the addition of RAM to Intel microcontroller is exactly the same as the case of AVR. In case of Intel it is also similar to that for the EPROM, except that the /RD and /WR lines are used instead of the /PSEN line. The 74LS373 latch performs the same function, and its /OC pin is again connected to ground.

6.7.4 Adding both External EPROM and RAM

In addition to the 64 kbytes of EPROM which can be connected to the microcontroller, a further 32 kbytes of RAM can be interfaced onto the same

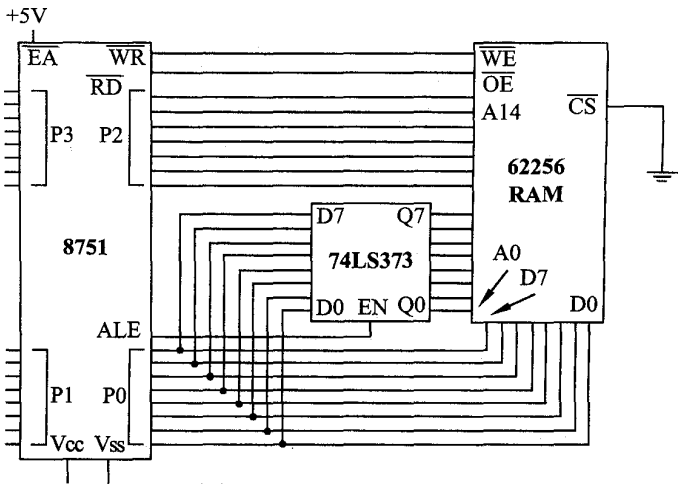


Figure 6.38 8751 chip with 32 kbyte external RAM.

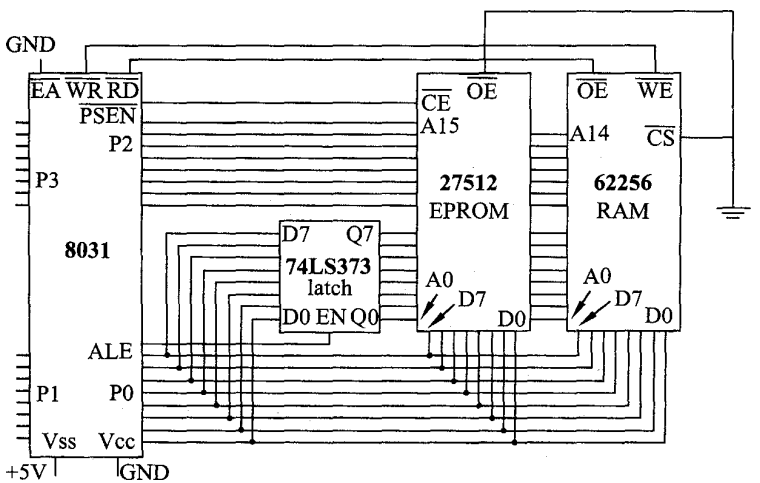


Figure 6.39 8031 chip with 64 kbyte EPROM and 32 kbyte RAM.

bus. As seen in Figure 6.39, the address and data lines are wired in parallel, but the use of the /PSEN, /WR and /RD lines ensures that only the correct bytes are transferred between the microcontroller and the EPROM or RAM. Although this configuration may be used with the EPROM or Flash versions of the MCS-5 1, it is most commonly used with the ROMless 8031 as shown.

Example 6.20: Connection of the 8255A at a specific memory address

In this example we are going to use the knowledge we gained about chip select decoder and about expanding the RAM.

The task of the example is to allocate the internal registers of the 8255A to unique addresses. To achieve that, it is necessary to decode all the address lines by means of logic circuits. The following example is given to illustrate a possible case. It is required to locate the four registers of the 8255A at the addresses shown in Table 6.8.

This can be achieved with a NOR gate and a NAND gate, as shown in Figure 6.40.

Example 6.21: Use of an FPGA for multi — peripheral systems

The circuit of Figure 6.40 requires a NOR gate with 13 inputs, which is obviously not a standard component. Therefore, in practice, decoding is not usually done using discrete chips. Instead an FPGA (Field Programmable Gate Array) is often employed. This is a chip containing a very large number of gates, whose interconnection can be made by means of software.

Figure 6.41 shows how several 8255A devices can be connected to a microcontroller using an FPGA to decode the addresses for the registers in the different chips.

Table 6.8 Specific addresses for the 8255A.

8255A internal register	Required address	Binary address format
Port A	8000h	10000000000000000
Port B	8001h	10000000000000001
Port C	8002h	1000000000000002
Control Register	8003h	1000000000000003

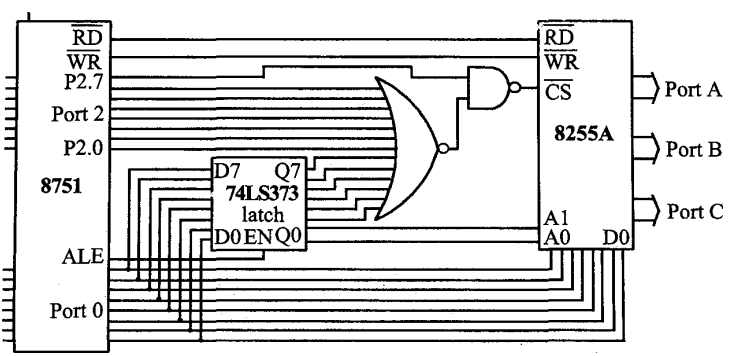


Figure 6.40 Address decoding to give specific addresses for 8255A registers.

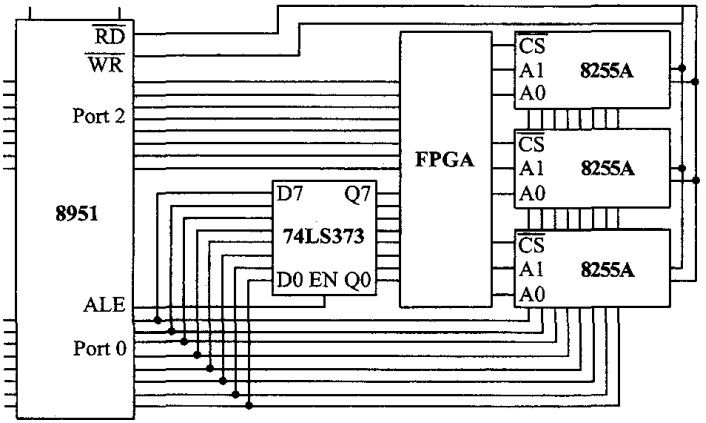


Figure 6.41 Using an FPGA to decode addresses for several 8255A chips.

6.8 Summary of the Chapter

In this chapter we introduced a detailed idea on memory systems; classification, types, operation, etc. The difference between memory map and memory space is explained. We concentrated on the semiconductor memories and we discussed the SRAM and DRAM. The idea of expanding the memory in width (word length) and in depth (total capacity) is given. The design of the chip select circuits is introduced. The memory of AVR microcontroller is given in detail. How to connect an external SRAM and EEPROM to AVR and Intel microcontrollers are discussed in some detail.

6.9 Review Questions

6.1 Name the three types of bus which are used when external memory is connected to the 8051.
6.2 From the experimental point of view, what is the difference between microcontrollers with EPROM or Flash memory?
6.3 What programming voltages need to be provided to programme either the EPROM or Flash type of code memory?
6.4 Draw the circuit diagram for connection of an 8031 microcontroller to a 32-bit external EPROM.
6.5 Assume you are working with 8051 microcontroller, on the table of Figure 6.42; draw a timing diagram for the instruction MOV A, 55 h. This instruction executes in one machine cycle, and the hexadecimal

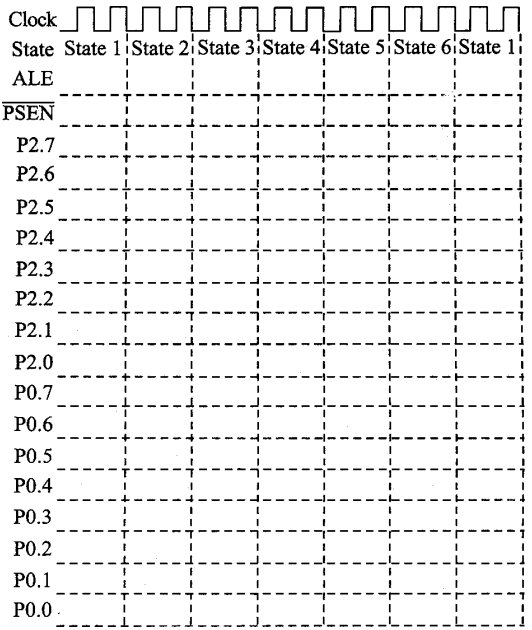


Figure 6.42 Timing diagram for Question 6.5.

value of the opcode is E5h. The programme is stored in an external EPROM, and the address of the instruction is O9FFh. Show the state cycles and the corresponding waveforms for the Clock, ALE, PSEN, and all lines of Ports 0 and 2. Draw the waveforms from State 2 to State 1 of the next machine cycle.

6.6 Draw the circuit diagram for the connection of an 8031 microcontroller to a 4 kbyte RAM and a 16 kbyte code EPROM.

6.7 Repeat question 6.6 in case of using ATmega8515.

6.8 In case of Intel 8051, when address and data multiplexing is used for writing to external memory, what are the four pieces of data that are outputted on the Port 0 pins in one machine cycle?

6.9 A weather station measures temperatures at four points, and four A-D converters send these as four 8-bit numbers to an ATmega8515 microcontroller. Design a system using two 8255A (or equivalent) chips to enable these temperatures to be inputted to the microcontroller. Write a programme to poll these temperatures and repeatedly calculate their average value. The temperatures are in the range 0 to 40° C. Store the average value in address 7Fh.

6.10 Estimate the memory requirement for a 500-images-capacity digital camera when the resolution is (a) 1024 × 768 pixels, (b) 640 × 480, (c) 320 × 240 and (d) 160 × 120 pixels and each image stores in a compressed jpeg format. Assume each pixel colour is defined by 24 bits.

6.11 How does a decoder help in memory and I/O devices interfacing? Draw four exemplary circuits.

6.12 We can assume that the memory of an embedded system is also a device. List the reasons for it. [Hint: The use of pointers, like access control registers, and the concept of virtual file and RAM disk devices.]

6.13 Refer to the material in this chapter and search the web. How does a boot block flash differ from a flash memory? How do flash, EEPROM and flash EEPROM differ? When do you use masked ROM for ROM image and when boot flash ROM in an embedded system?

6.14 Refer to the material in this chapter and search the Web. How does a memory map help in designing a locator programme? What are the Intel and Motorola formats for the ROM image records?

6.15 List various types of memories and application of each in the following: Robot, Electronic smart weight display system, ECG LCD display-cumrecorder, Router, Digital Camera, Speech Processing, Smart Card, Embedded Firewall! Router, Mail Client card, and Transceiver system with a collision control and jabber control [Collision control means transmission and reception when no other system on the network is using the network. Jabber control means control of continuous streams of random data flowing on a network, which eventually chokes a network.]

6.16 Tabulate hardware units needed in each of the systems mentioned in Question 42 above.

7

Timers, Counters and Watchdog Timer

THINGS TO LOOK FOR…

- To understand the importance of the timers in the field of digital system
- To understand the function and the operation of the watchdog timer.
- To study the timers of the AVR microcontroller.
- To study how to use the timer to control DC and servomotors.
- To study how to use the timer to measure the width of a pulse and to measure the frequency of an unknown signal.

7.1 Introduction to Timers and Counters

Timer/counters are probably the most commonly used complex peripheral in a microcontroller. A timer is an extremely common peripheral device that can measure time intervals. Such a device can be used to either generate events at specific times, or to determine the duration between two external events. Example applications that require generating events include keeping a traffic light green for a specified duration, or communicating bits serially between devices at a specific rate. An example of an application that determines inter-event duration is that of computing a car's speed by measuring the time the car takes to pass over two separated sensors in a road. Other applications might include measuring the rpm of a car's engine, timing an exact period of time, such as that necessary to time the speed of a bullet, producing tones to create music or to drive the spark ignition system of a car, or providing a pulse-width or variable-frequency drive to control a motor's speed.

A counter is a more general version of a timer. Instead of counting clock pulses, a counter counts pulses on some other input signal. For example, a counter may be used to count the number of cars that pass over a road sensor, or the number of people that pass through a turnstile. We often combine counters and timers to measure rates, such as counting the number of times a car wheel rotates in one second, in order to determine a car's speed.

Although timers/counters are used in two distinctly different modes, timing and counting, timers/counters are simply binary up-counters. When used in timing mode, the binary counters are counting time periods (pulses that occur on an input clock signal having a known period) applied to their input, and in counter mode, they are counting the events or pulses or something of this nature. For instance, if the binary counters had 1-millisecond pulses as their input, a time period could be measured by starting the counter at the beginning of an event and stopping the counter at the end of the event. The ending count in the counter would be the number of milliseconds that had elapsed during the event. If we counted for example 2,000 pulses on the clock signal, then we know that 2,000 milliseconds have passed.

When a timer/counter is used as a counter, the events to be counted are applied to the input of the binary counter, and the number of events occurring is counted. For instance, the counter could be used to count the number of cans of peas coming down an assembly line by applying one pulse to the input of the counter for each can of peas. At any time, the counter could be read to determine how many cans of peas had gone down an assembly line.

7.1.1 Counters

Two commonly used types of counters are binary counters and linear-feedback shift registers. An N-bit binary counter sequences through 2^N outputs in binary order. It has a minimum cycle time that increases with N. An N-bit linear-feedback shift register sequences through up to $2^N - 1$ outputs in pseudo-random order. It has a short minimum cycle time independent of N, so it is useful for extremely fast counters as well as pseudo-random number generation.

In general, divide-by-M counters ($M < 2^N$) can be built using an ordinary N-bit counter and circuitry to reset the counter upon reaching M. M can be a programmable input if an equality comparator is used.

7.1.1.1 Binary counters

The simplest binary counter is the asynchronous ripple-carry counter, shown in Figure 7.1. It is composed of N bistables (flip-flops) connected in toggle configuration, where the falling transition of each bistable clocks the subsequent bistable. Therefore, the delay can be quite long. It has no reset signal, making it extremely difficult to test. In general, asynchronous circuits introduce a whole assortment of problems, so the ripple-carry counter is shown mainly for historical interest and would not be recommended for commercial designs.

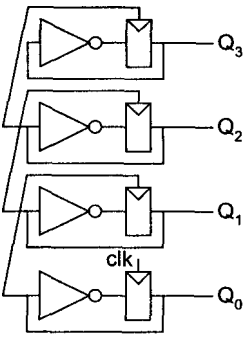


Figure 7.1 Asynchronous ripple-carry counter.

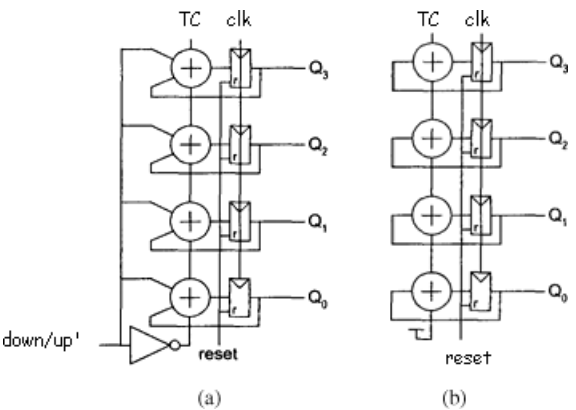


Figure 7.2 Synchronous counters.

A general synchronous up/down counter is shown in Figure 7.2(a). It uses a resettable register and full adder for each bit position. The cycle time is limited by the ripple-carry delay and can be improved by using any of the known faster adder techniques. If only an up counter (also called an incrementer) is required, the full adder degenerates into a half adder, shown in Figure 7.2(b). Including an input multiplexer allows the counter to load an initialization value. A clock enable is also often provided to each register for conditional counting.

7.1.1.2 Linear feedback shift register

A linear-feedback shift register (LFSR) consists of N bistables connected together as a shift register. The input to the shift register comes from the XOR of particular bits of the register, as shown in Figure 7.3 for a 3-bit LFSR. On

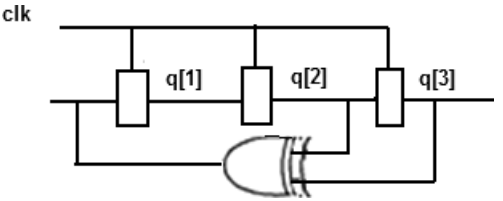


Figure 7.3 3-bit LFSR.

Table 7.1 LFSR sequence.

Cycle	Q[1]	Q[2]	Q[3]/Y
0	1	1	1
1	0	1	1
2	0	0	1
3	1	0	0
4	0	1	0
5	1	0	1
6	1	1	0
7	1	1	1
Repeats forever			

reset, the registers must be initialized to a nonzero value (e.g., all 1's). The pattern of outputs for the LFSR is shown in Table 7.1.

This LFSR is an example of a *maximal-length shift* register because its output sequences through all $2^n - 1$ combinations (excluding all 0's). The inputs fed to the XOR are called the tap sequence and are often specified with a *characteristic polynomial*. For example, the 3-bit LFSR of Figure 7.3 has the characteristic polynomial $1 + x^2 + x^3$ because the taps come after the second and third registers.

The output Y follows the 7-bit sequence [1110010]. This is an example of a *pseudo-random bit sequence* (PRBS) because it is spectrally random. LFSRs are used for high-speed counters and pseudo-random number generators. The pseudo-random sequences are handy for built-in self-test and bit-error-rate testing in communications links. They are also used in many spread-spectrum communications systems such as GPS and CDMA where their correlation properties make other users look like uncorrelated noise.

Table 7.2 lists characteristic polynomials for some commonly used maximal-length LFSRs. For certain lengths, N, more than two taps may be required. For many values of N, there are multiple polynomials resulting in different maximal-length LFSRs. Observe that the cycle time is set by the register and XOR delays, independent of N.

Table 7.2 Characteristic Polynomial.

N	Polynomial
3	$1 + X^2 + X^3$
4	$1 + X^3 + X^4$
5	$1 + X^3 + X^5$
6	$1 + X^5 + X^6$
7	$1 + X^6 + X^7$
8	$L + X + X^6 + X^7 + X^8$
9	$1 + X^5 + X^9$
15	$1 + X^{14} + X^{13}$
16	$1 + X^4 + X^{13} + X^{15} + X^{16}$
23	$1 + X^{18} + X^{23}$
24	$1 + X^{17} + X^{22} + X^{23} + X^{24}$
31	$1 + X^{28} + X^{31}$
32	$1 + X^{10} + X^{30} + X^{31} + X^{32}$

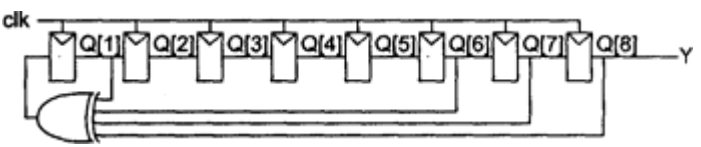


Figure 7.4 8-bit LFSR.

Example 7.1

Use Table 7.2 to sketch an 8-bit linear-feedback shift register. How long is the pseudo-random bit sequence that it produces?

Solution:

Figure 7.4 shows an 8-bit LFSR using the four taps after the first, sixth, seventh, and eighth bits, as given in Table 7.2. It produces a sequence of $2^8 - 1 = 255$ bits before repeating.

7.1.2 Timers

A timer, as mentioned before, is a series of divide-by-2 flip-flops that receive an input signal as a clocking source. The clock is applied to the first flip-flop, which divides the clock frequency by 2. The output of the first flip-flop clocks the second flip-flop, which also divides by 2, and so on. Since each successive stage divides by 2, a timer with n stages divides the input clock frequency by 2^n. The output of the last stage clocks a timer overflow flip-flop, or **flag**, which is tested by software or generates an interrupt. The binary value in the timer flip-flops can be thought of as a "count" of clock pulses since the timer was started. This 'count' can be converted to time by multiplying it by the

clock period, i.e.

$$\text{Time elapsed since timer started} = \text{count} \times 1/f_{clock}.$$

A 16-bit timer, for example, would count from 0000H to FFFFH. The overflow flag is set on the FFFFH–to–0000H overflow of the count.

To use a timer, we must configure its inputs and monitor its outputs. Such use often requires or can be greatly aided by an understanding of the internal structure of the timer. The internal structure can vary greatly among manufacturers. A few common features of such internal structures are shown in Figure 7.5.

Figure 7.5(a) provides the structure of a very simple timer. This timer has an internal 16-bit up counter, which increments its value on each clock pulse. Thus, the output value ***cnt*** represents the number of pulses since the counter was last reset to zero. To interpret this number as a time interval, we must know the frequency or period of the clock signal ***clk***. For example, suppose we wish to measure the time that passes between two button presses. In this case, we could reset the timer on the occurrence of the first press, and then read the timer output on the second press. Suppose the frequency of *clk* were 100 MHz, meaning the period would be 1/(100 MHz) = 10 nanoseconds, and that $cnt = 20{,}000$ at the time of the second button press. We would then compute the time that passed between the first and second button presses as $20,000 * 10$ nanoseconds = 200 microseconds. We note that since this timer's counter can count from 0 to 65,535, this particular timer has a measurement range of 0 to $65,535 * 10$

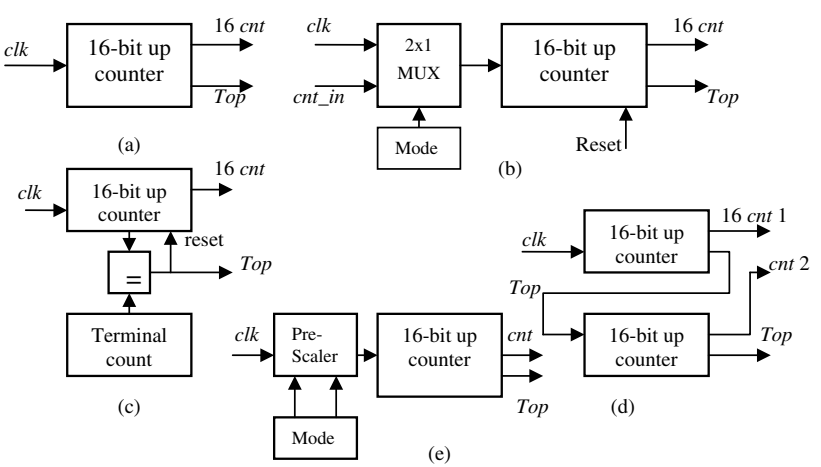


Figure 7.5 Timer structures: (a) a basic timer, (b) a timer/counter, (c) a timer with a terminal count, (d) a 16/32-bit timer, (e) a timer with a prescaler.

nanoseconds = 655.35 microseconds, with a resolution of 10 nanoseconds. We define a timer's ***range*** as the maximum time interval the timer can measure. A timer's ***resolution*** is the minimum interval it can measure.

The timer in Figure 7.5(a) has an additional output ***top*** that indicates when the top value of its range has been reached, also known as an overflow occurring, in which case the timer rolls over to 0. The output signal *top* is used in the applications that require generating events at specific times. When we use a timer in conjunction with a general-purpose processor, and we expect time intervals to exceed the timer range, we typically connect the top signal to an interrupt pin on the processor. We create a corresponding interrupt service routine that counts the number of times the routine is called, thus effectively extending the range we can measure. Many microcontrollers that include built-in timers will have special interrupts just for its timers, with those interrupts distinct from external interrupts.

Figure 7.5(b) provides the structure of a more advanced timer that can also be configured as a counter. A mode register holds a bit, which the user sets, that uses a 2×1 multiplexor to select the clock input to the internal 16-bit up counter. The clock input can be the external *clk* signal, in which case the device acts like a timer. Alternatively, the clock input can be the external *cnt_in* signal, in which case the device acts like a counter, counting the occurrences of pulses on *cnt_in. cnt_in* would typically be connected to an external sensor, so pulses would occur at indeterminate intervals. In other words, we could not measure time by counting such pulses.

Figure 7.5(c) provides the structure of a timer that can inform us whenever a particular interval of time has passed. A *terminal count register* holds a value, which the user sets, indicating the number of clock cycles in the desired interval. This number can be computed using the simple formula:

$$\text{number of clock cycles} = \text{desired time interval/clock period}$$

For example, to obtain duration of 3 microseconds from a clock cycle of 10 nanoseconds (100 MHz), we must count: $3 \times 10^{-6}\text{s}/10 \times 10^{-9}\text{s/cycle} = 300$ cycles. The timer structure includes a comparator that asserts its top output when the terminal count has been reached. This top output is not only used to reset the counter to 0, but also serves to inform the timer user that the desired time interval has passed. As mentioned earlier, we often connect this signal to an interrupt. The corresponding interrupt service routine would include the actions that must be taken at the specified time interval.

To improve efficiency, instead of counting up from 0 to terminal count, a timer could instead count down from terminal count to 0, meaning we would load terminal count rather than 0 into the 16-bit counter upon reset, and the

counter would be a down counter rather than an up counter. The efficiency comes from the simplicity by which we can check if our count has reached 0 — we simply input the count into a 16-bit NOR gate. A single 16-bit NOR gate is far more area- and power-efficient than a 16-bit comparator.

This configuration can be used to generate a one-shot or a periodic output signal. One-shot timers generate a pulse only once, and then stop counting. The pulse in such cases acts normally as an interrupt signal as in the case of watchdog timer (see latter). Periodic timers generate a pulse every time they reach a specific value. The pulses represent a periodic wave with a programmable frequency (such signal can be used, for example, to control traffic lights). By adjusting the reload value to the terminal count register, it is possible to adjust the frequency of the signal generated as top output. The timer in this mode is normally called programmable interval timer (PIT). The generated programmable frequency has an upper and a lower values as it will be shown in the next section.

Figure 7.5(d) provides the structure of a timer that can be configured as a 16-bit or 32-bit timer. The timer simply uses the top output of its first 16-bit up counter as the clock input of its second 16-bit counter. These are known as cascaded counters.

Finally, Figure 7.5(e) shows a timer with a prescaler. A prescaler is essentially a configurable clock-divider circuit, Figure 7.6. Depending on the mode bits being input to the prescaler, the prescaler output signal might

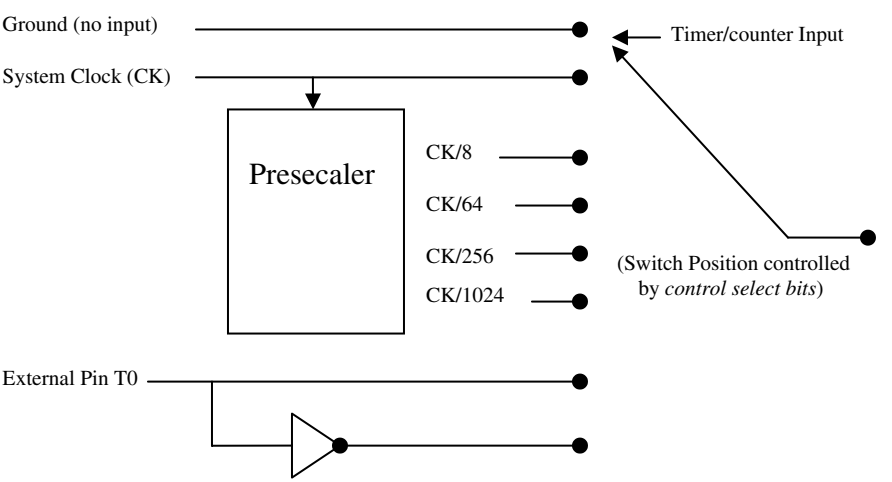


Figure 7.6 Prescaler configuration.

be the same as the input signal, or it may have half the frequency (double the period), one-fourth the frequency, one-eighth the frequency, etc. Thus, a prescaler can be used to extend a timer's range, by reducing the timer's resolution. For example, consider a timer with a resolution of 10 ns and a range of $65,535 * 10$ nanoseconds = 655.35 microseconds. If the prescaler of such a timer is configured to divide the clock frequency by eight, then the timer will have a resolution of 80 ns and a range of $65,535 * 80$ nanoseconds = 5.24 milliseconds. Figure 7.6 shows the prescaler configuration.

7.2 Uses and Types of Timers: Programmable Interval Timer (PIT)

7.2.1 Uses of Timers

Timers, as mentioned before, are the most commonly device used in embedded and digital systems. The following list shows some of the possible use of the timer devices.

1. Real Time Clocks: Real time clock is a clock, which, once the system starts, does not stop and cannot be reset and its count value cannot be reloaded.
2. Initiating an event after a preset delay time.
3. Initiating an event (or chain of events) after a comparison(s) between the preset time(s) with counted value(s). Preset time is loaded in a Compare Register.
4. Capturing the *count value* at the timer on an event. The information of *time* (instant of event) is thus stored at the *capture register.*
5. Finding the time interval between two events. *Time* is captured at each event and the intervals are thus found out.
6. Wait for a message from a queue or mailbox or semaphore for a preset time using RTOS (Real Time Operating System). There is a predefined waiting period before RTOS lets a task run.
7. Watchdog timer. It resets the system after a defined time.
8. Baud or Bit Rate Control for serial communication on a line or network. Timer timeout interrupts define the time of each bit or each baud.
9. Input pulse counting when using a timer, which is ticked by an external non-periodic event rather than the clock input. The timer in this case is acting in the "counter mode".
10. Scheduling of various tasks.

11. Time slicing of various tasks. RTOS (Real Time Operating System) switches after preset time-delay from one running task to the next. Each task can therefore run in a predefined slot of time.
12. Time division multiplexing (TDM). Timer device is used for multiplexing the input from a number of channels. Each channel input is allocated a distinct and fixed-time slot to get a TDM output.

7.2.2 Types of Timers

In Figure 7.5 we introduced some possible timer structures. Each structure can be used for one or more of the above listed applications. In general, the following types of counters can be used in any of the uses listed above:

1. Hardware internal timer.
2. Software timer (SWT)
3. User software-controlled hardware timer.
4. RTOS controlled hardware timer. An RTOS can define the clock ticks per second of a hardware timer at a system.
5. Timer with periodic time-out events (auto-reloading after overflow state).
6. One Shot timer (no reload after the overflow and finished state). It triggers on event-input for activating it to running state from its idle state. It is also used for enforcing time delays between two states or events.
7. Up count action Timer.
8. Down count action Timer.
9. Timers with its overflow-flag, which auto-resets as soon as interrupt service routine starts running.
10. Timers with overflow-flag, which does not auto reset.

7.2.2.1 Software timers

Sometimes, as we are going to see latter, the maximum time interval that the timer can give (the range) is less than needed by the application. In some other cases the number of hardware timers available in the system is not enough for the application. In such cases the software timer (SWT) is the solution. The SWT is an innovative concept- *a virtual timing device.* In such cases, the system clock or any other hardware-timing device ticks and generates one interrupt or a chain of interrupts at periodic intervals. This interval is as per the *count value* set. Now, the interrupt becomes a clock input to a SWT. This input is common to all the SWTs that are in the list of activated SWTs. Any number of SWTs can be made active in a list. Each SWT will set a flag on its timeout (*count value* reaching 0). SWT *control bits* are set as per application.

These can be up to nine types of control bits. There is no hardware input or output in a SWT.

SWT actions are analogous to that of a hardware timer. While there is a physical limits (1, 2, 3 or 4) on the number of hardware timers in a system, SWTs can be unlimited in number. Certain microcontrollers, now a day, define the interrupt vector addresses of 2 or 4 SWRs.

7.2.2.2 Hardware timers

At least one hardware timer device is a must in any embedded or digital system. The hardware timer to operate, it must (at least) combine the features shown in Figure 7.5, i.e.

- Use of one (Figure 7.5a) or more (Figure 7.5d) timers on the same device.
- Use of an up-down counter to increase the possible modes of operations of the timer.
- The possibility of using an internal or an external clock as an input of the timer. The system uses a mode bit to define the input of the counter (Figure 7.5b).
- The timer generates an output (the top signal in Figure 7.5) after predefined (programmed) time interval. The output signal can be periodic as the case of Figure 7.5c, or one-shot.
- The possibility of increasing the maximum time interval the timer can measure (the range) by increasing the counter length, e.g. to configure it as 16-bit counter or 32-bit counter (Figure 7.5d).
- Use of mode bits to define a prescaling value (Figure 7.5e) that can be used to change the resolution of the timer. Changing the resolution using the prescaler is normally used to change the range of the timer.

Other configurable features can be added. One such feature is a mode bit or additional input that enables or disables counting. Another bit mode can be added that enables or disables interrupt generation when top count is reached. The majority of the timers have all these features in the form of modes of operations. To familiarize ourselves with the possible features of a general timer, in the following we consider what is called a programmable interval timer (PIT). The PIT can be a separate IC, part from the motherboard of a PC, or part from the circuitry build-on a chip. Latter on the chapter we are going to consider the microcontroller timers.

Note: We note here that we could use a general-purpose processor to implement a timer. Knowing the number of cycles that each instruction requires, we could write a loop that executes the desired number of instructions; when this

loop completes, we know that the desired time passed. This implementation of a timer on a dedicated general-purpose processor is obviously quite inefficient in terms of size. One could alternatively incorporate the timer functionality into a main programme, but the timer functionality then occupies much of the program's run time, leaving little time for other computations. Thus, the benefit of assigning timer functionality to a special-purpose processor becomes evident.

7.2.3 PIT General Configuration

A PIT usually consists of three 16-bit counters which can be made to count down from a programmed value to zero. Most microcontrollers consist of two 16-bit and one 8-bit counters with at least one of them is an up/down counter. The rate at which counting down (or up) occurs can be fixed by using the microprocessor system clock or some other external clock. A block diagram of a PIT is shown in Figure 7.7 with two of the timers using the system clock to control decrementing and one as an alternative clock and provides an even greater flexibility when using the PIT, for producing programmable timer delays and programmable frequency signals.

The general programmer's model of the PIT consists of a command register for selecting the type of operation required from each timer and several timer count registers, usually 16 bits, into which is stored the initial count value. The general programmer's model of a PIT is shown in Figure 7.8.

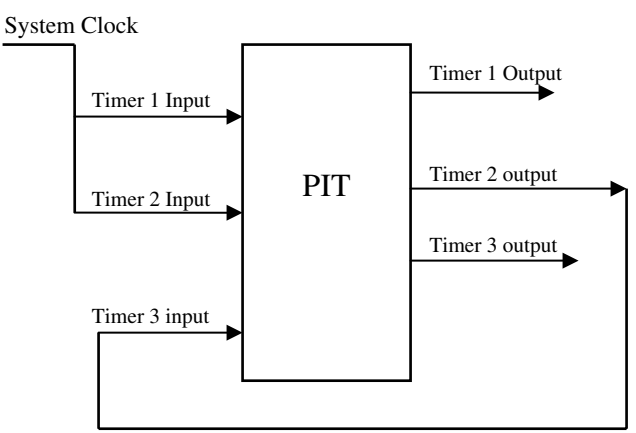


Figure 7.7 Block diagram of a PIT.

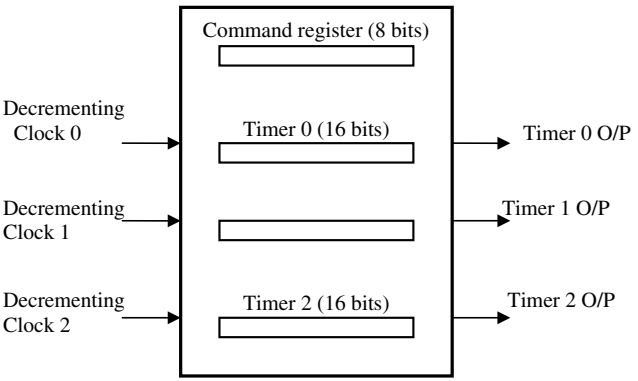


Figure 7.8 Programmer's model of a PIT.

The smallest count value that can be loaded into a timer count register is 02H to allow the highest programmed frequency to be obtained, as the output is activated half way through the count. On decrementing from 02H to 01H the timer output is activated and at the next decrementing clock pulse the count is decremented to 00H and the timer output is returned to a logic 0. The timer count register is then automatically reloaded with the initial count value, in this instance 02H, and the process starts again. Loading a value of FFFFH gives the largest count possible and produces the lowest programmed frequency. The rate at which each counter is decremented is determined by the decrement clock, which has an input to each individual timer and will be provided by the hardware of the microcontroller.

Example of programmable one-shot.

Assuming a hardware decrementing clock frequency of 1.19318 MHz, which is the frequency used in the IBM PC, the highest possible programmable frequency output produced by the timer is:

$$\begin{aligned}\text{Decrementing clock frequency/Count value} &= 1.19318/2 \\ &= 0.59659\,\text{MHz}\end{aligned}$$

The lowest possible programmable frequency output from the timer is obtained with the largest count value, which for a 16-bit register is $65536 (= 2^{16})$. This produces a programmed frequency of $1.19318/65536 = 18.206\,\text{Hz}$.

In the one-shot mode, the programmable delay can be calculated from the following formula:

$$\text{Programmed delay} = \text{Count value/Decrementing clock frequency}$$

Therefore the shortest possible delay is obtained when the count value = 01H which, for the same decrementing clock frequency used above, produces a programmed delay of $= 1/1.19318 \times 10^6 = 0.838\,\mu$s.

The longest programmed delay is when the largest count value of 65 536 is used. The programmed delay is then:

$$\text{Programmed delay} = 2^{16}/1.19318 \times 10^6 = 54.9\,\text{ms}$$

7.3 Microcontroller Timers/Counters: AVR Timers/Counters

The majority of the microcontrollers have one or more build-in timer/counter. For example, Intel 8051 has on board two 16-bit timers, Intel 8032/8052 has 3 timers, AVR has two timers (one 8-bit and one 16-bit) and ATmega8515 has a third timer (Timer 2) of 8-bit. The same is for PIC, Motorola and other microcontrollers. Normally the timers of the microcontrollers can fulfill all the modes of operation of the programmable interval timer (PIT). It is expected, accordingly, that all hardware of the PIT will be a part of the build-in hardware of the microcontroller. This includes the data registers, the control registers, and the prescaler. In the following the AVR timers/counters are taken as case study.

7.3.1 AVR Timers/Counters

The AVR microcontrollers have both 8-bit and 16-bit timer/counters. In either case, an important issue for the programme is to know when the counter reaches its maximum count and rolls over. In the case of an 8-bit counter, this occurs when the count reaches 255 (FFH), in which case the next pulse will cause the counter to roll over to 0. In the case of a 16-bit counter, the same thing occurs at 65,535 (FFFFH). The rollover events are extremely important for the programme to be able to accurately read the results from a timer/counter. In fact, rollovers are so important that an interrupt is provided that will occur when a timer/counter rolls over.

The AVR microcontrollers often have two 8-bit timers (Timer 0 and Timer 2) and one 16-bit timer (Timer 1), although this configuration will vary depending on the exact type of AVR microcontroller being used. For example, Timer 0 of ATmega8515 fulfils all the normal functions of Timer 0 and of timer 2 of other microcontroller. Accordingly ATMega8515 has two timers only, 8-bit timer (Timer 0), and 16-bit timer (Timer 1) covering all the

functions of the PIT and which are normally covered by Timers 0, 1, and 2 of other microcontrollers. The following sections will discuss the counter unit, the timer prescaler and input selector the features that are common to all of the timers. The compare circuit which is common for all Timer 1 and Timer 2 is given. The most common uses for each timer/counter will be discussed, although many timers have more functions than are discussed in this text. For any specific microcontroller, it is necessarily to check the specifications to determine all of the various functions possible with the timer/counters.

7.3.1.1 Prescalers and Input Selectors

Timer/counter units may use a variety of internal frequencies derived from the system clock as their input, or they may get their input from an external pin. Figure 7.9 shows the prescaler and input selector configuration for a timer/counter control register as used in most AVR microcontrollers. The Timer/Counter can be clocked directly by the system clock. This provides the fastest operation, with a maximum Timer/Counter clock frequency equal to system clock frequency (f_{CLK}_I/O). Alternatively, one of four taps from the prescaler can be used as a clock source. The prescaled clock has a frequency of either $f_{CLK}\_I/O/8$, $f_{CLK}\_I/O/64$, $f_{CLK}\_I/O/256$, or $f_{CLK}\_I/O/1024$.

The timer/counter control register (TCCRx) associated with the timer contains the counter select bits (CS×2, CS×1, CS×0) that control which input is used with a specific counter.

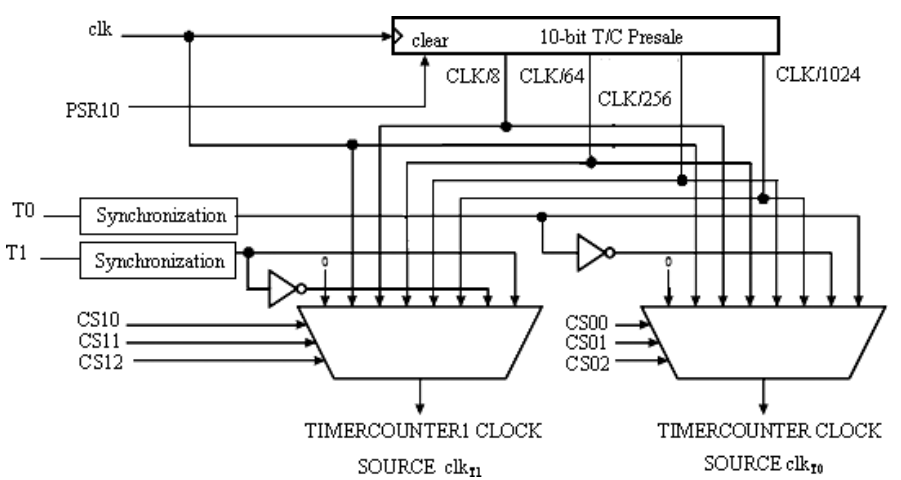


Figure 7.9 AVR Prescaler for Timer/Counter 0 and Timer/Counter 1.

The following code fragment shows, as an example, how to initialize Timer 0 to use the system clock divided by 8 as its clock source (the counter select bits are the three least significant bits of TCCR0):

```
TCCR0 = 0x2;     //Timer 0 uses clock/8
```

External Clock Source: As shown in Figure 7.9, an external clock source applied to the T1/T0 pin of the AVR microcontroller can be used as Timer/Counter clock (clkT1/clkT0). The T1/T0 pin is sampled once every system clock cycle by the pin synchronization logic. The synchronized (sampled) signal is then passed through the edge detector. Figure 7.10 shows a functional equivalent block diagram of the T1/T0 synchronization and edge detector logic. The registers are clocked at the positive edge of the internal system clock (clkI/O). The latch is transparent in the high period of the internal system clock

The edge detector generates one clkT1/clkT0 pulse for each positive (CSn2:0 = 7) or negative (CSn2:0 = 6) edge it detects.

7.3.2 Counter Unit

The main part of the Timer/Counter is the programmable bi-directional counter unit. Figure 7.11 shows a block diagram of the counter and its surroundings. In this figure, the size of the TCNTn register is 8 bits for Timer 0 and Timer 2, while it is 16 bits in case of Timer 1 (Note: in the following discussion, small "n" in the register and bit names indicates the device number, e.g. $n = 1$ for Timer/Counter 1). In Figure 7.11:

count:	Increment or decrement TCNTn by 1.
Direction:	Select between increment and decrement.
Clear:	Clear TCNTn (set all bits to zero).
clkTn:	Timer/Counter clock, referred to as clkT0 in the following.
Top:	Signalize that TCNTn has reached maximum value.
Bottom:	Signalize that TCNTn has reached minimum value (zero).

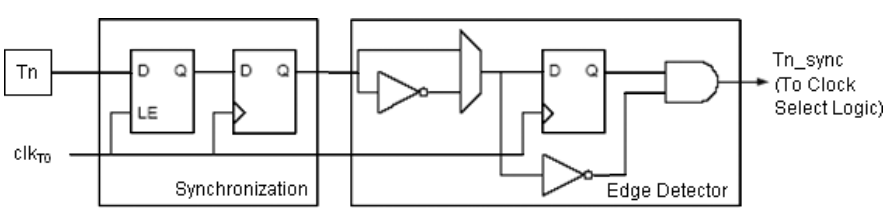


Figure 7.10 T1/T0 Pin Sampling.

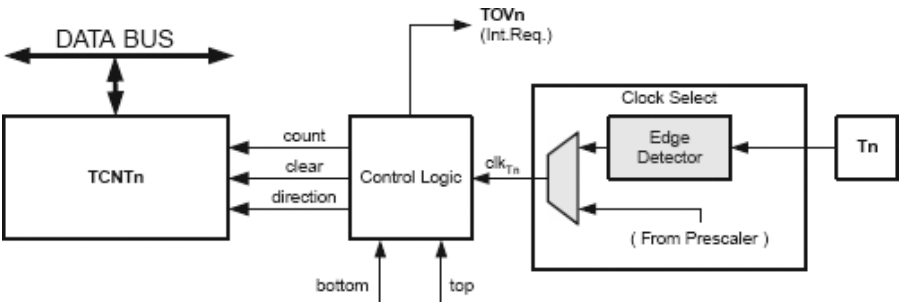


Figure 7.11 Counter unit block diagram.

Depending of the mode of operation used, the counter is cleared, incremented, or decremented at each timer clock (clkT0). clkT0 can be generated from an external or internal clock source, selected by the Clock Select bits (CSn2:0). When no clock source is selected (CSn2:0 = 0) the timer is stopped.

The counting sequence is determined by the setting of the WGMn1 and WGMn0 bits located in the Timer/Counter Control Register (TCCRn).

The Timer/Counter Overflow (TOVn) Flag is set according to the mode of operation selected by the WGM01:0 bits. TOVn can be used for generating a CPU interrupt.

7.3.3 Output Compare Unit

In this section we are going to consider the case of 16-bit timer, because the compare unit will be more general. The compare unit in case of 16-bit is shown in Figure 7.12. In this figure, the small "*n*" in the register and bit names indicates the device number ($n = 1$ for Timer/Counter1), and the "x" indicates output compare unit (A/B).The 16-bit comparator continuously compares TCNT1 with the *Output Compare Register* (OCR1x). If TCNT equals OCR1x the comparator signals a match. A match will set the *Output Compare Flag* (OCF1x) at the next timer clock cycle. If enabled (OCIE1x = 1), the Output Compare Flag generates an output compare interrupt. The OCF1x Flag is automatically cleared when the interrupt is executed. Alternatively the OCF1x Flag can be cleared by software by writing a logical one to its I/O bit location. The waveform generator uses the match signal to generate an output according to operating mode set by the *Waveform Generation mode* (WGM13:0) bits and *Compare Output mode* (COM1x1:0) bits. The TOP and BOTTOM signals are used by the waveform generator for handling the special cases of the extreme values in some modes of operation.

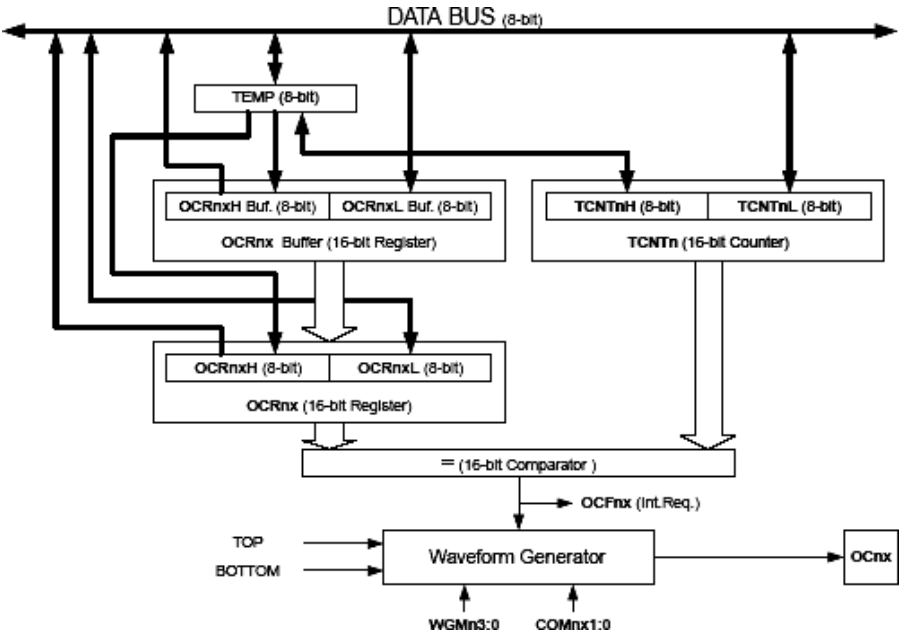


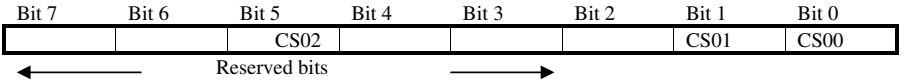
Figure 7.12 Output compare unit, block diagram.

A special feature of output compare unit A allows it to define the Timer/Counter TOP value (i.e., counter resolution). In addition to the counter resolution, the TOP value defines the period time for waveforms generated by the waveform generator.

7.4 TIMER 0

Timer 0 is typically an 8-bit timer, but this varies by specific processor type. It is capable of the usual timer/counter functions but is most often used to create a time base or tick for the programme. As any timer/counter, Timer 0 has a control register; Timer counter control register 0, TCCR0. TCCR0 controls the function of Timer 0 by selecting the clock source applied to Timer 0. Figure 7.13 shows the bit definitions for TCCR0. The five most significant bits are (normally) reserved. ATMega8515 is using them to control the additional modes added to Timer 0 to cover the modes of Timer 2 of other microcontrollers (This will be discussed latter).

Timer 0 can be used to provide a highly accurate timing event by using a simple programme. A programme *tick*, like the tick of a clock, is discussed



CS02	CS01	CS00	Interrupt function
0	0	0	Stop, Timer 0 is stopped
0	0	1	System Clock, CK
0	1	0	System Clock/8, CK/8
0	1	1	System Clock/64, CK/64
1	0	0	System clock/256, CK/256
1	0	1	System Clock/ 1024, CK/1024
1	1	0	External Pin T0, counts a falling edge
1	1	1	External Pin T0, counts a rising edge

Figure 7.13 TCCR0 register.

here. The overall scheme is that a number is selected and loaded into the timer. The timer counts from this number up to 255 and rolls over. Whenever it rolls over, it creates an interrupt (or a *tick*). The interrupt service routine reloads the same number into the timer, executes any time-critical activities that may be required, and then returns to the programme. The cycle then repeats, with the counter counting up from the number that was loaded to 255, and rolls over creating another interrupt. The interrupt, then, is occurring on a regular basis when each time period has elapsed. The number loaded into the counter determines the length of the period. The lower the number, the longer it will take the timer to reach to 255 and roll over and the longer the period of the tick will be.

Example 7.2

Write a programme that toggles the state of an LED every 0.5 seconds. Assume that the LED is attached to the most significant bit of port A as shown in Figure 7.14 and the microcontroller uses a 4 MHz clock.

Solution:

In this example we are using Timer 0 as a timer tick. The first necessary task is to determine the number that is loaded into the timer each time the interrupt occurs. In this example, we want the LED to toggle every 0.5 seconds. One obvious solution would be for the interrupt to occur every 0.5 seconds, in other words the timer has to generate a square wave of a period of 0.5 seconds. Each cycle of the square wave is a timer ***tick***. In the case of Timer 0, the slowest setting of the clock prescaler is system clock/1024.

$$4\,\text{MHz}/1024 = 3.906\,\text{kHz} \text{ which has a period of } 1/3,906\,\text{kHz} = 256\,\mu\text{us}$$

This shows that every 256 microseconds another clock pulse will be applied to Timer 0. Timer 0 is an 8-bit timer/counter, and so it can count up to

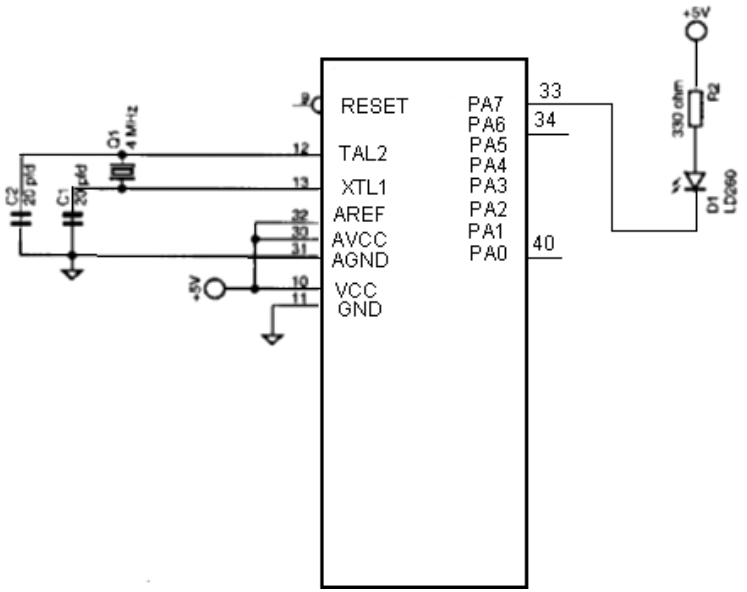


Figure 7.14 LED toggle hardware.

256 such periods before it rolls over. So the total time that is possible with this hardware is as follows:

$$256 * 256\mu s = 65.536\,\text{ms}$$

Sixty-five milliseconds are not sufficient to time 500 millisecond events. In order to accomplish longer periods, a global counter variable is placed in the interrupt service routine of the tick. So, for instance, if the tick occurs every 50 milliseconds, in this example we would want the counter to count up to 10 before toggling the LED ($10 * 50\,\text{ms} = 500\,\text{ms}$).

The choice of a reload number is up to the programmer. Usually it is desirable that the reload number produce an even and easy-to-use time delay period, such as 1 millisecond or 10 milliseconds. In the case of the 4 MHz clock shown in the example, using the divide-by-eight prescaler will give a clock period of 2 microseconds for each clock pulse applied to the counter. Given a 2-microsecond clock applied to an 8-bit counter, the maximum time possible using the counter alone would be as follows:

$$2\mu\text{s} * 256\,\text{counts} = 512\mu\text{s}$$

512 microseconds is not a very even period of time to work with, but using 250 counts would give a timeout period of 500 microseconds. Therefore the

counter reload number would be as follows:

$$256 - 250 = 6$$

Rollover occurs at count number 256 and it is desirable for the counter to count 250 counts before it rolls over, and therefore the reload number in this case is 6. This means that the interrupt service routine will be executed once every 500 microseconds (2 μs/clock cycle $*$ 250 clock cycles/interrupt cycle $=$ 500 μs/interrupt cycle). A global variable will be used to count to 1000 to produce the entire 500 milliseconds time period (500μs $*$ 1000 $=$ 500 ms). The entire programme is shown in Figure 7.15.

Figure 7.15 illustrates all of the concepts discussed above. The timer/counter register itself, *timer counter 0* (TCCNT0), is reloaded with the value 6 each time the ISR executes so that it can once again count up 250 steps to reach 256 and roll over.

A global variable called "timecount" is used to keep track of the number of times that the interrupt service routine is executed by incrementing it each time the ISR is executed. "*timecount*" is both incremented and checked inside

```
//***********************************************************
#include <ATmega8518.h>
unsigned int timecount = 0;           //global time counter

//Timer 0 overflow ISR
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
        TCNT0 = 6;                    //reload for 500 μs ticks
        If (++timecount ==1000)
        {
                PORTA = PORTA^ 0x80;  //toggle MSB of port A
                timecount = 0;        //clear for next 500 ms
        }
}
void main(void)
{
        DDRA = 0x80;                  //Port A MSB as output
        TCCR0 =0x02;                  //set for clock/8 as counter
        TCNT0 = 0x00;                 //start timer with 0 in timer
        //Timer 0 interrupt initialization
        TIMSK = 0x01;                 //unmask Timer 0 overflow interrupt

        // Global enable interrupts
        #asm("sei")
        while (1)
                ;                     //do nothing
}
//***********************************************************
```

Figure 7.15 LED blinker programme

the expression of the if statement. When "*timecount*" reaches 1000, the most significant bit of port A is toggled, and "time-count" is reset to zero to count up for the next 500-millisecond period. This tick could also handle any other event that occurs on an even number of 500 microsecond increments in the same manner.

7.5 Timer 1

The 16-bit timer, typically Timer 1, is a much more versatile and complex peripheral than Timer 0. In addition to the usual timer/counter, Timer 1 contains a 16-bit input capture register (ICR1 Register), two 16-bit output compare registers (OCR1A/B Registers) and is controlled through a control register "*Timer/counter control register 1* (TCCR1)". The input capture register is used for measuring pulse widths or capturing times. The output compare registers are used for producing frequencies or pulses from the timer/counter to an output pin on the microcontroller. Each of these modes will be discussed in the sections that follow. Remember, however, that although each mode is discussed separately, the modes may be, and often are, mixed together in a programme.

Timer 1 is also conceptually very different from Timer 0. Timer 0 is usually stopped, started, reset and so on in its normal use. Timer 1, on the other hand, is usually left running. This creates some considerable differences in its use. These differences will be discussed in detail in the sections that follow, covering the special uses of Timer 1.

7.5.1 Timer 1 Prescaler and Selector

In spite of its many special features, Timer 1 is still a binary up-counter whose count speed or timing intervals depend on the clock signal applied to its input just as Timer 0 was. As with all peripherals in the microcontroller, Timer 1 is controlled through a control register.

Timer/counter control register 1, **TCCR1** (the timer control register for Timer 1), is actually composed of two registers, TCCR1A and TCCR1B.

TCCR1A controls the compare modes and the pulse width modulation modes of Timer 1. These will be discussed later in the section.

TCCR1B controls the prescaler and input multiplexer for Timer 1, as well as the input capture modes. Figure 7.16 shows the bit definition for the bits of TCCR1B.

TCCR1B *counter select bits* control the input to Timer 1 in exactly the same manner as the counter select bits of Timer 0. In fact, the three select bits provide clock signals that are absolutely identical to those of Timer 0.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ICNC1	ICES1			CTC1	CS12	CS11	CS10

ICNC1	Input Capture Noise Canceller (1 = enabled)
ICES1	Input Capture Edge Select (1 = rising edge, 0 = falling edge)
ICC1	Clear Timer/Counter on Compare Match (1 = enabled)
CS12	Counter Input Select Bits Exactly the same definition as for Timer 0
CS11	
CS10	

Figure 7.16 TCCRIB bit definitions.

7.5.2 Accessing the 16-bit Timer 1 Registers

The ATmega8515 and the AT90S8515 are 8-bit processors. This means that the internal bus system is an 8 bits system. The data moves between the different blocks of the processor in the form of 8 bits units. It is expected, accordingly, that to read or to write 16-bit word, we must perform it in two cycles. To do a 16-bit write, the high byte must be written before the low byte. For a 16-bit read, the low byte must be read before the high byte.

The following code examples show how to access the 16-bit timer registers assuming that no interrupts updates the temporary register. The same principle can be used directly for accessing the OCR1A/B and ICR1 Registers.

```
...
; Set TCNT1 to 0x01FF
ldi r17,0x01
ldi r16,0xFF
out TCNT1H,r17
out TCNT1L,r16
; Read TCNT1 into r17:r16
in r16,TCNT1L
in r17,TCNT1H
...
```

7.5.3 Timer 1 Input Capture Mode

Measuring a time period with Timer 0 involves starting the timer at the beginning of the event, stopping the timer at the end of the event, and finally reading the time of the event, from the timer counter register. The same job with Timer 1 is handled differently because Timer 1 is always running. To measure an event, the time on Timer 1 is captured or held at the beginning of the event, the time is also captured at the end of the event, and the two are

subtracted to find the time that it took for the event to occur. In Timer 1, these tasks are managed by the input capture register (ICR1).

ICR1 is a 16-bit register that will capture the actual reading of Timer 1 when the microcontroller receives a certain signal. The signal that causes a capture to occur can be either a rising or a falling edge applied to the ***input capture pin***, **ICP**, of the microcontroller. As shown in Figure 7.16 the choice of a rising or falling edge trigger for the capture is controlled by the ***input capture edge select*** bit, **ICES1**. Setting ICES1 will allow ICR1 to capture the Timer 1 time on a rising edge, and clearing it will allow ICR1 to capture the time on a failing edge.

As is probably obvious by now, since there is only one capture register available to Timer 1, the captured contents must be read out as soon as they are captured to prevent the next capture from overwriting and destroying the previous reading. In order to accomplish this, an interrupt is provided that occurs whenever new data is captured into ICR1. Each time the capture interrupt occurs, the programme must determine whether the interrupt signals the beginning or the ending of an event that is being timed, so that it can treat the data in ICR1 appropriately.

Timer 1 also provides an input noise canceller feature, to prevent miscellaneous unwanted spikes in the signal applied to the ICP from causing a capture to occur at the wrong time. When the noise canceller feature is active, the ICP must remain at the active level (high for a rising edge, or low for a falling edge) for four successive samples before the microcontroller will treat the trigger as legitimate and capture the data. This prevents a noise spike from triggering the capture register. Setting the ***input capture noise canceller*** bit, **ICNC1**, in TCCR1B enables the noise canceller feature.

In Section 7.8.1, we gave an example of how to use the capture register to measure the width of a pulse applied to the input capture pin of the microcontroller.

7.5.4 Timer 1 Output Compare Mode

The output compare mode is used by the microcontroller to produce output signals. The outputs may be square or asymmetrical waves, and they may be varying in frequency or symmetry. Output compare mode, for instance, would be appropriate if you attempt to programme a microcontroller to play your school's fight song. In this case the output compare mode would be used to generate the musical notes that make up the song.

Output compare mode is sort of the antithesis of input capture mode. In input capture mode, an external signal causes the current time in a timer to be captured or held in the input capture register. In output compare mode, the programme loads an output compare register. The value in the ***output compare register*** is compared to the value in the timer/counter register, and an interrupt occurs when the two values match. This interrupt acts as an alarm clock to cause a processor to execute a function relative to the signal it is producing, exactly when it is needed.

In addition to generating an interrupt, output compare mode can automatically set, clear, or toggle a specific output port pin. For Timer 1, the output compare modes are controlled by timer counter control register 1 A, TCCR1A. Figure 7.17 shows the bit definitions for TCCR1A.

As shown in Figure 7.17, compare mode control bits determine what action will be taken when a match occurs between the compare register and the timer register. The associated output pin can be unaffected, toggled, set, or cleared. The match also causes an interrupt to occur. The goal of the interrupt service routine is to reset or reload the compare register for the next match that must occur.

TIMER1 compare mode can be used in many applications. One of the possible applications is to use it to generate square wave, i.e. to fulfill the rule of the programmable square wave generator mode of the PIT. This is given in Section 7.8.3 (Application 3).

7.5.5 Timer 1 Pulse Width Modulator Mode

The PWM is a simple and efficient method of converting a digital signal to a proportional drive current. PWM can be implemented using special hardware or by using software. Many microcontrollers now provide dedicated PWM outputs.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
COM1A1	COM1A0	COM1B1	COM1B0	-	-	PWM11	PWM10

COM1A0 & COM1A1 Control the compare mode function compare resister
COM1B0 & COM1B1 Control the compare mode function compare resister

COM1x1	COM1x0	Function (“x” is “A” or “B” as appropriate)
0	0	No Output
0	1	Compare match toggles the OC1x line
1	0	Compare match clears the OC1x line to 0
1	1	Compare match sets the OC1x line to 1

Figure 7.17 TCCR1A bit definitions.

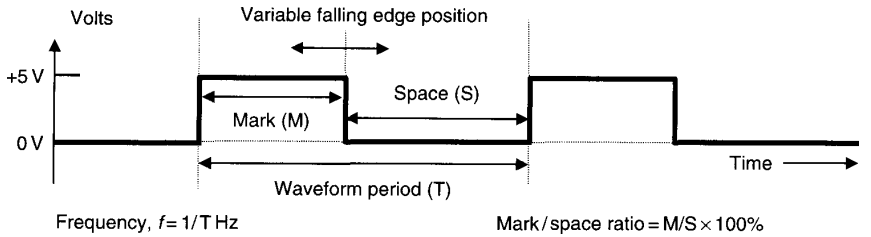


Figure 7.18 Pulse width modulated signal.

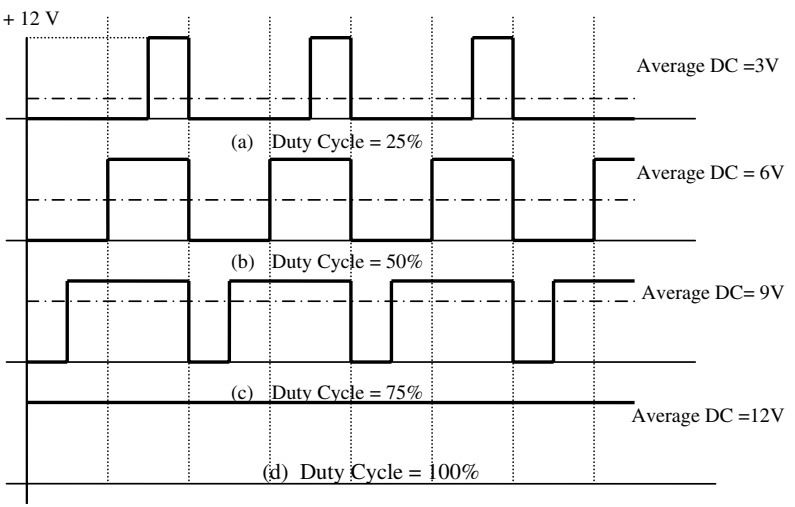


Figure 7.19 PWM waveforms.

PWM Basics

PWM is the scheme in which the duty cycle of a square wave is varied to provide a varying DC output by filtering the actual output waveform to get the average DC. The basic drive waveform is shown in Figure 7.18 and Figure 7.19 illustrates this principle.

As is shown in Figure 7.19, varying the duty cycle (proportion of the cycle that is high), or equally changing the mark/space ratio (MSR), will vary the average DC voltage of the waveform. The waveform is then filtered and used to control analogue devices, creating a digital-to-analogue converter (DAC). Examples of PWM applications in control and the circuits that can be used to provide the filtering action are shown in Figure 7.20.

Figure 7.20 demonstrates some typical circuits that are in use to filter the PWM signal. In Figure 7.20 at A, the RC circuit provides the filtering. The

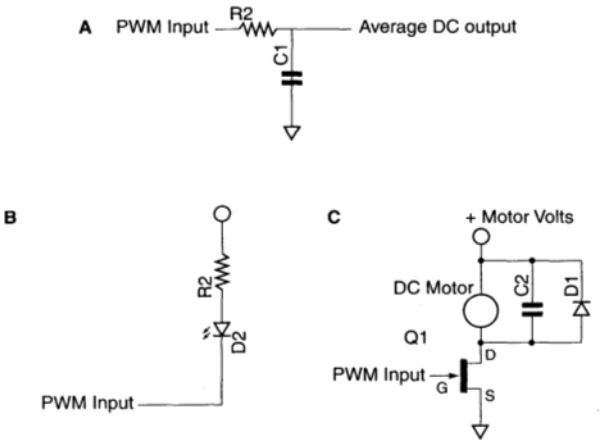


Figure 7.20 PWM examples

time constant of the RC circuit must be significantly longer than the period of the PWM waveform. Figure 7.20 at B shows an LED whose brightness is controlled by the PWM waveform. Note that in this example a logic 0 will turn the LED on, and so the brightness will be inversely proportional to the PWM. In this case, our eyes provide the filtering because we cannot distinguish frequencies above about 42 Hz, which is sometimes called the flicker rate. In this case the frequency of the PWM waveform must exceed 42 Hz, or we will see the LED blink.

The final example, Figure 7.20 at C, shows DC motor control using PWM. The filtering in this circuit is largely a combination of the mechanical inertia of the DC motor and the inductance of the windings. It simply cannot physically change speed fast enough to keep up with the waveform. The capacitor also adds some additional filtering, and the diode is important to suppress voltage spikes caused by switching the current on and off in the inductive motor.

One method of creating PWM with Timer 1 would be to use the output compare register and, each time a match occurs, vary the increment number being reloaded to create the PWM waveform. However, Timer 1 provides a built-in method to provide PWM without the need for the programme to be constantly servicing the compare register to create a pulse width modulator.

PWM Applications

1. PWM is the common technique used to control the speed of DC motors using an H-bridge. In this application, the duty cycle is varied between

0% and 100%. The physical properties of the motor itself effectively , as shown in Figure 7.20c before, average the values of the pulses such that they are interpreted as a voltage varying between stopped (0VDC) and full speed (e.g. 12VDC). In Section 7.8.4 we discuss this application in details.

2. PWM are also used to provide a dimming mechanism for LEDs in which, by using a pulse rate faster than the LED can flash, the effective voltage (and thus brightness) can be varied by varying the PWM duty cycle.
3. PWM can also be used to encode digital information. SAE protocol J1850-PWM (used in intra-vehicle networks) defines an encoding where a single bit 0 is represented by a (nominal) 33% duty cycle, and a bit 1 by a 66% duty cycle.
4. An encoding technique called Pulse Width Coding (PWC), which is directly related to PWM, is the most common technique that is in use to control RC servo motors (See Section 7.8.4). In this application, the pulse width is varied (typically) between 1 ms and 2 ms to position the servo mechanism; the refresh rate (PWC cycle period), however, may be freely (with some variation between units) chosen between 50 Hz and 2 kHz.

   The AVR's PWM generation unit may be used to generate a valid PWC signal.
5. PWC is also used to convey digital information in the MD5 and RECS80 protocols used by infrared-based television remote controls. While these protocols are different, both depend on the raw pulse width, and thus a calibrated clock, to discern between bits 0 and 1. The PWC cycle time, which includes the time between bits, is not defined and thus may (in theory) be arbitrarily long.

Use of Timer 1 to Generate PWM

Timer 1 changes its mode of operation in order to provide PWM. When operating in PWM mode, Timer 1 counts both up and down, making it difficult to use any of the other Timer 1 modes with PWM mode. During PWM operations, Timer 1 counts from zero up to a top value and back down again to zero. The top value is determined by the resolution desired. PWM is provided in 8-bit, 9-bit, or 10-bit resolutions as determined by the PWM select bits in TCCR1A. Figure 7.21 shows the effect of these bits on PWM.

Figure 7.21 shows that the resolution chosen will determine the top value for the counter to count up to and down from, and it will also affect the frequency of the resulting PWM wave form. The PWM frequency f_{PWM} is

PWM Select Bits		PWM Resolution	Timer Top Value
PWM11	PWM10		
0	0	PWM Disabled	
0	1	8-bit	255 (0xFF)
1	0	9-bit	511 (0x1FF)
1	1	10-bit	1023 (0x3FF)

Figure 7.21 PWM select bits.

given, as shown in Figure 7.22 by the relation:

$$f_{PWM} = f_{\text{system clock}}/(\text{prescaler} * 2 * \text{top count})$$

For example, choosing 9-bit resolution will result in a top count of 511 and the PWM frequency as calculated below (given an 8 MHz system clock):

$$\begin{aligned} f_{PWM} &= f_{\text{system clock}}/(\text{prescaler} * 2 * \text{top count}) \\ &= 8\,\text{Mhz}/(8 * 2 * 511) = 978.5\,\text{Hz} \end{aligned}$$

Resolution is the fineness, or accuracy, of PWM control. In 8-bit mode, PWM is controlled to one part in 256, in 9-bit mode PWM is controlled to one part in 512, and in 10-bit mode PWM may be controlled to one part in 1024. In PWM, resolution must be weighed against frequency to determine the optimum choice.

The actual duty cycle being output in PWM mode depends on the value loaded into the output compare register for the timer/counter. In normal PWM mode, as the counter counts down, it sets the output bit on a match to the output compare register, and as it counts up, it clears the output bit on a match to the output compare register. In this manner, loading the output compare register with the value equal to, say, 20 percent of the top value will produce a 20 percent duty cycle waveform. It is also possible to provide the inverted PWM for applications such as controlling the brightness of an LED connected in a current-sink mode directly to the PWM output pin — loading the output compare register to 80 percent of the top value in the inverted mode will produce an 80 percent low duty cycle square wave.

Setting the exact frequency output requires juggling timer prescaler and the resolution to get as close to the desired frequency as possible. The table in Figure 7.23 shows the frequency output (calculated using the formula presented previously) for all possible combinations of the prescaler and resolution, given the 8 MHz system clock. From the table we can pick the combination that provides the frequency closest to the one we desire. If the problem had required

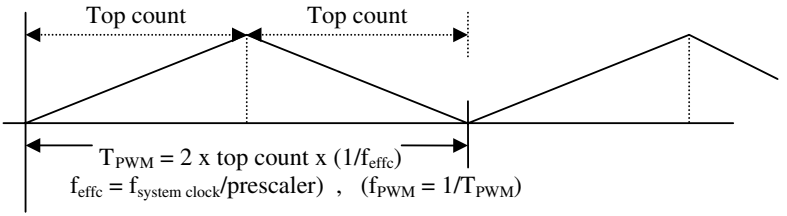


Figure 7.22 The up-down counter.

Prescaler	8-bit (Top=255)	9-bit (Top=511)	10-bit (Top=1023)
System Clock	f_{PWM} = 15.7 kHz	f_{PWM} = 7.8 kHz	f_{PWM} = 3.19 kHz
System Clock/8	f_{PWM} = 1.96 kHz	f_{PWM} = 978 Hz	f_{PWM} = 489 Hz
System Clock/64	f_{PWM} = 245 Hz	f_{PWM} = 122 Hz	f_{PWM} = 61 Hz
System Clock/256	f_{PWM} = 61 Hz	f_{PWM} = 31 Hz	f_{PWM} = 15Hz
System Clock/1024	f_{PWM} = 15 Hz	f_{PWM} = 8 Hz	f_{PWM} = 4 kHz

f_{PWM} = $f_{system\ clck}$/(prescaler *2* top count)

Figure 7.23 Frequency selection chart.

a particular resolution of the PWM, the choices from the table would be limited by the resolution, and the frequency might not have been as close.

Figure 7.23 also demonstrates that the frequency choices for any given combination of crystal, prescaler, and PWM resolution are fairly limited. If a project requires a very specific PWM frequency, the system clock crystal will likely have to be chosen to allow this frequency to be generated at the desired resolution.

In Section 7.8.4 (Application 4) we discussed how to use PWM to provide four different duty cycles (10, 20, 30, and 40 percent at a frequency close to 2 kHz).

7.6 Timer 2

Timer 2 is usually (depending on a specific microcontroller in use) an 8-bit timer/counter with output compare and PWM features similar to Timer 1.

The most interesting difference with Timer 2 is that it can use a crystal separate from the system clock as its clock source. Selection of the external oscillator as the clock source for Timer 2 is accomplished by setting the AS2 bit in the asynchronous status register, ASSR. The bit definitions for the bits of ASSR are shown in Figure 7.24.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
				AS2	TCN2UB	OCR2UB	TCR2UB
←----------Reserved ---------- ----------→							

Figure 7.24 ASS bit definition.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Reserved	PWM2	COM21	COM20	CTC2	CS22	CS21	CS20

Bit	Description
PWM2	Setting this bit enables the Timer 2 PWM function
COM21	These two bits set the output compare mode function. The bit definitions are the identical to COM1x1 and COM1x0 bits of Timer 1
COM20	
CTC2	Set to enable counter clear on compare match
CS22	Counter prescaler select bits for Timer 2.
CS21	
CS20	

Figure 7.25 TCCR2 bit definitions.

Setting the AS2 bit allows Timer 2 to use the external oscillator as its clock source. This means that the clock source for Timer 2 is running asynchronously to the microcontroller system clock. The other three bits in the ASSR register are used by the programmer to ensure that data is not written into the Timer 2 registers at the same moment that the hardware is updating the Timer 2 registers. This is necessary because the oscillator of Timer 2 is asynchronous to the system oscillator, and it is possible to corrupt the data in the Timer 2 registers by writing data to the registers as they attempt to update.

A single control register, TCCR2, controls the operation of Timer 2. The TCCR2 bit definitions are shown in Figure 7.25.

Using for instance, a 32.768 kHz crystal allows Timer 2 to function as the time base for a real-time clock. Using Timer 2 in this way will allow the microcontroller to keep accurate time when necessary.

ATMega8515 Timer 2 Implementation

The modes of operation of Timer 0 of ATMega8515 are extended to cover that of Timer 0 and Timer 2 mentioned above. To implement that, the five most significant bits of TCCR0 that are reserved in Figure 7.13 are used to accommodate the bits of TCCR2. The TCCR0 of ATmega8515 is shown in Figure 7.26.

Beside the use of the reserved bits, ATMega8515 Timer 0 has an additional register, Output Compare Register OCR0, exactly as that of Timer 1. The Output Compare Register contains an 8-bit value that is continuously compared

Bit	7	6	5	4	3	2	1	0	
	FOC0	WGM00	C)M01	COM00	WGM01	CS02	CS01	CS00	TCCR0
Read/Write	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
R/W	0	0	0	0	0	0	0	0	

Bit	Description
FOCO	**Force Output Compare: Set** only when the WGM00 bit specifies a non-PWM mode.
WGM00 WGM01	**Waveform Generation Mode:** These bits control the counting sequence of the counter, the source for the maximum (TOP) counter value, and what type of waveform generation to be used. Modes of operation supported by the Timer/Counter unit are: Normal mode, Clear Timer on Compare Match (CTC) mode, and two types of Pulse Width Modulation (PWM) modes. See Table -7.3.
COM01 COM00	**Compare Match Output Mode** These bits control the Output Compare pin (OC0) behaviour. If one or both of the COM01:0 bits are set, the OC0 output overrides the normal port functionality of the I/O pin it is connected to. However, note that the Data Direction Register (DDR) bit corresponding to the OC0 pin must be set in order to enable the output driver.
CS02, CS01, CS00	**Clock Select:** The three Clock Select bits select the clock source to be used by the Timer/Counter as given in case of Timer 0.

Figure 7.26 ATMega8515Timer/Counter Control Register — TCCR0.

Table 7.3 Waveform generation mode bit description.

Mode	WGM01 (CTC0)	WGM00 (PWM0)	Timer/Counter mode of operation	Top	Update of OCR0 at	TOV0 Flag Set on
0	0	0	Normal	0xFF	Immediate	MAX
1	0	1	PWM, Phase correct	0xFF	TOP	BOTTOM
2	1	0	CTC	OCR0	Immediate	MAX
3	1	1	Fast PWM	0xFF	TOP	MAX

with the counter value (TCNT0). A match can be used to generate an output compare interrupt, or to generate a waveform output on the OC0 pin.

The operation of ATmega8515 Timer 0 is the same as that of Timer 1; accordingly, no further explanation is included.

7.7 Watchdog Timer

7.7.1 Introduction to Watchdog Timer

A common problem is for a machine or operating system to lock up if two parts or programmes conflict, or, in an operating system, if memory management trouble occurs. Most embedded systems need to be self-reliant. In some cases, the system will eventually recover on its own but this may take an unknown and perhaps extended length of time. In many applications, it is not possible to wait for the system to recover on its own or for someone to restart them. Some embedded systems, such as remote data acquisition systems, are difficult

to reach, others, such as space probes, are simply not accessible to human operators. If their software ever hangs, such systems are permanently disabled. In other cases, the speed with which a human operator might reset the system would be too slow to meet the uptime requirements of the product.

For those embedded systems that can't be constantly watched by a human, and need to be automatically restarted after failing to give a response within a predefined period of time, watchdog timers may be the solution.

A watchdog timer is a piece of hardware, often built into a microcontroller that can be used to automatically detect software anomalies and reset the processor if any occur. Generally speaking, a watchdog timer is based on a counter that counts down, at constant speed, from some initial value to zero. The embedded software selects the counter's initial value (a real-time value) and periodically restarts it. It is the responsibility of the software to set the count to its original value often enough to ensure that it never reaches zero. If the counter ever reaches zero before the software restarts it, the software is presumed to be malfunctioning and the processor's reset signal is asserted. The processor (and the embedded software it's running) will be restarted as if a human operator had cycled the power.

The watchdog timer can be considered as a special type of timer. We configure a watchdog timer with a real-time value, just as with a regular timer. However, instead of the timer generating a signal for us every X time units, we must generate a signal for the watchdog timer every X time units. If we fail to generate this signal in time, then the timer "times out" and generates a signal indicating that we failed.

Timers and Watchdog timers: A watchdog is a special type of timer. We configure a watchdog timer with a real-time value, just as with a regular timer. However, instead of the timer generating a signal for us every X time units, we must generate a signal for the watchdog timer every X time units. If we fail to generate this signal in time, then the timer "times out" and generates a signal indicating that we failed.

Figure 7.27 shows a typical arrangement. In the figure, the watchdog timer is a chip external to the processor. However, it could also be included within the same chip as the CPU. This is done in many microcontrollers. In either case, the output from the watchdog timer is tied directly to the processor's reset signal.

A simple example is shown in Listing 1. Here we have a single infinite loop that controls the entire behavior of the system. This software architecture is

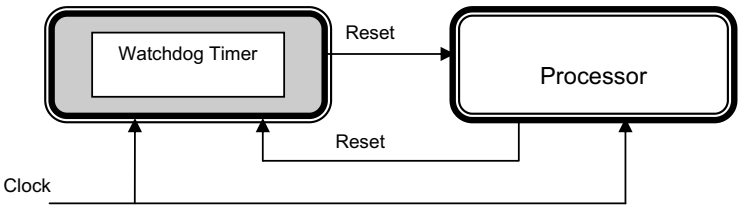


Figure 7.27 Typical watchdog timer setup.

common in many embedded systems with low-end processors and behaviors based on a single operational frequency. The hardware implementation of this watchdog allows the counter value to be set via a memory-mapped register.

Listing 1: Resetting the watchdog

```
uint16 volatile * pWatchdog =
    (uint16 volatile *) 0xFF0000;
main(void)
 {
        hwinit();
        for (;;)
        {
            *pWatchdog = 10000;
            read_sensors();
            control_motor();
            display_status();
        }
 }
```

Suppose that the loop must execute at least once every five milliseconds. (Say the motor must be fed new control parameters at least that often.) If the watchdog timer's counter is initialized to a value that corresponds to five milliseconds of elapsed time, say 10,000, and the software has no bugs, the watchdog timer will never expire; the software will always restart the counter before it reaches zero.

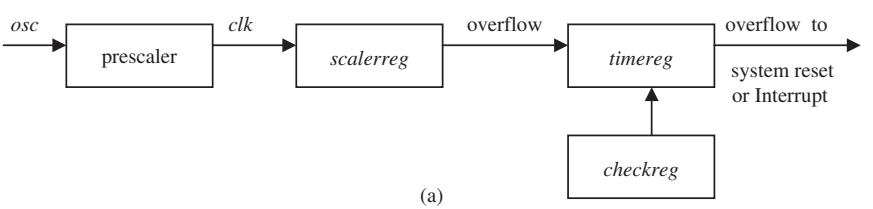
7.7.1.1 ATM timeout using watchdog timer

In this example, a watchdog timer is used to implement a timeout for an automatic teller machine (ATM). A normal ATM session involves a user inserting a bank card, typing in a personal identification number, and then answering about whether to deposit or withdraw money, which account will

be involved, how much money will be involved, whether another transaction is desired, and so on. We want to design the ATM such that it will terminate the session if at any time the user does not press any button for 2 minutes. In this case, the ATM will eject the bank card and terminate the session.

We will use a watchdog timer with the internal structure shown in Figure 7.28(a). An oscillator signal *osc* is connected to a prescaler that divides the frequency by 12 to generate a signal *clk*. The signal *clk* is connected to an 11-bit up-counter *scalereg*. When *scalereg* overflows, it rolls over to 0, and its overflow output causes the 16-bit up-counter *timereg* to increment. If *timereg* overflows, it triggers the system reset or an interrupt. To reset the watchdog timer, *checkreg* must be enabled. Then a value can be loaded into *timereg*. When a value is loaded into *timereg*, the *checkreg* register is automatically reset. If the *checkreg* register is not enabled, a value cannot be loaded into *timereg*. This is to prevent erroneous software from unintentionally resetting the watchdog timer.

Now let's determine what value to load into *timereg* to achieve a timeout of 2 minutes. The *osc* signal frequency is 12 MHz. *Timereg* is incremented at



(a)

```
/* main.C */

main () {
    wait until card inserted
    Call watchdog_reset_routine

     While (transaction in progress){
            If (button pressed){
               Perform corresponding action
               Call watchdog_reset_routine
          }

   /* if watchdog_rest_routine not called
    every <2 minutes,
   interrupt_service_routine is called */
}
```

(b)

```
Watchdog_reset_routine (){
/* checkreg is set so we can load value
   into timereg. Zero is loaded into scalereg
   and 11070 is loaded into timereg */

   checkreg  = 1
   scalereg  = 0
   timereg  =11070

}

void interrupt_service_routine (){
    eject card
    reset screen
}
```

(c)

Figure 7.28 ATM timeout using a watchdog timer : (a) timer structure, (b) main pseudo-code, (c) watchdog reset routine.

every t seconds where:

$$\begin{aligned} t &= 12 * 2^{11} * 1/(\text{osc frequency}) = 12 * 2^{11} * 1/(12 * 10^6) \\ &= 12 * 2,048 * (8.33 * 10^{-8}) = 0.002 \text{ second} \end{aligned}$$

So this watchdog timer has a resolution of 2 milliseconds. Since *timereg* is a 16-bit register, its range is 0 to 65,535, and thus the timer's range is 0 to 131,070 milliseconds (approximately 2.18 minutes). Because *timereg* counts up, then to attain the watchdog interval time X milliseconds, we load the following value into the *timereg* register:

$$\text{Timereg value} = 131,1070 - X$$

If *timereg* is not reset within X milliseconds, it will overflow. Figure 7.28(b) and (c) provide pseudo-code for the main routine and the watchdog reset routine to implement the timeout functionality of the ATM. We want timeout to be 2 minutes (120,000 millisecond). This means that the value of X should be 11070 (i.e., $131,070 - 120,000$).

7.7.1.2 Errors that the watchdog timer can detect

A watchdog timer can get a system out of a lot of dangerous situations. This section discusses the sort of failures a watchdog can detect. A properly designed watchdog mechanism should, at the very least, catch events that hang the system. In electrically noisy environments, a power glitch may corrupt the programme counter, stack pointer, or data in RAM. The software would crash almost immediately, even if the code is completely bug free. This is exactly the sort of transient failure that watchdogs will catch.

Bugs in software can also cause the system to hang. However, if it is to be effective, resetting the watchdog timer must be considered within the overall software design. Designers must know what kinds of things could go wrong with their software, and ensure that the watchdog timer will detect them, if any occur. Systems hang for any number of reasons. Some of them are

A logical fallacy resulting in the execution of an infinite loop is the simplest. Suppose such a condition occurred within the read_sensors() call in Listing 1. None of the other software (except ISRs, if interrupts are still enabled) would get a chance to run again.

If the software bug leads to an accidental jump out of the code memory.

Another possibility is that an unusual number of interrupts arrives during one pass of the loop. Any extra time spent in ISRs is time not spent executing the main loop. A dangerous delay in feeding the motor new control instructions could result.

When multitasking kernels are used, deadlocks can occur. For example, a group of tasks might get stuck waiting on each other and some external signal that one of them needs, leaving the whole set of tasks hung indefinitely.

If such faults are transient, the system may function perfectly for some length of time after each watchdog-induced reset. However, failed hardware could lead to a system that constantly resets. For this reason it may be wise to count the number of watchdog-induced resets, and give up trying after some fixed number of failures.

When using the watchdog timer to enable an embedded system to **restart** itself in case of a failure, we have to modify the system's programme to include statements that reset the watchdog timer. We place these statements such that the watchdog timer will be reset at least once during every time out interval if the programme is executing normally. We connect the fail signal from the watchdog timer to the microprocessor's reset pin. Now, suppose the programme has an unexpected failure, such as entering an undesired infinite loop, or waiting for an input event that never arrives. The watchdog timer will time out, and thus the microprocessor will reset itself, starting its programme from the beginning. In systems where such a full reset during system operation is not practical, we might instead connect the fail signal to an interrupt pin, and create an interrupt service routine that jumps to some safe part of the programme. We might even combine these two responses, first jumping to an interrupt service routine to test parts of the system and record what wrong, and then resetting the system. The interrupt service routine may record information as to the number of failures and the causes of each, so that a service technician may later evaluate this information to determine if a particular part requires replacement. Note that an embedded system often must self-recover from failures whenever possible, as the user may not have the means to reboot the system in the same manner that he/she might reboot a desktop system.

Another common use is to support time outs in a programme while keeping the programme structure simple. For example, we may desire that a user respond to questions on a display within some time period. Rather than sprinkling response-time checks throughout our programme, we can use a watchdog timer to check for us, thus keeping our programme neater.

It is important to note here that, it is also possible to design the watchdog timer hardware so that a reset signal that occurs too soon will cause the watchdog timer to reset the system. In order to use such a system, very precise knowledge of the timing characteristics of the main loop of your programme is required.

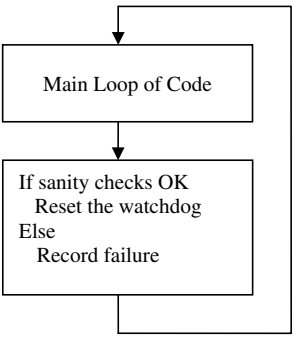


Figure 7.29 Reset the watchdog after some sanity checks.

7.7.1.3 Increasing the family of errors

Resetting the watchdog on a regular interval proves that the software is running. It is often a good idea to reset the watchdog only if the system passes some sanity check, as shown in Figure 7.29. Stack depth, number of buffers allocated, or the status of some mechanical component may be checked before deciding to reset the watchdog. Good design of such checks will increase the family of errors that the watchdog will detect.

One approach is to clear a number of flags before each loop is started, as shown in Figure 7.30. Each flag is set at a certain point in the loop. At the bottom of the loop the watchdog is reset, but first the flags are checked to see that all of the important points in the loop have been visited.

For a specific failure, it is often a good idea to try to record the cause (possibly in NVRAM), since it may be difficult to establish the cause after the reset. If the watchdog timeout is due to a bug then, any other information you can record about the state of the system, or the currently active task will be valuable when trying to diagnose the problem.

7.7.1.4 Protecting watchdog timer from malfunctioning

An actual watchdog implementation would usually have an interface to the software that is more complex than the one in Listing 1. When the set of instructions required to reset the watchdog is very simple, it's possible that buggy software could perform this action by accident. To prevent the possibility of resetting or disabling the watchdog timer accidentally, many watchdog implementations require that a complex sequence of two or more consecutive writes be used to restart the watchdog timer.

If the watchdog is part of the microcontroller, it may not be enabled automatically when the device resets. The user must be sure to enable it

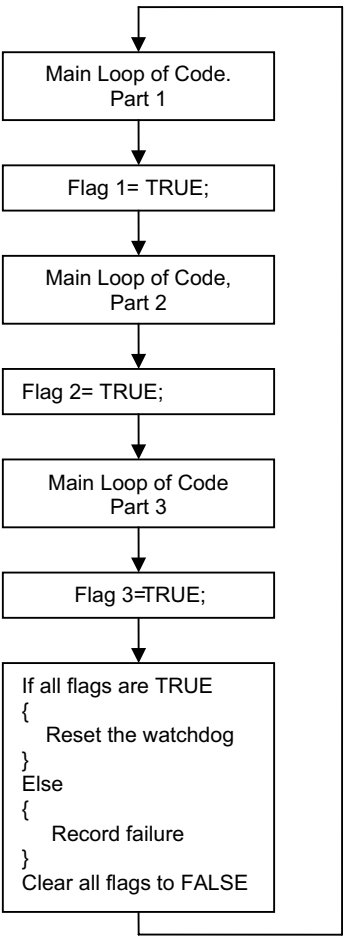


Figure 7.30 Use three flags to check that certain Points within the main loop have been visited.

during hardware initialization. To provide protection against a bug accidentally disabling the watchdog, the hardware design usually makes it impossible to disable the watchdog timer once it has been enabled.

If the software can do a complete loop faster than the watchdog period, your programme in this case may work fine for you. It gets more challenging if some part of your software takes a long time to complete. Say you have a loop that waits for an element to heat to a certain temperature before returning. Many watchdog timers have a maximum period of around two seconds. If you are going to delay for more than that length of time, you may have to reset the

watchdog from within the waiting loop. If there are many such places in your software, control of the watchdog can become problematic.

System initialization is a part of the code that often takes longer than the watchdog timer's maximum period. Perhaps a memory test or ROM to RAM data transfer slows this down. For this reason, some watchdogs can wait longer for their first reset than they do for subsequent resets.

As threads of control are added to software (in the form of ISRs and software tasks), it becomes ineffective to have just one place in the code where the watchdog is reset.

Choosing a proper reset interval is also an important issue, one that can only be addressed in a system-specific manner.

7.7.1.5 Failing to reset watchdog timer

Once you failed to reset the watchdog, you have to decide what action to take. The hardware will usually assert the processor's reset line, but other actions are also possible. In the example of the ATM machine if the customer fails to give a response for 2 minutes, the watchdog timer will end the session and the ATM machine will eject the card. In other cases, when the watchdog timeouts (you failed to reset it) it may directly disable a motor, engage an interlock, or sound an alarm until the software recovers. Such actions are especially important to leave the system in a safe state if, for some reason, the system's software is unable to run at all (perhaps due to chip death) after the failure.

A microcontroller with an internal watchdog will almost always contain a status bit that gets set when a watchdog timeout signal occurs. By examining this bit after emerging from a watchdog-induced reset, we can decide whether to continue running, switch to a fail-safe state, and/or display an error message. At the very least, you should count such events, so that a persistently errant application won't be restarted indefinitely. A reasonable approach might be to shut the system down if three watchdog bites occur in one day.

If we want the system to recover quickly, the initialization after a watchdog reset should be much shorter than power-on initialization.

A possible shortcut is to skip some of the device's self-tests. On the other hand, in some systems it is better to do a full set of self-tests since the root cause of the watchdog timeout might be identified by such a test.

In terms of the outside world, the recovery may be instantaneous, and the user may not even know a reset occurred. The recovery time will be the length of the watchdog timeout plus the time it takes the system to reset and perform its initialization. How well the device recovers depends on how much

persistent data the device requires, and whether that data is stored regularly and read after the system resets.

7.7.1.6 Choosing the timeout interval

Any safety chain is only as good as its weakest link, and if the software policy used to decide when to kick the dog is not good, then using watchdog hardware can make your system less reliable. In case of the ATM machine, the designer knows exactly the timing characteristics of his system, and then it was easy to define the timeout period of the watchdog timer (2 minutes in the example). If the designer does not fully understand the timing characteristics of his programme, he might pick a timeout interval that is too short. This could lead to occasional resets of the system, which may be difficult to diagnose. The inputs to the system, and the frequency of interrupts, can affect the length of a single loop.

One approach is to pick an interval which is several seconds long. Use this approach when you are only trying to reset a system that has definitely hung, but you do not want to do a detailed study of the timing of the system. This is a robust approach. Some systems require fast recovery, but for others, the only requirement is that the system is not left in a hung state indefinitely. For these more sluggish systems, there is no need to do precise measurements of the worst case time of the program's main loop to the nearest millisecond.

When picking the timeout you may also want to consider the greatest amount of damage the device can do between the original failure and the watchdog biting. With a slowly responding system, such as a large thermal mass, it may be acceptable to wait 10 seconds before resetting. Such a long time can guarantee that there will be no false watchdog resets. On a medical ventilator, 10 seconds would have been far too long to leave the patient unassisted, but if the device can recover within a second then the failure will have minimal impact, so a choice of a 500 ms timeout might be appropriate. When making such calculations the programmer must be sure that he includes the time taken for the device to start up as well as the timeout time of the watchdog itself.

One real-life example is the Space Shuttle's main engine controller. The watchdog timeout is set at 18ms, which is shorter than one major control cycle. The response to the watchdog timeout signal is to switch over to the backup computer. This mechanism allows control to pass from a failed computer to the backup before the engine has time to perform any irreversible actions.

While on the subject of timeouts, it is worth pointing out that some watchdog circuits allow the very first timeout to be considerably longer than the timeout used for the rest of the periodic checks.

This allows the processor time to initialize, without having to worry about the watchdog biting.

While the watchdog can often respond fast enough to halt mechanical systems, it offers little protection for damage that can be done by software alone. Consider an area of non-volatile RAM which may be overwritten with rubbish data if some loop goes out of control. It is likely that such an overwrite would occur far faster than a watchdog could detect the fault. For those situations you need some other protection such as a checksum. The watchdog is really just one layer of protection, and should form part of a comprehensive safety net.

Multiplying the interval

If you are not building the watchdog hardware yourself, then you may have little say in determining the longest interval available. On some microcontrollers the built-in watchdog has a maximum timeout on the order of a few hundred milliseconds. It you decide that you want more time, you need to multiply that in software.

Say the hardware provides a 100 ms timeout, but your policy says that you only want to check the system for sanity every 300 ms. You will have to reset the watchdog timer at an interval shorter than 100 ms, but only do the sanity check every third time the kick function is called. This approach may not be suitable for a single loop design if the main loop could take longer than 100 ms to execute.

One possibility is to move the sanity check out to an interrupt. The interrupt would be called every 100 ms, and would then reset the watchdog. On every third interrupt the interrupt function would check a flag that indicates that the main loop is still spinning. This flag is set at the end of the main loop, and cleared by the interrupt as soon as it has read it.

If you take the approach of resetting the watchdog from an interrupt, it is vital to have a check on the main loop, such as the one described in the previous paragraph. Otherwise it is possible to get into a situation where the main loop has hung, but the interrupt continues to reset the watchdog timer, and the watchdog never gets a chance to reset the system.

7.7.2 AVR Internal Watchdog Timer

The ***internal watchdog timer*** is a controlled timer that is used as a safety device. It is designed to work as a wakeup device in the case where the software is lost in some infinite loop or in case of faulty programme execution or the case where the processor is doing anything other than running the programme

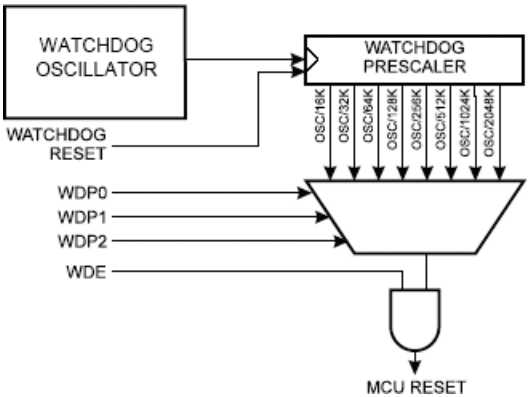


Figure 7.31 AVR watchdog timer.

it is supposed to be running. The watchdog timer has an output that has the capability to reset the controller.

The Watchdog Timer (Figure 7.31) is clocked from a separate on-chip oscillator which runs at 1 MHz. This is the typical value at VCC = 5 V. (See characterization data for typical values at other VCC levels). By controlling the Watchdog Timer prescaler, the Watchdog reset interval can be adjusted, see Table 7.4 for a detailed description. The WDR-Watchdog Reset — instruction resets the Watchdog Timer. Eight different clock cycle periods can be selected to determine the reset period. If the reset period expires without another Watchdog reset, the AT90S4414/8515 resets and executes from the reset vector.

To prevent unintentional disabling of the watchdog, a special turn-off sequence must be followed when the watchdog is disabled. Refer to the description of the Watchdog Timer Control Register for details.

7.7.3 Handling the Watchdog Timer

The different conditions for the watchdog timer are defined by the contents of the watchdog timer control register (WDTCR).

The Watchdog Enable bit WDE enables the watchdog function if set. Otherwise, the watchdog is disabled and no check of programme flow occurs.

The bits WDP2, WDP1 and WDP0 determine the watchdog timer prescaling when the watchdog timer is enabled. The different prescaling values and their corresponding timeout periods are shown in Table 7.4.

Table 7.4 Watchdog timer prescaler select.

WDP2	WDP1	WDP0	Number of WDT Oscillator Cycles	Timeout @ V_{cc} = 3.0 V	Timeout @ 5 V_{cc}
0	0	0	16 K (16,384)	17.1 ms	16.3 ms
0	0	1	32 K(32,768)	34.3 ms	32.5 ms
0	1	0	64 K (65,536)	68.5 ms	65 ms
0	1	1	128 K (131,072)	0.14 s	0.13 s
1	0	0	256 K (262,144)	0.27 s	0.26 s
1	0	1	512 K (524,288)	0.55 s	0.52 s
1	1	0	1024 K (1,048,576)	1.1 s	1.0 s
1	1	1	2,048 K (2,097,152)	2.2 s	2.1 s

To prevent unintentional disabling of the Watchdog, a special turn-off sequence must be followed when the Watchdog is disabled.

As mentioned before, when WDE is set (One) the Watchdog Timer is enabled, and if the WDE is cleared (zero) the Watchdog Timer function is disabled. WDE can only be cleared if WDTOE bit is set (one). To disable an enabled Watchdog Timer, the following procedure must be followed.

In the same operation, write a logical "1" to WDTOE and WDE. A logical "1" must be written to WDE even though it is set to one before disable operation starts.

Within the next four clock cycles, write a logical "0" to WDE. This disabled the Watchdog.

This sequence is shown in the following C language programme fragment.

Watchdog Timer Control Register — WDTCR

The watchdog Timer Control Register (WDTCR) is used to enable and disable the timer. It is used also to define the prescaler factor. WDTCR and bit definition are shown next.

Bit	7	6	5	4	3	2	1	0
$21 ($41)	-	-	-	WDTOE	WDE	WDP2	WDP1	WDP0
Read/Write	R	R	R	R/W	R/w	R/W	R/W	R/W
Initial value	0	0	0	0	0	0	0	0

Bit	Description
OWDTE	**Watch Dog Turn-Off Enable: S**et when the WDE bit is cleared. Otherwise, the watchdog will not be disabled. Once set, hardware will clear this bit to zero after four clock cycles
WDE	**Watch Dog Enable: S**et to enable the Watchdog Timer. If the WDE is cleared (zero) the Watchdog Timer function is disabled. WDE can only be cleared if the WDTOE bit is set.
WDP2	**Watchdog Timer Prescaler**: They define the Watchdog Timer prescaling when the Watchdog Timer is enabled (see Table-4.3)
WDP1	
WDP0	

Watchdog Timer Control Register WDTCR

The WDR-Watchdog Reset — instruction should always be executed before the Watchdog Timer is enabled. This ensures that the reset period will

be in accordance with the Watchdog Timer prescale settings. If the Watchdog Timer is enabled without reset, the watchdog timer may not start to count from zero.

Watchdog Programme Example

Optimum placement of watchdog refreshes within the programme is no simple task and generally is the last procedure before the programme is finalized. Normally, the user should examine the software flow and timing for all interrupt handlers and subroutines, critical and noncritical applications.

Ideally, one watchdog refresh (wdr instruction) in the whole programme would be the best, but because microcontrollers have large programmes and more on-chip functionality, this is rarely achievable. If possible, the watchdog refresh routines should never be placed in interrupt handlers or subroutines; they should be placed strategically in the main loop. The following C programme fragment shows this idea and also shows the sequence required to follow to disable the Watchdog timer.

```
;************************************************
    void main(void)
    {
      WDTCR = 0xB;       //enable WDT and set to
                         //120ms timeout
      while (1)
      {
        #asm(''wdr'')        // reset the watchdog
                             // timer
        If  (expression)     // if true disable the
                             // WDT
        {
        WDTCR = 0x18         // set both WDE and
                             // WDTOE
        WDTCR = 0x00         // clear WDE
        \ldots .
        \ldots ..
        /* other code here*/
        }
      }
    }
;************************************************
```

In the given programme fragment, the third line enables the watchdog timer and sets the timeout to 120 milliseconds. This is accomplished by setting the WDE bit and the watchdog prescaler bits in a single write to WTCR. The watchdog timer is reset each time the *while(1)* loop is executed by the "#asm("wdr")" instruction. This is an assembly code instruction (*Watch Dog Reset*) not available in the C language, and it must be executed before the watchdog timer has an opportunity to timeout.

The "if (expression)" is used to determine when to disable the watchdog timer. When "expression" is found to be True, the WDT will be disabled. Disabling the watchdog timer, as mentioned before, is a two-step process. Lines nine and ten represent the two steps.

7.8 Timer Applications

The timers have wide applications in the field of digital systems. Some of the possible applications are:

1. Measuring digital signals in time domain. This includes:
    a. The period of a periodic signal.
    b. The duration of a signal.
    c. The elapsed time between two events.
2. Measuring digital signals in frequency domain. This includes:
    a. The frequency of a periodic signal
    b. The number of events per time for a periodic or aperiodic signal.
3. Generating a periodic wave with arbitrary duty cycle.
4. Control external devices: This includes:
    a. Speed and position of DC and Servo motors.
    b. Generating a "tick" that can be used to start or control some external operations.

7.8.1 Application 1: Measuring Digital Signal in Time Domain

The Approach

To implement a time domain measurement, in general, a known signal is measured for an unknown time. In our case, the known signal is the timer clock and the unknown signal is the pulse that needs to be measured (See Figure 7.32).

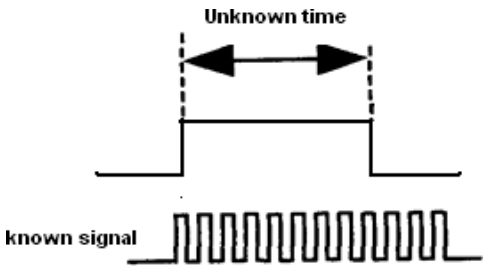


Figure 7.32 Measuring a known signal for an unknown time.

In Figure 7.32, if the rising edge of the unknown signal is used to enable a second signal of known frequency into a counter which will be disabled with the falling edge of the unknown signal, the value in the counter at the end of the unknown time will provide a measure of the duration of the unknown signal. The resolution of the measurement is a direct function of the frequency of the clock to the counter, $f_{counter}$.

$$\texttt{Resolution = 1/}f_{counter}\,\texttt{sec}$$

If from the rising edge to the falling edge of the unknown pulse the counter counts N, the time period T_p of the unknown pulse will be:

$$T_{\mathrm{p}} = \mathrm{N}/f_{counter}$$

For example, if the known frequency to the counter is 10 MHz and at the end of measurement 7500 counts have accrued, then:

$$\text{Resolution} = 0.1\mu\text{sec, and}$$

Duration of the unknown pulse $= 0.1 \times 7500 = 750\ \mu$sec.

The designer must take into consideration the case when the pulse duration T_p is larger than the range of the counter. If the counter width is m bits, then:

$$\texttt{Counter range = } 2^{\mathrm{m}}\texttt{x Resolution} = 2^{\mathrm{m}}/f_{counter}$$

Handling such situation is given in the next examples.

Inside vs. Outside

Many microcontrollers (also microprocessors) have one or more built-in timers that can be utilized for making both time and frequency measurements. The decision of the designer as to whether he is going to use one of the

internal timers or build the measurement circuitry outside of the microcontroller/microprocessor is based on the given design constraints and also on the speed and the duration of the signal to be measured. For systems with lower performance constraints but with more restricted cost constraints, utilizing as much of the internal circuitry as possible is the preferable alternative. Alternatively, when measuring signals with higher frequencies or shorter time durations combined with possible hard real-time constraints, a greater portion of the design will need to be implemented in hardware outside of the microprocessor.

A third alternative utilizes a combination of both external hardware and the internal timers. The hardware component manages the higher speed/shorter duration measurement, whereas the internal counter/timer deals with the slower portion.

In the following we are considering the use of the internal timers of the microcontroller to run the required measurements.

7.8.1.1 Measuring time interval using internal timer

The use of the Capture mode of Timer 1 is the best way to measure the width (the duration) of an unknown signal. The unknown signal is connected to the input capture pin of the microcontroller and is used as an interrupt signal. The interrupt is configured to trigger on the start edge of the pulse and trigger again on the tail edge. When the first interrupt occurs, the counter is started. At that point, the counter is incrementing at the known rate: the clock rate applied to the counter. When the second interrupt occurs, the counter is disabled. Its contents, combined with the frequency of the incrementing signal, provide sufficient information to compute the frequency of the unknown signal, thus,

The following example shows how to use the capture mode of Timer 1 of AVR to measure the width of a positive going pulse.

Example: Use of Input Capture Register:

Show the hardware and software that can be used to measure the width of a positive-going pulse applied to the input capture pin ICP of the microcontroller and that output the result, in milliseconds, on port C.

Solution

The hardware and software shown in Figures 7.33 and 7.34, respectively, demonstrate the use of the input capture register. This hardware and software are used to measure the width of a positive-going pulse applied to the ICP of the microcontroller and output the result, in milliseconds, on port C.

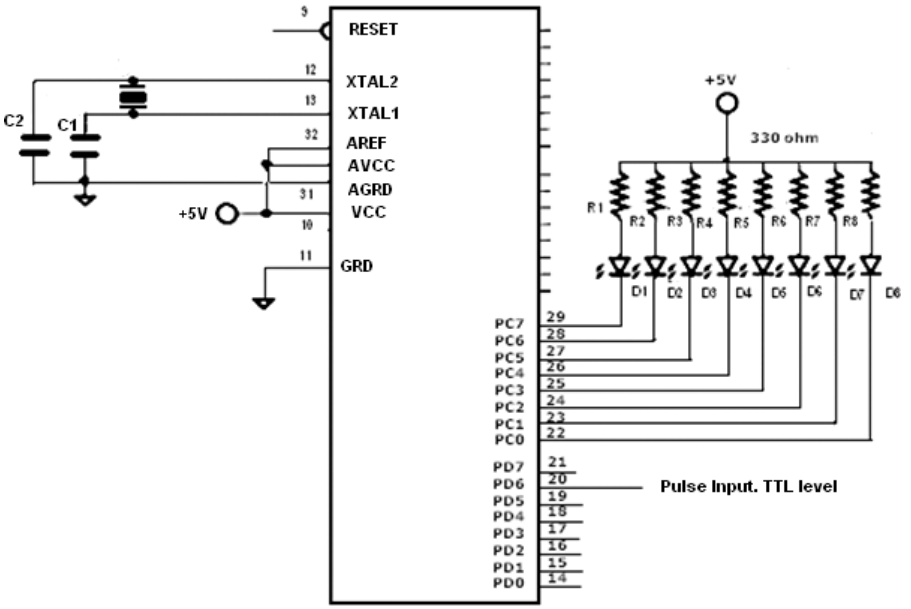


Figure 7.33 Pulse width measurement hardware.

The software in Figure 7.34 has several features worth noting. #define statements are used to connect a meaningful name with the output port and also to define the input capture pin, ICP, so it can be tested in the software.

The Interrupt Service Routine (ISR) for the Timer 1 overflow is used to handle the possibility that the period of the unknown pulse is greater than the range of the timer/counter. The ISR does nothing more than increment an overflow counter when the overflow occurs during a pulse measurement. The number of overflows is used in the calculation for the pulse width.

The ISR for the Timer 1 input capture event checks the actual state of the ICP to determine whether a rising or a falling edge has occurred. If the interrupt was triggered by a rising edge (ICP is now high), the programme must be starting to measure a new pulse, and the TSR records the rising edge time, changes the trigger sense on the input capture register to capture on a falling edge (to capture the end of the pulse), and clears the overflow counter for this measurement. If, on the other hand, ICP is found to be low, the interrupt must have been triggered by a falling edge, and the pulse has ended. In this case the length of the pulse is calculated, the result is output to port C, and the trigger sense on the input capture register is set to catch a rising edge for the start of the next pulse. Note that the CodeVisionAVR compiler allows access

```
//*********************************************************************
//define Port C as output for pulse width
#define pulse_out PORTC

//define ICP so we can read the pin
#define ICP PIND.6
unsigned char ov_counter;                    //counter for timer 1 overflow
unsigned int rising_edg, falling_edge;       //storage for times
unsigned long pulse_clocks;          //storage for actual clock count in the pulse
interrupt [TIM1_OVF] void timer1_ovf_isr(void)       //Timer 1 overflow ISR
{
        ov_counter++;                 //increment counter when overflow occurs
}
interrupt [TIM1_CAPT] void timer1_capt_isr(void)            //Timer 1 input
                                                 //capture ISR
{
        //Check for rising or falling edge by checking the level on ICP.
        //If high, the interrupt must have triggered by a rising edge,
        //and if not, the trigger must have been a falling edge.
        If (ICP)
        {
                rising_edge = ICR1;    //save start time for pulse
                TCCR1B = TCCR1B & 0xBF; //set to trigger on falling edge next
                Ov_counter = 0; //clear overflow counter for this measurement
        }
        else
        {
                falling_edge = ICR1;          //save falling edge time

                TCCR1B = TCCR1B    0x40;      //set for rising edge trigger next

                //calculation
                Pulse_clocks = (unsigned long) falling_edge - (unsigned long)
                rising_edge + (unsigned long) ov_counter * 0x10000;
                pulse_out = pulse_clocks / 500;      //output milliseconds to port C
        }
}
void main (void)
{
        DDRC = 0xFF;           //set Port C for output
        TCCR1B = 0xC2;         //Timer 1 input to clock/8, enable input
                               //capture on rising edge and noise canceller
        TIMSK = 0x24;          //unmask timer 1 overflow and capture interrupts
        #asm ("sei")           //enable global interrupt bit
        while (1)
                ;       //do nothing- interrupts do it all
}
//*********************************************
```

Figure 7.34 Pulse width measurement software.

to the 16-bit input capture register, ICR1, under a single name, ICR1. This is defined in the ATmega8515.h file. Not all compilers provide for 16-bit access to these registers, nor does CodeVisionAVR allow 16-bit access to all of the 16-bit registers. The reader has to check the appropriate header file for his processor.

The actual pulse width, in milliseconds, is calculated using the clock rate applied to the counter. The clock applied to Timer 1 is system clock (4 MHz) divided by 8, in this case 500 kHz, as initialized in TCCR1B at the beginning

of the *main*() function. The period of the 500 kHz clock is 2 microseconds. At 2 microseconds per clock tick, it takes 500 clock ticks to make 1 millisecond. Therefore the programme divides the resulting clock ticks by 500 to calculate the correct result in milliseconds.

Limitations of using Timer 1 for time measurements:

The only limitation of the above technique is when the resolution of the timer is less than the accuracy needed to measure the time. It is possible to change the timer/counter clock to get better resolution, accordingly better accuracy, but it may happened that the maximum accuracy that we can get from the timer is less than the needed accuracy. In such cases using external hardware with the required accuracy is the only solution.

7.8.1.2 Measuring time interval using external circuits

When Timer 1 resolution is not fulfilling the accuracy constraint needed, it is possible to achieve much greater accuracy and precision by the implementation of the measurement using external hardware. Another possible advantage of external implementation is that the microprocessor/microcontroller will not have direct participation in the measurement the matter which removes a large burden from the processor. One such implementation is shown in Figure 7.35.

The timing diagram for the system is shown in Figure 7.36. The Start signal from the microprocessor/microcontroller initiates the measurement. In

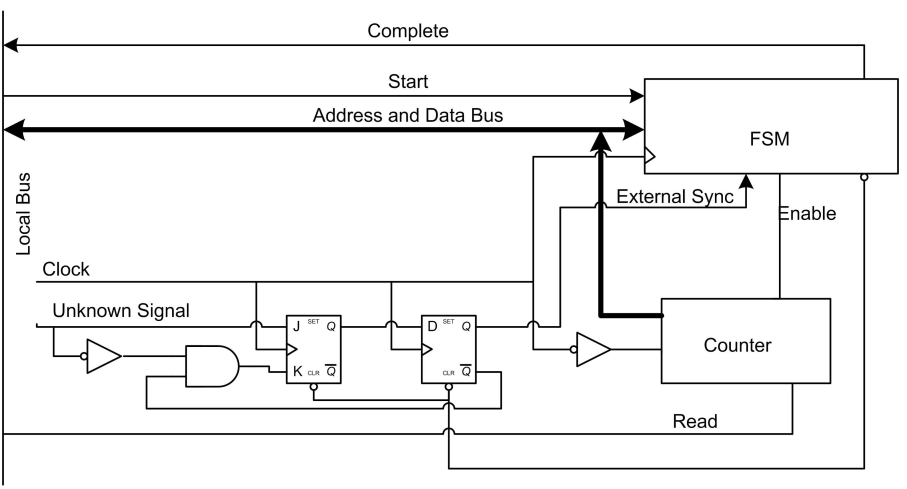


Figure 7.35 Measure the period of a signal based on a hardware implementation outside of the microprocessor.

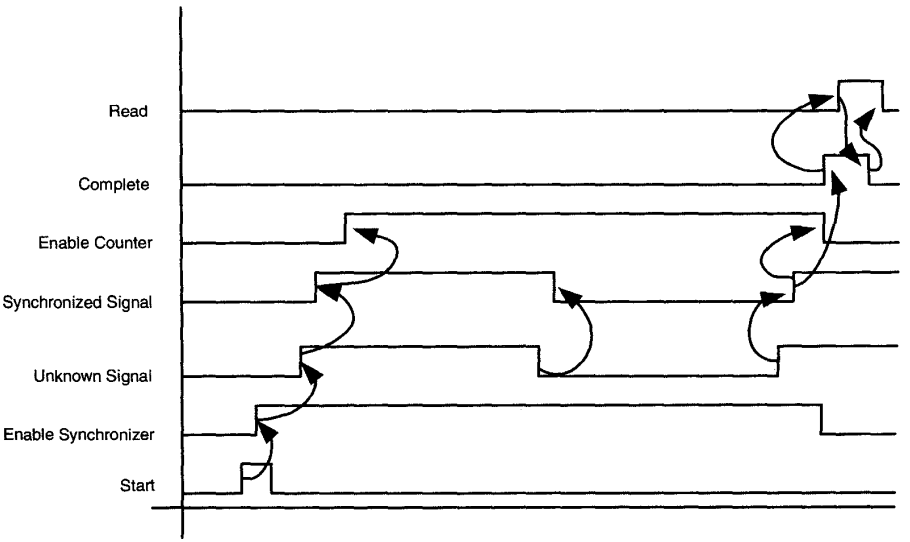


Figure 7.36 Timing diagram for system to measure the period of a signal based on a hardware implementation outside of the microprocessor.

response, the state machine enables the synchronizer. The synchronizer serves a dual role. In addition to synchronizing the external signal to the internal clock, the second synchronizer flip-flop serves to delimit the measurement. One to two *clock1* pulses after the external signal makes a 0 to 1 transition, the *Enable* signal is asserted by the state machine, thereby starting the counter. On the second 0 to 1 transition by the external signal, the *Enable* is deasserted and the *Complete* event is sent to the microprocessor. The state of the counter, representing the period of the unknown signal, can be read at any time after that.

7.8.2 Application 2: Measuring Unknown Frequency

To implement a frequency domain measurement, in general, an unknown signal is measured for a known amount of time (see Figure 7.37). As shown in Figure 7.37, the time period is known and the goal is to determine the frequency of the unknown signal. Thus, if the known duration is used as an enable to a counter, at the end of the time the counter will have accumulated a number of counts. If the duration of the known time is T_p and the counters counts N events during this period of time, the frequency of the unknown

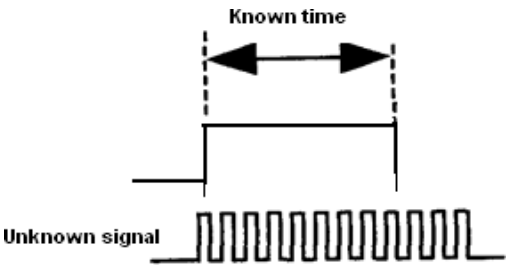


Figure 7.37 Measuring an unknown signal for known period of time.

signal will be

$$f_{unknown} = N/T_p$$

When using microcontroller, by applying an unknown signal to the timer/counter *capture input* we succeeded to measure time; the duration of the unknown signal. If we apply it to the *counter input* we can measure frequency. More accurately, to make frequency measurement we need two counters, one for the time base, and one counting the edges of the unknown signal. Ideally, the second counter must clear-on-capture, from the period signal of the time base.

As in the case of measuring time period, measuring the frequency of an unknown signal can be implemented using the built-in timer of the microcontroller, or using an external circuit.

a) Use of the Internal Timer/counter for Frequency Measurement

As shown in Figure 7.37, we use one counter as time base to open a gate for a known and specified interval. The unknown signal is used to increment another counter. If the known interval is T_{known}, and when the interval ends, the counter contains the value N, the frequency of the unknown signal will be

$$f_{unknown} = N/T_{known}.$$

The width of the window T_{known} depends on the precision needed and the resolution of the timer/counter.

To implement the measurement, a timer and a counter are needed. The unknown signal is used as an interrupt into the microcontroller. When the first interrupt occurs, the timer is started and the counter is incremented. The counter is incremented for each subsequent interrupt until the timer expires. At that point, the external interrupt is disabled. The counter will now contain the number of accrued counts, which translates directly to the unknown frequency.

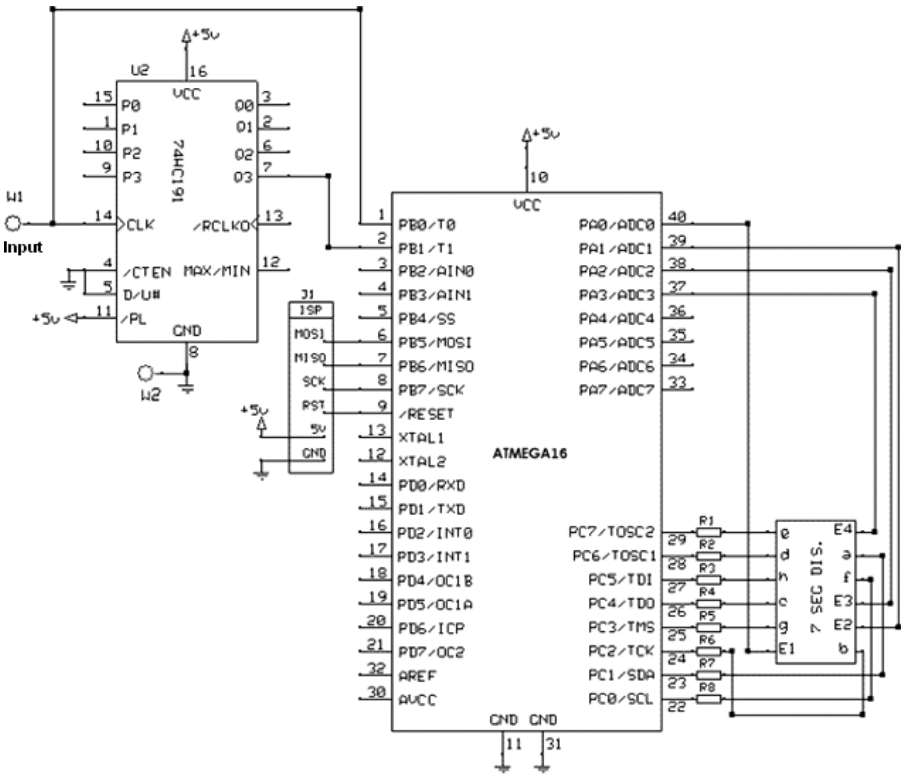


Figure 7.38 Hardware for measuring frequency.

In the following we show how to build a frequency meter using the timer of the microcontroller. The circuit given in Figure 7.38 can be used to measure frequencies up to 40 MHz with a very high accuracy (less than 1% error).

i. The hardware

The hardware shown in Figure 7.38 uses ATMEGA16 microcontroller with an internal oscillator of 8 MHz.

The frequency measurement probe is connected to W1 terminal, which is then fed to the pin PB0 and the clock input of the 74191 4-bit counter. The 4-bit counter is used to divide the measured frequency by 16 before feeding it to the microcontroller (pin PB1). The frequency of output Q3 of the counter is all the time input frequency to the counter divided by 16. In the figure, Q3 is connected to pin PB1 of the microcontroller.

The figure allows displaying the measured frequency by using standard four cells 7-segment module. The four cells are integrated in one package. Pins E1 to E4 are used to enable one of the 4 cells. ATMEGA16 microcontroller can source up to 40mA of current per I/O pin. This allows the 4 enable signals of the 7 segment display to be directly connected to the microcontroller.

It is known that to use a frequency meter to measure the frequency generated by another circuit, the two must share the same ground. For this reason a ground probe is proposed to connect W2 to ground. The ground probe is only used when the power supply of the frequency meter is different from the power supply of the device being tested, to connect both grounds together.

ii. Frequency measurement algorithm

There is an upper limit on the frequency that can be sampled by the counters of any microcontroller. In case of ATMEGA16, the maximum frequency that can be sampled by any one of its counters cannot exceed the CPU clock divided by 2.5. Let us call this frequency F_max. Assuming the ATMEGA16's CPU is clocked at 8 MHz, the maximum frequency that can directly measured is 3.2 Mhz. Frequencies above that limit will be measured as 3.2 MHz or less, since not all the pulses will be sampled. To be able to measure frequencies above F_max, we used the 4 bit counter as a frequency divider, dividing the measured frequency by 16. By this way it is also possible to measure frequencies up to 16 times F_max. Due to the limitation of the 74191 counter, the actual maximum measurable frequency will not exceed 40 MHz.

The algorithm used here measures both the original frequency (we call it F_o) and divided frequency (F_d). In normal conditions, when the frequency is below F_max, the following relation is true:

$$F\_o = 16 * F\_d$$

But as F_o approaches to F_max, more and more pulses won't be sampled, and the relation above will obviously become:

$$F\_o < 16 * F\_d$$

And hence the limit of the microcontroller can be automatically detected.

The frequency meter starts by selecting the original frequency for processing and display, and as soon as it detects that it reaches F_max (using the method described above), the divided frequency is selected instead.

This algorithm can be summarized by the chart shown in Figure 7.39. It is clear from the figure that timer T0 is used to measure the frequencies that is less than F_max, and timer T1 is used when the unknown frequency is more

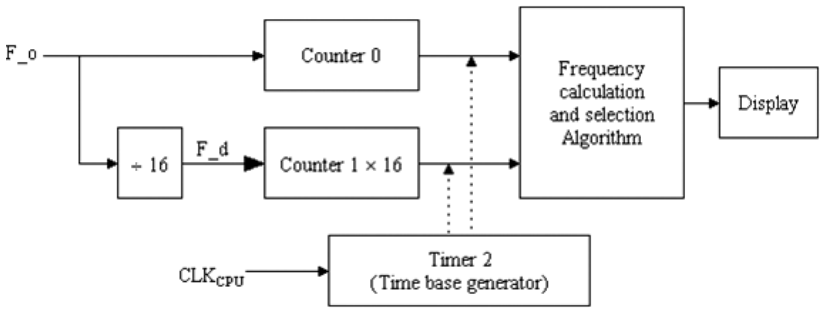


Figure 7.39 Summary of the algorithm.

than F_max. In the two cases timer T2 is used to generate the known period shown in Figure 7.39.

iii. The software

The C source code for this frequency meter is given on the site of the book. Besides the comments given in the source code, the following explanations will help understanding the code:

The code is made such that the number being displayed is in KHz. For example, if the 7 segments displays the number "325.8" that means 325.8 KHz, "3983" would mean 3983 KHz (or 3.983 MHz). If the number is in tenth of megahertz, it is displayed with an "m" at the end like "22.3 m" meaning 22.3 MHz.

Timer/Counter 0 is used to count incoming pulses directly.

Timer/Counter 1 is used to count incoming pulses divided by 16.

Timer/Counter 2 is configured as timer with a 1024 prescaller (counting CPU frequency divided by 1024). It is used to create the time base (the known window) T. T is defined as "$1024 * 256/(F\_cpu)$" (T2 is an 8-bit).

The constant "factor" defined in the top of the programme as "31.78581" have to be calibrated by measuring a known frequency. This factor was initially calculated as the following:

$$\text{factor} = F\_cpu/(1024 * 256) = 8 \times 106/(1024 * 256) = 30.51757$$

But due to the fact that there is some delay between the time the Timer_2 overflows and the time where the counted frequency is actually processed, this factor need some calibration to reflect more accurately the reality of the frequency being measured.

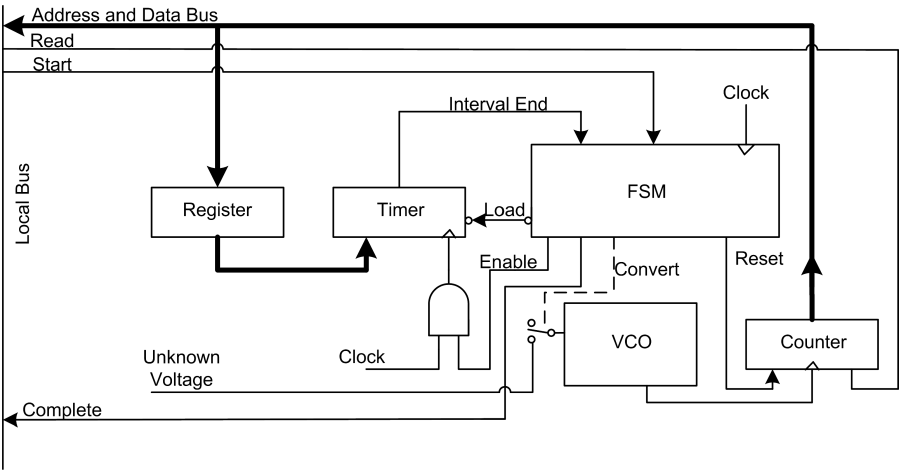


Figure 7.40 Block diagram for system to measure the frequency of a signal based on a hardware implementation outside of the microprocessor.

An Anti-flickering algorithm is used to totally prevent the display from quickly switching between various values, while still showing accurate values, and quickly change if the frequency being measured "really" changes.

b) Frequency Measurement Using an External Implementation

The frequency measurement can also be implemented in hardware outside of the microcontroller/microprocessor. Such a design is presented in Figure 7.40.

The high level timing diagram is given in Figure 7.41. In this design, as in the previous external hardware designs for other measurements, the process commences when the microprocessor issues the Start command. In turn, the state machine enables the time base, the synchronizer, and the counter. The time base generates a window, Enable Counter, for a length of time consistent with the measurement being made. When the window duration has expired, the Complete signal is asserted to the microprocessor, which responds with a Read command to read the state of the counter.

7.8.3 Application 3: Wave Generation

The compare mode of Timer 1 can be used to generate a programmable square wave. The following example shows the idea.

Example: Use TIMER1 compare mode to produce a 20 kHz square wave.

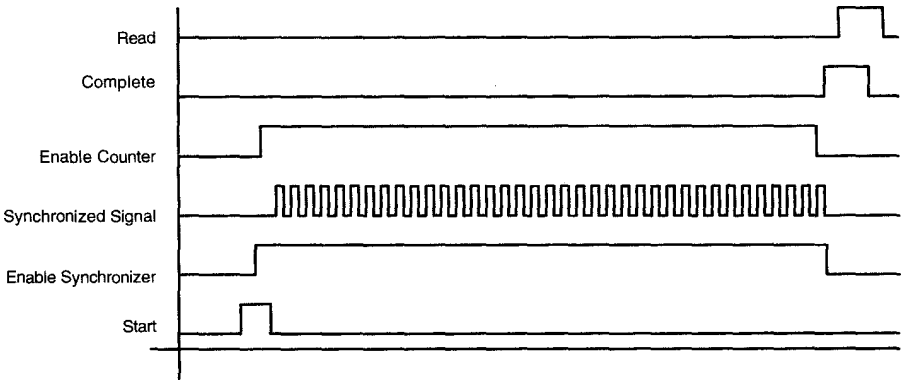


Figure 7.41 Timing diagram for system to measure the frequency of a signal based on a hardware implementation outside of the microprocessor.

Solution:

The scheme that can be used to generate the square wave using the compare mode works something like this:

1. When the first match occurs, the output bit is toggled, and the interrupt occurs.
2. In the interrupt service routine, the programme will calculate when the next match should occur and load that number into the compare register. In this case, the 20 kHz waveform has a period of 50 microseconds with 25 microseconds in each half of the waveform. So, the time from one toggle to the next is 25 microseconds. Using the frequency of the clock applied to the timer, the programme calculates the number of clock ticks that will occur in 25 microseconds.
3. This number of clock ticks is added to the existing contents of the compare register and reloaded into the compare register to cause the next toggle and to repeat the calculation and reload cycle.

An example of hardware and software demonstrating these concepts is shown in Figures 7.42 and 7.43, respectively. Note that no external hardware is required because the signal is produced entirely by the microcontroller.

In the programme in Figure 7.43, notice first the initializations that occur in main . Register DDRD is set to 0×20 so that the output compare bit, OC1A, that relates to output compare register A, OCR1A, is set for output mode, so that the output signal can appear on the bit. TCCR1A and TCCR1B set the prescaler to clock/8 (in this case clock = 8 MHZ, so clock/8 = 1 MHZ) and set the output compare mode for output compare register A to toggle the

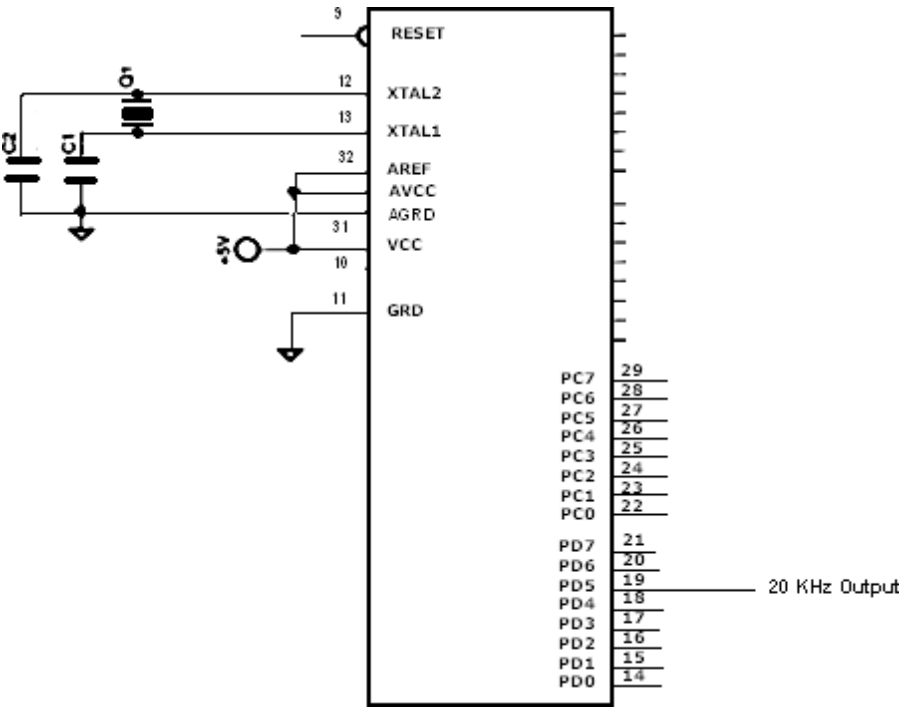


Figure 7.42 20 kHz hardware.

```
//*****************************************************************
// Timer 1 output compare A interrupt service routine
interrupt [TIM1_COMPA]  void timer_compa_isr (void)
{
        OCR1A = OCR1A + 25;    //set to next match (toggle) point
}
void main(void)
{
        DDRD = 0x20;           //set OC1A bit for output
        TCCR1A = 0x40;         //enable output compare mode to toggle OC1A pin
                               // on match
        TCCR1B = 0x02;         //set prescaler to clock/8 (1 microsec. Clocks)
        TIMSK = 0x10;          //unmask output compare match interrupt for register
A

        #asm("sei")            //set global interrupt bit

        while (1)
           ;                   //do nothing
}
//***********************************************
```

Figure 7.43 20 KHz software.

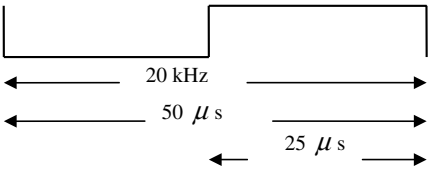


Figure 7.44 20 kHz waveform.

output bit, OC1A, when a match occurs. And finally, the compare interrupt is unmasked by setting the OCIE1A bit in the timer interrupt mask register, TIMSK.

Information such as that shown in Figure 7.44 is used to calculate the number that is added to the compare register contents to determine the point of the next toggle. In this case one- half of the waveform is 25 microseconds long. Since the clock applied to the counter is 1 MHz (dock/8), the number of clock cycles between match points (waveform toggle points) is given by the following:

Length of time between toggle points/clock cycle time = interval number

or, for this example:

```
25 μs/1 μs per clock cycle
   = 25 clock cycles per match point
```

And so, for this example, each time the compare match interrupt occurs, 25 is added to the current contents of the compare register to determine when the next match and toggle of the output waveform will occur.

One additional important point to consider relative to the output compare registers, and specifically relative to the calculation of the next match point number, is the situation in which adding the interval number to the current contents of the compare register results in a number bigger than 16 bits. For example, if the output compare register contains a number 65,000, and the interval is 1000, then

$$65,000 + 1000 = 66000 (\text{a number greater than } 65,535)$$

As long as unsigned integers are used for this calculation, those bits greater than 16 will be truncated and so the actual result will be:

$$65,000 + 1000 = 464 (\text{drop the 17th bit from 66000 to get 464})$$

This will work out perfectly, since the output compare register is a 16-bit register and the timer/counter is a 16-bit device as well. The timer counter will count from 65,000 up to 65,536 (a total of 536 counts) and then an additional 464 counts to reach the match point. The 536 counts plus the 464 counts is exactly 1000 counts as desired. In other words, in both the timer/counter and the compare register, rollover occurs at the same point, and as long as unsigned integers are used for the related math, rollover is not a problem.

7.8.4 Application 4: Use of PWM Mode: DC and Servo Motors Control

The ability to control different kinds of motors is very important in many applications. The applications ranging from assembly robots to remotely controlled vehicles to the precision positioning of medical instruments. Motors that are typically found in such applications fall into three categories: DC motors, servo motors, and stepper motors. DC and servo motors can be controlled using the same technique: use of PWM. PWM cannot be used to control stepper motor.

7.8.4.1 Introduction to DC Motors

Figure 7.45 gives a high-level diagram of the basic components of a DC motor.

These components comprise a stationary permanent magnet called a stator, a movable (rotating) electromagnetic called a rotor, and a system to connect power to the electromagnetic called brushes and a commutator.

Operation of the motor proceeds as follows: When a voltage is applied across the electromagnetic, the magnetic poles of the rotor are attracted to the

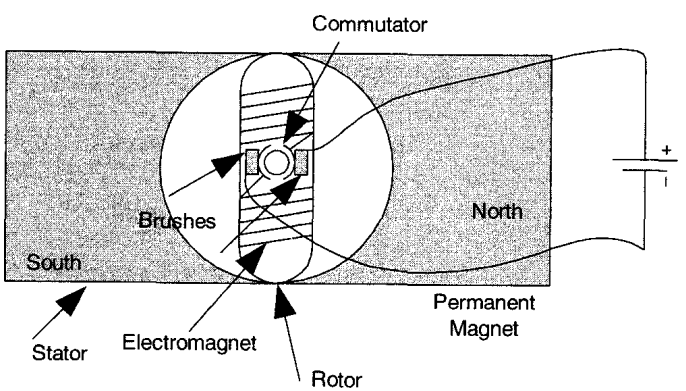


Figure 7.45 A basic DC motor.

opposite poles of the stator, thereby causing the rotor to turn. As the rotor turns, the electromagnet becomes polarized in the opposite direction and the poles of the rotor are now repelled by the nearer poles and are attracted to the opposite poles of the permanent magnet causing the rotor to turn once again.

The commutator is a split ring against which the brushes make physical contact. One portion of the commutator is connected to one end of the electromagnet, and the other portion is connected to the opposite end. Through the action of the commutator, the direction of the field in the electromagnet is continually switched, thus causing the rotor to continue to move.

Figure 7.46 is a simple illustration showing the actions of the commutator, brushes, and electromagnet. The brushes are fixed. However, as the rotor rotates, the commutator (which is attached to the rotor) acts like a switch, connecting the voltage source first one way then the opposite way across the electromagnetic thereby changing its polarization.

The DC motor has the ability to turn through 360 degrees, continuously, in one direction, when power is applied. Two important features can be noted here:

- If the applied voltage is held constant, the speed of the motor is also held constant; increasing or decreasing the applied voltage will have a corresponding effect on the speed of the motor.
- If the polarity of the applied voltage is reversed, the motor will run in the opposite direction.

These features define the mechanism that can be used to control the speed of the motor: control the applied voltage across the electromagnetic to control

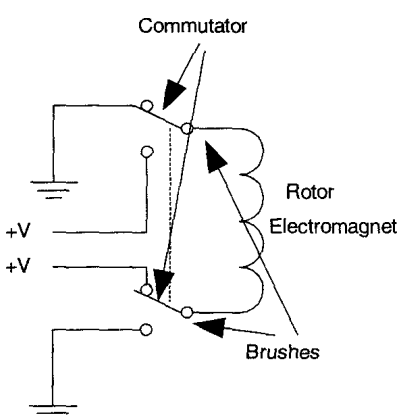


Figure 7.46 Schematic of the commutator, brushes, and electromagnet in a DC motor.

the speed and control the polarity of the applied voltage to control the direction of rotation. As an example, assume that a DC motor is driven by a voltage signal ranging from 0 to 12 V. To run the motor at full speed, a 12 V signal is applied; to run the motor at half speed a 6 V signal is applied; to run the motor at one-quarter speed a 3 V signal is applied; and so on.

This can be achieved using PWM. By using PWM scheme the average magnitude of the applied voltage can effectively be controlled and so can the speed of the motor. H-bridge may be used to manage the reversal.

Generally, a DC motor is not used for positioning tasks unless it is incorporated into a control system that can provide position (and possibly velocity) feedback information.

7.8.4.2 Introduction to servo motors

A servo motor is a special case of a DC motor to which position or velocity feedback circuitry has been added to implement a closed loop control system. Like the DC motor, the servo motor can rotate in either direction; however, generally the range is less than 360 degrees. Also like the DC motor, the servo motor can be controlled by a pulse width modulated signal; however, the signal is used to control position rather than speed. The servo motor finds common use in systems such as remotely controlled systems, robotics applications, numerically controlled machinery, plotters, or similar systems where starts and stops must be made quickly and accurate position is essential.

In conclusion, both the DC motor and the servo motor require a PWM signal to control either the speed or position.

7.8.4.3 Controlling DC Motors

As mentioned before, to control the speed of a DC motor we must control the applied voltage. Assume that a DC motor is driven by a voltage signal ranging from 0 to 12 V. The relation between the applied voltage and the speed of the motor is shown in Table 7.5. From the table, to run the motor at full speed of 3600 rpm, a 12 V signal is applied; to run the motor at 1440 rpm (40% of the full speed) 4.8 V signal is applied; to run the motor at 20% of the full speed a 2.4 V signal is applied; and so on.

This can be achieved using Timer 1 in the PWM mode of the microcontroller. When timer 1 is in PWM mode, the duty cycle of the waveform can be controlled by loading the compare register with the corresponding value. For example, to generate a waveform of 20% duty cycle it is enough to load the compare register with a value equals to 20% of its top value. If the compare register is of 8-bit width, the top value will be 255, and the 20% will be 51.

Table 7.5 Speed/Applied voltage relation.

Speed	Applied voltage
3600	12 V
2880	9.6 V
2160	7.2 V
1440	4.8 V
720	2.4V

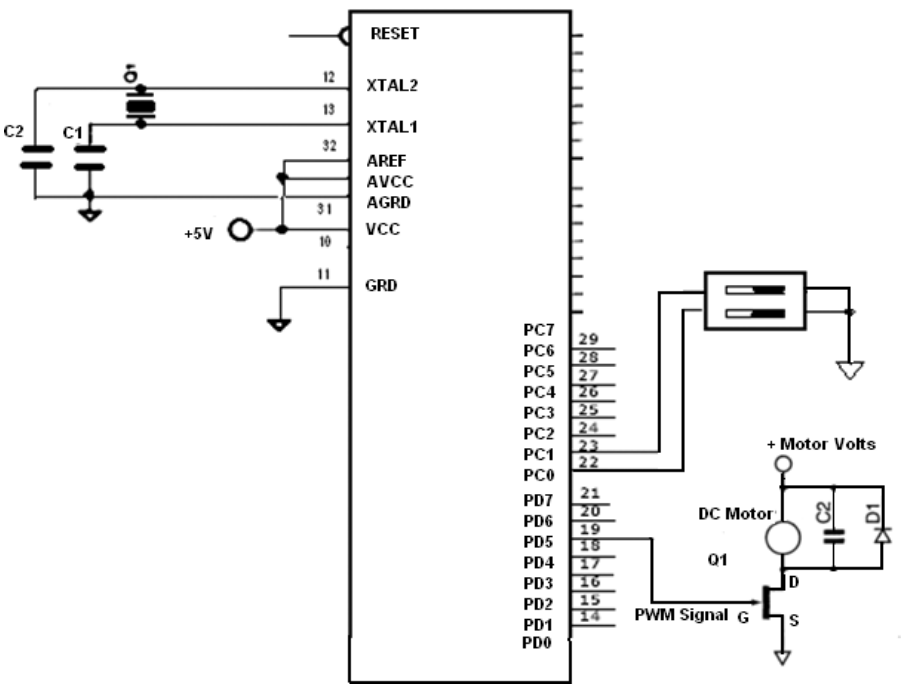


Figure 7.47 PWM hardware.

Figures 7.47 and 7.48 show the hardware and software, respectively, of an example programme using PWM. This programme provides four different duty cycles (10, 20, 30, and 40 percent at a frequency close to 2 kHz) based on the logic levels applied to port C.

Using Figure 7.23 it is clear that 8-bit resolution and system clock/8 would give 1.96 kHz, which is very close to the desired frequency.

The programme shown in Figure 7.48 uses a switch/case statement to select the duty cycle of the PWM output. A #define statement is used to mask port C so that only the two lower bits are used as input, and port C is set to

```
//*****************************************************
#include <Mega8515.h>
//use a 'define' to set the two lowest bits of port C as control input.
#define PWMselect (PINC & 3)
void main (void)
{
        unsigned int oldtogs;  //storage for past value of input
        PORTC = 0x03;          //enable the internal pull-ups for
                               //the input bits
        TCCR1A = 0x91;         //Compare A for non-inverted PWM and
                               //8 bit resolution
        TCCR1B = 0x02;         //clock/8 presacler
        While (1)
        {
                if (PWM_select != oldtogs)
                {
                        oldtogs = PWM_select;  //save toggle switch value
                        switch (PWM_select);
                        {
                                Case 0:
                                    OCR1A = 25; //10 % duty cycle desired
                                    Break;
                                Case 1:
                                    OCR1A = 51; //20 % duty cycle desired
                                     Break;
                                Case 2:
                                    OCR1A = 76;        //30% duty cycle desired
                                    Break;
                                Case 3:
                                    OCR1A = 102;       //40% duty cycle desired
                        }
                }
        }
}
//*************************************************
```

Figure 7.48 PWM software.

0x3 so that the internal pull-ups are enabled on the same lower two bits. This avoids the need for external pull-up resistors.

TCCR1A and TCCR1B are initialized to select the PWM mode and the prescaler to provide the desired frequency output.

Processor time in many microcontroller applications is precious. In this programme, an if statement is used along with the variable "oldtogs" so that new data is written to the pulse width modulator only if the switches are actually changed, thereby preventing continuous writes to the output compare register.

7.8.5 Application 5: Stepper Motors

A stepper motor is different from, and yet similar to, both the DC and the servo motor discussed before. One major difference is that each of the latter motors

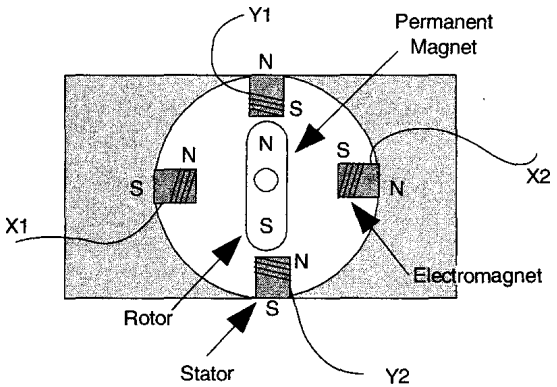


Figure 7.49 Basic components of a stepper motor.

moves in either the forward or reverse direction with a smooth and continuous motion; the stepper motor moves in a series of increments or steps. The size of each step is specified in degrees and varies with the design of the motor. The step size is selected based on the precision of the positioning required. The accompanying diagram in Figure 7.49 presents a high-level view of the essential elements of a stepper motor.

From Figure 7.49 it is clear that the rotor of stepper motor is a permanent magnet rather than the stator as in the DC and servo motors. Also it is easy to observe that the rotor of the stepper motor shown in Figure 7.49 has two teeth and has four poles and four electromagnets. This configuration has a step angle of 90 degrees, based on the spacing of the poles.

The first point to observe is that the rotor is a permanent magnet rather than the stator as in the DC and servo motors. The rotor in the motor in Figure 7.51 has two teeth, and the stator has four poles and four electromagnets.

In a stepper motor, the size of each step is specified in degrees and varies with the design of the motor. The step size is selected based on the precision of the positioning required. The simple motor given above has a step angle of 90 degrees, based on the spacing of the poles. Connections are made to the electromagnets through the signals marked Xl, X2, Yl, and Y2. Like the DC motor, the stepper can rotate through 360 degrees and in either direction. The speed of the motor is determined by the repetition rate of the steps.

7.8.5.1 Controlling stepper motors

Controlling stepper motors is not that much more complicated than controlling DC motors, although it may seem a lot like juggling as one tries to keep

all the signals straight. The control is based on the view that the stepper motor is an electromagnetic device that converts digital pulses (current) into mechanical shaft rotation. By controlling the sequence of the pulses applied to electromagnets we can control the angle of rotation, direction of rotation and speed of rotation. For example, to rotate the motor one step, we pass current through one or two of the coils; which particular coil or coils depends on the present orientation of the motor. Thus, rotating the motor 360 degrees requires applying current to the coils in a specified sequence. Applying the sequence in reverse causes reversed rotation.

To understand that we start by Figure 7.49 again. The polarization of the electromagnets as illustrated in Figure 7.50 requires that the indicated input signals are applied to Xl, X2, Yl, and Y2: V to Xl and Y2 and 0 to X2 and Yl.

If the input signals to V on Xl and Yl and 0 to X2 and Y2 are now changed, the polarization on the electromagnets changes to that shown in Figure 7.50. The two north poles at the top of the drawing will repel, and the north pole on the rotor will be attracted to the south pole on the right-hand side of the stator. The rotor will thus move 90° in the clockwise direction (see Figure 7.51).

Similarly, changes to the input signal levels shown in the accompanying table will produce the rotor movements shown in Figures 7.52 and 7.53.

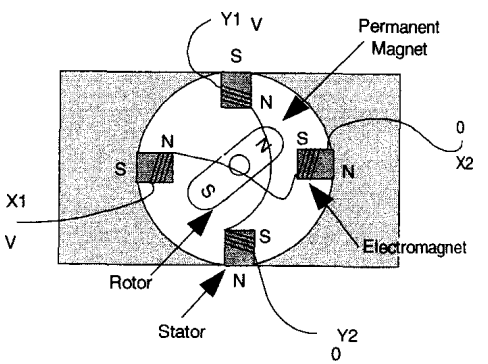


Figure 7.50 Controlling a stepper motor.

X1	X2	Y1	Y2	Position
V	0	0	V	0°
V	0	V	0	90°
0	V	V	0	180°
0	V	0	V	270°

Figure 7.51 Stepper Motor with 90° per step.

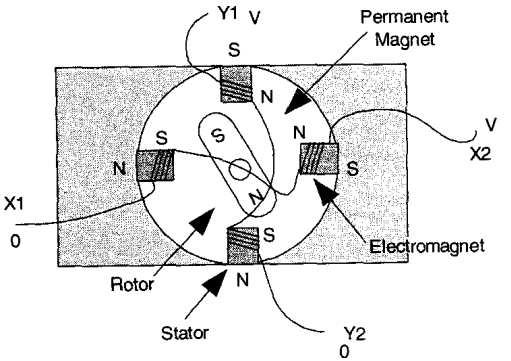


Figure 7.52 Controlling a stepper motor.

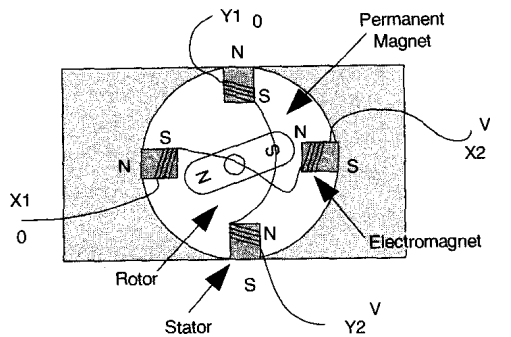


Figure 7.53 Controlling a stepper motor.

Extending the design to motors with a greater number of poles or stator teeth is a straightforward application of the pattern illustrated. The variable will be the number of times that the pattern will have to be repeated to achieve a full rotation as the number of degrees per step will decrease.

The timing diagram for one cycle (not full rotation) is given in Figure 7.54.

In some cases, the stepper motor comes with four inputs corresponding to the four coils, and with documentation that includes a table indicating the proper input sequence. To control the motor from software, we must maintain this table in software, and write a step routine that applies high values to the inputs based on the table values that follow the previously applied values. Microcontroller can be used to implement this technique.

In other cases the stepper motor comes with a built-in controller, which is an instance of a special-purpose processor, implementing this sequence. Thus, we merely create a pulse on an input signal of the motor causing the controller

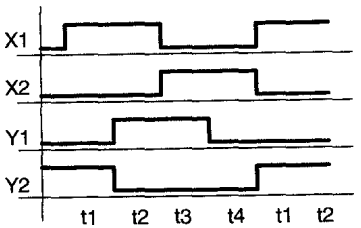


Figure 7.54 Stepper motor control timing diagram.

sequence	X1	Y1	X2	Y2
1	+	+	-	-
2	-	+	+	-
3	-	-	+	+
4	+	-	-	+
5	+	+	-	-

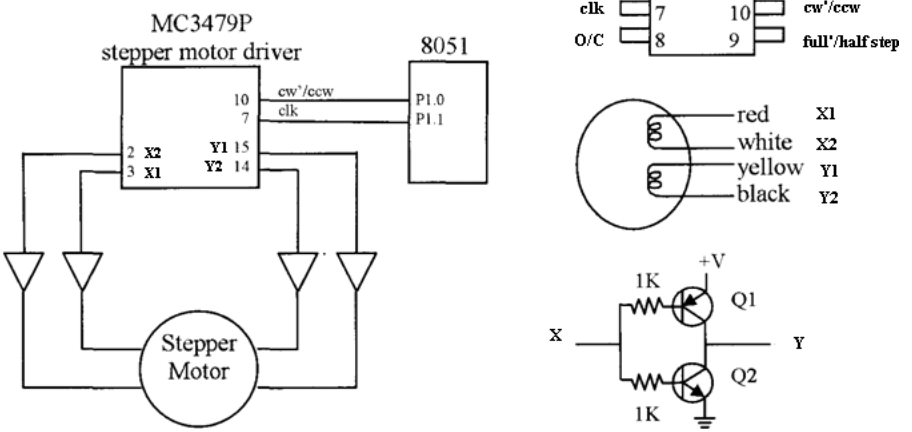


Figure 7.55 Controlling a stepper motor using a driver — hardware.

to generate the appropriate high signals to the coils that will cause the motor to rotate one step.

In the following we discuss the two cases.

a) Use of Stepper Motor Driver

Controlling a stepper motor requires applying a series of voltages to the four (typically) coils of the stepper motor. The coils are energized one or two at a time causing the motor to rotate one step. In this example, we are using a 9-volt, 2-phase bipolar stepper motor. Figure 7.55 shows a table indicating the input sequence required to rotate the motor. In this figure the motor step is

7.5 degrees. The entire sequence given in the table shown in the figure must be applied to get the motor to rotate 7.5 degrees. To rotate the motor in the opposite direction, we simply apply the sequence in reverse order. To rotate the motor an angle of $7.5 * N (N = 1, 2, \ldots.)$ the sequence must be repeated N times.

In the figure we used an 8051 microcontroller and a stepper motor driver (MC3479P) chip to control the stepper motor. We need only worry about setting the direction on the clockwise/counterclockwise pin *(cw'/ccw)* and pulsing the clock pin (*clk*) on the stepper motor driver chip using the 8051 microcontroller. Figure 7.55 gives the schematic showing how to connect the stepper motor to the driver, and the driver to the 8051. Figure 7.56 gives some sample code to run the stepper motor.

b) Controlling a Stepper Motor Using Microcontroller

In some cases, the stepper motor comes with four inputs corresponding to the four coils, and with documentation that includes a table indicating the proper input sequence. To control the motor from software, we must maintain

```
*/*********************************************
/* main.c */
sbit clk=Pl ^l;
sbit cw=Pl^0

void delay (void) {
        int     i, j;
        for (i=0; i<l000; i++)
                for (j=0; j<50; j++)
                        i = i + 0;
}

void main(void) {
        */turn the motor forward*/
         cw=0;          /*set direction*/
        clk=0;          /*pulse clock*/
        delay ( );
        clk=1;

        /*turn the motor backwards */
        cw=l;           /* set direction */
        clk=0;          /* pulse clock */
        delay ();
        clk=l;
}
*/*********************************************8
```

Figure 7.56 Controlling a stepper motor using a driver — software.

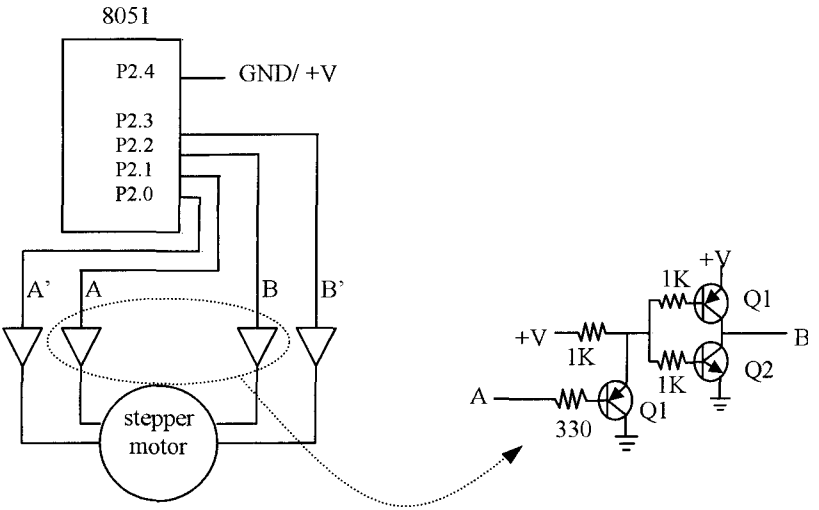


Figure 7.57 Controlling a stepper motor directly hardware.

this table in software, and write a step routine that applies high values to the inputs based on the table values that follow the previously applied values.

In the second example, the stepper motor driver is eliminated. The stepper motor is connected directly to the 8051 microcontroller. Figure 7.57 gives the schematic showing how to connect the stepper motor to the 8051. The direction of the stepper motor is controlled manually. If P2.4 is grounded, the motor rotates counterclockwise, otherwise the motor rotates clockwise. Figure 7.58 gives the code required to execute the input sequence to turn the motor.

Note that the 8051 ports are unable to directly supply the current needed to drive the motor. This can be solved by adding buffers. A possible way to implement the buffers is show in Figure 7.57. The 8051 alone cannot drive the stepper motor, so several transistors were added to increase the current going to the stepper motor. Q1 are M5E3055T NPN transistors and Q2 is an MJE2955T PNP transistor. A is connected to the 8051 microcontroller and B is connected to the stepper motor.

7.9 Summary of the Chapter

In this chapter we studied one of the most popular devices used in embedded and digital systems; Timers. Any system must include at least one hardware

```
;*****************************************************************
sbit    notA =P2^0;
sbit    isA
sbit    notB=P2^2;
sbit    isB=P2^3;
sbit    dir=P2^4;
void delay () {
        int     a, b;
        for(a=0; a<5000; a++)
                for(b=0;  b<l0000; b++);
}
void    move(int dir, int steps) {
        int y, z;
        if (dir = 1) {
                for(y=0; y<=steps; y++){
                        for(z=0; z<=l9; z+=4) {
                                isA= lookup[z];
                                isB=lookup[z+1];
                                notA=lookup[z+2];
                                notB=lookup[z+3];
                                delay ();
                        }
                }
        }
if(dir=0)
        for(y=0;  y<=step; y++){
                for(z=l9; z>=U; z -= 4){
                isA=lookup[z];
                isB=lookup[z-1];
                notA=lookup [z-2];
                notB=lookup [z-3];
                delay( );
                }
             }
        }
}
int 1ook[20] = {
        1, 1, 0, 0, 0, 1, 1, 0, 0, 0
        1, 1, 1, 0, 0, 1, 1, 1, 0, 0};
void main ()
         while (1) {
                /* move forward  15 degrees */
                move (1, 2);
                 /* move backwards 7.5 degrees */
                move (0, 1);
        }
}
;*****************************************************************
```

Figure 7.58 Controlling a stepper motor directly-software.

timer and there can be a number of software timers. The SWT is a virtual timing device. A timer is essentially a counter getting the count-input at regular time interval. This can be the system clock or the system clock divided by prescalar.

The internal timer of the microcontroller is a programmable timer that can be used in many applications and to generate interrupts for the ticks for the software timer. We studied a number of applications of the timer.

7.10 Review Questions

7.1 What is timer? How does a counter perform:
(a) Timer function
(b) Prefixed time initiated event
(c) Time capture function?

7.2 Why do we need at least one timer device in an embedded system?

7.3 An embedded system uses a processor that provides two hardware timers. The multitasking application comprises four tasks that require intervals of 1, 2, 6, and 10 μs seconds be timed and three tasks that intervals of 2, 5, 15, and 25 ms be timed.
Each timer is 16 bits, and each is clocked at 10 MHz. A 16-bit value can be loaded into a timer using a software load command.
Give the pseudo code for a time base that will support all of the tasks.

7.4 Design a software driver that will poll a periodic digital signal on an input port of your processor and, using a timer, determine the period of the signal on the port.
(a) What is the shortest period that your design will support, assuming that no other tasks are running?
(b) What is the worst case error in your measurement?
(c) What is the longest period that you can measure without timer overflow?
(d) How would you modify your design to accommodate timer overflow?

7.5 Design a software driver that will poll a periodic digital signal on an input port of ATmega16 microcontroller and, using a timer, determine the frequency of the signal on the port.
(a) What is the highest frequency that your design will support assuming that no other tasks are running?
(b) What is the worst case error in your measurement?
(c) What is the lowest frequency that you can measure without timer overflow?

7.6 Give a full hardware and software that use the PWM mode of AVR ATmega16 to control the speed of a DC motor. The output waveform should support controlling the motor from full OFF to full ON based on the value of an 8-bit control word. The hexadecimal value 0×00

should correspond to a speed of 0 RPM, and 0xFF should correspond to full speed.
(a) What is the smallest change in motor speed that you can control?
(b) Based on your answer to part (a), what is the worst case error in motor speed control?

7.7 Design the external hardware and software driver to control the position of a stepper motor assuming changes only in the forward direction.
(a) What is the smallest change in motor position that you can command?
(b) Based on your answer to part (a), what is the worst case error in motor position?

7.8 What is meant by a software timer (SWT)? How do the SWTs help in scheduling multiple tasks in real time?
Suppose three SWTs are programmed to timeout after the overflow interrupts 1024, 2048 and 4096 times from the overflow interrupts from the timer in Question 7.8. What will be the rate of timeout interrupts from each SWT?

7.9 Write a programme to output on pin P3.0 a signal which produces the notes of the musical scale (in square wave form) in sequence from C4 to CS. Each note should last for 0.5 seconds. The frequencies of the relevant notes are given in next table. The clock frequency is 12 MHz.

Frequencies of the C-major musical scale

Musical note	Frequency (Hz)
C4	262
D4	294
E4	330
F4	349
G4	392
A4	440
B4	494
C4	523

7.10 Write a programme to produce the following four lines of music as a signal on pin P3.0.

doh(3)	re(l)	mi(3)	doh(l)	mi(2)	doh(2)	mi(4)
re(3)	mi(l)	fa(l)	fa(1)	mi(1)	re(1)	fa(8)
mi(3)	fa(l)	so(3)	mi(l)	so(2)	mi(2)	so(4)
fa(3)	so(1)	la(1)	la(l)	so(1)	fa(l)	la(8)

The numbers in brackets indicate the duration of each note in units of (approximately) 0.5 seconds. The programme should repeat the tune

continuously. The following table lists the names and frequencies of the notes.

Names and frequencies of musical notes

Musical note	Tonic sol-fa	Frequency (Hz)
C4	Doh	262
D4	Re	294
E4	Mi	330
F4	Fa	349
G4	So	392
A4	La	440
B4	Ti	494
C4	Doh5	523

7.8 A 16-bit counter is getting inputs from an internal clock of 12 MHz. There is a prescaling circuit, which prescales by a factor of 16. What are the time intervals at which overflow interrupts will occur from this timer?
What will be the period before these interrupted must be serviced?

8

Interface to Local Devices — Analogue Data and Analogue Input/Output Subsystems

THINGS TO LOOK FOR…

- The meaning of data acquisition system.
- Performance criteria to evaluate our overall data acquisition system.
- The specifications necessary to select the proper transducer for our data acquisition system.
- How to analyze the sources of noise and how to suggest methods to reduce their effect.
- The main building blocks of data acquisition system.
- How to interface ADC and DAC to processors.
- Some generic ADC and DAC ICs.
- The software implementation of ADC.

8.1 Introduction

The processors considered in the previous chapters process digital, or discrete, signals. Many applications require monitoring or controlling of physical quantities such as temperature, position, or pressure. For instrumentation applications, monitoring, or measuring, such signals are the primary goal. The quantity being measured is referred to as the measurand. Signals associated with physical measurands are analogue, or continuous, signals. For each physical measurand a transducer is required that generates an electrical signal that varies in proportion to the measurand. Such a signal is called an analogue signal. For an analogue signal to be input to an embedded (or digital) system, an analogue input subsystem is required. An *analogue input subsystem* converts an analogue signal to a digital number. This number can then be input by the microprocessor/microcontroller and processed.

Control applications can be broadly characterized as either open loop or closed loop. In an open loop control application, a setpoint value, entered by a user or computed by the system, must be converted to an analogue signal to control a process. This requires an *analogue output subsystem*. The analogue output signal is transformed by an output transducer to some other form of energy.

In a closed loop control application, the value of the physical measurand to be controlled must first be measured before it can be controlled. A closed loop control system requires both an analogue input subsystem and an analogue output subsystem. A signal from the measurand's transducer is fed back and compared to the setpoint value to control the process.

An understanding of analogue input and analogue output subsystems and the circuits commonly used to construct them is crucial for many embedded system designs. Analogue-to-digital converters and digital-to-analogue converters provide the fundamental interface between analogue and digital signals. Other circuits used in analogue input and output subsystems include operational amplifiers (op-amps), filters, track-holds, analogue multiplexers, and analogue demultiplexers.

The chapter starts by providing a brief overview of analogue data and analogue input and output subsystems which generally known as analogue *data acquisition systems*. This brief also highlights some topics related to analogue input/output subsystems. The chapter next analyzes several different methods by which one can generate analogue signals, and then look at how various physical world analogue signals can be converted into a digital form that is more appropriate for use in the microcontroller/microprocessor. Three specific conversion algorithms — flash, counting and successive approximation converters — will be presented. Because the outputs of the various sensors and transducers are typically nonlinear, we examine how to deal with such nonlinearity. In the market now, many microcontrollers and microprocessors have on chip a built-in analogue to digital converters with all the needed interface circuitry. The chapter considers two of such microcontrollers and discusses the way of using them.

8.2 Analogue Data and Analogue I/O Subsystems

Analogue data is data that can have any one of an infinite number of values within some range. In electronic systems, analogue data is represented by analogue signals. At any time, an analogue signal has one of a continuous set

of values in a restricted range. Except for limitations imposed by electrical noise, the resolution of an analogue signal is infinite.

Like all electrical signals, an analogue signal is characterized by a pair of variables, voltage and current. The information in an analogue signal is associated with one or the other of these two variables. Most commonly, but certainly not exclusively, the voltage carries the information. For simplicity, we will usually assume that the information is represented by a signal's voltage. Operational amplifiers are normally used as the circuits that converts an information-carrying current to a voltage.

The exact voltage at a given time is important because this value represents the information embodied in the signal. For example, Motorola's MPX41 15 Altimeter/Barometer Absolute Pressure Sensor generates an output voltage that ranges from 0.2 V to 4.8 V. This voltage is directly proportional to pressures that range from 15 to 115 kPa (kilopascal). Reading a voltage of 2.0 V from this sensor corresponds to a pressure of 54 kPa.

8.2.1 Analogue Input and Analogue Output Subsystems

Figure 8.1(a) shows a block diagram of an analogue input subsystem. The input to the analogue input subsystem comes from an analogue input device. The analogue input device is usually an input transducer, or sensor, that converts some form of energy, such as heat, pressure, position, or light, into an analogue voltage or current.

For almost every type of physical measurand, numerous input transducers exist. A few transducers directly generate a digital output, like the optical shaft encoder, which transforms angular position to digital output pulses. However, we are now interested in input transducers that generate an analogue output. An analogue input subsystem must convert the transducer's analogue output voltage into a digital number or code so that it can be read by the microprocessor.

The task of an analogue output subsystem is just the opposite of that of an analogue input subsystem (Figure 8.1(b)). A digital number or code is converted to an analogue voltage or current. A system may need to generate output data in analogue form to provide analogue control signals, drive analogue output transducers, or to synthesize analogue waveforms. An analogue output device is often called an output transducer or actuator. The output transducer changes the analogue voltage to some other form of energy, for example mechanical position, rotation, pressure, or light.

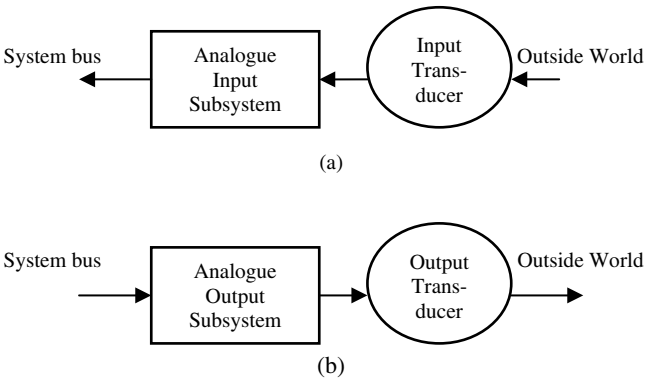


Figure 8.1 Analogue I/O subsystems: (a) analogue input device or input transducer provides analogue input signal to analogue input subsystem; (b) analogue output subsystem provides analogue signal to output device or output transducer.

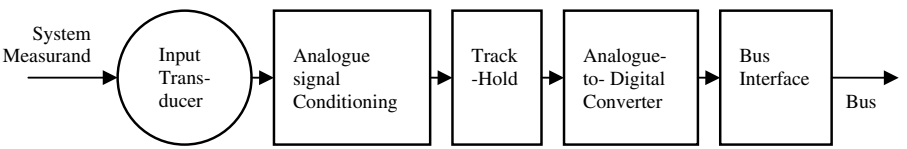


Figure 8.2 Single channel analogue data acquisition subsystem.

8.2.2 Components of an Analogue Input Subsystem: Data Acquisition System (DAS)

Analogue data conversion typically requires several analogue functions. Together, these functions comprise a data acquisition system (DAS). As shown in Figure 8.2, the analogue input signal to a data acquisition system is often the output of an input transducer, or sensor. The input transducer converts energy from the quantity being measured to an analogue voltage or current. The transducers provide a raw signal which must be conditioned and finally measured in digital form.

The flowchart for a typical data acquisition loop is shown in Figure 8.3.

8.2.2.1 Signal conditioning

The analogue signal from a transducer or other analogue input device is often not in the form needed for direct input to an analogue-to-digital converter (ADC). The signal is either too small, too noisy, has too high output impedance, is not referenced to ground, or has some deficiency. Consider as an example the case of thermocouple, the output voltage is frequently in the millivolt or even

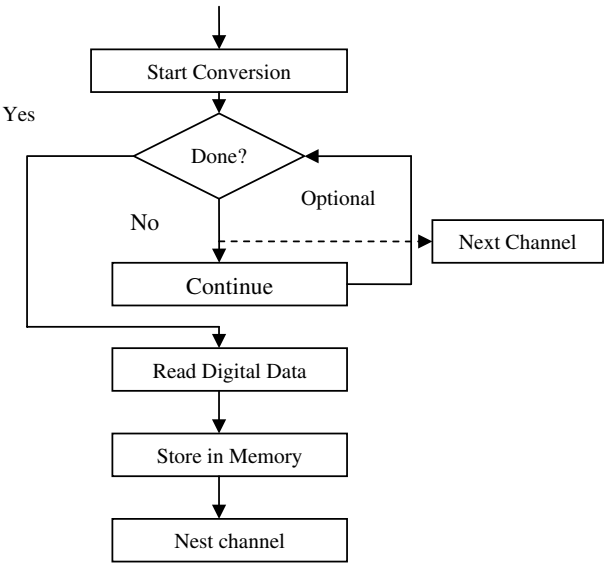


Figure 8.3 Typical data acquisition loop.

microvolt range. Amplification is then required to convert a low level analogue signal to a high level analogue signal (1 to 10 V) compatible with the input voltage range of the ADC. In general we need to use signal conditioning.

In an analogue input subsystem, the purpose of signal conditioning is to provide the ADC with a noise free signal compatible with its input range. Amplification and filtering are the most commonly required signal conditioning operations.

8.2.2.2 Filtering: Analogue filter

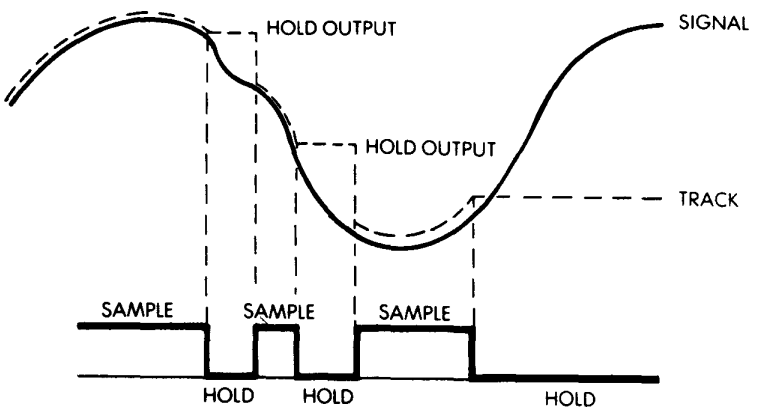
Filtering (i.e. use of analogue filter) is another form of signal conditioning. It may be necessarily to use it to eliminate noise or undesired frequency components in an analogue signal to remove, for example, aliasing error caused by ADC sampling. Other analogue signal conditioning circuits implement operations such as buffering, clamping, and linearization. All these operations are used to transform the output of a transducer into a signal suitable for conversion.

8.2.2.3 Track-hold or sample-and-hold

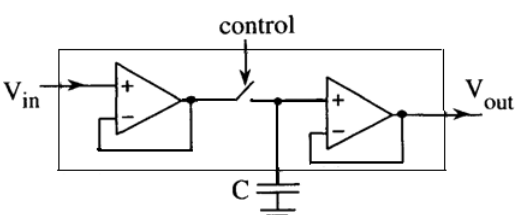
Because of their architecture, some types of ADCs require that their analogue input remain constant for the duration of the time it takes to complete

conversion. In addition, in some applications it is required to determine the input analogue value at a precise instant. Use of a ***track-hold*** (or *sample* and *hold, S/H*) circuit allows both these requirements to be met. When used at the input of an ADC, a track-hold latches the analogue input signal at the precise time dictated by its control signal; normally it holds it constant until the next sample. The ADC then converts the output voltage of the track-hold. The device holds the sample in a high-quality capacitor, buffered by an operational amplifier. The operation of a sample-and-hold circuit is shown in Figure 8.4.

In Figure 8.4 the control signal (the digital input signal) determines the S/H mode. The S/H is in *sample mode*, where V_{out} equals V_{in}, when the switch is closed. The S/H is in *hold mode*, where V_{out} is fixed, when the switch is open. The *acquisition time* is the time for the output to equal the input after the control is switched from hold to sample. This is the time to charge the capacitor C. The *aperture time* is the time for the output to stabilize after the control is switched from sample to hold. This is the time to open the switch



(a)



(b)

Figure 8.4 Sample and hold (S/H): (a) Operation (b) Circuit.

that is usually quite fast. The *droop rate* is the output voltage slope (dV_{out}/dt) when control equals hold. Normally the gain K should be 1 and the offset V_{off} should be 0. The gain and offset error specify how close the V_{out}, is to the desired V_{in} when control equals sample,

$$V_{\text{out}} = K V_{\text{in}} + V_{\text{off}}$$

where K should be one and V_{off} should be zero.

To choose the external capacitor C:

- One should use a polystyrene capacitor because of its high insulation resistance and low dielectric absorption
- A larger value of C will decrease (improve) the droop rate. If the droop current is I_{DR} then the droop rate will be:

$$dV_{\text{out}}/dt = I_{\text{DR}}/C$$

- A smaller C will decrease (improve) the acquisition time

8.2.2.4 Analogue to Digital Converter (ADC)

The appropriately conditioned analogue signal is converted to digital data by an *analogue-to-digital converter* (ADC or A/D). The output of the ADC is a binary number or code that is proportional to the analogue signal at the input of the ADC.

8.2.2.5 Analogue multiplexer

Normally, data acquisition system is a multichannel system. In the multichannel data acquisition system of Figure 8.5, an analogue multiplexer is used to select an analogue input for conversion. The multiplexer has a number of control signals representing the "channel address" inputs. The channel address bits are used to select which of the analogue input channels voltage appears as the output of the multiplexer.

In the multichannel data acquisition system of Figure 8.5, a single ADC converts several analogue inputs in a time multiplexed fashion. The microprocessor selects the channel to be input to the ADC by supplying the appropriate channel address to the multiplexer (Figure 8.6). The microprocessor then causes the track-hold to latch the multiplexer's output. Finally, the microprocessor initiates the conversion. After the conversion is complete, the microprocessor can select another channel and initiate the next conversion.

Multichannel systems can also be implemented using a dedicated converter for each channel. This approach is called "converter per channel". Using a

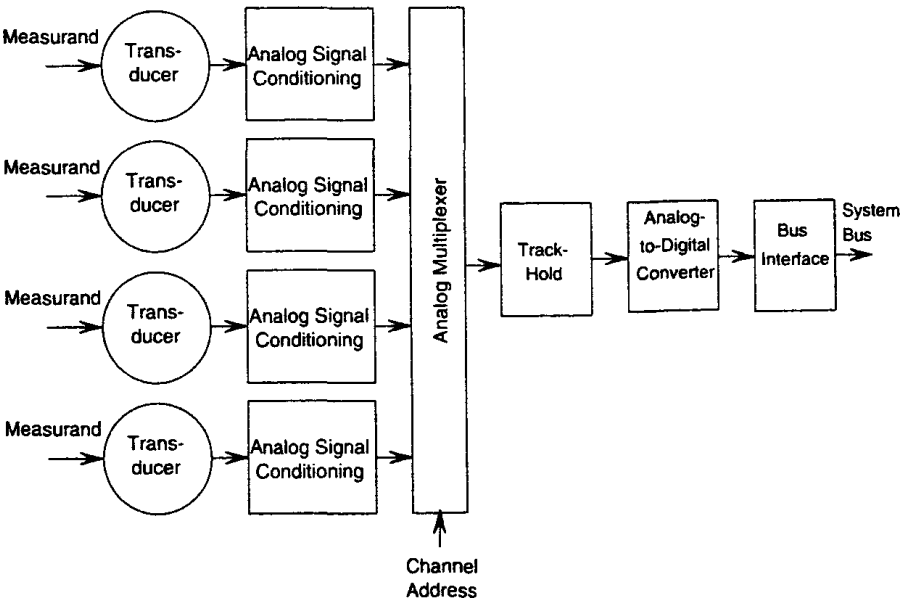


Figure 8.5 Multichannel analogue data acquisition subsystem.

converter per channel eliminates the need for multiplexing but is more costly. Each approach has its advantages and disadvantages.

Example 8.1: A 16-channel multiplexer

Figure 8.6 shows the structure of a 16-channel multiplexer. One of 16 inputs may be selected by a combination of 4 input signals derived from A0, A1, A2, A3 on the address bus.

Whenever all 16 lines must be sampled sequentially, a hardware counter may be used. The resulting organization is shown on Figure 8.7, and the resulting switching is shown on Figure 8.8.

Note:

Sensors and Transducers: A distinction is sometimes made between a sensor and a transducer. The definitions that follow correspond to the most commonly made distinction. An element that senses one form of energy, or a change in that form of energy, and transforms it into another form of energy, or change in another form of energy, is a ***sensor***. An **electrical sensor** converts a measurand, or a change in measurand, to an electrical signal. In contrast, a ***transducer*** is a device composed of a sensor and other elements that convert the output of the sensor to a signal that is more appropriate to the application.

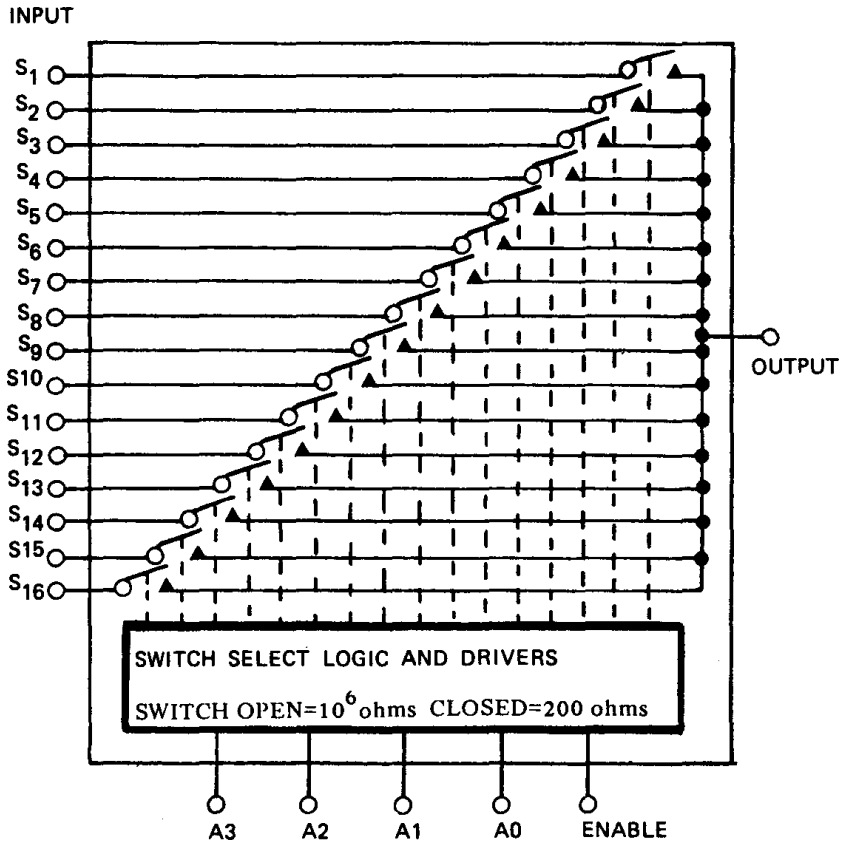


Figure 8.6 An analogue multiplexer.

8.2.3 Components for an Analogue Output Subsystem

An analogue output subsystem typically comprises of the components shown in Figure 8.9. The bus interface allows the microprocessor/microcontroller to write a digital value to the digital-to-analogue converter (DAC). The DAC converts digital data from the system bus to a piecewise continuous analogue output. The output of the DAC is normally connected to an analogue signal conditioning to amplify, filter, buffer, or scale the analogue voltage to the output transducer. Since a single analogue output is generated in Figure 8.9, this subsystem is referred to as a single channel analogue output subsystem.

A multichannel analogue output subsystem, sometimes called a data distribution system, provides several independent analogue outputs. A separate DAC can be used to produce each output, creating a DAC per channel mul-

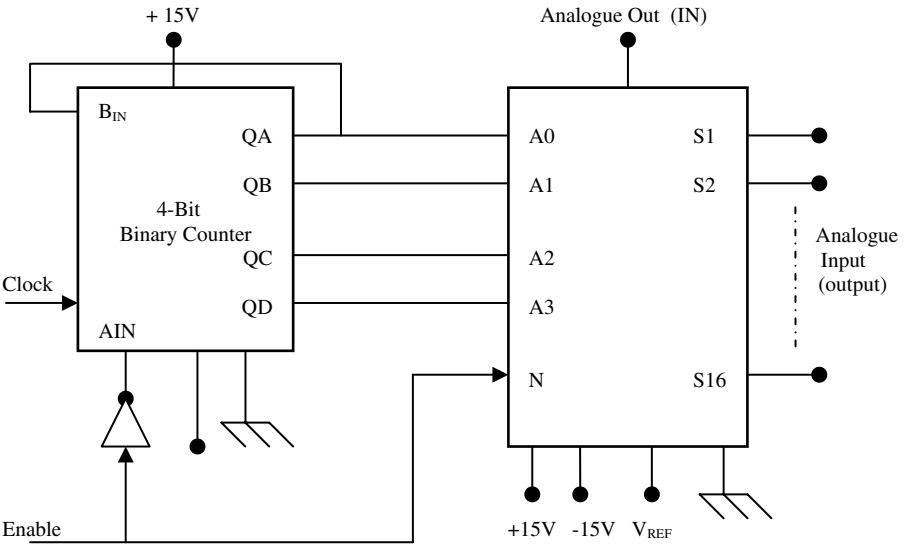


Figure 8.7 16 Channel sequential multiplexer.

ENABLE	MUX SEQUENCE RATE	MUX INPUTS				DG506 SWITCH STATES (-DENOTES OFF)															
		A_0	A_1	A_2	A_3	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}	S_{13}	S_{14}	S_{15}	S_{16}
0	0	X	X	X	X	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
1	0	0	0	0	0	ON	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
1	1PULSE	1	0	0	0	—	ON	—	—	—	—	—	—	—	—	—	—	—	—	—	—
1	2 PULSES	0	1	0	0	—	—	ON	—	—	—	—	—	—	—	—	—	—	—	—	—
1	3 PULSES	1	1	0	0	—	—	—	ON	—	—	—	—	—	—	—	—	—	—	—	—
1	4 PULSES	0	0	1	0	—	—	—	—	ON	—	—	—	—	—	—	—	—	—	—	—
1	5 PULSES	1	0	1	0	—	—	—	—	—	ON	—	—	—	—	—	—	—	—	—	—
1	6 PULSES	0	1	1	0	—	—	—	—	—	—	ON	—	—	—	—	—	—	—	—	—
1	7 PULSES	1	1	1	0	—	—	—	—	—	—	—	ON	—	—	—	—	—	—	—	—
1	8 PULSES	0	0	0	1	—	—	—	—	—	—	—	—	ON	—	—	—	—	—	—	—
1	9 PULSES	1	0	0	1	—	—	—	—	—	—	—	—	—	ON	—	—	—	—	—	—
1	10 PULSES	0	1	0	1	—	—	—	—	—	—	—	—	—	—	ON	—	—	—	—	—
1	11 PULSES	1	1	0	1	—	—	—	—	—	—	—	—	—	—	—	ON	—	—	—	—
1	12 PULSES	0	0	1	1	—	—	—	—	—	—	—	—	—	—	—	—	ON	—	—	—
1	13 PULSES	1	0	1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	ON	—	—
1	14 PULSES	0	1	1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	ON	—
1	15 PULSES	1	1	1	1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	ON
1	16 PULSES	0	0	0	0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—

Figure 8.8 Sequential Mux truth-table.

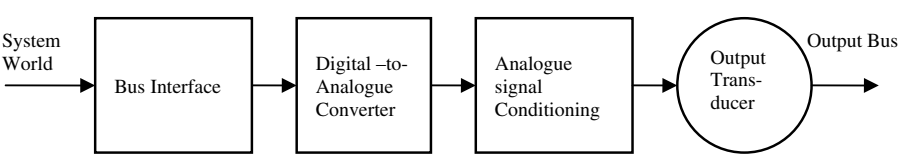


Figure 8.9 Single channel analogue output subsystem.

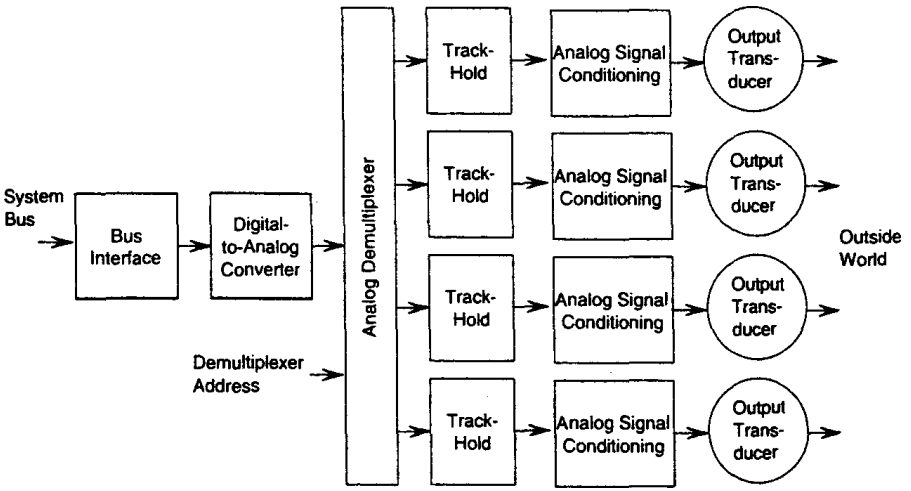


Figure 8.10 Multichannel multiplexed analogue output subsystem.

tichannel subsystem. However, an alternative approach using a demultiplexer can generate multiple analogue outputs by time multiplexing a single DAC. The single DAC is followed by an analogue demultiplexer with a separate track- hold for each output (Figure 8.10). The analogue demultiplexer is a digitally controlled analogue switch that routes the DAC's output to the selected track-hold.

When in its track mode, a track-hold (or sample-hold) circuit's output tracks (follows) its analogue input. This mode is used to acquire a new output value. When switched to hold mode, the track-hold holds (latches) its analogue input's value at the instant the track-hold was switched to hold mode.

In the above note we defined three of the main components of Figures 8.2 and 8.9. In the next sections we discuss the details of the ADC and the DAC converters.

8.3 Digital-to-Analogue Converters (DACs)

A DAC, or D/A, accepts n-bits as input and provides an analogue current or voltage as output (Figure 8.9). The output of the DAC is usually a linear function of its digital input. Some DAC have an intentionally nonlinear output function. These types of DACs are less common and are not of interest in this text. The DAC in Figure 8.11 accepts parallel input data. Many DACs can be

interfaced to SPI synchronous serial port. In such serial DACs the input data is serially shifted into the DAC.

The DAC ***precision*** is the number of distinguishable DAC outputs (e.g., 256 alternatives, 8 bits). The DAC ***range*** is the maximum and minimum DAC output (volts, amperes).

The DAC ***resolution*** is the smallest distinguishable change in output. The units of resolution are in volts or amperes depending on whether the output is voltage or current. The resolution is the change that occurs when the digital input changes by 1.

$$\text{Range (volts)} = \text{precision (alternatives)} * \text{Resolution (volts)}$$

A simple DAC consisting of a register, a DAC circuit, a voltage reference, and an op amp current-to-voltage converter is shown in Figure 8.12. The register holds the digital input to the DAC circuit, allowing the DAC to be interfaced to the system bus.

The DAC circuit consists of electronic analogue switches and a network of either precision resistors or capacitors. Each bit of the digital input controls one or more switches. The switches in turn control currents or voltages derived from the reference voltage. The result is an output current or voltage that is proportional to the input code. The voltage reference and current-to-voltage

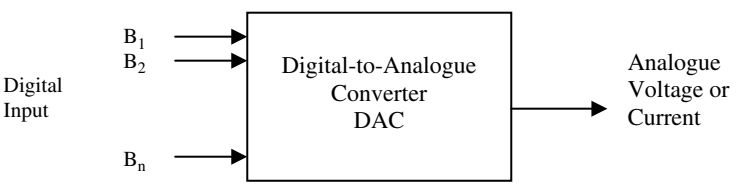


Figure 8.11 Parallel input DAC.

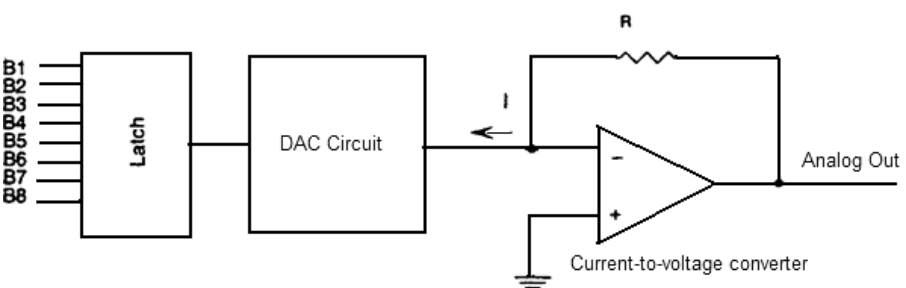


Figure 8.12 Digital-to-Analogue Converter (DAC).

converter may or may not be included on a DAC IC. If not, an external voltage reference and an external op-amp are used.

The output of an ideal (error free, i.e. no offset voltage) DAC that accepts an n-bit straight binary code is:

$$V_o = V_{\text{REF}} \times (B_1 2^{-1} + B_2 2^{-2} + \cdots + B_n 2^{-n}) \tag{8.1}$$

where B_1-the most significant bit, and B_n-the least significant bit of the binary input. The output voltage of DAC is the product of the reference voltage and the binary fraction $0.B_1B_2\ldots B_n$. Some DACs require V_{REF} to be a constant value for the equation to be valid. Some DACs have a negative output voltage. The sign of the output voltage is a function of the type of DAC circuit and the polarity of the voltage reference.

In the case when the DAC has an offset voltage V_{os}, the above equation takes the form:

$$\text{for unsigned: } V_o = V_{\text{REF}} * (B_1 2^{-1} + B_2 2^{-2} + \cdots + B_n 2^{-n}) + V_{os} \tag{8.2}$$

and for 2's complement:

$$V_o = V_{\text{REF}} * (-B_1 2^{-1} + B_2 2^{-2} + \cdots + B_n 2^{-n}) + V_{os} \tag{8.3}$$

In such case we use the term "DAC ***accuracy***" which is defined as:

$$\text{Accuracy} = (\text{actual} - \text{ideal})/\text{ideal},$$

where ideal is referred to the standard given by the National Bureau of Standards (NBS) (See Figure 8.13).

Multiplying DACs are DACs for which the previous equation is valid even if V_{REF} varies from 0 to full scale. The output of a multiplying DAC is the product of two variables, the digital input and the analogue voltage reference.

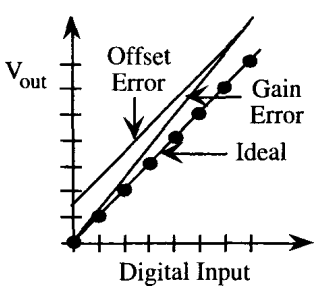


Figure 8.13 Ideal performance and actual values.

One can choose the full-scale range of the DAC to simplify the use of fixed-point mathematics. For example, if an 8-bit DAC had a full-scale range of 0 to 2.55 V, then the resolution would be exactly 10 mV. This means that if the D/A digital input were 123, then the D/A output voltage would be 1.23 V.

Linearity and monotonicity: To define linearity, let m, n be digital inputs, and let $f(n)$ be the analogue output of the DAC. Let Δ be the DAC resolution. The DAC is linear if:

$$f(n+1) - f(n) = f(m+1) - f(m) = \Delta \quad \text{for all } n, m$$

The DAC is monotonic if:

$$sign[f(n+1) - f(n)] = sign[f(m+1) - f(m)] \quad \text{for all } n, m$$

Conversely, the DAC is nonlinear if

$$f(n+1) - f(n) \neq f(m+1) - f(m) \quad \text{for some } m, n$$

Practically speaking all DACs are nonlinear, but the worst nonlinearity is nonmonoticity. The DAC is nonmonotonic if:

$$sign[f(n+1) - f(n)] \neq sign[f(m+1) - f(m)] \quad \text{for some } n, m$$

8.3.1 Ideal DACs

An error-free DAC is referred to as being ideal. The transfer function of an ideal 3-bit DAC with a straight binary input code is shown in Figure 8.14. The transfer function plots the output of the circuit versus its input. The analogue output is not continuous, but has one of eight possible values corresponding to the eight possible 3-bit input codes. In general, an n-bit DAC with a fixed reference voltage can have only 2^n possible output values. Thus, using a DAC you can only approximate a desired analogue value by using the closest discrete output value from the DAC.

The binary inputs of DAC are written as numbers that range from 000 through 111. Although a binary point is not written, these numbers actually represent eight fractions from 0.000 through 0.111 (from 0 through the maximum).

The maximum output voltage of an ideal DAC is:

$$V_{OMAX} = V_{REF} \times (1 - 2^{-n})$$

For $n = 3$, the maximum output is 7/8 of the nominal full scale (FS), or full scale range (FSR) output. If a 3-bit DAC has a FS value of 10 V, the actual

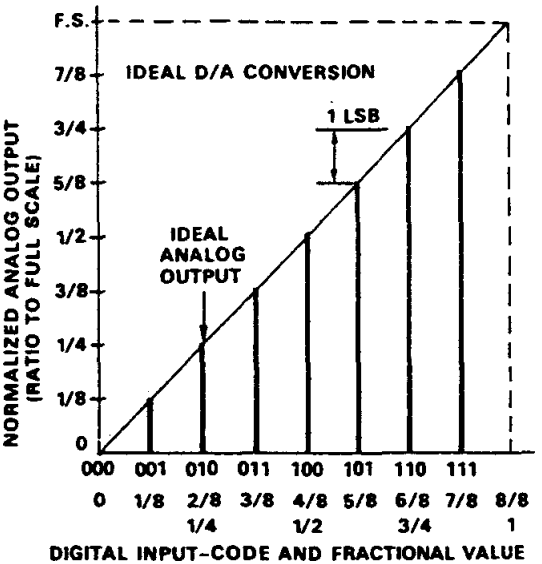


Figure 8.14 Conversion relationship for an ideal 3-bit straight binary D/A converter.

Table 8.1 Output vs. binary coding for a +10 V FS 8-bit unipolar DAC.

Binary	Scale	V_{out}
0000 0000	0	0.00
0000 0001	+1 lsb	+0.04
0010 0000	+1/8 FS	+1.25
0100 0000	+1/4 FS	+2.50
1000 0000	+1/2 FS	+5.0
1100 0000	+3/4 FS	+7.5
1111 1111	+FS – 1 lsb	+9.96

maximum output is 8.75 V. Table 8.1 gives some output voltages for an 8-bit, 10 V FS DAC as a function of its binary input.

The size of the change at the output of DAC for a one least significant bit (1 lsb) change in its input code is called a step and equals $FS/2^n$. For a 3-bit, 0 to 10 V DAC, the step size is $10/2^3$ or 1.25 V.

IC DACs with 8- to 16-bit inputs are common. The determination of the required size (number of bits) for a DAC is a function of the application. DACs with a larger number of bits have a correspondingly reduced step size. A DAC with an 8-bit straight binary input code has 256 distinct output values, one for each possible input code. The step size for an 8-bit 10 V FS DAC is 39.06 mV.

Table 8.2 shows the resolution of a DAC with a straight binary input as a function of the number of input bits. ***Resolution*** is the measure of the output

Table 8.2 Number of output values and resolution as a function of the number of bits for a binary DAC.

		Resolution			
Bits	Output values	Percentage	PPM Part per million)	dB	Millivolts (for 10V FS)
1	2	50	500,000	−6	5000 mV
6	64	1.6	15,625	−36	156 mV
8	256	0.4	3,906	−48	39.1 mV
10	1,024	0.1	977	−60	9.77 mV
12	4,906	0.024	244	−72	2.44 mV
16	65,536	0.0015	15	−96	0.135 mV
20	1,048,576	0.0001	1	−120	0.00954 mV
24	16,777,216	0.000006	0.06	−144	0.000596 mV

step size relative to FS and equals $1/2^n$. Resolution is also a measure of a DAC's precision and is expressed in bits or as a percentage. An 8-bit binary DAC has a resolution of 1 part in 256 or 0.3906 percent or 3906 ppm (parts per million).

Consider an application where a system must generate an analogue control voltage that ranges from 0 to +5 V. This control voltage is used to set a high-voltage power supply that has an output voltage varies within the range of 0 to 10,000 V. If it is necessary to cause changes as small as 1 V in the output of high-voltage power supply, a DAC with 14 or more bits is required. A 14-bit DAC provides a resolution of 1 part in 2^{14} or 1 part in 16,384. With this resolution you can cause the output of the high-voltage power supply to change 0.6 V (10,000/16,384) for a 1 lsb change at the input of the DAC. A 13-bit DAC has a resolution of only 1 part in 8,192. A 1 lsb input change for a 13-bit DAC would change the output of the high-voltage power supply by 1.2 V.

8.3.2 DAC Implementation Techniques

Typically, basic DAC circuits are either networks of electronic switches and resistors or networks of electronic switches and capacitors. A very simple 2-bit resistor divider DAC circuit is shown in Figure 8.15. This circuit uses a resistor divider to divide down a reference voltage into all of the possible discrete analogue outputs needed for a 2-bit DAC. The equal value resistors in the divider produce the desired voltages at the divider taps as long as no current is drawn from the divider.

The 2-bit input is decoded into four mutually exclusive outputs by a two-to-four decoder. The switch corresponding to the asserted output is closed, and the analogue voltage from the selected divider tap is buffered and then

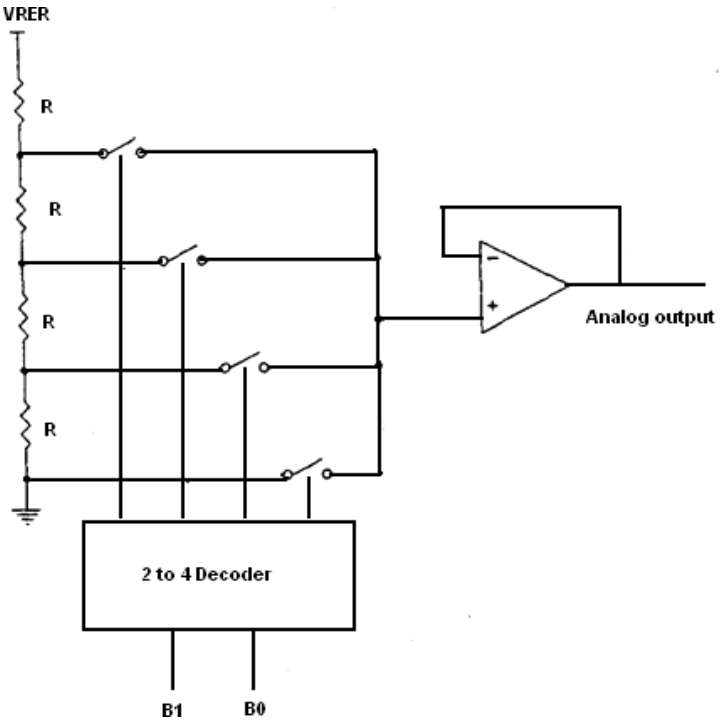


Figure 8.15 Resistor divider 2-bit DAC.

output. The unity gain op-amp buffer prevents a load connected to the DAC from drawing current from the resistor divider.

The main drawback with this circuit is that the number of components needed increases exponentially with the number of bits. A n-bit resistor divider circuit requires 2^n resistors, 2^n switches, and an n-to-2^n decoder. Thus, this circuit is impractical for values of n above 8.

Two versions of resistor DAC circuits where the number of components required increases linearly with n are presented in the following subsection. These circuits are both R-2R ladder implementations.

8.3.2.1 R-2R Ladder DAC

An R-2R ladder DAC circuit is shown in Figure 8.16. This circuit has the advantage of only requiring resistors with the precise ratios of R and 2R. The circuit in Figure 8.16 is a current steering mode R-2R circuit. This circuit creates a current I_s at the summing junction of the op-amp that is proportional

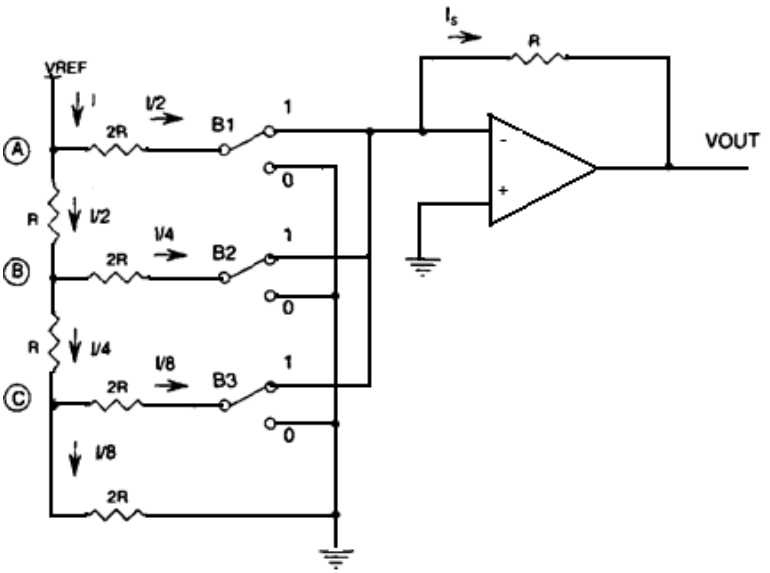


Figure 8.16 Current steering R-2R ladder network voltage output DAC.

to its input code. The op-amp converts this current to an output voltage with:

$$V_{out} = -I_s R$$

where I_s is the sum of the currents through those switches that are in their logic 1 positions.

The input to the circuit is a 3-bit word, $B_1B_2B_3$, which represents a straight binary fraction. B_1, the MSB (most significant bit), has a weight of 2^{-1}, B_2 has a weight of 2^{-2}, and B_3 the LSB (least significant bit), has a weight of 2^{-3}. Each bit controls an analogue switch. When B_i is 1, its analogue switch passes a current through the associated horizontal 2R resistor and into the op-amp's summing junction. The summing junction of the op-amp is held at ground potential by negative feedback through feedback resistor R. The current into the summing junction due to B_i being a 1 is:

$$I_i = (V_{REF}/R)2^{-i}$$

When B_i is 0, the analogue switch directs the current through its associated horizontal resistor to circuit ground, instead of into the op-amp's summing junction.

Due to the virtual ground effect of the op-amp in this negative feedback configuration, the inverting (-) terminal of the op-amp is effectively at ground

potential. Thus, the right side of each 2R resistor is connected to ground through its associated switch, regardless of whether the switch is in its 0 or 1 position.

The Thevenin equivalent of the resistor, ladder looking into the terminal connected to V_{RFF}, is simply R. Thus, the current into the circuit from the reference supply is always:

$$I = V_{REF}/R$$

The equivalent resistance of the ladder network below node A is 2R. Thus, the current I into node A splits in half, resulting in a current I/2, through the horizontal 2R resistor associated with B_1. The other half of the current is through the vertical resistor R and into node B. The equivalent resistance below node B is also 2R. Thus, the current 1/2 into node B splits, causing a current I/4 in the horizontal resistor associated with bit B_2. A continuation of this analysis shows that the current splits at each subsequent node, resulting in currents through the horizontal resistors that are weighted by powers of two. For each switch in its 1 position, the current through its associated horizontal resistor is directed into the summing junction of op amp. Thus, I_s is the sum of these weighted currents.

$$I_s = I.(B_1 2^{-1} + B_2 2^{-2} + B_3 2^{-3})$$

The op-amp current-to-voltage converter converts I_s into an output voltage:

$$\begin{aligned} V_{OUT} &= -I_s R \\ &= -I(B_1 2^{-1} + B_2 2^{-2} + B_3 2^{-3})R \\ &= -V_{REF}(B_1 2^{-1} + B_2 2^{-2} + B_3 2^{-3}) \end{aligned}$$

This output voltage is directly proportional to the input code. The factor of proportionality is equal to V_{REF}. If V_{REF} is -10 V, the DAC's output ranges from 0 V to +8.75 V (seven eighths of full scale), corresponding to input codes ranging from 000 to 111, respectively.

A **voltage switching mode** R-2R circuit can also be constructed (Figure 8.17). Here, the reference terminal and the output terminal are interchanged compared to the current steering mode. The effective resistance to ground of all resistors below a given node is 2R. If B_1 alone is 1, the voltage at node A is 1/2 V_{REF}. If B_2 alone is 1, the voltage at node A is 1/4 V_{REF}. The contribution of each bit is a voltage at node A that is 1/2 the voltage contributed by the preceding bit. By superposition, the voltage at node A, and correspondingly at the output of the unity gain op-amp is:

$$V_{OUT} = V_{REF}(B_1 2^{-1} + B_2 2^{-2} + B_3 2^{-3})$$

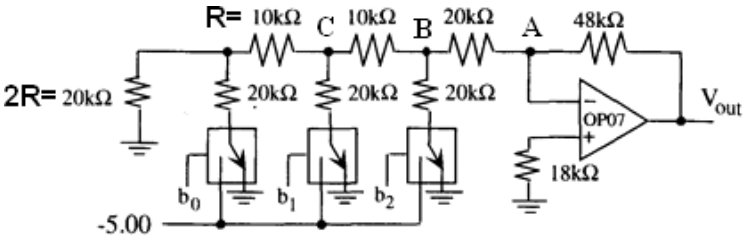


Figure 8.17 Voltage switching R-2R ladder network voltage output DAC.

The output of a voltage switching mode DAC has the same polarity as its reference voltage. In contrast, the polarity of a current steering DAC's output is the negative of its reference voltage.

8.3.3 DAC to System Bus Interface

In principle, any DAC can be interfaced to a system bus. However, microprocessor/microcontroller compatible DACs include, on the DAC IC, the necessary registers and control logic to make interfacing easy. The digital input to a DAC may be serial or parallel. In either case, part of the interface logic is the register that holds the code that is the input to the DAC circuit.

8.3.3.1 Case of Parallel Input DAC

The interface logic on a microprocessor compatible, parallel input DAC with 8, or fewer, bits is simply a single register. The inputs of the register are connected to the data bus, and its outputs are the inputs to the DAC circuit. The register appears to the microprocessor as a byte output port. All that is required is an output device select pulse to write the register.

Interfacing a DAC with 9 or more bits to an 8-bit data bus requires at least two registers. For example, for a 12-bit DAC designed to be interfaced to an 8-bit data bus, one register holds the least significant 8 bits and the other the most significant 4 bits. However, these two registers alone still do not provide an adequate interface.

In many applications, it is important that the analogue output change monotonically from one output value to the next. The two registers of the DAC are loaded by two Write I/O bus cycles. These Write I/O bus cycles result from the execution of either two separate instructions that write byte ports or a single instruction that writes a word port. After the first register is written, and until the second register is written, the data in one of the registers is from the new 12-bit word and the data in the other register is from the

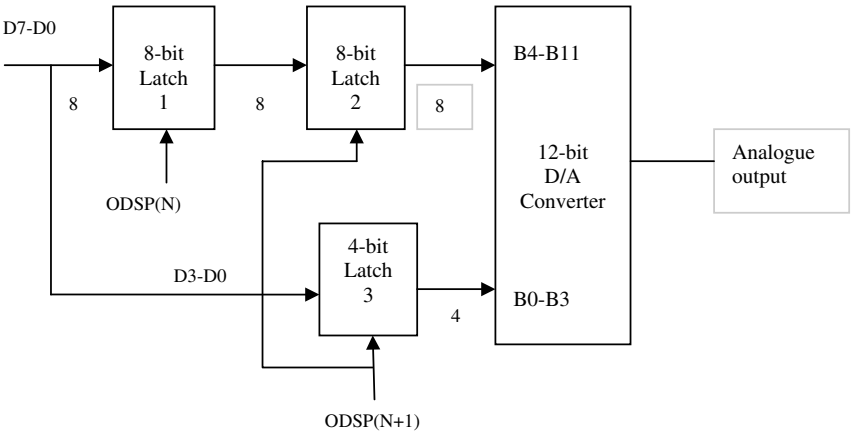


Figure 8.18 Double buffering the inputs to a DAC.

old 12-bit word. This intermediate input to the DAC circuit may produce a significant glitch.

Double buffering is used to solve this problem (Figure 8.18). The eight least significant bits of a 12-bit word are first output to latch 1. The 4 most significant bits are then output to latch 3. The output device select pulse that clocks the 4 bits from the data bus into latch 3, simultaneously clocks the output of latch 1 into latch 2. Thus, all 12 bits appear at the input to the DAC circuit nearly simultaneously.

Even with double buffering, glitches from data loading can exist. These glitches result from the timing relationship between /WR and valid data during a write or output bus cycle and the fact that most DACs use level-triggered latches. Some microprocessors/ microcontrollers assert /WR at the same time that it starts to drive valid data onto the data bus. If the DAC's registers are level-triggered, they will be transparent as soon as /WR goes low. The DAC will initially respond to invalid data. During this time a glitch can be generated.

8.3.3.2 DAC Selection

Many manufacturers, like Analog Devices, Burr Brown, Motorola, Sipex, and Maxim, produce DACs. These DACs have a wide range of performance parameters and come in many configurations. In the following we are going to discuss the various issues to consider when selecting a DAC.

Precision/range/resolution. These three parameters affect the quality of the signal that can be generated by the system. The more bits in the DAC,

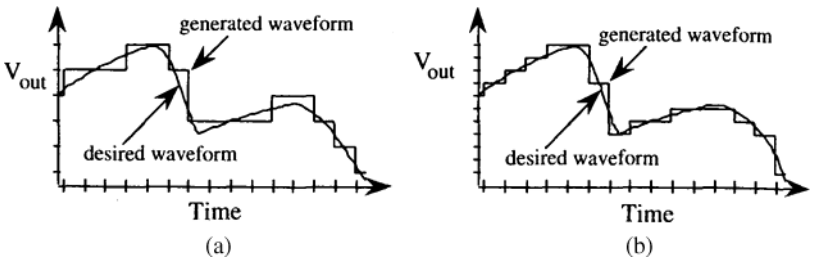


Figure 8.19 The waveform at (b) is created by a DAC with one bit more than that at (a).

the finer the control the system has over the waveform it creates. To establish a priori specification for this important parameter represents a real difficulty for the designer. One effective way that helps in such a case is to design a prototype system with a very high precision (e.g. 12, 14, or 16 bits). The prototype is simulated either by using standard programmes (usually given by manufacturers) or by tailored software. The goal of the simulation is to give as an output the waveform corresponding to each DAC precision. The software must be flexible such that the designer can modify it for the available precision. Figure 8.19 gives an example of the expected output we can get out of the simulation. From the simulation output waveforms, and according to specifications given to the designer, he can decide the needed precision.

***Channels*:** Even though multiple channels could be implemented using multiple DAC chips, it is usually more efficient to design a multiple-channel system using a multiple-channel DAC. Some advantages of using a DAC with more channels than originally conceived are future expansion, automated calibration, and automated testing.

***Configuration*:** DACs can have voltage or current outputs. Current-output DACs can be used in a wide spectrum of applications (e.g., adding gain and filtering) but do require external components. DACs can have internal or external references. An internal-reference DAC is easier to use for standard digital-input/analogue-output applications, but the external- reference DAC can be used often in variable-gain applications (multiplying DAC). We note also that some DACs can sometimes generate a unipolar output, while other times the DAC produces bipolar outputs.

***Speed*:** There are a couple of parameters manufacturers use to specify the dynamic behavior of the DAC. The most common is *settling time* and sometimes the *maximum output rate*. When operating the DAC in variable-gain mode, we are also interested in the "gain/Band Width" product of the analogue amplifier. When comparing specifications reported by different manufacturers,

it is important to consider the exact situation used to collect the parameter. In other words, one manufacturer may define settling time as the time to reach 0.1% of the final output after a full-scale change in input given a certain load on the output, while another manufacturer may define settling time as the time to reach 1% of the final output after a 1 V change in input under a different load. The speed of the DAC together with the speed of the computer/software will determine the effective frequency components in the generated waveforms. Both the software (rate at which the software outputs new values to the DAC) and the DAC speed must be fast enough for the given application. In other words, if the software outputs new values to the DAC at a rate faster than the DAC can respond, then errors will occur. Figure 8.20 illustrates the effect of DAC output rate on the quality of generated waveform. According to Nyquist theorem, to prevent such error, the digital data rate must be greater than twice the maximum frequency component of the desired analogue waveform.

***Power*:** There are three power issues to consider. The first consideration is the type of power required. Some devices require three power voltages (e.g., +5, +12, and −12 V), while many of the newer devices will operate on a single voltage supply (e.g., +2.7, +3.3. +5, or +12 V). If a single supply can be used to power all the digital and analogue components, then the overall system costs will be reduced. The second consideration is the amount of power required, i.e. the power consumption. Some devices can operate on less than 1 mW and are appropriate for battery-operated systems or for systems where excess heat is a problem. The last consideration is the need for a low-power sleep mode. Some battery-operated systems need the DAC only intermittently. In these applications, we wish to give a shutdown command to the DAC so that it won't draw much current when the DAC is not needed.

***Interface*:** Three approaches exist for interfacing the DAC to the computer (Figure 8.21); use of digital logic, μ P-bus, and use of SPI/DCA. In a digital

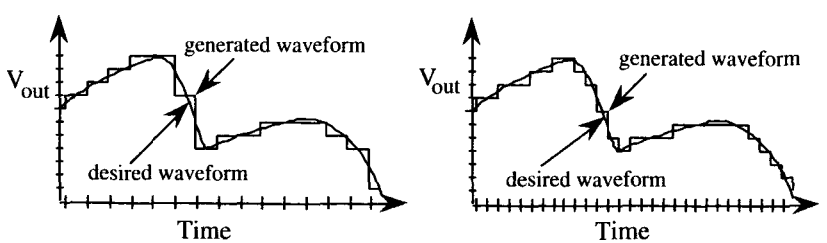


Figure 8.20 The waveform at (b) was generated by a system with twice the output rate than in (a).

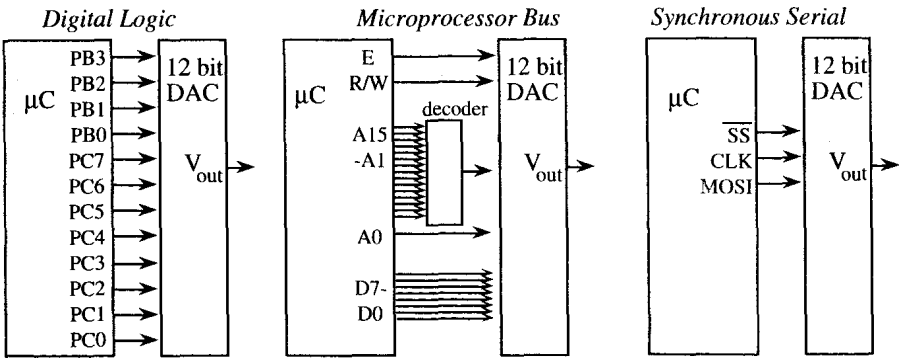


Figure 8.21 Three approaches to interfacing a 12-bit DAC to the microcomputer.

logic interface, the individual data bits are connected to a dedicated computer output port. For example, a 12-bit DAC requires 12-bit output port bits to interface. The software simply writes to the parallel port(s) to change the DAC output. The second approach is called μ P-bus or microprocessor-compatible. These devices are intended to be interfaced onto the address/data bus of an expanded-mode microcomputer. Some processors, e.g. MC68HC8 12A4, have built-in address decoders enabling most μ P-bus DACs to be interfaced to the address/data bus without additional external logic. The third approach is to use SPI (Serial Peripheral Interface) to interface DAC. The SPI/DAC8043 interface is an example. This approach requires the fewest number of I/O pins. Even if the microcomputer does not support the SPI directly, these devices can be interfaced to regular I/O pins via the bit-banging software approach.

Package: The standard DIP is convenient for creating and testing an original prototype. On the other hand, surface-mount packages like the SO and μ Max require much less board space. Because surface-mount packages do not require holes in the printed circuit board, circuits with these devices are easier/cheaper to manufacture (Figure 8.22).

Cost: Cost is always a factor in engineering design. Beside the direct costs of the individual components in the DAC interface, other considerations that affect cost include (1) power supply requirements, (2) manufacturing costs, (3) the labour involved in individual calibration if required, and (4) software development costs.

Example of Practical DAC: The DAC0832 integrated circuit

The DAC0832 is a CMOS 8-bit DAC designed to interface directly with several popular microprocessors. A resistor ladder network divides the

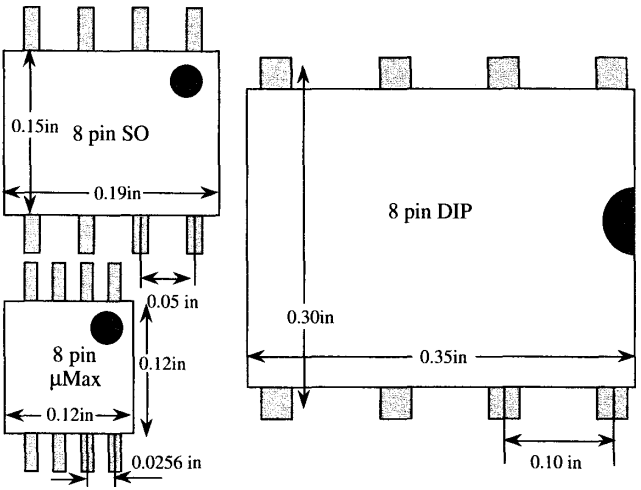


Figure 8.22 IC comes in a variety of packages.

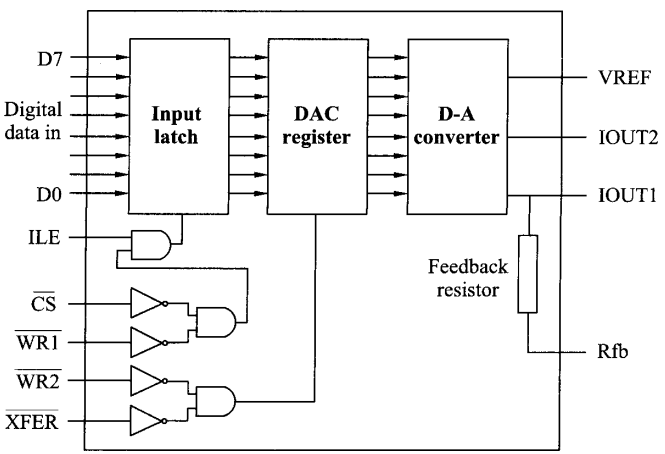


Figure 8.23 Block diagram of the DAC0832 structure.

reference current and provides the circuit with good temperature tracking characteristics. The circuit uses CMOS current switches and control logic to achieve low power consumption and low output leakage current errors. The use of an input latch and a DAC register allows the device to output a voltage corresponding to one digital number while holding the next digital number. This permits the simultaneous updating of any number of these devices. Figure 8.23 shows the block diagram of the device.

In order to use the DAC0832, an external operational amplifier must be connected. This functions in conjunction with a feedback resistor inside the chip to give an analogue output.

In Figure 8.23, the connections to the DAC0832 are as follows:

D0-D7 = Digital data input
/CS = Chip Select
/WR1 = Write Enable 1
/WR2 = Write Enable 2
/XFER = Pin to start transfer of data
ILE = Input Latch Enable
VREF = Reference voltage for the conversion
lOUT 1 = Current output 1 for op-amp connection
IOUT2 = Current output 2 for op-amp connection
Rfb = Connection to internal feedback resistor

8.4 Analogue-to-Digital Conversion (ADC)

In this section we are presenting some fundamental analogue-to-digital conversion concepts. These concepts are common for all possible ADC architectures. To convert an analogue signal to a digital code, ADCs perform two basic operations, quantization and coding. Quantization is the mapping of the analogue signal into one of several possible discrete ranges, or quanta. Coding is the assignment of a digital code to each quantum. The same codes used for inputs to DACs are also used in coding outputs from ADCs.

Figure 8.24 is the transfer characteristic of an ideal 3-bit binary unipolar ADC. An n-bit binary ADC has 2^n distinct output codes. Thus, the 3-bit converter has eight output codes represented on the vertical axis. On the horizontal axis, the analogue input range is divided into quanta by transition points or decision levels. Quantization size is $Q = FS/2^n$, where FS is the nominal full scale input voltage. The quantization size represents the maximum input voltage change required to cause a change in the output of one least significant bit. Equivalently, a quantum change is the smallest change in the input signal that the ADC is guaranteed to detect. This defines the ADC resolution; the *resolution* is the change in input that causes the digital output change by 1.

In an ideal ADC, the first transition point occurs at Q/2, and each of the remaining transition points is spaced Q volts apart. The midpoint of each quantum is the voltage that is exactly (without error) represented by the output code. For example, transition points at 1/16 FS and 3/16 FS bracket the 1/8 FS

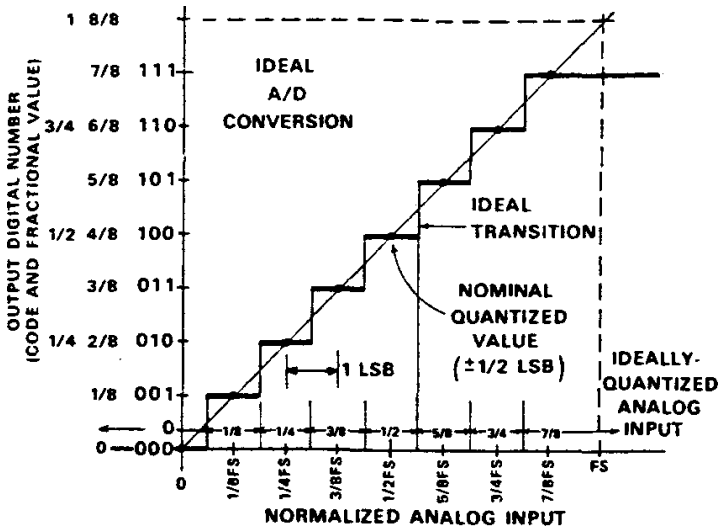


Figure 8.24 Conversion relationships for an ideal 3-bit ADC.

point. Any analogue input in the quantum from 1/16 FS to 3/16 FS is assigned the output code representing 1/8 FS (001_b). Thus, quantization produces an inherent quantization error of $\pm$ Q/2. Accordingly an output, M, from an ADC indicates that the analogue input has a value of M $\pm$ Q/2 (M $\pm$ FS/2^{n+1}).

Since quantization error is inherent in the conversion process, the only way quantization error can be reduced is by selecting an ADC with a larger number of bits. In a practical converter, the transition points are not perfectly placed and nonlinearity, offset, and gain errors result. These errors are in addition to the inherent quantization error.

Errors in ADCs are defined and measured in terms of the location of the actual transition points in relation to their ideal locations (Figure 8.25). If the first transition does not occur at exactly +1/2 LSB (+1/2 Q), there is an ***offset*** error. If the difference between the points at which the last transition and first transition occur is not equal to FS -2 LSB, there is a ***gain error***. ***Linearity error*** exists if the differences between transition points are not all equal, in which case the midpoints of some decision quanta do not lie on a straight line between 0 and FS. The midpoints between transition points should be 1 LSB apart. **Differential nonlinearity** is the deviation between the actual difference between midpoints and 1 LSB, for adjacent pairs of codes. If the differential nonlinearity is equal to or more negative than -1 LSB, then one or more codes will be missing.

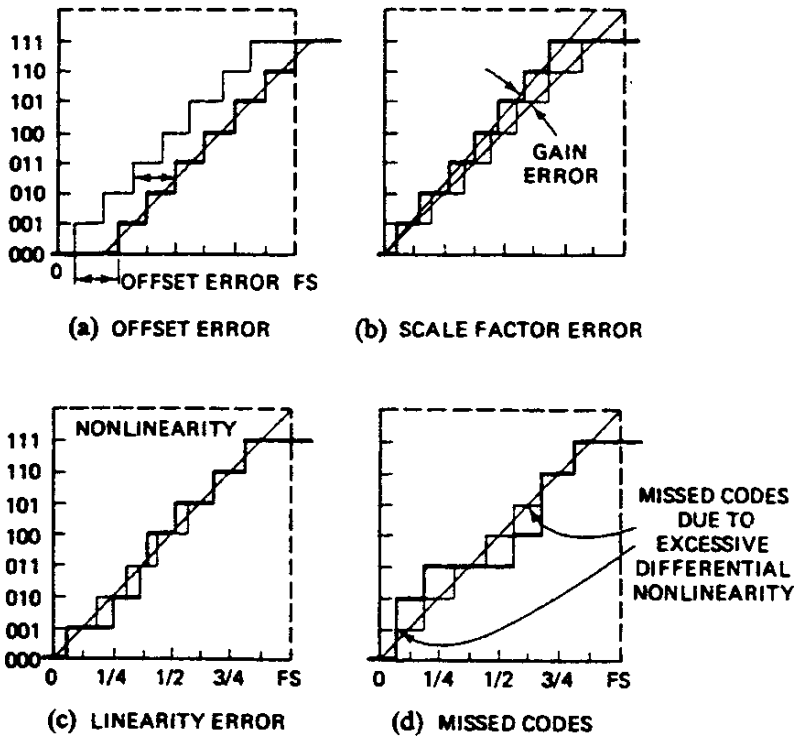


Figure 8.25 Typical sources of error for a 3-bit A/D converter.

ADCs differ in how the operations of quantization and coding are accomplished. Several types of ADCs and their associated conversion techniques are presented in the next two sections.

8.4.1 Conversion Techniques: Direct Conversion Techniques

To construct ADCs of n bits, two broad categories of techniques are used, **direct conversion** and **indirect conversion**. With ***direct conversion***, the analogue voltage to be converted is compared to the output of a reference DAC. The output of the reference DAC is changed as a result of each comparison, until it is eventually equal to the analogue input voltage. The input of the DAC is then equal to the desired output code. The various direct conversion techniques differ in the way the inputs to the DAC are sequenced to obtain a value equal to the unknown analogue voltage. For ADCs with binary outputs,

the desired binary code has been obtained when

$$\left| V_I - V_{FS} \sum_i B_i 2^{-i} \right| < (1/2)LSB$$

where V_I is the analogue voltage to be converted.

With ***indirect conversion***, the analogue input voltage is transformed to the time or the frequency domain. Digital logic converts the time interval or the frequency into a digital output code. Indirect conversion techniques are slower than direct conversion techniques.

Four direct conversion techniques are known in literature: flash, half-flash, counting, and successive approximation. Flash, counting and successive approximation converters are presented next.

8.4.1.1 Flash converters

The fastest and conceptually simplest ADC is a flash (parallel or simultaneous) converter. This converter directly implements the quantization operation using a voltage divider consisting of 2^n resistors and $2^n - 1$ comparators. The coding operation is implemented by a digital encoder that translates the digital outputs of the $2^n - 1$ comparators into an n-bit code. Figure 8.26 represents the general block diagram of the flash ADC and Figure 8.27 is practical example of a two-bit flash ADC and Table 8.3 shows the output results.

The resistors in the voltage divider divide the reference voltage into $2^n - 1$ voltages corresponding to the $2^n - 1$ transition points needed for quantizing an analogue input. The inverting input of each comparator is connected to one of the transition point reference voltages. The resistor voltage divider provides a threshold for each comparator that is one LSB higher that the threshold for the comparator immediately below. The unknown analogue voltage is connected to the noninverting input of each comparator. All comparators with transition point threshold voltages below that of the unknown voltage have outputs of 1. All comparators with transition point threshold voltages above the unknown voltage have outputs of 0.

The output from the comparators is a thermometer code. A thermometer code is a code consisting of a column with j consecutive 1s at the bottom and k consecutive 0s at the top, where $j + k$ is a constant.

The comparator's thermometer code outputs are latched. The latched outputs provide the input to a digital encoder. The digital encoder has $2^n - 1$ inputs, but only 2^n possible output codes. This combinational circuit can be designed to produce any output code format desired.

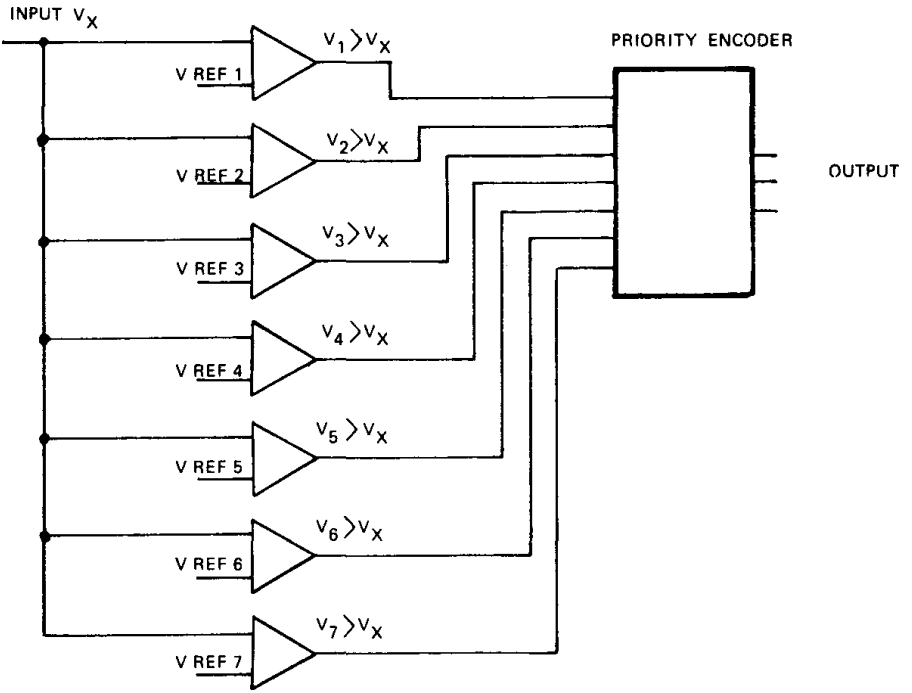


Figure 8.26 Flash ADC.

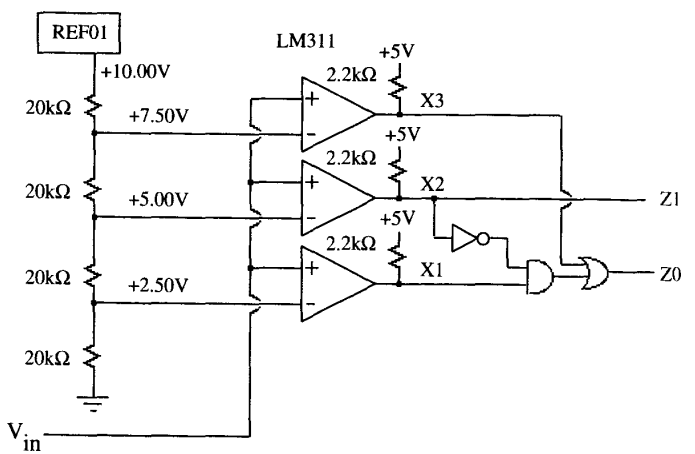


Figure 8.27 Two-bit flash ADC.

Table 8.3 Outputs of two-bit flash ADC.

Vin	X3	X2	X1	Z1	Z0
$2.5 > V_{in}$	0	0	0	0	0
$5.0 > V_{in} \geq 2.5$	0	0	1	0	1
$7.5 > V_{in} \geq 5.0$	0	1	1	1	0
$V_{in} \geq 7.5$	1	1	1	1	1

Flash ADCs are the fastest converters available. Conversion time is the sum of the delay through the comparator stage and the encoders. Because of the flash converter's conversion speed, a track-hold is usually not needed.

The flash ADC's drawback is the number of comparators required. An 8-bit converter requires 255 comparators! Practical IC flash converters range in size from 4 to 10 bits.

It is possible to build a signed flash ADC. There are two approaches to build such produce a signed flash ADC. The direct approach is to place the equal resistors from a $+10.00$ V reference to a -10.00 V reference (instead of $+10$ V and ground shown in Figure 8.27). The middle of the series resistors ("zero" reference) should be grounded. The digital circuit would then be modified to produce the desired digital code.

The other method would be to add analogue preprocessing to convert the signed input to an unsigned range. This unsigned voltage is then converted with an unsigned ADC. The digital code for this approach would be offset binary. This approach will work to convert any unsigned ADC into one that operates on bipolar inputs.

8.4.1.2 Counting (Ramp) converters

A simple direct conversion technique is that of the counting or ramp converter. The ramp ADC starts at the minimum voltage and counts up until the DAC output V_o just exceeds the input voltage V_{in}. Hardware for a counting converter that has its conversion algorithm implemented in software is shown in Figure 8.28. To produce the ideal transfer characteristics shown in Figure 8.13, the output of the DAC must be offset to that when the DAC is loaded with all zeros, its output voltage corresponds to $+1/2$ least significant bit.

The conversion process consists of repeatedly outputting incrementally higher binary values to the DAC until the comparator changes state. Starting from a value of 0, each subsequent value output is 1 LSB greater than the previous value. The resulting output from the DAC is a piecewise continuous ramp.

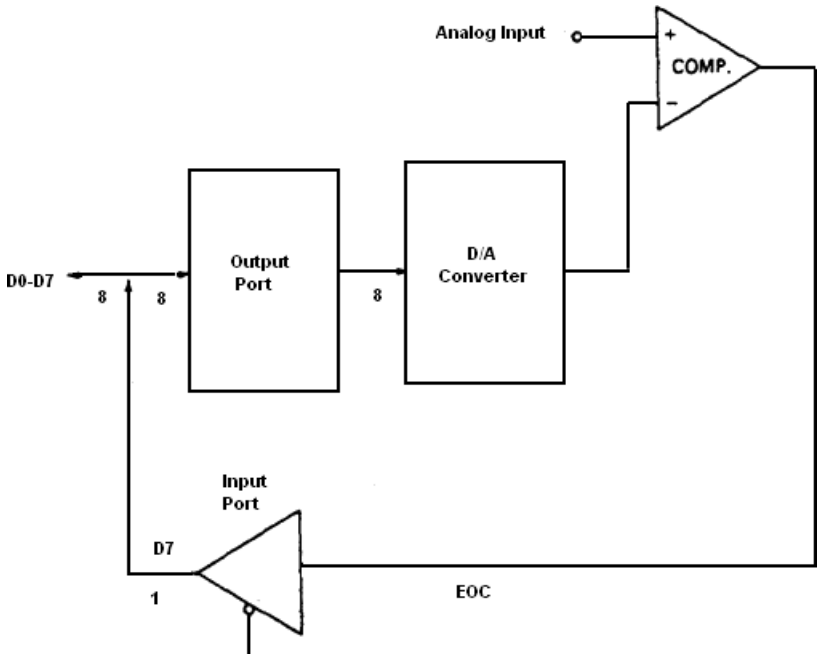


Figure 8.28 Basic hardware structure for software ADC conversion routines.

8.4.1.3 Successive approximation converters

The most popular direct conversion method is successive approximation. This method has the advantage of a fixed conversion time proportional to the number of bits, n, in the code and independent of the analogue input value. If high speed A/D conversion is necessary this approach can be very effective. The successive approximation algorithm consists of successive comparisons of the unknown voltage to a voltage generated by a DAC (the approximation). The simple block diagram shown in Figure 8.29 illustrates the essence of the approach.

To make a measurement, the switch is placed in the sample position for some time, t_0, to allow the capacitor to charge to the unknown voltage. In the ideal case, t_0 will approach 0. The switch is then opened (placed in the hold position), and the reference voltage is adjusted until the output of the comparator is 0. That is, until the two input voltages are equal.

The successive approximations technique is analogous to the "*logarithmic search*" technique used in programming a search on an interval. The concept is to jump to the middle of the interval, then jump to the middle of either the bottom or the top remaining halves, depending on the value of the comparison.

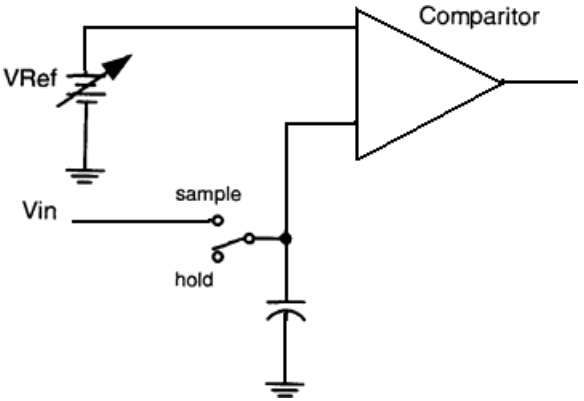


Figure 8.29 Schematic diagram for a basic Successive approximation analogue-to-digital converter.

This guarantees that the final value will be reached in $log_2 n$ operations, where n is the number of elements. The logarithmic search is used in programming to retrieve an element within a file. A simpler comparison consists in opening a book in the middle, then jumping to the middle of the first section or else the middle of the next section depending on whether the item one is looking for was before or after the middle of the book, and so on. This is also called *"binary search"*. The intent is to reduce the amount of time required for the search process. The successive approximation algorithm is, in effect, a binary search; Starting with the most significant bit, each comparison determines one bit of the final result and the approximation to be used for the next comparison.

A successive approximation converter uses a DAC to generate the voltage that is compared to the unknown voltage. A software implementation of a successive approximation converter uses the same hardware used for the counting converter (Figure 8.28). However, for a successive approximation converter, the offset required when the DAC is loaded with all 0s must be $-1/2$ LSB, rather than $+1/2$ LSB.

The steps in the successive approximation algorithm are as follows:

1. An initial approximation code is used to determine the most significant bit of the result. This code has a 1 as its most significant bit, and all its other bits are 0s.
2. The approximation code is output to the DAC
3. The comparator output is read. If it is 0, the bit being determined in the approximation is changed to a 0. If not, it is left as a 1.

4. If all bits have not been determined, the next most significant bit is tested. To test the next most significant bit, the next bit in the approximation code that resulted from step 3 is made a 1, and then step 2 is repeated. When all bits have been determined, the conversion process is complete.

The effect of this algorithm is to first test whether the analogue input voltage is greater or less than 1/2 FS. If the analogue input is greater than 1/2 FS, having a 1 as the most significant bit of the approximation code output to the DAC for the first comparison makes the comparator output a 1. When this is the case, the most significant bit is left as a 1. The next most significant bit in the approximation code is then set to 1, and the analogue input is again tested (second comparison) to see whether it is greater than 3/4 FS.

In contrast, if during the first comparison the analogue input is less than 1/2 FS, the comparator's output is a 0, and the most significant bit is changed to a 0. The next most significant bit is then set to 1, and a test is made to determine whether the unknown voltage is greater than 1/4 FS. This process is repeated in succession for each bit until the last bit is tested.

8.4.2 Conversion Techniques: Indirect Conversion

One of the indirect conversion techniques for implementing an A/D conversion is based on a voltage-controlled oscillator (VCO). A VCO is a circuit that produces a frequency proportional to an input voltage. If the VCO is designed to produce a frequency of 50 KHz in response to a 5.0-VDC signal, gating the frequency to a counter for 1 second will yield a value of 50,000 counts. Appropriate placement of the decimal point gives the digital equivalent of the analogue voltage.

A high-level design for an A/D based on such a concept is given in Figure 8.30.

A measurement begins when the microprocessor loads a value into the Register and issues a Start command. Following the Start command, the state machine issues the following commands:

1. A *Load* command to transfer the contents of the Register into the Timer
2. An *Enable* to the gate controlling the Clk to the *Timer*
3. A *Convert* command to connect the unknown voltage to the input of the VCO

After one second, the timer issues an Interval End to the State Machine, which responds by disconnecting the unknown signal from the VCO and issues

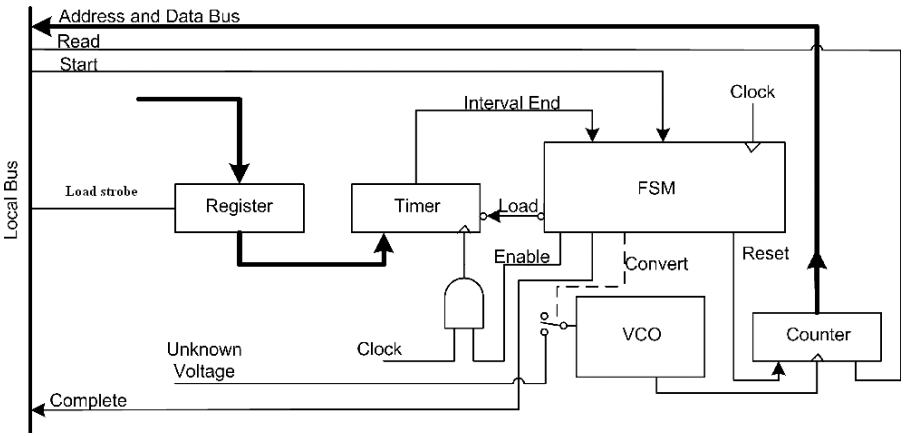


Figure 8.30 Block diagram for a voltage-controlled oscillator-based A/D converter.

a *Complete* event to the microprocessor. The microprocessor responds by asserting the Read signal to capture the state of the *Counter*.

8.4.3 Summing up ADC: Data Acquisition System Design

In Figure 8.5 we introduced a general block diagram for data acquisition system (DAS). In the following we are discussing how to design the various components in the block diagram using the information introduced in this section.

1. ***How to Determine the Sampling Rate: Using Nyquist Theory***

There are two types of errors introduced by the sampling process: errors due to voltage quantizing and errors due time quantizing. *Voltage quantizing* is caused by the finite word size of the ADC. This type of error can be controlled by using the proper *precision.* The *precision* is determined by the number of bits in the ADC.

Time quantizing is caused by the finite discrete sampling interval. The errors that arise due to sampling can be avoided by using the correct sampling rate f_s. *Nyquist theory* defines that rate. Assume that we want to sample a sine wave

$$V(t) = A\sin(2\pi f t + \Phi)$$

using a sampling rate f_s, *Nyquist theory* says that if f_s is strictly greater than twice f, then one can determine A, f, and Φ from the digital samples. In other words, the entire analogue signal can be reconstructed from the digital

samples. But if f_s less than or equal to twice f, then one cannot determine A, f, and Φ. For example, if the frequency of an input sine wave is 1000 Hz we must sample with sampling rate $f_s > 2000$ samples per second.

In general, the choice of sampling rate f_s is determined by the maximum useful frequency f_{max} contained in the signal. One must sample at least twice this maximum useful frequency. Faster sampling rates may be required to implement various digital filters and digital signal processing.

$$f_s > 2f_{max}.$$

We must note here that, even though the largest signal frequency of interest is f_{max}, there may be significant signal magnitudes at frequencies above f_{max}. These signals may arise from the input signal x, from added noise in the transducer, or from added noise in the analogue processing. Once the sampling rate is chosen at f_s, then, as discussed before, a low-pass analogue filter maybe required to remove frequency components above $0.5f_s$. It is worth to mention here also that a digital filter cannot be used to remove aliasing.

2. ***How Many Bits Does One Need for the ADC?***

The choice of the ADC precision is a compromise of various factors. The overall objective of the DAS will dictate the potential number of useful bits in the signal. If the transducer is nonlinear, then the ADC precision must be larger than the precision specified in the problem statement. For example, let y be the transducer output and let x be the real-world signal. Assume for now that the transducer output is connected to the ADC input. Let the range of x be r_x, let the range of y be r_y, and let the required precision of x be n_x. The resolutions of x and y are Δx and Δy, respectively. Let the following expression describes the nonlinear transducer:

$$y = f'(x)$$

The required ADC precision n_y (in alternatives) can be calculated by:

$$\Delta x = \frac{r_x}{n_x}$$

$$\Delta y = \min\{f(x + \Delta x) - f(x)\} \quad \text{for all } x \text{ in } r_x$$

$$n_y = \frac{r_y}{\Delta y}$$

For example, consider the nonlinear transducer $y = x^2$. The range of x is $0 \leq x \leq 1$. Thus, the range of y is also $0 \leq y \leq 1$. Let the desired resolution

be $\Delta x = 0.01$. $n_x = r_x/\Delta x = 100$ or, alternatively, about 7 bits. From the above equation,

$$\Delta y = \min\{(x + 0.01)^2 - x^2\} = \min\{0.02x + 0.0001\} = 0.00001$$

This is almost 15 bits.

3. ***How Fast Must the ADC Be?***

The ADC conversion time must be smaller than the quotient of the sampling interval by the number of multiplexer signals. Let m be the number of multiplexer signals that must be sampled at a rate of f_s, let t_{mux} be the settling time of the multiplexer, and let t_c be the ADC conversion time. Then without a S/H,

$$m.(t_{mux} + t_c) < 1/f_s$$

With a S/H, one must include both the acquisition time, t_{aq} and the aperture time, t_{ap}:

$$m.(t_{mux} + t_{aq} + t_{ap} + t_c) < 1/f_s$$

4. ***Specifications for the S/H***

A S/H is required if the analogue input changes more than one resolution during the conversion time. Let *dz/dt* be the maximum slope of the ADC input voltage, letΔz, be the ADC resolution, and let t_c be the ADC conversion time. A S/H is required if

$$\frac{dz}{dt}.t_c > 0.5\Delta z$$

If the transducer and analogue signal processing are both linear, the determination of whether or not to use a S/H can be calculated from the input signals. A S/H is required if:

$$\frac{dx}{dt}.t_c > 0.5\Delta z$$

5. ***Specifications for analogue Signal Processing***

This depends on the nature of the signal and the noise. The reader is referred to the references at the end of the chapter to get the details of this topic.

8.5 AVR Analogue Peripherals

A microcontroller is able to handle analogue data by first converting the data to digital form. An AVR microcontroller includes both an analogue to

digital conversion peripheral and an analogue comparator peripheral. Each of these analogue interfaces will be covered in this section, along with a brief background on analogue to digital conversion.

Microcontrollers use analogue-to-digital converters to convert analogue quantities such as temperature and voltage (for example, a low-battery monitor), convert audio to digital formats, and perform a host of additional functions.

8.5.1 ADC Peripheral

The ADC peripheral in the AVR microcontrollers is capable of 10-bit resolution and can operate at speeds as high as 15 kSPS (kilo-samples per second). It can read the voltage on one of eight different input pins of the microcontroller, meaning that it is capable of reading from eight different analogue sources.

Two registers control the analogue-to-digital converter: The ADC control and status register (ADCSR) controls the functioning of the ADC and the ADC multiplexer select register (ADMUX) controls which of the eight possible inputs are being measured. Figure 8.31 shows the bit definitions for the ADC control and status register.

The ADC requires a clock frequency in the range of 50 kHz to 200 kHz to operate at maximum resolution. Higher clock frequencies are allowed but at decreased resolution. The ADC clock is derived from the system clock by means of a prescaler in a manner similar to the timers. The least significant three bits of ADCSR control the prescaler division ratio. These bits must be set so that the system clock, when divided by the selected division ratio, provides

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADSP1	ADSP0

Bit	Description
ADNE	ADC Enable bit. Set to enable the ADC
ADSC	ADC Start Conversion bit. Set to start a conversion.
ADFR	ADC Free Running Select bit. Set to enable free run mode.
ADIF	ADC Interrupt Flag bit. Is set by hardware at the end of a conversion cycle.
ADIE	ADC Interrupt Mask bit. Set to allow the interrupt to occur at the end of a conversion
ADPS2	ADC Prescaler select bit
ADPS1	
ADPS0	

Figure 8.31 ADCSR bit definition.

ADSP2	ADSP1	ADSP0	Division factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Figure 8.32 ADC pre-selector division ratios.

an ADC clock between 50 kHz and 200 kHz. The selection bits and division ratios are shown in Figure 8.32.

Although it could be done by trial and error, the most direct method for choosing the ADC pre-selector factor is to divide the system clock by 200 kHz and then choose the next higher division factor. This will ensure an ADC clock that is as fast as possible but under 200 kHz.

The ADC, like the serial UART, is somewhat slower than the processor. If the processor were to wait for each analogue conversion to be complete, it would be wasting valuable time. As a result, the ADC is usually used in an interrupt-driven mode.

Although the discussion that follows uses the more common interrupt-driven mode, it is also possible for the ADC to operate in free-running mode, in which it continuously does conversions as fast as possible. When reading the ADC output in free-running mode, it is necessary to disable the interrupts or stop the free-running conversions, read the result, and then re-enable the interrupts and free-running mode. These steps are necessary to ensure that the data read is accurate, in that the programme will not be reading the data during the time that the processor is updating the ADC result registers.

`The ADC is usually initialized as follows:`

1. Set the three lowest bits of ADCSR for the correct division factor.
2. Set ADIE high to enable interrupt mode.
3. Set ADEN high to enable ADC.
4. Set ADSC to immediately start a conversion.

For a division factor of 8, the following lines of code would initialize the ADC to read the analogue voltage on the ADC2 pin:

```
ADMUX = 2;          //read analogue voltage on ADC2
ADCSR = 0xcb; //ADC on, interrupt mode, /8,
                & started
```

The initialization above sets up the ADC, enables it, and starts the first conversion all at once. This is useful because the first conversion cycle after the ADC is enabled is an extra long cycle to allow for the setup time of the ADC. The long cycle, then, occurs during the balance of the programme initialization, and the ADC interrupt will occur immediately after the global interrupt enabled bit is set. Notice that the ADMU)(register is loaded with the number of the ADC channel to be read.

Figures 8.33 and 8.34 show the hardware and software, respectively, for a limit detector system based on the analogue input voltage to ADC channel 3. Briefly, the system lights the red LED if the input voltage exceeds 3 V, lights the yellow LED if the input voltage is below 2 or lights the green LED if the input voltage is within the range of 2 V to 3 V.

The limit detector programme in Figure 8.34 shows a typical application for the ADC. The ADC is initialized and started in main 0 by setting ADCSR to 0xCE. ADC channel 3 is selected by setting ADMUX to 3. This starts the sequence so the ADC interrupt will occur at the end of the first conversion. Checking the ADC output to see which LED to light and lighting the appropriate LED are all handled in the ADC interrupt ISR.

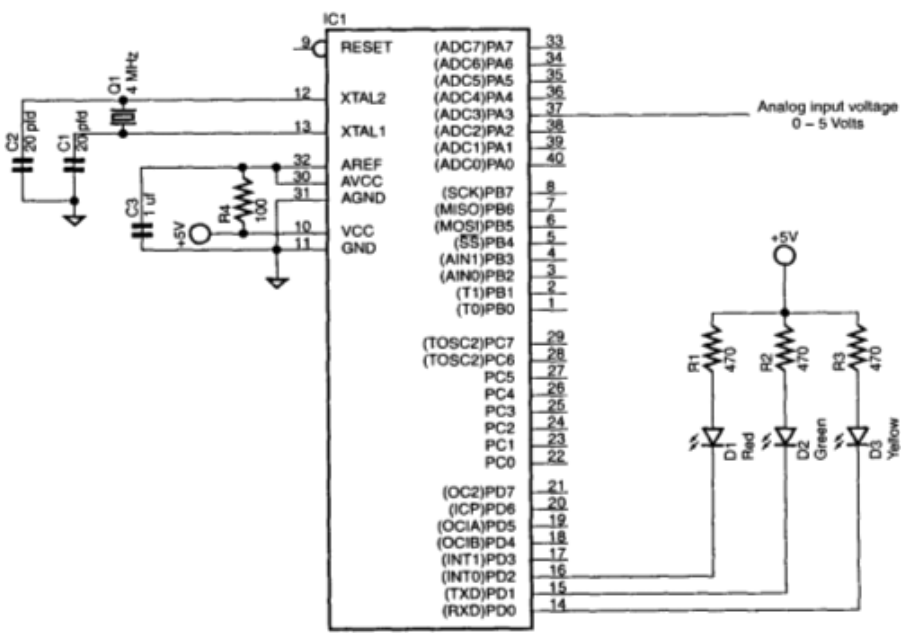


Figure 8.33 ADC example hardware.

```
#include <ATMega8515.h>
//Define output port and light types
#define LEDs PORTD
#define red 0b111
#define green 0b101
#define yellow 0b110

interrupt [ADC_INT] void adc_isr(void)       //ADC ISR
{
        unsigned int adc_data;        //variable for ADC results
        adc_data = ADCW;                      //read all 10 bits into variable
        if (adc_data > (3*1023)/5)
                LEDs = red;                   //too high (> 3V)
        Else if (adc_data < (2*1023)/5)
                LEDs = yellow;        //too low (<2V)
        Else
                LEDs = green;         //must be just right

        ADCSR = ADCSR | 0x40;         //start the next conversion
}
void main(void)
{
        DDRD = 0x07;          //least significant 3 bits for output
        ADMUX = 0x3;          //select to read only channel 3
        ADCSR = 0xCE;         //ADC on, /64, interrupt unmasked, and started
        #asm("sei")                   //global interrupt enable bit
        while (1)
            ;                         // do nothing but wait on ADC interrupt
}
```

Figure 8.34 ADC example software.

Notice that the 10-bit output from the ADC is read by reading the data from ADCW. ADCW is a special register name provided by CodeVisionAVR that allows retrieving the data from the two ADC result registers, ADCL and ADCH, at once. Other compilers would more likely require the programmer to read both ADC registers (in the correct order, even) and combine them in the 10-bit result as a part of the programme.

Also notice that the programmer has used the analogue-to-digital conversion formula in the ISR. The compiler will do the math and create a constant that will actually be used in the programme. You will find that this technique can be used to your advantage in several other places, such as loading the UBRR for the UART.

The ADC peripheral in the AVR microcontrollers varies somewhat according to the specific microcontroller in use. All of the ADCs require some noise suppression on the ADC Vcc connection (see Figure 8.33). Some also have

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ACD	-	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0

Bit	Description
ACD	Analogue Comparator Disable bit. Set to disable the analogue comparator
ACO	Analogue Comparator Output bit
ACI	Analogue Comparator Interrupt flag
ACIE	Analogue Comparator Interrupt mask bit
ACIC	Analogue Comparator Interrupt Capture bit Set enable input capture on comparator change-of-state
ACIS1 ACIS0	Analogue Converter Comparator Mode Select bits (See definition below)

ACIS1	ACIS0	Interrupt Mode
0	0	Comparator interrupt on ACO toggle
0	1	Reserved
1	0	Comparator interrupt on ACO falling edge. (AIN1 becomes greater than AIN0)
1	1	Comparator interrupt on ACO rising edge. (AIN0 becomes greater than AIN1)

Figure 8.35 ACSR bit definition.

a built-in noise canceller function, and some have the ability to control V_{ref} internally. As mentioned in Chapter 3, you will need to check the specification for your particular microcontroller when using the ADC.

8.5.2 Analogue Comparator Peripheral

The analogue comparator peripheral is a device that compares two analogue inputs: AINO, the positive analogue comparator input, and AIN1, the negative analogue comparator input. If AINO > ATN1, then the analogue comparator output bit, ACO, is set. When ACO changes state, either positive-going, negative-going, or both, it will cause an interrupt to occur, provided that the analogue comparator interrupt is unmasked, or else the change of state may be set to cause an input capture to occur on timer/counter 1.

The *analogue comparator control and status register*, ACSR, shown in Figure 8.35, is used to control the analogue comparator functions.

As a simple example of analogue comparator functioning, consider the system shown in Figure 8.36. A battery is used to power the system, so it is important to know when the battery voltage becomes dangerously low.

The system shown in Figure 8.36 is continuously monitoring the battery without expending any processor time whatsoever, because it is all handled

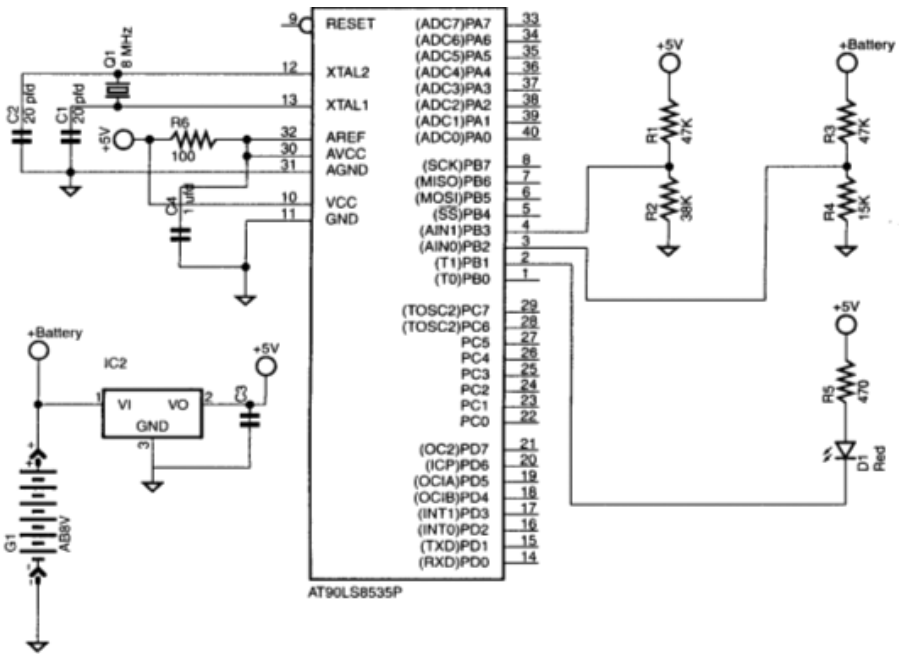


Figure 8.36 Analogue Comparator Example Hardware.

by the analogue comparator. When the battery voltage becomes too low, the LED is lighted.

Caution: the circuit shown has a design flaw. When the battery gets low, the LED is turned on and left on, further draining an already ailing battery. The LED should be pulsed in any actual circuit of this type. However, the circuit does demonstrate the use of the analogue comparator peripheral.

The two analogue converter inputs are connected to voltage dividers: one is powered by the regulated +5 V as a reference, and the other is powered directly by the battery. The voltage divider powered by the +5 V is designed to provide approximately 2.2 V at its center point. The other voltage divider is designed so that when the battery discharges to approximately 6 V, the center point of the voltage divider will also measure approximately 2.2 V. The analogue comparator is going to be set to detect when the voltage from the battery's voltage divider drops below the 2.2 V provided by the reference divider.

As Figure 8.37 shows, using the analogue comparator is relatively simple. In this example, ACSR is loaded with 0x0A to enable the analogue comparator,

```
#include <ATMega8515.h>
// Analogue Comparator interrupt service routine
interrupt [ANA_COMP] void ana_comp-isr (void)
{
        PORTB.1 = 0;            //light the LED
}
void main(void)
{
        PORTB = 0x02;           //start with LED off
        DDRB = 0x02;            //set bit 1 for output
        ACSR = 0x0A;            //enable analogue comp, AC interrupt, falling edge

        #asm("sei")                     //set global interrupt enable bit

        while (1)
           ;                            //do nothing
}
```

Figure 8.37 Analogue comparator example software.

to enable its interrupt, and to set it so that the interrupt occurs on a failing edge when AIN0 drops below AIN1 (AIN1 becomes greater than A1NO).

Example: Measuring Engine Temperature Using the Analogue-to-Digital Converter (ADC)

A simple data acquisition system is used to measure the engine temperature and sending the collected data to a PC. The practical measurements taken from the output of the thermocouple and corresponding conditioning circuitry showed that the temperature of the motor changes in the range 100°F to 250°F. The DAC is 10-bit which takes one sample every one second during the free-running mode. Analyze the system and write the corresponding programmes.

Step 1: Using the 10-bit measurement mode on the ADC means that the resulting measured values will be as follows:

$$100°\text{F} = 0 \times 000 = 0_{10}$$
$$250°\text{F} = 0 \times 3\text{FF} = 1023_{10}$$

This sets the conversion formula to be

$$\text{Temp} = (150°\text{F} * \text{ADC reading})/1023 + 100°\text{F}$$

The ADC may most conveniently be run in free-running mode for this use. In this way the temperature will be kept up to date as fast as possible so that when the data is stored at the one-second interval, the most recent value will be recorded. In free-running mode, the ADC interrupt occurs at the end of each conversion and can be used to update the current temperature value.

Step 2: Selecting a prescaler value

Part of the ADC initialization is to select an appropriate clock frequency by choosing the prescaler value for the ADC clock. The ADC clock must be between 50 kHz and 200 kHz. In this case, you have a system clock of 8 MHz, and so a prescaler of 64 will give an ADC clock of 125 kHz.

The ADC initialization process involves choosing the channel to be measured, enabling the free-running mode, and starting the first conversion so that an interrupt will occur at the end of the conversion, keeping the process running:

```
// ADC initialization
ADMUX = 0x3;     //select channel 3 and AREF pin as
                 //the reference voltage input
ADCSR = 0xE9;    //enable ADC, free run, started,
                 //clock prescaler of 64
```

Step 3: ADS ISR

The ADC ISR has the job of reading the current conversion value and converting it to temperature:

```
//ADC interrupt service routine
interrupt [ADC_INT] void adc_isr(void)
{
    unsigned int ADC_DATA;
    ADC_DATA = ADCW; //get data from ADC result
                     //register
    Current_temp = ((long)150 * (long ADC_DATA)/
                   (long) 1023 + (long 100;
}
```

Step 4: Sending collected data to the PC

The collected data is already converted to units of rpm and °F, so this function needs only to send the data to a PC. This involves initializing the UART and using built-in functions to format and send the data.

```
The UART is initialized for 9600 baud, 8 bits, and
no parity as follows:
    // UART initialization
    UCSRA = 0x00;
    UCSRB = 0x18; //enable transmitter and receiver
    UBRR = 0x33;  //select 9600 baud
    UBRRHI = 0x00;
```

The data sent to the PC is going to be analyzed using a Microsoft Excel spreadsheet. Investigation shows that data can be input directly using a comma-delimited (CSV) format. This means that commas separate the data on each line of the spreadsheet, and CR separates each line. That is, data sent as

data1, data2, data3
data4, data5, data6

will appear in the spreadsheet exactly as shown above occupying a space two cells high by three cells wide. In this case, you decide that the first column in the spreadsheet will contain the engine rpm, the second will contain the shaft rpm, and the third will contain the engine temperature. Each line of the spreadsheet will contain one set of data, so each line will contain data taken one second after the data in the previous line. The following code is the while (1) from the main() function of the code:

```
while (1)
{
    if (! PINA.0)    //Note: switch must be released
                     //before data is all
sent
    {
        unsigned char x; //temporary counter
                         //variable
                       //print column titles into
                       //the spreadsheet in the
                       //first row
        printf (''%s , %s , %s \n'',
                ''Engine RPM'', ''Shaft RPM'',
    ''Temperature'');
            for (x = 0; x< 120; x++)
            {
                    // print one set of data into
                    // one line on spreadsheet
              printf (''%d , %d , %d \n'',
                       e_rpm[x], s_rpm [x],
    temp[x]);
            }
    }
};
```

This routine sends the 120 sets of data (and a line of column titles) using the built-in prinf() function. A new line character ('\n') is included with each set of data to cause the next set to appear in the following line of the spreadsheet.

8.6 Some Practical ADC: The ADC0809 IC

In this section we are introducing one of the available ADC IC, ADC0809, and explain how to interface it to an Intel 8051 microcontroller. The ADC0809 is a CMOS device with an 8-bit analogue-to-digital converter, 8-channel multiplexer and microprocessor compatible control logic, shown in Figure 8.38.

The connections to the ADCO809 are as follows:

IN0-IN7 = Analogue input channels 0 to 7
ADDA = Bit 0 (LSB) of channel selection address
ADDB = Bit 1 of channel selection address
ADDC = Bit 2 (MSB) of channel selection address
ALE = Address Latch Enable for channel selection
START = Pin for starting conversion process
CLOCK = Synchronizing clock input
REF+, REF− = Upper and lower reference voltages for input
D0-D7 = Digital output
OE = Output Enable input pin
EOC = End of Conversion output pin

The 8-bit ADC uses successive approximation as the conversion technique. The converter features a high impedance chopper stabilized comparator, a voltage divider with analogue switch tree and a successive approximation

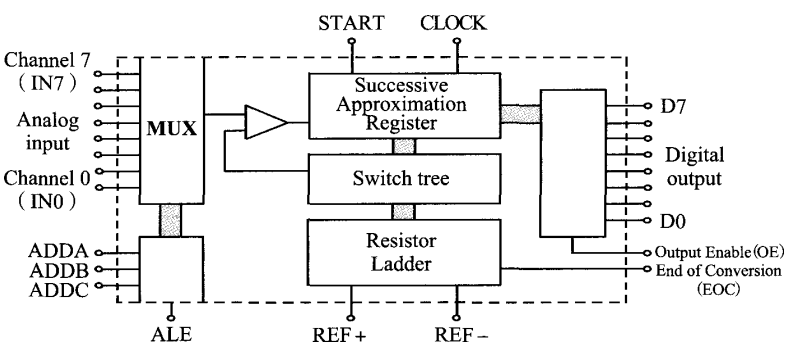


Figure 8.38 Block diagram of the ADCO8O9 structure.

Table 8.4 Addresses for selection of analogue input channels.

Selected analogue channel	Address lines		
	ADDC	ADDB	ADDA
IN0	0	0	0
IN1	0	0	1
IN2	0	1	0
IN3	0	1	1
IN4	1	0	0
IN5	1	0	1
IN6	1	1	0
IN7	1	1	1

register. The 8-channel multiplexer can directly access any of eight analogue input signals.

The 8-channel analogue multiplexer can switch the converter to any of the input channels which are numbered from 0 to 7. These are selected by means of the 3-bit address bus (ADDA-ADDC), as shown in Table 8.4. The upper and lower limits of the analogue range are determined by setting the values on the REF+ and REF− pins. The eight output pins D0/D7 are latched. The input pin START will cause the conversion to commence, and the device will signal the completion of the conversion by setting EOC (End Of Conversion). If it is required to run the converter in continuous operation, then the START and EOC pins can be connected together.

8.6.1 Connecting the ADC0809 to Intel 8051

The ADC0809 is intended for direct memory-mapping into a microcontroller system. It can be connected to the microcontroller by using one port for entering the analogue channel number and control bits, and a second port for receiving the output of the ADC. The ALE lines are connected to the corresponding lines on the two devices, and the use of the MOVX instruction enables data from the ADC to be copied straight to the accumulator on the 8051. A typical arrangement is shown in Figure 8.39.

Timing of data transfer from the ADC to the 8051

The MOVX instruction transfers data from an external device into the accumulator A. In this case there are no address lines in use, and therefore the data can be read by Port 0, regardless of the value in the DPTR. Usually the DPTR is cleared, but any number is valid for the operation. The timing diagram is shown in Figure 8.40, in which it can be seen that the transfer of the byte from the A-D converter takes place during the second machine cycle.

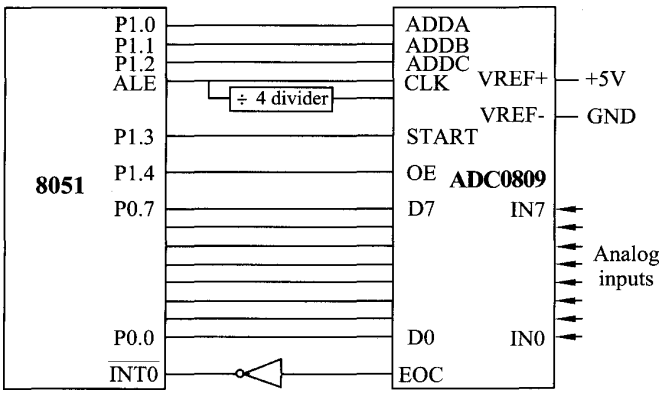


Figure 8.39 Connection of ADC to a microcontroller.

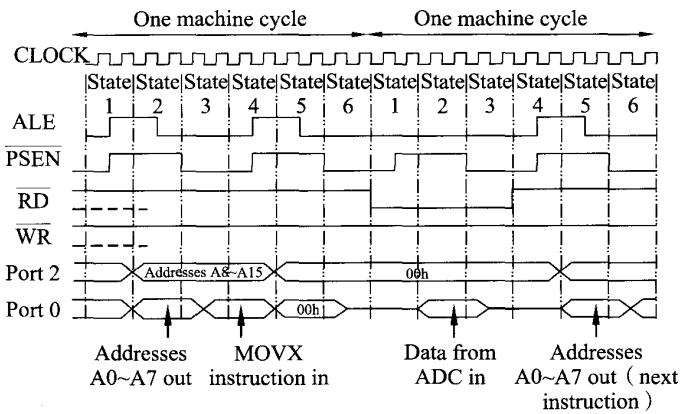


Figure 8.40 Timing diagram for reading the digital value from the ADC.

8.6.2 Examples of Software Needed for ADC

This is an example of simple conversion of data from analogue to digital

Task:

Using the circuit arrangement of Figure 8.39, write a programme, using external interrupt 0, to put the number from the A-D conversion into address 50h. Channel 0 is the selected analogue input.

Possible solution:

The MOVX instruction is used to input the data from the ADC0809 at the appropriate time. The analogue channel is selected by outputs on Port 1 pins.

The programme is as follows:

```
            ORG 0000h
            AJMP MAIN
            ORG 0003h
            AJMP EOC_INT
MAIN: MOV   DPTR, #0000h
                        ;DPTR = ext address 0000h
            SETB P1.4     ; Enable output with OE
            SETB IT0      ; Edge triggering of INT0
            SETB EA       ; Enable interrupts
            SETB EX0
            CLR P1.0      ; Select input Channel 0
            CLR P1.1
            CLR P1.2
            SETB P1.3     ;Start the conversion
            SJMP $
EOC_INT:    MOVX MOV A, @DPTR
            MOV 50h,A     ;Put result in location 50h
            RETI
```

8.7 Digital-to-Analogue Conversion Interface

8.7.1 The DAC0832 IC

The DAC0832 is a CMOS 8bit DAC designed to interface directly with several popular microprocessors. A resistor ladder network divides the reference current and provides the circuit with good temperature tracking characteristics.

The circuit uses CMOS current switches and control logic to achieve low power consumption and low output leakage current errors. The use of an input latch and a DAC register allows the device to output a voltage corresponding to one digital number while holding the next digital number. This permits the simultaneous updating of any number of these devices. Figure 8.41 shows the block diagram of the device.

In order to use the DAC0832, an external operational amplifier must be connected. This functions in conjunction with a feedback resistor inside the chip to give an analogue output.

The connections to the DAC0832 are as follows:

D0-D7 = Digital data input
/CS = Chip Select

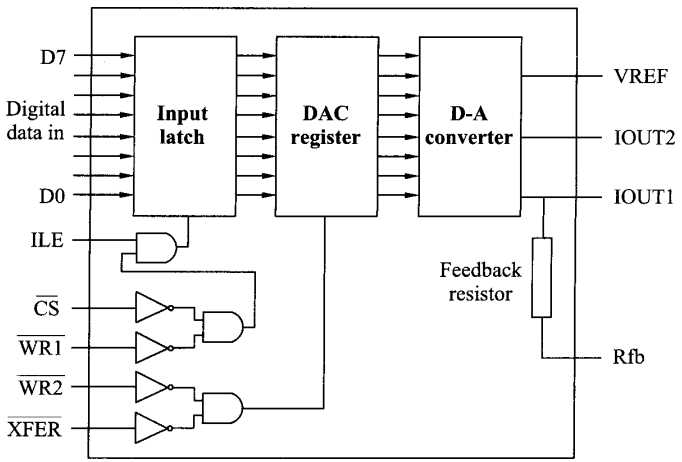


Figure 8.41 Block diagram of the DAC0832 structure.

/WR1 = Write Enable 1
/WR2 = Write Enable 2
/XFER = Pin to start transfer of data
ILE = Input Latch Enable
VREF = Reference voltage for the conversion
lOUT 1 = Current output 1 for op-amp connection
IOUT2 = Current output 2 for op-amp connection
Rfb = Connection to internal feedback resistor

8.7.2 Connecting the 8051 to the DAC0832

The DAC0832 can be connected to the microcontroller by the circuitry shown in Figure 8.42. The DAC produces two outputs intended for driving an operational amplifier, and the feedback resistor for this part of the circuit is provided internally within the DAC. The Op-Amp should have suitable input current rating and speed for the particular application.

8.7.3 Example of Software Needed for DAC

Task:

Using the circuit arrangement of Figure 8.42, write a programme to convert the digital value at internal RAM address 50h to an analogue signal at the output of the Op-Amp.

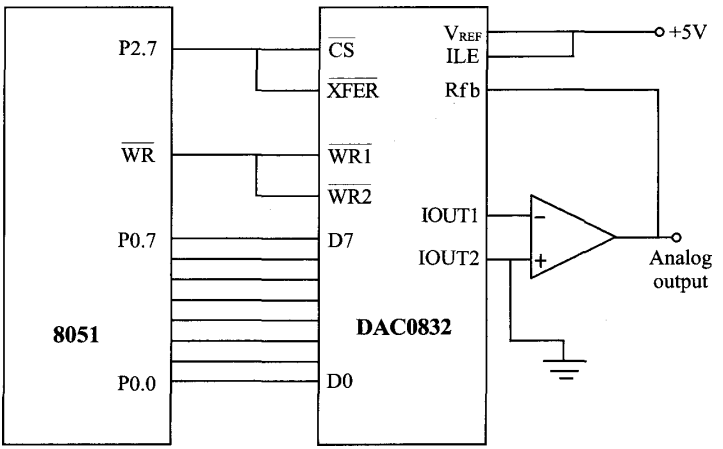


Figure 8.42 Connection of DAC to a microcontroller.

Possible solution:

The MOVX instruction is again used to read data from the external device. The pins /CS and /XFER on the DAC are connected to P2.7 on the 8051, which is the MSB of the 16-bit address. Therefore any address can be put into the DPTR so long as the MSB is zero. In this case 7FFFh is chosen.

```
ORG         0000h
MOV         A, 50h
MOV         DPTR, #7FFFh
MOVX        @DPTR, A          ; Send data to DAC
SJMP        $
```

8.7.4 Examples of controlling two DACs from an 8051

Task:

Design a circuit and write the software for an 8051 to continuously read the values on Ports 1 and 3, and output two analogue voltages corresponding to the digital inputs.

Possible solution:

Two DAC0832 chips are connected to the 8051 as shown in Figure 8.43. Most of the connections for the two DACs are in parallel, but the exceptions are the CS pins which connect independently to P2.6 and P2.5 of the 8051. This arrangement means that the MOVX instruction can be used to write to either of the DACs using different values of the high-byte of the address. One

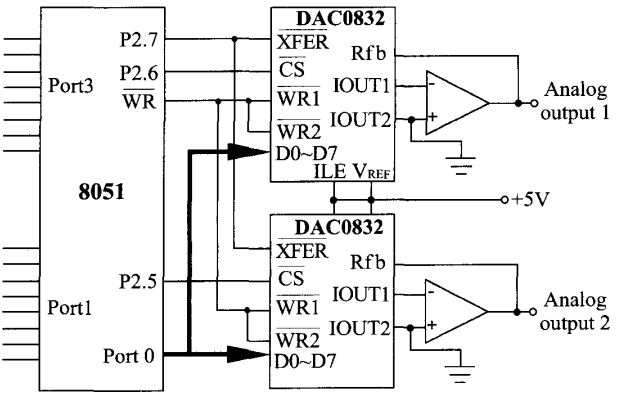


Figure 8.43 Connection of two DACs to an 8051.

DAC can be selected when bit 14 is zero, while the other requires bit 13 to be zero. Typical values of addresses which would meet these requirements are BFFFh and DFFFh respectively.

The operation is done in two stages. Firstly, the data for each DAC is written to the relevant input register. Then the XFER line is brought low to transfer the data into the DAC register for conversion. This can be done by using an address with bit 15 cleared, such as 7FFFh.

The programme to achieve this is as follows:

```
        ORG   0000h
LOOP:   MOV   A, P1           ; Put P1 data into A
        MOV   DPTR, #0BFFFh   ; Address bit 14 clear
        MOVX  @DPTR, A        ; Data to DAC input reg
        MOV   A, P3           ; Put P3 data into A
        MOV   DPTR, #0DFFFh   ; Address bit 13 clear
        MOVX  @DPTR, A        ; Data to DAC input reg
        MOV   DPTR, #7FFFh    ; Address bit 15 clear
        MOVX  @DPTR, A        ; Output both DACs
        SJMP  LOOP
```

8.8 Summary of the Chapter

In this chapter we introduced the meaning of data acquisition system and the performance criteria that can be used to evaluate the overall data acquisition

system. The specifications necessary to select some of the components of data acquisition systems are introduced.

The fundamentals of A/D conversion and D/A conversion have been discussed and some of the most popular ADC and DAC hardware are introduced. The ADC and DAC as part of the resources of the AVR and Intel microcontrollers are discussed. The question of how to interface ADC and DAC to processors is discussed.

8.9 Review Questions

8.1 Binary Weighted Digital-to-Analogue Converter is one of the main techniques that we use in a digital systems to generate analogue signals, explain its drawbacks?

8.2 How does the number of bits in a digital word affect the quality of the output of a digital-to-analogue converter?

8.3 What is an R/2R ladder and what is it used for?

8.4 Give three different ways by which we can measure voltage. Briefly explain the measurement technique used in each case.

8.5 What is the purpose of the sample and hold circuit in a successive approximation A/D?

8.6 How does the operation of the sample and hold circuit affect the accuracy of the conversion?

8.7 What are some sources of error in a sample and hold circuit?

8.8 Explain how to use A/D converters to measure current.

8.9 Give at least two ways by which you can measure temperature.

8.10 Give several ways by which we can measure resistance.

8.11 In the chapter, we introduced three different ways by which we can convert a digital signal to the analogue equivalent, compare the advantages and disadvantages of each method.

8.12 Give two examples of embedded systems, which need one or more of following units. (i) DAC (Using a PWM) (ii) ADC (iii) LCD display (iv) LED Display (v) Keypad (vi) Pulse Dialer (vii) Modem (viii) Transceiver (ix) GPIB (IEEE 488) Link.

8.13 An ADC is a 10-bit ADC? It has reference voltages, V_{ref-} = 0.0V and V_{ref+} = 1.023V. What will be the ADC outputs when inputs are (a) -0.512 V (b) + 0.5 12 V and (c) +2047V? What will be the ADC outputs in three situations when (i) V_{ref-} = 0.512 V and V_{ref+} = 1 .023V (ii) V_{ref-} = 1.024 V and V_{ref+} = 2.047V and (iii) V_{ref-} = - 1.024 V and V_{ref+} = +2.047V.

8.14 Write the expression for analogue current, *Ia* of a 4-bit D/A converter. Calculate values of *Ia* for input codes b3b2b1b0 = 0000, 0001, 1000, 1010, and 1111, if Iref = 1 mA.

8.15 a. Calculate the output voltage of an 8-bit bipolar DAC (offset binary) for code = 00000000. FS = 8 V.
b. Calculate the output voltage of the same DAC for code = 11111111.
c. Write the code for FS/2 and state the value of the output voltage.

8.16 a. Calculate the output voltage of an 8-bit bipolar DAC (2's complement) for code = 00000000. FS = 8 V.
b. Calculate the output voltage of the same DAC for code = 11111111.
c. Write the code for FS/2 and state the value of the output voltage.

8.17 a. Determine the range of voltages that will generate a code of 00000000 for an 8-bit bipolar ADC (2's complement coding) with a full scale range of - 4 V to +4 V.
b. Determine the range of voltages that will generate a code of 11111111 for the same ADC.
c. Calculate the output code when the input voltage is at the halfway point of its range.

8.18 a. Determine the range of voltages that will generate a code of 00000000 for an 8-bit unipolar ADC with a full scale value of 8 volts.
b. Determine the range of voltages that will generate a code of 11111111 for the same ADC.
c. Calculate the output code when the input voltage is at the halfway point of its range.

8.19 An ADC has a full scale voltage of 8 V. Write the codes for 0 V, 2 V, 4 V, and 6 V for the cases where the code is 4 bits, 6 bits, and 8 bits. Also write the voltage for the highest code for each case.

Question Using Verilog

8.20 Design a Verilog model, hardware interface, and software driver to an 8-bit successive approximation analogue-to- digital converter. Assume that the analogue input signal will be in the range of 0-5.0 VDC.
(a) What is the value, in volts, of the least significant data bit?
(b) What is the smallest full-scale error in a sample?
(c) Use your measurement subsystem to sample the following voltages: 0.0, 1.0, 2.0, 3.0, 4.0, and 5.0. Compare your measured values with the input voltages. What are the errors in your readings?

(d) Write a simple software loop to repeat the measurements in part (c) ten times. Plot your samples and the minimum and maximum error for each cardinal point. Are the errors consistent across the range of values? If not, how do you explain the differences?

9

Multiprocessor Communications (Network — Based Interface)

THINGS TO LOOK FOR…

- The need for serial communication.
- UART/USART and RS-232.
- The most commonly used communication protocols for network-based interface: I2C, CAN and SPI.
- The strengths and weaknesses of each protocol.
- The different transport mechanisms
- How we can achieve synchronization in each protocol

9.1 Introduction

The main target of using microcontrollers and microprocessors in an embedded system is to process data coming from input devices and/or send data to control some output devices. System I/O devices and timing devices play the most significant role in any embedded system. Some of the I/O devices, e.g. timers and UART, are build-in the microcontroller chip. Such devices are accessing the system processor internally. The majority of the I/O devices are outside the microcontroller. Such I/O devices are connected and accessing the microcontroller through a port with each port having an assigned port address similar to a memory address.

Till recently the embedded system has two main features: (a) the external I/O devices and the processor are local to the immediate environment, (b) all the needed processing are taking place within the microcontroller/microprocessor, i.e. the I/O devices are passive and have no role in processing. Now, embedded systems are expanded into nearly every corner of the modern world. Embedded systems, accordingly, become complex systems comprising potentially several processors, programmable logic devices,

networks, communication systems, ASICs, and much more. Some of these networking devices such as transceivers and encrypting/decrypting devices operate at MHz and GHz rates. The new embedded systems are characterized by; (a) many of the I/O devices are acting as sub-processors (sometimes processors or single-purpose processors), (b) many of the I/O devices are substantially remote and distributed. To meet the new features, distributed devices are networked using the sophisticated I/O buses in a system. For example, take the case of an automobile. Most devices are distributed at different locations in any automobile. These are networked using a bus called Controller Area Network (CAN) bus. Similarly a number of sub-processor ICs (e.g. encryptors and decryptors ICs) can be networking using a bus called I2C (Inter-Integrated Circuits).

This leads us to the case of network-based interfaces. The microcontroller network can be looked as a single communications medium (i.e., wire or bus) and multiple devices connected to this medium which can initiate message transfers and respond to messages directed towards them. In other words, we are targeting bus-based communication that offers multiple devices to share the same physical connection lines.

We must remember also the fact that it is usually associated with each I/O device a software procedure called a *driver* or *device driver* that supports the interaction with the device. Accordingly, when studying input and output interaction with the external world, several things must be considered, notably:

- The source or destination for any exchange.
- The I/O ports of the microcontroller or the microprocessor.
- The memory address space and map.
- The location of the I/O driver.
- The protocol for the data exchange.
- Timing requirements.
- The physical medium of the exchange.

We start this chapter with some data communications terminology that is important to embedded systems design. This is followed by giving a brief idea on Universal Asynchronous Receiver/Transmitter (UART) and Universal Synchronous Asynchronous Receiver/Transmitter (USART) and how microcontrollers are implementing them as essential part of the resources of any microcontroller. After that we will study four different, commonly used network-based input/output designs: RS-232 (which is now EIA-232), Inter-Integrated I2C, Controller Area Networks (CAN) and Serial Peripheral Interface (SPI). For each interface we will consider: the problems that

motivated designers to develop the interface, the transport mechanism, study how the control and synchronization are achieved, investigate how message senders and receivers are identified, identify the contributions that each has introduced to the embedded world, and cite the strengths and weaknesses of each approach.

We selected these busses because each represents a change in thinking about the way information is exchanged in the context in which the design is applied. The architecture and control/synchronization mechanisms utilized in these busses are representative of those found in most of the network-type busses in use today. In this way the reader will understand the key aspects, strengths, and weaknesses of each.

9.2 Serial Communications Channels

Serial data transfer involves a transmitter (source) and a receiver (destination). The electrical transmission path between the two is called a communications channel. This channel can be as simple as a single wire or coaxial cable, or more complex such as fiber optic cables, telephone lines, or a portion of the radio frequency spectrum.

Serial channels are classified in terms of their direction of transfer as simplex, half duplex, or full duplex (Figure 9.1).

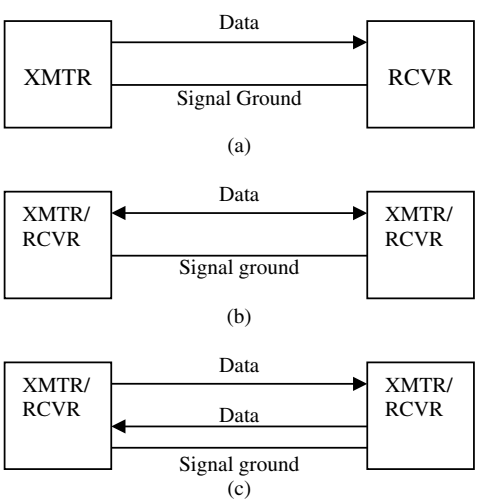


Figure 9.1 Serial data connections classified by direction of transfer: (a) Simplex; (b) Half-duplex; (c) Full-duplex.

A ***simplex*** channel can transfer data in only one direction. One application of the simplex channel is the ring network.

A ***half-duplex*** communication system, Figure 9.1b, allows information to transfer in either direction, but in only one direction at a time. Half-duplex is a term usually defined for modem communications, but it is possible to expand its meaning to include any serial protocol that allows communication in both directions, but in only one direction at a time. One application of the half-duplex protocol is the desktop serial bus, or multidrop network.

A ***full-duplex*** communication system, Figure 9.1c, allows information (data, characters) to transfer in both directions simultaneously. A full-duplex channel allows bits (information, error checking, synchronization, or overhead) to transfer simultaneously in both directions.

When the communications channel consists of wires or lines, the minimum number required for a simplex channel is two: the data, or signal line, and a signal ground. A half-duplex channel also requires only two lines, signal and ground. However, there must be some method to "turn the line around" when the direction of transfer is to change. This involves a protocol and logic to establish one device as the transmitter and the other as the receiver. After transfer in one direction is complete, the roles of transmitter and receiver are reversed and transfer takes place in the opposite direction, on the same signal line. Full- duplex channels require two signal lines and a signal ground. Each signal line is dedicated to data transfer in a single direction, allowing simultaneous transfer in opposite directions.

A ***frame*** is a complete and nondivisable packet of bits. A frame includes both information (e.g., data, characters) and overhead (start bit, error checking, and stop bits). A frame is the smallest packet that can be transmitted. The RS232 and RS422 protocols, for example, have one start bit, seven/eight data bits, no/even/odd parity, and one/1.5/two stop bits (Figure 9.2).

Parity is generated by transmitter and checked by the receiver. Parity is used to detect errors. The parity bit can be calculated using a built-in Parity Generator that calculates automatically the parity bit for the serial frame data

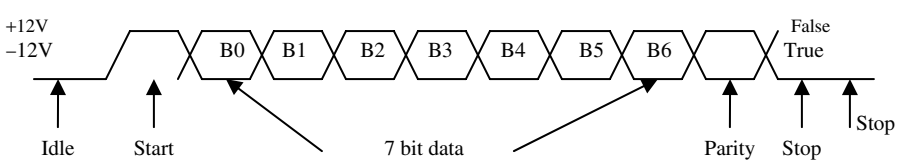


Figure 9.2 A R232 Frame (R232 uses negative logic).

or can be calculated manually then entered into a register. A simple algorithm for calculating the parity bit is to take the byte to be transmitted and XOR all the bits together. If odd parity is used, the result of the exclusive should be XORed with 1 or is inverted. The relation between the parity bit and data bits is as follows:

$$P_{even} = d_{n-1} \oplus \cdots \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 0$$
$$P_{odd} = d_{n-1} \oplus \cdots \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 1$$

P_{even} Parity bit using even parity
P_{odd} Parity bit using odd parity
d_i Data bit i of the character

Baud and ***Baud Rate***: The bit time is the basic unit of time used in serial communication. Bit time is the time that a single data bit is valid during a serial transfer. The transmitter outputs a bit, waits one bit time, and then outputs the next bit. The start bit is used to synchronize the receiver with the transmitter. The receiver waits on the idle line until a start bit is first detected. After the true-to-false transition the receiver waits a half bit time. The half bit time wait places the input sampling time in the middle of each data bit, giving the best tolerance to variations between the transmitter and receiver clock rates. The detailed timing will be discussed latter in the chapter.

The number of signal changes per second on a communications channel is its ***baud rate***. Since pure binary signals can only change between one of two possible values, their baud rate is equivalent to the number of bits per second (bps) transferred. The baud rate can be looked on as the total number of bits (information, overhead, and idle) per time that is transmitted in serial communication. The bit time is the reciprocal of the baud rate.

$$\text{Baud rate} = 1/(\text{bit time}) \tag{9.1}$$

A communication channel's capacity is the maximum baud rate the channel can support. If data is transmitted at a rate beyond a channel's capacity, the number of transmission errors increases significantly.

Information can be defined as the data that the "user" intends to be transmitted by the communication system. Examples of information include:

- Characters to be printed by a printer
- A picture file to be transmitted to another computer
- A digitally encoded voice message communicated to another person
- The object code files to be downloaded from the PC to another device.

The overhead may be defined as signals added by the system ("operating system") to the communication to affect reliable transmission. Examples of

overhead include:

- Start bit(s), start byte(s), or start code(s)
- Stop bit(s), stop byte(s), or stop code(s)
- Error checking bits such as parity, cyclic redundancy check (CRC), and checksum
- Synchronization messages like ACK, NAK, XON, XOFF

Although, in a general sense overhead signals contain "information", overhead signals are not included when calculating bandwidth or considering full-duplex, half-duplex, or simplex.

An important parameter in all communication systems is bandwidth. In many cases the three terms "***bandwidth***", "*bit rate*" and "*throughput*" are used interchangeably to specify the number of information bits per time that is transmitted. These terms apply to all forms of communication: serial, parallel, and mixed parallel/serial.

For serial communication systems, the following equation can be used to calculate the bandwidth:

$$Bandwidth = \frac{\text{number of information bits/frame}}{\text{total number of bits/frame}} \times \text{baud rate} \qquad (9.2)$$

9.2.1 Synchronization Techniques

A clock signal is needed to synchronize the receiver's timing to the incoming serial data so that the receiver can latch each data bit. Serial data transfers can be classified as either asynchronous or synchronous based on the nature of the clock signal.

In ***asynchronous*** transfer, the transmitter and receiver do not share a common clock signal. Each provides its own independent clock. Both clocks run at the same nominal frequency. Asynchronous serial data is transmitted as frames (Figure 9.3). Synchronization is established on a frame-by-frame basis, and only for the duration of each frame. The advantage of asynchronous transfer is its low cost. It is used primarily where transmission of frames is irregular, or where large blocks of data do not need to be transmitted at high speed.

In ***synchronous*** transfer, either a common clock signal exists between the transmitter and receiver, or the receiver is capable of deriving a clock signal from the transitions of the received data signal.

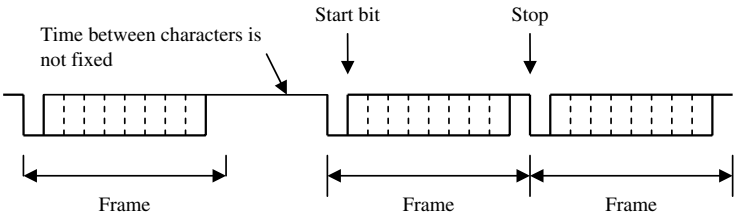


Figure 9.3 Asynchronous serial transfers.

9.3 Asynchronous Serial Communication: UART

Asynchronous transmission recognizes and accepts that there are irregular intervals between the sending of piece of data. It allows data to be transmitted without the sender having to send a clock signal to the receiver. Instead, the sender and receiver must agree on transmission protocol. The transmission protocol uses the framing procedure mentioned above. For successful transmission and reception of the frames, the protocol includes timing parameters (e.g. baud rate) and the special bits to be added to each word which are used to synchronize the sending and receiving units. Character encoding, data compression and error detection must also be agreed upon. The sequence of pulses illustrated in Figure 9.2 represents a sample transmission protocol.

The most popular serial transmission protocol is the Universal Asynchronous Receiver Transmitter (UART). The transmission protocol used by UART determines also the rate at which bits are sent and received, i.e. the baud rate. The protocol also specifies the number of bits of data and the type of parity sent during each transmission. Finally, the protocol specifies the minimum number of bits used to separate two consecutive data transmissions. Stop bits are important in serial communication as they are used to give the receiving UART a chance to prepare itself prior to the reception of the next data transmission.

When a word is given to the UART for asynchronous transmissions, a "Start Bit" is added to the beginning of each word that is to be transmitted. The Start Bit is used to alert the receiver that a word of data is about to be sent, and to force the clock in the receiver into synchronization with the clock in the transmitter.

After the Start Bit, the individual bits of the word of data are sent, with the Least Significant Bit (LSB) being sent first. Each bit in the transmission is transmitted for exactly the same amount of time as all of the other bits, and the receiver "looks" at the wire at approximately halfway through the period assigned to each bit to determine if the bit is a 1 or a 0. For example, if it takes

two seconds to send each bit, the receiver will examine the signal to determine if it is a 1 or a 0 after one second has passed, then it will wait two seconds and then examine the value of the next bit, and so on.

The sender does not know when the receiver has "looked" at the value of the bit. The sender only knows when the clock says to begin transmitting the next bit of the word.

When the entire data word has been sent, the transmitter may add a **Parity Bit** that the transmitter generates. The Parity Bit may be used by the receiver to perform simple error checking. After the parity bit, if any, at least one **Stop Bit** is sent by the transmitter.

When the receiver has received all of the bits in the data word, it may check for the Parity Bits (both sender and receiver must agree on whether a Parity Bit is to be used), and then the receiver looks for a Stop Bit. If the Stop Bit does not appear when it is supposed to, the UART considers the entire word to be distorted and will report a ***Framing Error*** to the host processor when the data word is read. The usual cause of a Framing Error is that the sender and receiver clocks were not running at the same speed, or that the signal was interrupted.

Regardless of whether the data was received correctly or not, the UART automatically discards the Start, Parity and Stop bits. If the sender and receiver are configured identically, these bits are not passed to the host.

If another word is ready for transmission, the Start Bit for the new word can be sent as soon as the Stop Bit for the previous word has been sent.

Because asynchronous data is "self synchronizing", if there is no data to transmit, the transmission line can be idle.

Along with "odd" and "even" parity, there is also "No", "Mark," and "Space" parity. "No" parity means that no parity bit is transmitted with the data in the packet. "Mark" or "Space" parity means that a "1" or "0," respectively, is always sent after the data in the packet. This type of parity is very rarely used, and when it is used, the reason for its use is to allow the receiver time to process the data before the next data byte comes along.

The number of stop bits is also an option, for the same reasons as mark and space parity. A second stop bit can be inserted in the data packet to give the receiver more time to process the received byte before preparing to receive the next one.

In virtually all modern asynchronous communications, the data format is "8-N-1," which means 8 data bits, no parity, and one stop bit as the packet format. The parity and the additional end interval are generally not required for serial communications.

Definition: **Serial Frame and Frame Format**

A serial frame is defined to be one character of data bits with synchronization bits (start and stop bits), and optionally a parity bit for error checking. The format defining the number of bits used to represent each of the three components and there sequence within the frame is called the Frame Format.

Example 9.1: AVR ATmega8515 Frame Format

The USART accepts all 30 combinations of the following as valid frame formats:

- 1 start bit
- 5, 6, 7, 8, or 9 data bits
- no, even or odd parity bit
- 1 or 2 stop bits

Figure 9.4 illustrates the possible combinations of the frame formats. Bits inside brackets are optional.

9.3.1 Data Recovery and Timing

With an asynchronous transmission scheme, the transmitting and receiving devices are each operating on independent local clocks. Data recovery depends on the transmission protocol that the transmitter and the receiver agreed to use. The transmission protocol defines and guarantees that the transmitter and receiver agreed on at least the following key parameters:

1. The sender and receiver must be operating at the same baud rate.
2. The sender and receiver must agree on the number of bits comprising a character.
3. The idle state of the data lines is known. The EIA-232 standard specifies this to be the logical one state.
4. The sender must place the data line into the idle state for at least one bit time between characters. This time is the stop bit.

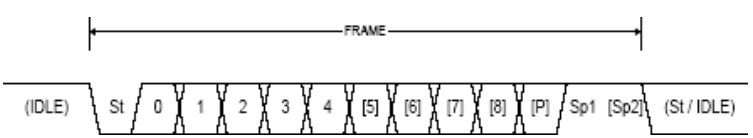


Figure 9.4 AVR frame formats.

With such key elements in place, the asynchronous data recovery at the receiving end takes place, roughly, in two phases:

1. Synchronize the Internal Clock: The system uses this phase for defining a valid "start bit" that will be used to synchronize the internal clock to the incoming serial frame. This phase is called "***Asynchronous Clock Recovery"***.
2. Data Recovery: When the Receiver clock is synchronized to the start bit, the second phase, which is the data recovery, starts. This phase is called "***asynchronous data recovery***".

9.3.1.1 Asynchronous clock recovery

The receiver uses a special hardware called the "***clock recovery logic***" to synchronize the internal clock to the incoming serial frames. It uses an over-speed clock to sample the start bit of an incoming frame. The use of over-speed clock for sampling the received signal is called "***oversampling***". Oversampling is used to detect any false start bit, to make the synchronization and to achieve noise filtering that guarantee correct detection of the data bits. Figure 9.5 illustrates the sampling process of the start bit of an incoming frame. The oversampling rate is normally 16 times or 8 times the baud rate.

Normally a *"bit timer"* is used that runs with the oversampling speed:

$$\textit{Timer period} = \textit{clock rate/baud rate}.$$

For example if the baud rate is 9600 and the oversampling rate is selected to be 16, then the bit timer is running with clock rate of 153.6 KHz.

When the clock recovery logic detects a high (idle) to low (start) transition on the input line (RxD line), the start bit detection sequence is initiated; the *bit timer* is enabled. Let sample 1 denotes the first zero-sample as shown in the Figure 9.5b. The clock recovery logic then uses samples 8, 9, and 10 for Normal mode, and samples 4, 5, and 6 for Double Speed mode (indicated with sample numbers inside boxes on the figure), to decide if a valid start bit is received. If two or more of these three samples have logical high levels (the majority wins), the start bit is rejected as a noise spike "glitch" and the receiver starts looking for the next high to low-transition. If however, a valid start bit is detected, the clock recovery logic is synchronized and the data recovery can begin. The synchronization process is repeated for each start bit. The block diagram in Figure 9.5a illustrates the design.

The above mentioned process of clock synchronization is used for both hardware (use of clock recovery logic) and software asynchronous data

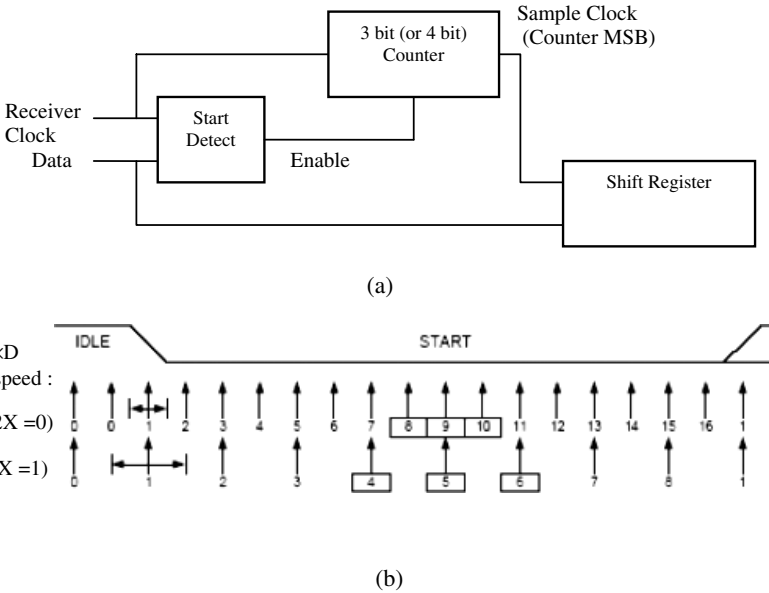


Figure 9.5 (a) System utilization bit sampling at the receiver, (b) Start Bit Sampling.

receiving. In case of software receiver, a timed loop is used for the half bit and full bit delays.

It is important to mention here that when using oversampling of 16, the output of the most significant bit of the bit timer makes a 0-1 transition when the count advances from 7 (0111) to 8 (1000). That transition can be used to implement other functions that will be used by the system for data recovery. For example, it can be used to clock the incoming data into a shift register. The same transition can be used to increment a second ***bit counter*** that counts the number of incoming data bits that have been stored. When the agreed upon (by the sender and receiver) number of bits have been stored, it is known that a full character has been received. Both the bit timer and bit counter return to the quiescent state awaiting the next start bit.

9.3.1.2 Asynchronous data recovery

When the receiver clock is synchronized to the start bit, the data recovery can begin. Figure 9.6 shows the sampling of the data bits and the parity bit. The *bit timer*, which has been enabled at the leading edge of the start bit, will reach a count of 8, at the centre of the first data bit and because it is incrementing modulo 16 (or modulo 8), it will again reach a count 8 by the centre of the

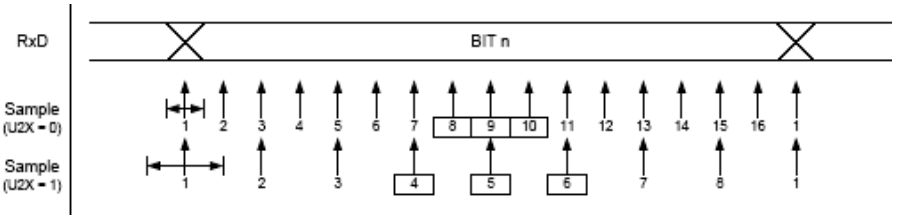


Figure 9.6 Sampling of data and parity bit.

second and succeeding data bits. As in case of start bit the decision of the logic level of the received bit is taken by doing a majority voting of the logic value to the three samples in the center of the received bit. This majority voting process acts as a low pass filter for the incoming signal on the input pin (RxD pin). The recovery process is then repeated until a complete frame is received, including the first stop bit. Note that the receiver only uses the first stop bit of a frame.

The same majority voting is done to the stop bit as done for the other bits in the frame. If the stop bit is registered to have a logic 0 value, a Frame Error (FE) Flag in the UART status register will be set. The *bit counter* helps to achieve that; after counting the agreed number of data bits, it is expected that the next bit is a stop bit. A new high to low transition indicating the start bit of a new frame can come right after the last of the bits used for majority voting.

9.3.1.3 Baud rate

To use a UART, we must configure its baud rate by writing to the configuration register, and then we must write data to the transmit register and/or read data from the received register. Unfortunately, configuring the baud rate is usually not as simple as writing the desired rate (e.g., 4,800) to a register. For example, to configure the UART of an 8051 microcontroller, we must use the following equation:

$$\text{Baud rate} = (2^{\text{smod}}/32) * \textit{oscfreq}/(12 * (256 - \text{TH1})).$$

Here, *smod* corresponds to 2 bits in a special-function register, *oscfreq* is the frequency of the oscillator, and TH1 is an 8-bit rate register of a built-in timer (Baud Rate Register). (The equation used in case of AVR are given latter.)

9.3.1.4 Other UART functions

In addition to the basic job of converting data from parallel to serial for transmission and from serial to parallel on reception, a UART will usually

provide additional circuits for signals that can be used to indicate the state of the transmission media, and to regulate the flow of data in the event that the remote device is not prepared to accept more data. For example, when the device connected to the UART is a modem, the modem may report the presence of a carrier on the phone line while the computer may be able to instruct the modem to reset itself or to not take calls by raising or lowering one more of these extra signals. The function of each of these additional signals is defined in the EIA RS232-C standard.

9.3.2 Serial Communication Interface

Any serial communication port needs an interface system that guarantee its proper functionality as it is defined by the designer or the programmer. The functionality covers, the frame format, other parameters related to serial communication as baud rate, converting the received signals from serial to parallel form, converting the transmitted signals from parallel to serial form, the current status of the port e.g. if it is ready to receive (or to transmit) the next message, etc. To achieve that, the serial communication interface consists normally of four registers:

- **Data Register — UDR:** The UDR is normally two registers sharing a single I/O address: one of them, a read-only register and the other a write-only register. The read-only register contains any serial byte received, and the write-only register contains any serial byte to be transmitted. So, when a programme reads the UDR, it is reading the receive UDR to get data that has been received serially. When the programme writes to the transmit UDR, it is writing data to be serially transmitted.
- **Control Register — UCR:** The serial port (UART/USART) control register is used to initialize and set the function of the port (the UART/USART). The most significant three bits of this register are normally mask bits for the three interrupts associated with the UART/USART. The first interrupt is one that occurs when a serial character is received by the UART/USART, the second interrupt is one that occurs when the UART/USART has successfully completed transmitting a character, and the third interrupt occurs when the transmit UDR is ready to transmit another character.
- **Status Register — USR:** The current state of the UART/USART is reflected in the status register. The USR has bits to indicate when a character has been received, when a character has been transmitted, when the transmit UDR is empty, and any errors that may have occurred.

- **Baud Rate Register — UBRR:** This register determines the baud rate at which the serial port transmits and receives data.

The following sections discuses the serial communication as implemented by AVR microcontrollers. At first we are going to deal with the AVR AT90S series that has single UART, and then we are going to discuss the USART of the AVR Mega series.

9.3.2.1 Relation between UART and RS-232: Communication Medium

The serial waveform and other parameters relating to the serial communication discussed above are all involved with the information being transmitted. The serial information is independent of the medium being used to carry the serial information. The medium may consist of a wire, an infrared beam, a radio link, or other means.

The most common medium is defined as RS-232. It was developed in order to provide a greater distance for reliable serial communication using wire to carry the signal. RS-232 is an inverted scheme, in that a logic 1 is represented by a negative voltage more negative than −3 V and a logic 0 is represented by a positive voltage more positive than +3 V. Most microcontroller-to-PC communication is RS-232. RS-232 is the subject of the next section (Section 9.4). The differences between the functions of RS-232 and UART are shown clearly in Figure 9.7.

9.3.3 AVR UART/USART

The AVR AT90 series, e.g. AT90S4414/8515/8535 features a full duplex (separate receive and transmit registers) Universal Asynchronous Receiver and Transmitter (UART). The main features are:

- Baud rate generator that can generate a large number of baud rates (bps)
- High baud rates at low oscillator frequencies

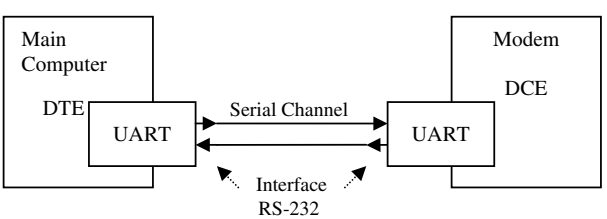


Figure 9.7 A serial Channel connects a DTE to a DCE.

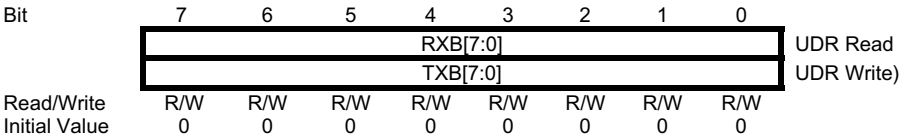


Figure 9.8 The UDR registers.

- 8 or 9 bits data
- Noise filtering by oversampling
- Overrun detection
- Framing Error detection
- False Start Bit detection
- Three separate interrupts on TX Complete, TX Data Register Empty and RX Complete

The UART Data Register or UDR (Figure 9.8) is actually two registers sharing a single I/O address ($0C): one a read-only register and the other a write-only register.

- The read-only register, or the USART Receive Data Buffer Register (RXB), contains any serial byte received, and
- The write-only register, or the USART Transmit Data Buffer Register (TXB), contains any serial byte to be transmitted.

So, when a programme reads the UDR, it is reading the receive UDR to get data that has been received serially. When the programme writes to the transmit UDR, it is writing data to be serially transmitted.

In other words, the Transmit Data Buffer Register (TXB) will be the destination for data written to the UDR Register location. Reading the UDR Register location will return the contents of the Receive Data Buffer Register (RXB).

UART Control Register (UCR)

Figure 9.9 shows the bit definitions for the UART control register, UCR. The UART control register is used to initialize and set the function of the UART. The most significant three bits, as mentioned before, are the mask bits for the three interrupts associated with the UART.

The UART Status Register (USR)

The current state of the UART is reflected in the UART status register. The status register has bits to indicate when a character has been received, when a character has been transmitted, when the transmit UDR is empty,

Bit	7	6	5	4	3	2	1	0
	RXCIE	TXCIE	UDRIE	RXEN	TXEN	CHR9	RXB8	TXB8

Bit	**Description**
RXCIE	Mask bit for the receiver interrupt enable. Set to unmask the interrupt. .
TXCIE	Mask bit for the Transmit interrupt enable. Set to unmask the interrupt.
UDRIE	Mask bit for the UART Data Register Empty interrupt enable. Set to unmask the interrupt.
RXEN	Set to enable the serial receiver.
TXEN	Set to enable the serial transmitter
CHR9	Set to enable 9 bit serial character.
RXB8	9th bit received in 9 bit mode
TXB8	9th bit transmitted in 9 bit mode.

Figure 9.9 UART control register.

Bit	7	6	5	4	3	2	1	0	
	RXC	TXC	UDRE	FE	DOR	PE	U2X	MPCM	UCSRA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	(b)
Initial value	0	0	1	0	0	0	0	0	
	RXC	TXC	UDRE	FE	DOR				USR
									(a)

Figure 9.10 USR Status register.

Bit	Bit name		Description
	AT90S 8518	ATmega 8515	
7	RXC		**Receive Complete**: Set to indicate receipt of a serial character
6	TXC		**Transmit Complete:** Set to indicate that a serial character has been sent
5	UDRE		**Data Register Empty**: Set to indicate that the transmit UDR is empty
4	FE		**Frame Error**: Set to indicate a framing error
3	(D)OR		**Data Over Run**: Set to indicate an overrun error
2	-	PE	**Parity Empty:** This bit is set if the next character in the receive buffer had a Parity Error when received.
1	-	U2X	**Double the USART Transmission Speed:** Set this bit to double the transfer rate in case of asynchronous UART transmission. Write this bit to zero when using synchronous transmission.
0	-	MPCM	**Multi-processor Communication Mode**: Set to enable the Multi-processor Communication Mode

Figure 9.11 Definition of the USR bits.

and any errors that may have occurred. The microcontrollers that have UART (e.g.AT90S8515) have one 8 bits status register USR that is completely separate from the control register UCR. Figure 9.10 shows the contents of the 8-bit USR of the AVR AT90 series, and Figure 9.11 gives the definition of each bit.

The status register USR/USCR is important due to the fact that serial communication is always slower than parallel communication. During the

1.04 milliseconds that it takes to transmit a single serial byte at 9600 baud, a microcontroller using a system clock of 8 MHz can execute as many as 8000 instructions. So, in order for the microcontroller not to be waiting around for the serial port, it is important for the programme to be able to tell the state of the serial port. In the case of a received character, it takes the same 1.04 milliseconds from the time the start bit is received until the character has been completely received. After the eighth bit is received, RXC bit is set in the USR/USCR to indicate that a serial byte has been received. The programme uses the RXC bit to know when to read valid data from the receive UDR. The data must be read from the UDR before the next character is received, because it will overwrite and destroy the data in the UDR. This explains the provision for an interrupt to occur when a serial character is received so that it may be read promptly without consuming large amounts of processor time polling to see if a byte has been received.

In a similar manner, the UDRE bit is used to indicate that the transmit UDR is empty and that another byte may be loaded into the transmit UDR to be transmitted serially. Again, this is necessary because the microcontroller can load out bytes much faster than the UART can transmit them. In order to keep the programme from having to poll the USR continuously to see when it is available to send another byte, an interrupt is provided that indicates when the transmit UDR is empty.

The transmitter side of the UART is actually double buffered. That is, the UDR that the programme writes to holds the data until the actual transmit register is empty. Then the data from the UDR is transferred to the transmit register and begins to serially shift out on the transmit pin. At this point, the UDR is available to accept the next data word from the programme, UDRE flag is set high and, if enabled, the UDRE interrupt occurs. Occasionally it is necessary for the microcontroller to actually know when a byte has been sent. The TXC flag is provided to indicate that the transmit register is empty and that no new data is waiting to be transmitted. The programme uses this flag, and its associated interrupt, when it is necessary to know exactly when the data has been sent.

Baud Rate Register UBRR

The final register associated with the UART is the UART baud rate register, UBRR. This register determines the baud rate at which the serial port transmits and receives data. The number entered in the UBRR is determined in case of UART according to the following formula:

$$\text{UBRR} = (\text{System Clock}/(16 * \text{baud rate})) - 1 \tag{9.3}$$

Bit	7	6	5	4	3	2	1	0	
$09 ($29)	MSB							LSB	UBRR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 9.12 AVR Baud Rate Register (UBRR).

Example 9.2: Calculate the number to be entered in the UBRR to transmit the data asynchronously with baud rate of 9600. Assume the internal clock has a frequency of 8 MHz.

Answer: Using the formula of the normal UART, the UBRR number is calculated as follows:

$$\begin{aligned} \text{UBRR} &= (\text{System Clock}/(16 * \text{baud rate})) - 1 \\ &= (8\,\text{Mhz}/(16 * 9600\text{baud})) - 1 = 51 = 0 \times 33 \end{aligned}$$

The microcontrollers that handle asynchronous mode only, i.e. that have UART, uses an 8-bit read/write register to store the UBRR number. The contents of this register, the UBRR register, specify the UART Rate according to the above equation. In many AVR microcontroller this register is at I/O location $09 ($29). Figure 9.12 shows the AVR UBRR register.

The following example shows how to handle the UBRR register

Example 9.3: The following example prints the message "Yabba Dabba Do" on a terminal device. The programme considers the case of asynchronous transmission by baud rate of 9600 and internal frequency clock of 8 MHz.

To transmitter with baud rate of 9600 using an internal frequency of 8MHz, the UBRR number as calculated in Example 9.2 is 0x33. The programme in case of using AT90S8535 microcontroller is as follows:

```
#include <90s8535.h>
#include <stdio.h>       //Standard Input/Output
                         //functions

void main(void)
{
      UCR=0x18; //serial port initialized
      UBRR = 0x33; //Set baud rate
      Printf (''\n\rYabba Dabba Do''); //print
                                       //phrase
      While (1)
```

```
      ;                  // do nothing else
}
```

As the simple serial example shows, instead of initializing the serial port, the first two lines of main() serial communication is extremely easy using the built-in library functions of CodeVisionAVR. CodeVisionAVR, like most C language compilers, provides built-in library functions to handle the common serial communication tasks. These tasks usually involve communicating with a terminal device, for example, a PC executing a terminal programme to transmit serially the characters typed on the keyboard and to display in a window the characters received serially from the microcontroller. CodeVisionAVR provides a built-in terminal emulator in the development environment as a convenience to the programmer. The standard library functions are included in the header file stdio.h, which may be included in the C language programme. Using the header file with its built-in functions makes serial communication very easy. This will be shown during the following discussions.

Data Transmission Using UART

A block schematic of the UART transmitter is shown in Figure 9.13.

Data transmission is initiated by writing the data to be transmitted to the UART I/O Data Register, UDR. The data is transferred from UDR to the Transmit shift register when:

- A new character has been written to UDR after the stop bit from the previous character has been shifted out. The shift register is loaded immediately.
- A new character has been written to UDR before the stop bit from the previous character has been shifted out. The shift register is loaded when the stop bit of the character currently being transmitted has been shifted out.

If the 10(11)-bit Transmitter shift register is empty, data is transferred from UDR to the shift register. At this time the UDRE (UART Data Register Empty) bit in the UART Status Register, USR, is set. When this bit is set (one), the UART is ready to receive the next character. At the same time as the data is transferred from UDR to the 10(11)-bit shift register, bit 0 of the shift register is cleared (start bit) and bit 9 or 10 is set (stop bit). If 9 bit data word is selected (the CHR9 bit in the UART Control Register, UCR is set), the TXB8 bit in UCR is transferred to bit 9 in the Transmit shift register.

On the Baud Rate clock following the transfer operation to the shift register, the start bit is shifted out on the TXD pin. Then follows the data, with LSB

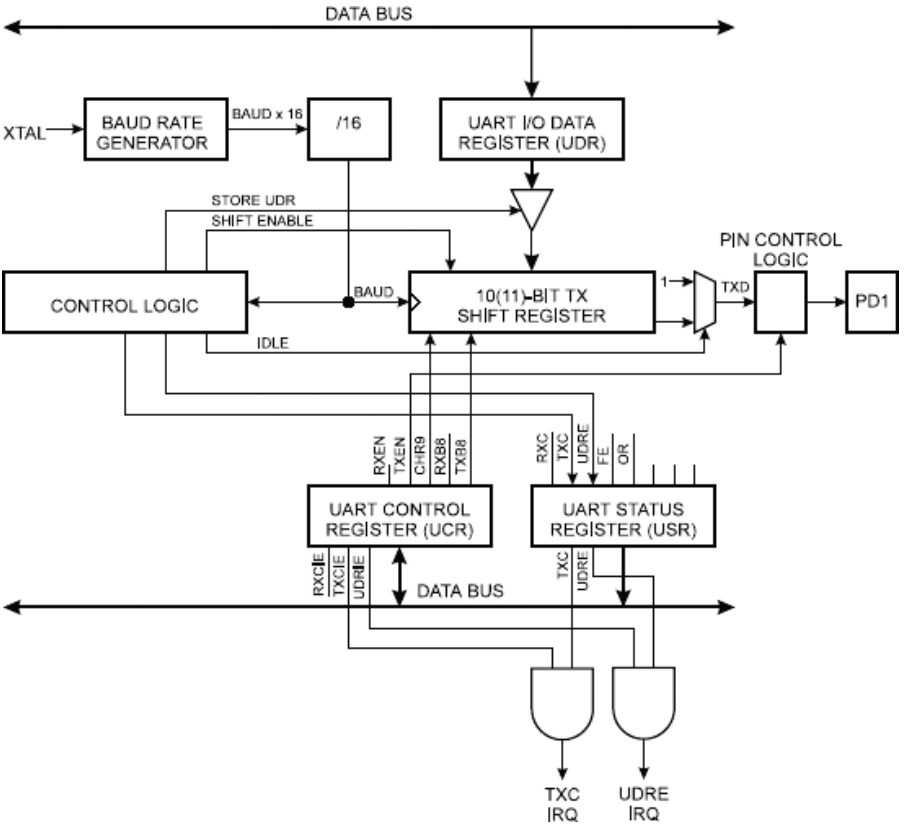


Figure 9.13 UART transmitter.

first. When the stop bit has been shifted out, the shift register is loaded if any new data has been written to the UDR during the transmission. During loading, UDRE is set. If there is no new data in the UDR register to send when the stop bit is shifted out, the UDRE flag will remain set until UDR is written again. When no new data has been written, and the stop bit has been present on TXD for one bit length, the TX Complete Flag, TXC, in USR is set.

The TXEN bit in UCR enables the UART transmitter when set (one). When this bit is cleared (zero), the PD1 pin can be used for general I/O. When TXEN is set, the UART Transmitter will be connected to PD1, which is forced to be an output pin regardless of the setting of the DDD1 bit in DDRD.

Data Reception

Figure 9.14 shows a block diagram of the UART Receiver.

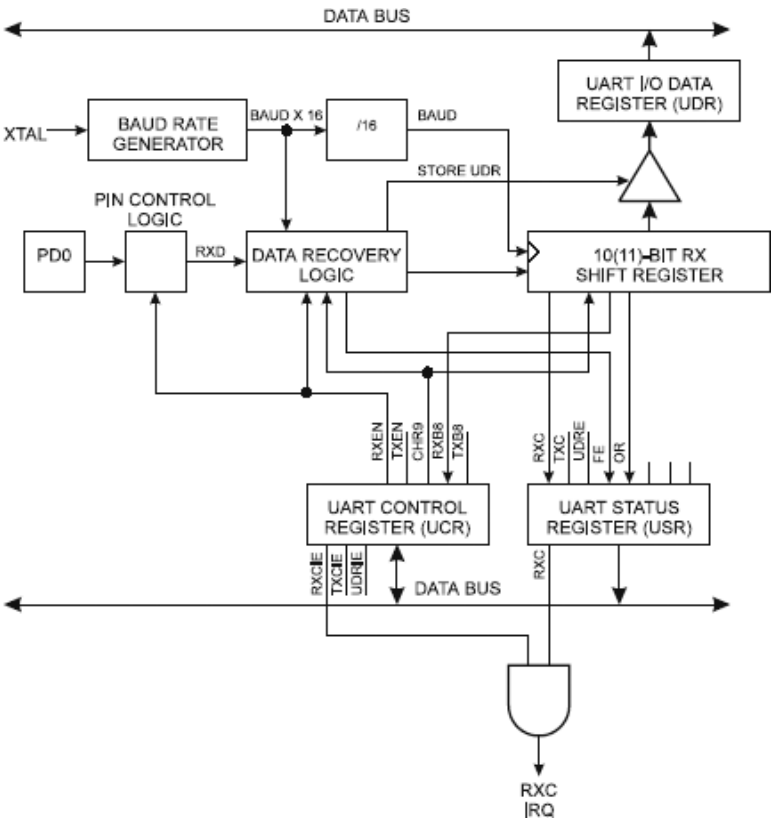


Figure 9.14 UART Receiver.

The asynchronous data recovery technique, explained before, is used to identify the start bit, to read the incoming data bits and the stop bit as mentioned before.

9.4 The EIA-232 Standard

The EIA-232 standard specifies the signals and interconnecting lines between DTE and DCE devices. Figure 9.15 depicts a high-level picture of a communications system. The figure shows clearly what we mentioned before that the RS-232 interface was strictly defined between a piece of DTE, a computer of one form or another, and a piece of DCE, typically a modem.

The full standard specifies that the DTE device uses a 25-pin DB25P (male) connector and the DCE end uses a mating 25 pin DB25S (female)

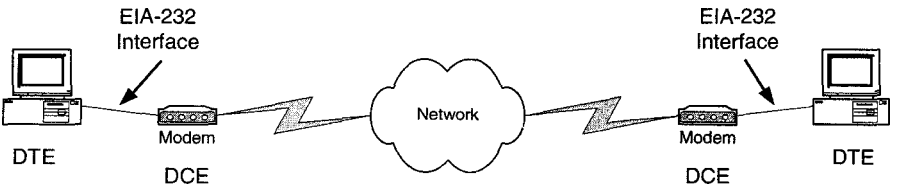
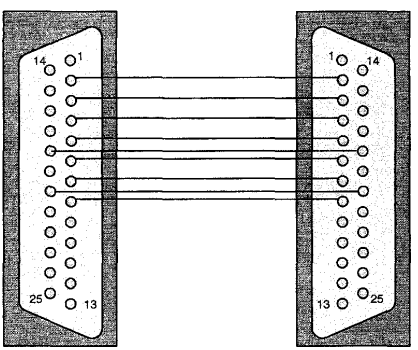
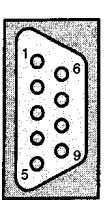


Figure 9.15 A high-level representation of an early communication system.



PIN

2	TXD- Transmitted Data
3	RXD-Received Data
4	TTS- request to Send
5	CTS-Clear to Send
6	DSR- Data Set Ready
7	SG- Signal Ground
8	CD- Carrier Detect
20	DTR- Data Terminal Ready
22	RI- Ring Indicator



1	CD - Carrier Detect
2	TXD - Transmitted Data
3	RXD - Received Data
4	DTR - Data Terminal Ready
5	SG - Signal Ground
6	DSR - Data Set Ready
7	RTS - Request to Send
8	CTS - Clear to Send
9	RI - Ring Indicator

Figure 9.16 Pin Numbering for the DB-9 and DB-25 EIA-232 Connectors.

connector. The standard specifies signals for 22 of the available pins. Today a subset of these appears on a wide variety of computing equipment, with cables terminating in DB9P and DB9S connectors. In either case, the original cabling is parallel, straight through — no crossover connections. The drawings in Figure 9.16 show the pin numbers on the DB-9 and DB-25 connectors as well as the EIA-232 inputs/outputs to which they correspond.

In addition to the lines for exchanging data, the standard also specified a number of handshaking and status lines. In Figure 9.16, the more commonly used signals are given. In the following we give them in more detail.

TXD Data transmission line from DTE to DCE.
RXD Data transmission line from DCE to DTE.
DSR Data Set Ready from DCE to DTE: intended to inform the DTE that the data set has a valid connection, has completed whatever initialization might be necessary, and is ready to engage in a message exchange. If the connection drops during the exchange, this signal is deasserted.
DTR Data Terminal Ready from DTE to DCE: intended to inform the DCE that the data terminal has completed whatever initialization might be necessary, is ready to engage in a message exchange, and would like to open a communication channel.
RTS Request to Send from DTE to DCE: intended to inform the DCE that the DTE had data to send and for the DCE to do whatever was necessary (dial, ensure carrier, ensure a line, etc.) to effect the communication.
CTS Clear to Send from DCE to DTE: intended to inform the DTE that the DCE had done its job and data could now be sent.
CD Carrier detect: intended to indicate that a connection has been established and an answer tone has been received from the remote modem.
SG Signal ground: the name says it.

Timing signals

Some synchronous devices provide a clock signal to synchronize data transmission, especially at higher data rates. Two timing signals are provided by the DCE on pins 15 and 17. Pin 15 is the transmitter clock, or send timing (ST); the DTE puts the next bit on the data line (pin 2) when this clock transitions from OFF to ON (so it is stable during the ON to OFF transition when the DCE registers the bit). Pin 17 is the receiver clock, or receive timing (RT); the DTE reads the next bit from the data line (pin 3) when this clock transitions from ON to OFF.

Alternatively, the DTE can provide a clock signal, called transmitter timing (TT), on pin 24 for transmitted data. Again, data is changed when the clock transitions from OFF to ON and read during the ON to OFF transition. TT can be used to overcome the issue where ST must traverse a cable of unknown length and delay, clock a bit out of the DTE after another unknown delay, and return it to the DCE over the same unknown cable delay. Since the relation between the transmitted bit and TT can be fixed in the DTE design, and since both signals traverse the same cable length, using TT eliminates the issue. TT may be generated by looping ST back with an appropriate phase change

to align it with the transmitted data. ST loop back to TT lets the DTE use the DCE as the frequency reference, and correct the clock to data timing.

9.4.1 Standard Details

In RS-232 data is sent as a time-series of bits. Both synchronous and asynchronous transmissions are supported by the standard. In addition to the data circuits, the standard defines a number of control circuits used to manage the connection between the DTE and DCE. Each data or control circuit only operates in one direction, i.e. signaling from a DTE to the attached DCE or the reverse. Since transmit data and receive data are separate circuits, the interface can operate in a full duplex manner, supporting concurrent data flow in both directions. The standard does not define character framing within the data stream, or character encoding.

RTS/CTS handshaking

Handshaking is a form of flow control. It is a way for one piece of DTE to synchronize actions or the exchange of data with another piece of DTE. For example, if data is sent to a printer at a rate higher than the printer can handle because of the speed of printing, it will send a signal to the sending device to stop until it catches up.

In older versions of the specification, the signals CTS and RTS were used for handshaking; The DTE asserts RTS to indicate a desire to transmit to the DCE, and the DCE asserts CTS in response to grant permission. This allows for half-duplex modems that disable their transmitters when not required, and must transmit a synchronization preamble to the receiver when they are re-enabled. This scheme is also employed on present-day RS-232 to RS-485 converters, where the RS-232's RTS signal is used to ask the converter to take control of the RS-485 bus — a concept that doesn't otherwise exist in RS-232. There is no way for the DTE to indicate that it is unable to accept data from the DCE.

EIA-232 Addressing

Because the original focus of the RS-232 interface was the exchange between a single piece of DTE and a single piece of DCE, addressing was unnecessary. Addressing at a higher level between systems connected through modems to the telephone system occurred naturally through the corresponding telephone numbers.

Today, the majority of EIA-232 interfaces neither use a modem nor the telephone system. Consequently, source and destination addressing must be

implemented as part of the message exchange protocol utilized by the specific application. This standard makes no provision for such addressing.

Configuring the Interface

The EIA interface can be implemented either as an integrated component within a microprocessor or microcomputer or externally using an LSI UART/USART (Universal Synchronous/Asynchronous Receiver Transmitter). The external device may be a peripheral processor that is a component included in the supporting chip set of the microcomputer or a general-purpose device that is provided by another vendor.

The EIA device must still be configured to ensure that the sender and the receiver are using the same transmission protocol; the same transmission parameters. The typical minimum set of parameters that must be configured are: baud rate, bits per character, parity and number of stop bits.

Data Recovery and Timing

As explained before, the SR-232 data exchange is accomplished by the framing procedure explained before. Each character is framed by a start and a stop bit as illustrated in Figure 9.4.

The start bit signal is used to enable the receiving device to temporarily synchronize with the transmitting device. The stop bit signals the end of the data character and provides time for the receiving device to get ready for the next one. RS-232 use the mechanism explained before while discussing UART to recover the data.

EIA-232 Interface Signals

The EIA-232 data and control signals along with their voltage levels are summarized in Table 9.1. A logical 1, known as a mark, corresponds to a voltage level between -3 and -15 volts. A logical 0, or space, corresponds to a voltage level between $+3$ and $+15$ volts. Such values are intended to improve noise immunity of the system.

The internal logic levels in most contemporary embedded systems remain at the 5.0 VDC level. Consequently, outgoing and incoming EIA-232 signals must be buffered and/or level shifted from or to standard logical levels before

Table 9.1 EIA-232 Levels.

	Interchange Voltage	
Notion	Negative	Positive
Binary State	1	0
Signal Condition	Marking	Spacing
Function	OFF	ON

they can be used. Finally, note that the data link remains in the marking state ($< -3\,\text{V}$) until the start bit, a space ($> +3\,\text{V}$), is sent.

9.4.2 Implementation Examples

In the following, we give two examples to explain the relation between UART and RS-232 and how to use them in real world.

Example 9.4:

This example is a general one showing a typical interface between microprocessor and an external UART device. The block diagram in this case is shown in Figure 9.17. The UART device appears for the microprocessor either as a memory mapped I/O device or as a special purpose peripheral processor.

In the block diagram, the UART is a peripheral device sitting on the microprocessor bus. The address lines serve the dual purpose of selecting the UART and identifying registers internal to the device that must be written to configure the device. The data lines are bidirectional and contain data that is to be transmitted from or received by the UART. Each exchange is accompanied by either an *RD* strobe to read a received character from the UART or a *WR* strobe to send one. The *Data Rec*. line can be either polled or used to generate an interrupt when a character has been received. The *Tx Rdy* line indicates that the internal transmit buffer of the UART is empty and a new character can be transmitted. *Like Data Rec.*, the line can either be polled or used as an interrupt.

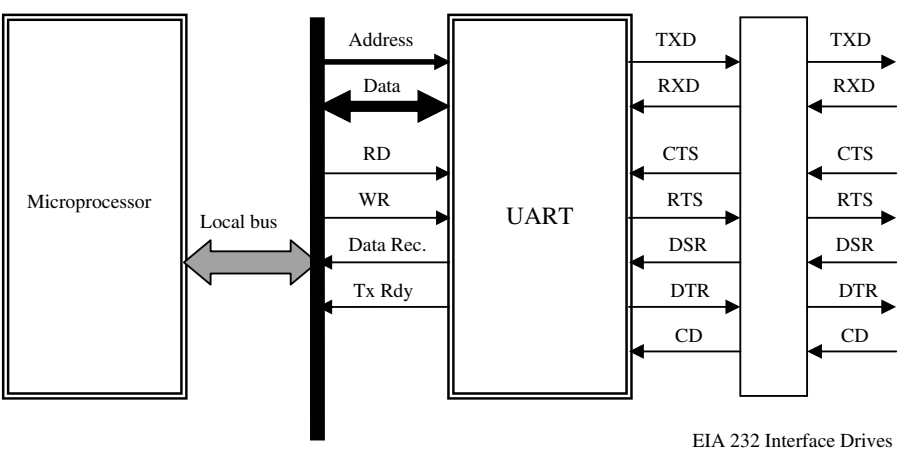


Figure 9.17 Block diagram for a UART type device.

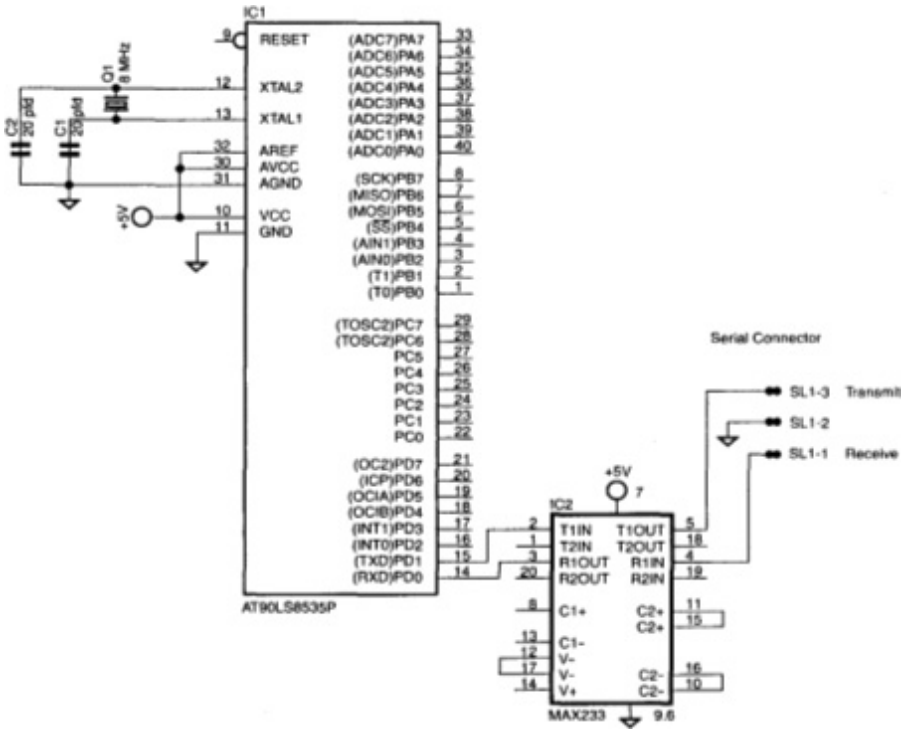


Figure 9.18 Serial communication example hardware.

Example 9.5: In this example we consider a simple system to implement serial communication between a microcontroller and a PC using RS-232. The hardware and software implementation of such a system are shown in Figures 9.18 and 9.19 respectively.

Figure 9.18 shows a standard AVR processor with an attached MAX233 media driver to convert the TTL serial signals used by the microcontroller to the RS-232 signals used for serial communication to the PC. The hardware shown in this figure is used with the software shown in Figure 9.19 to make the serial communication example.

The serial communication example software, shown in Figure 9.19, is a programme that uses a switch construct to print one of three messages on the PC. The exact message will depend on whether an '*a*', or '*b*', or another key is pressed on the PC. The ASCII code for the character pressed is sent serially to the microcontroller. This example demonstrates both manual and interrupt handling of UART functions.

```
#include <90S8535.h>
unsigned char qcntr, sndcntr;         //indexes into the que
unsigned char queue[50];                      //character queue

char msg1 [ ] = {"That was an a."};
char msg2 [ ] = {"That was a b, not an a."};
char msg3 [ ] {"That was neither b nor a."};

interrupt [UART_TXC] void UART_transmit_isr (void)
{
       if (qcntr  != sndcntr)  UDR = queue[sndcntr++];
                     //send next character and increment index
}
void sendmsg (char *s)
{
       qcntr = 0;                     //preset indices
       sndcntr = 1;                   //set to one because first character already
sent
       queue [qcntr++] = 0x0d;        //put CRLF into the queue first
       queue [qcntr++] = 0x0a;
       while (*s)  queue [qcntr++] = *s++;//put characters into queue
       UDR = queue [0];       //send first character to start process
}

void main (void)
{
       UCR = 0x58;    //enable receiver, transmitter
                      //and transmit interrupt
       UBRR = 0x33;           //baud rate = 9600
       # asm("sei")           //global interrupt enable

       while (1)
       {
              if (USR & 0x80)        //check for character received
              {
                  ch = UDR;          //get character sent from PC
                  switch (ch)
                 {
                      case 'a':
                          sendmsg (msg1);    //send first message
                          break;
                      case 'b':
                          sendmsg (msg2);    //send second message
                          break;
                      default:
                          sendmsg (msg3);    //send default message
                  }
              }
       }
}
```

Figure 9.19 Serial communication example (Software).

The first two lines in main() initialize the UART The UCR is loaded with 0×58 to enable the receiver, the transmitter, and the transmit interrupt. The UBRR is set to 0×33 to set the baud rate to 9600. Once the UART is set up and the interrupt is enabled, the programme enters a while (1) loop that continuously checks for a received character using an If statement, whose expression will evaluate to TRUE after a character is received. Receiving the character will set the RXC bit in the USR, which is being tested by the If statement. This is an example of manually polling the status of the serial port to determine when a character is received.

When the If statement determines that a character has been received, the character is read from the UDR and used in a switch statement to determine which message to send. The message being transmitted (such as "That was an a.") is composed of several characters, each of which must be sent serially to the PC along with a carriage return, CR, and a line feed, LF, character so that each message will start on a new line. The longest message is composed of 25 characters, and adding the CR and LF characters makes the total message 27 characters long. At 9600 baud (1.04 ms per serial byte) this message will take over 27 milliseconds to transmit. It is not appropriate for the microcontroller to wait around for 27 milliseconds while this message is being sent, because it could be executing as many as 216,000 instructions (8 instructions per microsecond $*27$ milliseconds $*1000$ microseconds per milliseconds $= 216{,}000$ instructions). In order to free the microcontroller from the need to wait while messages are being transmitted, a FIFO queue is used in conjunction with the transmit interrupt to actually transmit the message.

A queue is a temporary holding device to hold bytes of data and return them on a first in, first out (FIFO) basis. In this case, the queue is mechanized as a variable array of data called, appropriately, "queue". An index, "qcntr", is used to indicate where new data is to be placed in the queue. As each new byte is added to the queue, the index is incremented so the next byte is added in sequence and so on, until the queue holds all of the necessary data.

A separate index, "sndcntr", is used to retrieve the data from the queue. As each byte is retrieved, this index is incremented and finally, when the two indices are equal, the programme knows that the queue has been emptied.

Actually, the CodeVisionAVR C language compiler can provide either a transmitter queue or a receiver queue or both using the CodeWizardAVR code generator feature. This example is provided to demonstrate how the queue works for educational reasons.

Getting back to the example programme, the sendmsg() function called from the switch state ment puts the message in the queue and starts the transmit

function. The switch statement passes a pointer (an address) to the appropriate message when it calls the function. The function first puts the CR and LF characters in the queue and then puts the message to be transmitted in the queue using the pointer. Finally, the function writes the first byte in the queue into the UDR to start the transmission process. After this character has been transmitted, the TXC interrupt occurs, the ISR loads the next character from the queue into the UDR, and so the cycle continues until the queue is empty, as indicated by the two indices being equal.

9.5 Inter-Integrated Circuits (I2C)

The most popular form of serial bus protocol is "I2C", which stands for "Inter-Integrated Circuit". Sometimes the bus is called IIC or I^2C bus.This standard was originally developed by Philips Semiconductors in the late 1970s as a method to provide an interface between microprocessors and peripheral devices without wiring full address, data, and control busses between devices; more specifically, it was designed to provide an easy way to connect a CPU to peripheral chips in a TV-set.

The problem that faced the designers at that time, and till now facing many of them, is that the peripheral devices in embedded systems are often connected to the microcomputer unit (normally a microcontroller) as memory-mapped I/O devices, using the microcontroller's parallel address and data bus. This result in lots of wiring on the PCB's to route the address and data lines. The number of wires increases if we consider the need to connect also a number of address decoders and glue logic to connect everything. In mass production items such as TV-sets, VCR's and audio equipment, this is not acceptable. In such applications every component that can be saved means increased profitability for the manufacturer and more affordable products for the end customer. Another problem directly related to the use of a lot of wires is the possible interference: lots of control lines imply that the system is more susceptible to disturbances by Electromagnetic Interference (EMI) and Electrostatic Discharge (ESD).

The need for a more effective and economical way of connecting different devices (normally ICs) resulted in introducing the 2-wire communication bus that we call it now I2C bus protocol. This protocol enables peripheral ICs in electronic systems to communicate with each other using simple communication hardware. The ICs can be on the same board or linked via cables. The length of the cable is limited by the total bus capacitance and the noise generated on the bus.

Today, I2C has become a de facto world standard that is now implemented in over 1000 different ICs and is licensed to more than 50 companies including some of the leading chip manufacturers like Xicor, ST Microelectronics, Infineon Technologies, Intel, Texas Instruments, Maxim, Atmel, Analog Devices and others.

9.5.1 The I2C Bus Hardware Structure

The I2C bus physically consists of 2 active wires: a data line wire called serial-data-line (SDA) and a clock wire called serial-clock-line (SCL), and a ground connection, as shown in Figure 9.20. Both SDA and SCL are bi-directional. This means that in a particular device, these lines can be driven by the IC itself or from an external device. To achieve this functionality, these signals use open-collector or open-drain outputs (depending on the technology).

The bus interface is built around an input buffer and an open drain or open collector transistor (Figure 9.20). When the bus is IDLE, the bus lines are in the logic HIGH state. This is achieved by using an external pull-up resistor, which is a small current source. To put a signal on the bus, the chip drives its output transistor, thus pulling the bus to a LOW level.

The two wires are acting accordingly as wired-AND. This concept achieves a "built-in" bus mastering technique. If the bus is "occupied" by a chip that is sending a 0, then all other chips lose their right to access the bus. This will be used for "bus arbitration".

Electrical Consideration

Some electrical considerations must be taken by the designer of an I2C system. The two main considerations are:

- The electrical considerations when connecting different devices to I2C bus.
- The electrical considerations that limits the length of the bus.

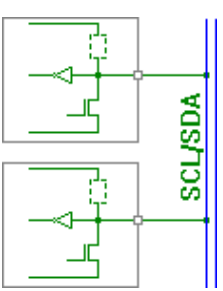


Figure 9.20 I2C Bus.

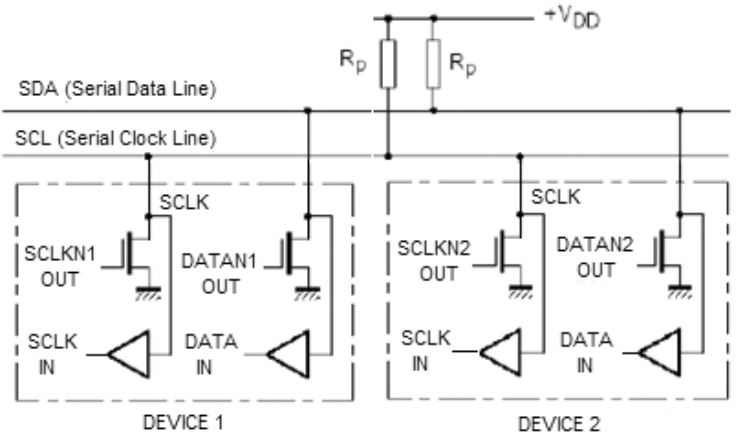


Figure 9.21 Connecting devices of the same type to form I2C system.

a. Types of Devices that can be connected to I2C bus

The design of the I2C bus places no restrictions on the type of devices that may be connected to the bus. Figure 9.21 represents the hardware needed to connect two devices of the same logic family (e.g. both are using TTL or both are using CMOS) using I2C bus. The two devices use the same supply voltage. The SCL is used to strobe data transmitted on SDL from or to the master that currently has control over the bus. Because the devices are wired-ANDed, this means that the bus is free if SDA and SCL are high and also that the device output is ANDed with the signal on the bus. This, as mentioned before, represents the basics of achieving the bus arbitration when there is more than one master.

It is possible, in general to configure a system with a mixture of devices from different logic families, i.e. a mixture of various TTL and CMOS logic families. In this case each device has its own different supply voltage. To accommodate bidirectional communication with devices operating on different voltages, e.g. 5 V and 3.5 V, it is necessarily to incorporate into the SCL and SDA lines a form of buffering circuit as shown in Figure 9.22. The two devices, operating at different voltage levels, are now separated from one another.

b. Bus Capacitance Limitation

The open-collector technique in use with I2C has a drawback that limits the length of the bus. Long lines present a capacitive load for the output drivers. Since the pull-up is passive, the *RC constant* of the bus will affect the wave shape of the signals propagating on the bus, which in turn limits the speed that the bus can handle; the higher *RC constant* of the bus, the slower the speed

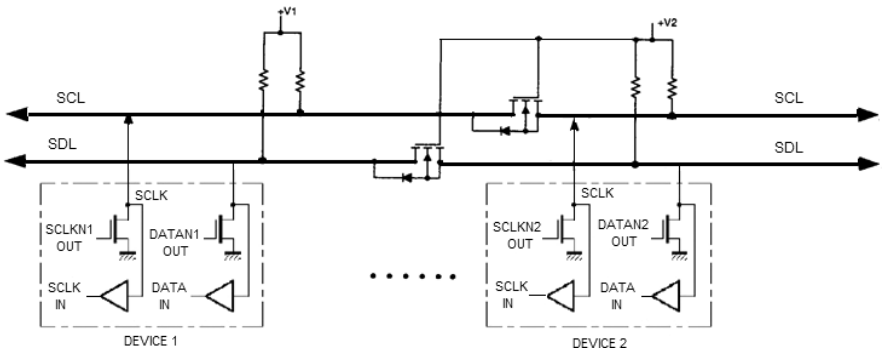


Figure 9.22 I2C bus Supporting Devices Operating on different voltage levels.

of the bus. This results because of the effect of the *RC constant* on the slew rate of the edge of the signals. At a certain point, and because of the slew rate, the ICs will not be able to distinguish clearly between logic 1 and logic 0. Increasing the length of the line has another effect: it increases the possibility of getting reflections at high speed. The reflected waves can corrupt the data on the bus to the extent that it becomes unreadable.

To avoid such problems a number of strict electrical specifications are to be followed when using I2C bus. According to the specification the length of the bus is limited by the total bus capacitance: the total capacitance of the bus remains under 400 pF.

The original I2C specification allows a data transfer rates of up to 100 kbits/s and 7-bit addressing. Seven-bit addressing allows a total of 128 devices to communicate over a shared 1 bus. With increased data transfer rate requirements, the I2C specification has been recently enhanced (Version 1.0 — 1992 and Version 2.0-1998) to include fast-mode, 3.4 Mbits/s, with 10-bit addressing. At present I2C range includes more than 150 CMOS and bipolar I2C-bus compatible types for performing communication functions between intelligent control devices (e.g. microcontrollers), general-purpose circuits (e.g. LCD drivers, thermal sensors, remote I/O ports, RAM, Flash and EPROM memories) and application-oriented circuits (e.g. digital tuning and signal processing circuits for radio and video systems). Other common devices capable of interfacing to an I2C bus include real time clocks and watchdog timers.

9.5.2 Basic Operation: How it works?

In Figure 9.23 we depict a sample of I2C network with four devices attached to the bus. In reality any number of devices may be connected to the bus as long

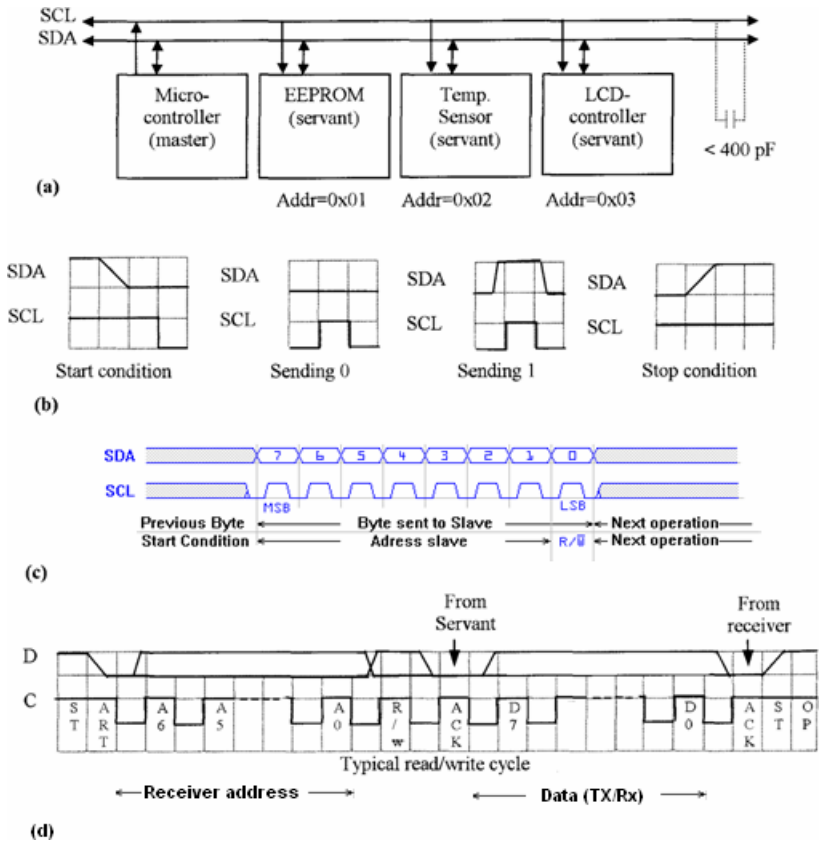


Figure 9.23 I2C Bus.

as its length will keep its total capacitance under 400 pF. The system given in Figure 9.23 is used here to explain the basic operation of I2C. In Figure 8.23, the device that initiates a transaction on the I2C bus is termed the master. The master normally controls the clock signal. A device being addressed by the master is called a slave. In our case, the master is the microcontroller, the other three devices, a temperature sensor, an EEPROM, and a LCD-controller, are slaves. Each of the slave devices is assigned a unique address (Figure 9.23a). The address comprises 7 bits. The four most significant bits (A7 to A3) identify the category of the device being addresses. The three least significant bits (A2 to A0) identify a programmable hardware address assigned to the device. Thus, up to eight instances of the same device can be included in the system. For example, as we are going to see later, the code 1010 is reserved for the serial

EEPROM, and then no more than eight EEPROMs may be connected to the system. The address of all the eight devices starts with 1010 concatenated with one of the combinations 000 to 111. The seven bits of the address is followed by a direction bit (W/R bit) which is used to inform the slave if the master is writing (W/R = 0) to it or reading from it (W/R = 1).

Only master devices can initiate a data transfer on an I2C bus. The protocol is a multi-master bus; it does not limit the number of master devices on an I2C bus, but typically, in a microcontroller-based system, the microcontroller serves as the master. Both master and slave devices can be senders or receivers of data. This will depend on the function of the device. In the example given in Figure 9.23, the microcontroller and EEPROM send and receive data, while the temperature sensor sends data and the LCD-controller receives data. In Figure 9.23(a), arrows connecting the devices to the I2C bus wires depict the data movement direction. Normally, all the slave devices connected to I2C bus assert high-impedance on the bus while the master device maintains logic high, signaling an idle condition.

Frame- Start/Stop conditions: When the I2C bus is in the "idle state", both the clock and the data lines are not being driven and are pulled high. To begin any data transfer on an I2C bus, a "*start*" condition is put on the bus. A **start condition** is shown in Figure 9.23(b). A high to low transition of the SDA line while the SCL signal is held high signals a start condition. All data transfers on an I2C bus are terminated by a "*stop*" condition. A **stop condition** is shown in Figure 9.23(b). A low to high transition of the SDA line while the SCL signal is held high signals a stop condition.

Data Transfer: Actual data is transferred in between start and stop conditions. Data is transmitted in a synchronous fashion, with the most significant bit sent first. Data can only change when SCL line is in the LOW state. A command is sent from the master to receiver in the format shown in Figure 9.23(d). Next we explain the steps needed for a master write and master read operations (normally called bus events).

I2C Bus Events: Transmitting a byte to a slave device

To write to a slave, the write cycle is as follows:

- The master device initiates the transfer by a start condition. This acts as an 'Attention' signal to all of the connected devices. All ICs on the bus will listen to the bus for incoming data.
- Once the start condition has sent, the master sends a byte to the slave. This first byte is used to identify the slave on the bus and to select the mode of operation. As shown in Figure 9.23c, the 7 bits forming the address of the

slave starting with the most significant bit (MSB) is send at first followed by the eighth bit that defines the direction of the data. The eighth bit is labeled R/W in the figure with 1 for "read" and 0 for "write". Here, the bit value is placed on the SDA line by the master device while the SCL line is low and maintained stable until after a clock pulse on SCL.

- If performing a write, directly after sending the address of the receiving device, the master sends a zero.
- Having received the address, all the ICs will compare it with their own address. If there is no match, the ICs will ignore the rest of this transaction and wait for the next, i.e. wait until the bus is released by the stop condition. On the other hand, if the address matches, the receiving device will respond by producing an acknowledge signal. The receiving device produces this signal by holding the SDA line low during the first ACK clock cycle.
- Having sent the I2C address (and the internal register address) the master can now send the data byte (or bytes). The master device transmits a byte of data starting with the most significant bit down to the least significant bit. The receiving device, in this case one of the slaves, acknowledges the reception of data by holding the SDA line low during the second ACK clock cycle. This means that for every 8 bits transferred, the device receiving the data sends back an acknowledge bit, so there are actually 9 SCL clock pulses to transfer each 8 bit byte of data. If the receiving device sends back a low ACK bit, then it has received the data and is ready to accept another byte. If it sends back a high, then it is indicating it cannot accept any further data and the master should terminate the transfer by sending a stop sequence.

I2C Bus Events: Read a byte from a slave device

To read from a slave device, the read cycle will be as follows:

- If performing a read operation, the master initiates the transfer by a start condition, sends the address of the device that is being read, sends a one (logic high on SDA line) requesting a read and waits to receive an acknowledgment.
- The protocol format is the same as in transmitting a byte to a slave, except that now the master is not allowed to touch the SDA line. Prior to sending the 8 clock pulses needed to clock in a byte on the SCL line, the master releases the SDA line. The slave will now take control of this line. The line will then go high if it wants to transmit a '1' or, if the slave wants to send a '0', remain low.

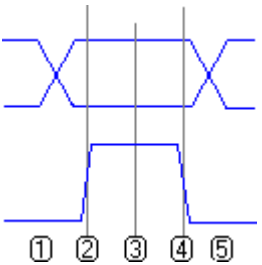


Figure 9.24 Read event.

- All the master has to do (see Figure 9.24) is generate a rising edge on the SCL line (2), read the level on SDA (3) and generate a falling edge on the SCL line (4). The slave will not change the data during the time that SCL is high. (Otherwise a Start or Stop condition might inadvertently be generated.). During (1) and (5), the slave may change the state of the SDA line.
- In total, this sequence has to be performed 8 times to complete the data byte.

The meaning of all bytes being read depends on the slave. There is no such thing as a "universal status register". The user needs to consult the data sheet of the slave being addressed to know the meaning of each bit in any byte transmitted

Possible modifications on the timing diagram: Repeat start condition

Figure 9.23(d) depicts a timing diagram of a typical read/write cycle. Sometimes this timing diagram may change. Some of the possible changes are (See Figure 9.25):

- ***Acknowledge Extension:*** Under normal circumstances, following the ACK bit time, the master will release the SCL line so that transmission may continue with the next byte. If, however, the receiver (which is usually a “slave device” or “master”) is temporarily unable to proceed, it will hold the SCL line LOW, thereby extending the ACK interval. In other words it is allowed for the acknowledge bit to float high. When the receiver is ready able to proceed again, it will release the SCL line and transmission continues. The timing diagram in Figure 9.25 illustrates the ACK interval extension.
- ***Multi-register Slaves:*** While explaining the write/read cycle we assumed that the slave is simple and has only one register. In such a case just sending the address of the slave is enough. In many cases, the slave has

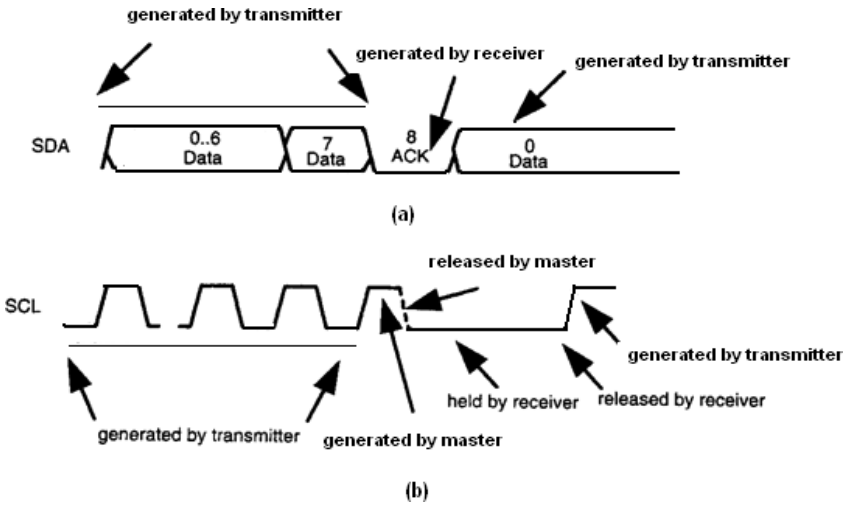


Figure 9.25 Acknowledge Extension (a) without, (b) with extension.

many registers each has its own address within the slave device. If the master wants to write at specific register within the slave, it must send this address. In such cases, having addressed the slave device the master must now send out the internal location or register number inside the slave that it wishes to write to or read from. This number is obviously dependant on what the slave actually is and how many internal registers it has. In such a case the master can continue to send data bytes to the slave and these will normally be placed in the following registers because the slave will automatically increment the internal register address after each byte. When the master has finished writing all data to the slave, it sends a stop sequence which completes the transaction.

- ***Repeated Start:*** During an I2C transfer there is often the need to first send a command and then read back an answer right away. This has to be done without the risk of another (multimaster) device interrupting this atomic operation. The I2C protocol defines a so-called repeated start condition. After having sent the address byte (address and read/write bit) the master may send any number of bytes followed by a stop condition. If, however, it wishes to establish a connection with a different slave, rather than issue the Stop, the master will issue another Start, using the address of the new device and then sends more data. This is defined recursively allowing any number of start conditions to be sent. The purpose of this is, in general, to allow combined write/read operations to one or more devices without

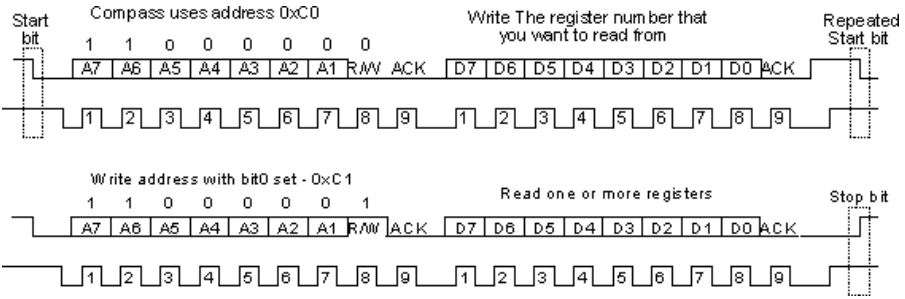


Figure 9.26 Multiple address.

releasing the bus and thus with the guarantee that the operation is not interrupted.

- Also we have to mention here that, in some devices, a start bit has to be resent to reset the receiving device for the next command e.g., in a serial EEPROM read, the first command sends the address to read from, the second reads the data at that address

Multiple address is shown in Figure 9.26.

I2C Addresses Standard: Special Addresses and Exceptions

The first byte of an I2C transfer contains the slave address and the data direction (Figure 9.23). The address is 7 bits long, followed by the direction bit (W/R bit). A 7-bit wide address allows, theoretically 128 I2C addresses — however, some addresses are reserved for special purposes. Only 112 addresses are available with the 7 bit address scheme.

As a matter of fact, I2C gives a loose standard for the address. It uses the most significant four bits to identify the type of the device and the next three bits are used to specify one of eight devices of this type (or further specify the device type). Also, some devices require certain patterns for the last three bits, while others (such as large serial EEPROMs) use these bits to specify an address inside the device. This shows the importance of mapping out the devices to be put on the bus and all their addresses.

In the I2C address map there are also what is called “reserved addresses”. Table 9.2 shows the special addresses

9.5.3 I2C Modes

There are applications where the I2C transfer speed is a limiting factor. To allow for higher transmission rates while retaining a certain amount of

Table 9.2 I2C special addresses.

Address	R/W	Designation
0000-000	0	General Call address
0000-000	1	START byte
0000-001	X	reserved for the (now obsolete) C-Bus format
0000-010	X	Reserved for a different bus format
0000-011	X	Reserved for future purposes
0000-1xx	X	High Speed master code
1111-1xx	X	Reserved for future purposes
1111-0xx	X	10-bit slave addressing mode

compatibility Philips in 1998 has introduced the HS I2C standard. Before discussing the high speed mode (Hs-mode) we introduce at first the other I2C modes.

a. Low-Speed Mode

The very first specification of I2C dates back to the year 1982. It only covered Standard mode: up to 100 kbit/s and 7-bit addressing.

b. Enhanced I2C (Fast Mode)

In the FAST mode, the physical bus parameters are not altered. The protocol, bus levels, capacitive load etc., remain unchanged. However, the data rate has been increased to 400 Kbit/s and a constraint has been set on the level of noise that can be present in the system. To accomplish this task, a number of changes have been made to the I2C bus timing.

c. High Speed Mode (Hs-mode)

High-speed mode (Hs-mode) devices offer a quantum leap in I2C-bus transfer speeds. Hs-mode devices can transfer information at bit rates of up to 3.4 Mbit/s, yet they remain fully downward compatible with Fast- or Standard-mode (F/S-mode) devices for bi-directional communication in a mixed-speed bus system. With the exception that arbitration and clock synchronization is not performed during the Hs-mode transfer, the same serial bus protocol and data format is maintained as with the F/S-mode system. Depending on the application, new devices may have a Fast or Hs-mode I2C-bus interface, although Hs-mode devices are preferred as they can be designed-in to a greater number of applications.

Transmission Format of High Speed Mode

The addressing scheme for high speed transfers differs from the normal addressing procedure. In Figure 9.27, we show the beginning of a high speed transfer.

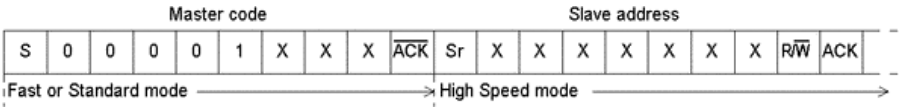


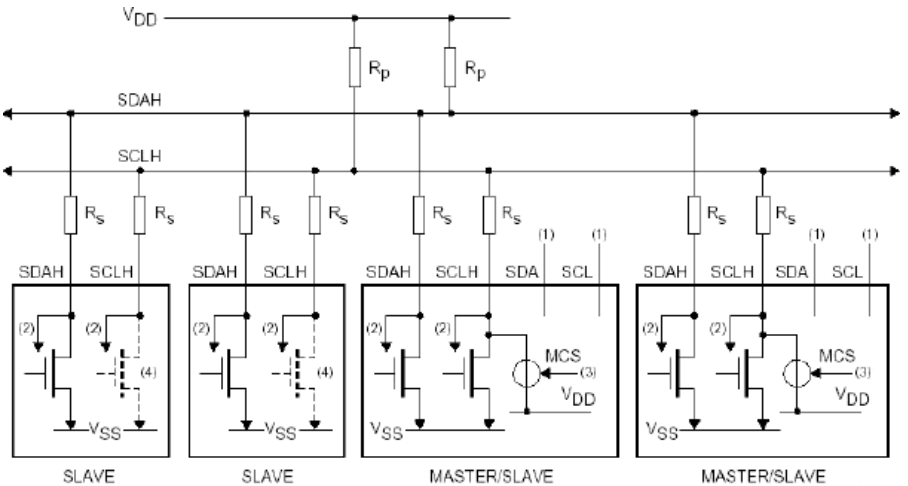
Figure 9.27 High speed transfer.

The format is as follows:

- Start condition is sent.
- After the start condition a so-called master code is transmitted '00001XXX' (See Table 9.2), followed by a mandatory not-acknowledge bit. The master code is sent in Fast- or Standard-mode (this is with at most 400 bkit/s).
- The three lowest bits are used to identify different I2C masters on the same bus — each one has its unique identifier. During transmission of the master code, arbitration takes place (see arbitration latter), so that only the winning master can perform the following high speed transfer. The winning master is the "active master"
- The master codes are selectable by the designer and allow up to eight high speed masters to be connected in one system (master code '00001000' should be reserved for test and diagnostic purposes).
- Active master switches to high-speed communication.
- Arbitration and clock synchronization only take place during master code transmission, not during HS transfer
- After the acknowledge phase following the master code, the high speed transfer begins with a repeated start condition, followed by the slave address and the succeeding data, just like in Fast or Standard mode, but with a higher bit rate.
- After each address the slave must respond by *ack* or *nack* signal
- The current source circuit is disabled after each repeated start condition and after each *ack* or *nack* to give slaves a chance to stretch the clock
- The high speed mode remains active until a stop condition is transmitted, on which the connected high speed devices switch back to slow transmission rates like Fast or Standard mode.
- All devices return to fast mode operation after a stop condition is sent

Electrical Characteristics of HS mode

Figure 9.28 shows the physical I2C-bus configuration in a system with only Hs-mode devices. The pins, SDA and SCL, on the master devices are



(1) SDA and SCL are not used here but may be used for other functions.
(2) To input filter
(3) Only the active master can enable its current-source pull-up circuit
(4) Dotted transistors are optional open-drain outputs which can stretch the serial signal SCLH

Figure 9.28 HS-device connection.

only used in mixed-speed bus systems and are not connected in an Hs-mode only system. In such cases, these pins can be used for other functions.

9.5.4 I2C as a Multi-Master Bus: Bus Arbitration

So far we have seen the operation of the I2C bus from the master's point of view and using only one master on the bus. The I2C bus was originally developed as a multi-master bus. This means that more than one device initiating transfers can be active in the system.

When using only one master on the bus there is no real risk of corrupted data, except if a slave device is malfunctioning or if there is a fault condition involving the SDA/SCL bus lines. This situation changes when using more than one master, e.g. use of the 2 microcontrollers units CPU1 and CPU2 in Figure 9.29.

When CPU 1 issues a start condition and sends an address, all slaves will listen (including CPU 2 which at that time is considered a slave as well). If the address does not match the address of CPU 2, this device has to hold back any activity until the bus becomes idle again after a stop condition.

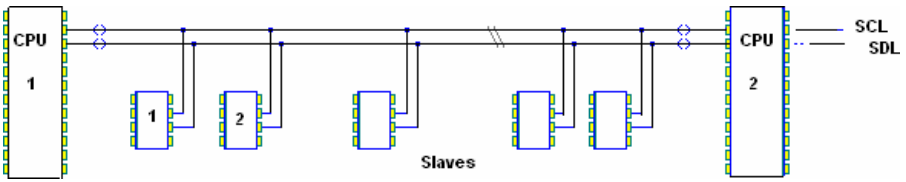


Figure 9.29 Multi-master bus.

As long as the two CPUs monitor what is going on the bus (start and stop) and as long as they are aware that a transaction is going on because the last issued command was not a STOP, there is no problem. The problems arise if the two CPUs decided to initiate the "start condition" at the same time or if one of the CPUs missed the start condition and still think that the bus is idle. As a fact this is not the only problem that may arise when a system has multiple masters. A second problem that can arise is the case when having multiple clocks in the system. The first problem is resolved by "***arbitration***" and the second by "***synchronization***". The two problems and their solution are discussed next.

9.5.4.1 Arbitration

For proper functioning in case of multi master, each device needs to be able to cooperate with the fact that another device is currently talking and the bus is therefore busy. This can be translated into:

- *Being able to follow arbitration logic.* If two devices start to communicate at the same time the one writing more zeros to the bus (or the slower device) wins the arbitration and the other device immediately discontinues any operation on the bus.
- *Bus busy detection.* Each device must detect an ongoing bus communication and must not interrupt it. This is achieved by recognizing traffic and waiting for a stop condition to appear before starting to talk on the bus.

The physical I2C bus setup is designed not only to help the devices to monitor what is going on the bus but more importantly, it is designed to prevent any risk of data corruption that may arise from data collision. The bus monitoring and collision avoidance is discussed next.

Bus monitoring:

The I2C bus structure is a wired AND. This means that if one device pulls a line low it stays low and accordingly any device can test if the bus is idle or

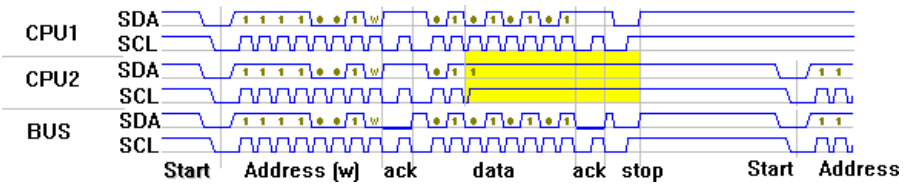


Figure 9.30 Bus arbitration.

occupied. When a master (the sender) changes the state of a line to HIGH, it MUST always check that the line really has gone to HIGH. If it stays low then this is an indication that the bus is occupied and some other device is pulling the line low.

Therefore the general rule of thumb in I2C bus is: If a master (a sender) cannot get a certain line to go high, it lost arbitration and needs to back off and wait until a stop condition is seen before making another attempt to start transmitting.

Possibility of Collision:

Since the previous rule says that a master loses arbitration when it cannot get either SCL or SDA to go high when needed, the problem of data corruption (or data collision) does not exist. It is the device that is sending the '0' that rules the bus. One master cannot disturb the transmission of other master because if it cannot detect one of the lines to go high, it backs off, and if it is the other master that cannot do so, it will behave the same.

This kind of back-off condition will only occur if the two levels transmitted by the two masters are not the same. As an example, let's consider Figure 9.30, where two CPUs start transmitting at the same time.

The two CPUs are accessing a slave in write mode at address 1111001. The slave acknowledges this. So far, both masters are under the impression that they "own" the bus. Now CPU1 wants to transmit 01010101 to the slave, while CPU2 wants to transmit 01100110 to the slave. The moment the data bits do not match anymore (because what the CPU sends is different than what is present on the bus) one of them loses arbitration and backs off. Obviously, this is the CPU which did not get its data on the bus. For as long as there has been no STOP present on the bus, it won't touch the bus and leave the SDA and SCL lines alone (shaded zone). The moment a STOP was detected, CPU2 can attempt to transmit again.

From the example above we can conclude that it is the master that is pulling the line LOW in an arbitration situation that always wins the arbitration. The master which wanted the line to be HIGH when it is being pulled low by

the other master loses the bus. We call this a loss of arbitration or a back-off condition. When a CPU loses arbitration, it has to wait for a STOP condition to appear on the bus. Then it knows that the previous transmission has been completed.

9.5.4.2 Clock Synchronization and Handshaking

Clock Synchronization:

All masters generate their own clock on the SCL line to transfer messages on the I2C-bus. Data is only valid during the HIGH period of the clock. A defined clock is therefore needed for the bit-by-bit arbitration procedure to take place.

Clock synchronization is performed using the wired-AND connection of I2C interfaces to the SCL line. This means that a HIGH to LOW transition on the SCL line will cause the devices concerned to start counting off their LOW period and, once a device clock has gone LOW, it will hold the SCL line in that state until the clock HIGH state is reached. However, the LOW to HIGH transition of this clock may not change the state of the SCL line if another clock is still within its LOW period. The SCL line will therefore be held LOW by the device with the longest LOW period. Devices with shorter LOW periods enter a HIGH wait-state during this time, see Figure 9.31.

When all devices concerned have counted off their LOW period, the clock line will be released and go HIGH. There will then be no difference between the device clocks and the state of the SCL line, and all the devices will start counting their HIGH periods. The first device to complete its HIGH period will again pull the SCL line LOW. In this way, a synchronized SCL clock is generated with its LOW period determined by the device with the longest clock

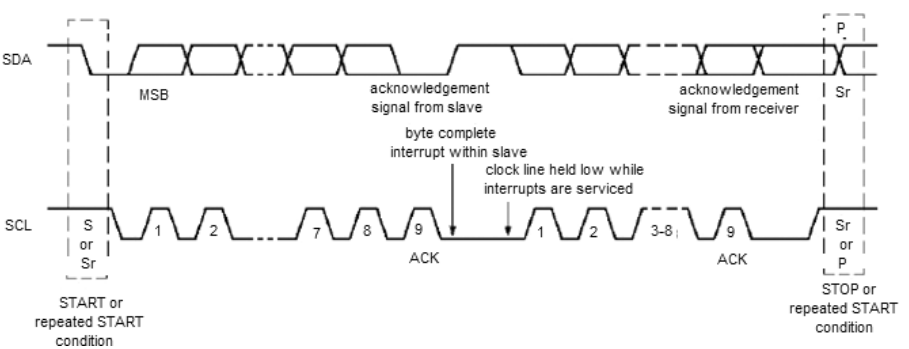


Figure 9.31 Clock synchronization.

LOW period, and its HIGH period determined by the one with the shortest clock HIGH period.

Handshaking: Using the clock synchronizing mechanism as a handshake

The I2C protocol also includes a synchronization mechanism, which can be used as a handshake mechanism between slow and fast devices or between masters in a multi-master session.

When a slow slave (slow in terms of internal execution) is attached to the bus then problems may occur. Let us consider a serial EEPROM. The actual writing process inside the EEPROM might take some time. Now if you send multiple bytes to such a device, the risk exists that you send new data to it before it has completed the write cycle. This would corrupt the data or cause data loss.

The slave must have some means to tell the master that it is busy. It could of course simply not respond to the acknowledge cycle. This would cause the master to send a stop condition and retry. (That is how it is done in hardware in EEPROMs.) Other cases might not be so simple. Think about an A/D converter. It might take some time for the conversion to complete. If the master would just go on it would be reading the result of the previous conversion instead of the newly acquired data.

Now the synchronization mechanism can come in handy. This mechanism works on the SCL line only. The slave that wants the master to wait simply pulls the SCL low as long as needed. This is like adding "wait states" to the I2C bus cycle. The master is then not able to produce the acknowledge clock pulse because it cannot get the SCL line to go high. Of course the master software must check this condition and act appropriately. In this case, the master simply waits until it can get the SCL line to go HIGH and then just goes on with whatever it was doing.

There are a number of minor drawbacks involved when implementing this. If the SCL gets stuck due to an electrical failure of a circuit, the master can go into deadlock. Of course this can be handled by timeout counters. Plus, if the bus gets stuck like this, the communication is not working anyway.

Another drawback is speed. The bus is locked at that moment. If you have rather long delays (long conversion time in our example above), then this penalizes the total bus speed a lot. Other masters cannot use the bus at that time either.

This technique does not interfere with the previously introduced arbitration mechanism because the low SCL line will lead to back-off situations in other devices which possibly would want to "claim" the bus. So there is no real

drawback to this technique except the loss of speed / bandwidth and some software overhead in the masters.

It is possible to use this mechanism between masters in a multi-master environment. This can prevent other master from taking over the bus. In a two-master system this is not useful. But as soon as the system has three or more masters this is very handy. A third master cannot interrupt a transfer between master 1 and 2 in this way. For some mission-critical situations this can be a very nice feature.

It is possible to make this technique more rigid by not pulling only the SCL line low, but also the SDA line. Then any master other than the two masters talking to each other will immediately back off. Before continue, the master must first let SDA go back high, and then SCL, representing a stop condition. Any master which attempted to communicate in the meantime would have detected a back-off situation and would be waiting for a STOP to appear

9.5.5 Applications Using I2C Bus

Today, a great variety of integrated circuit devices, including a plethora of different microprocessors and microcontrollers, support the 12C bus. Such devices range from displays and serial EEPROMS to analogue-to-digital converters and video acquisition systems. The bus can provide a highly effective lower speed network for locally distributed embedded applications. The extent of the topographic distribution is subject to the 400 pf capacitive loading specification. Such a constraint suggests 20 to 30 devices and a maximum signal path of approximately 10 meters.

9.6 Controller Area Network (CAN)

The ***controller area network*** (CAN) bus is a broadcast serial communication standard for real-time applications that allows multiple processors in a system, possibly connected over a twisted pair multidrop cable, to communicate efficiently with each other. CAN does not use the conventional address scheme for identifying the devices; Data messages transmitted from any node on a CAN bus do not contain addresses of either the transmitting node (data source), or of any intended receiving node (destination node). Instead, CAN broadcasts the messages to all the nodes in the network using a unique identifier for the contents of the message. The identifier is unique throughout the network. All the nodes receive the message and each node runs an acceptance test on the identifier to determine if the contents of the message are relevant to it or not.

Any node finds that the contents are relevant to it, this node processes the message otherwise it ignores the message.

CAN uses the identifier also to determine the priority of the message in terms of competition for bus access. The lower the numerical value of the identifier, the higher the priority of the message. CAN standards use the identifier to avoid collision on the bus. This technique is used by CAN to achieve that is called "non-destructive arbitration". It guarantees that in situations where two or more nodes attempt to transmit at the same time, the messages are sent in order of priority and that no messages are lost.

Controller area network was first developed by Robert Bosch in 1986. It is documented in ISO 11898 (for applications up to 1 Mbps) and ISO 11519 (for applications up to 125 Kbps).

9.6.1 Some Features of CAN Bus

Some of the features and uses of CAN bus are:

- To make the communication between different microprocessors more efficient without increasing the wiring needed, CAN does not use central control. Any processor in the system can send and receive messages directly without relying on any central control (central processor). This reduces the wiring, in some cases, by 90 percent. The non-destructive arbitration technique used by CAN guarantee the arrival of the messages without any collisions.
- Measurements needed by several controllers can be transmitted via the bus, thereby removing the need for each controller to have its own individual sensor.
- Flexibility and expansion: The use of message identifier technique instead of using addressing, simplifying the system configuration and gives its high degree of flexibility. The CAN specification does not specify the actual layout and structure of the physical bus itself. This gives the design engineers the possibility to reconfigure CAN systems by adding or removing network nodes easily. In the cases when the new nodes are purely receivers, and which need only existing transmitted data, it is possible to add them to the network without the need to make any changes to existing hardware or software. In general, any new node can be connected to CAN bus as it is able to transmit, or detect, on the physical bus, one of two signals called dominant or recessive. For example, a dominant signal may be represented as logic '0' and recessive as logic '1' on a single data wire. Furthermore, the physical CAN bus must guarantee that if one of

two devices asserts a dominant signal and another device simultaneously a recessive signal, the dominant signal prevails.

- Due to flexibility, expandability and content-oriented nature of CAN bus, it is easy to use by any project manager. Each unit in CAN system can be developed and tested separately. If all the individual functional units are proved to work independently, there is a very high probability that when all of them connected together to form a CAN system they will work correctly as a system.
- The individual nodes of a CAN system are connected together in a daisy chain. All the nodes are equal. Any processor can send a message to any other processor, and if any processor fails, the other systems in the machine will continue to work properly and communicate with each other.
- CAN messages are short, no more than eight bytes long. It is generally used for sending signals to trigger events, such as to lock seat belts during heavy braking, and measurement values, such as temperature and pressure readings. The content-oriented nature of CAN and the non-destructive arbitration technique used guarantee interrupt-free transmission. The message identifier defines the priority of the message which will prevent any conflicts. It also guarantees that the urgent messages (with the highest priority) will be delivered first.
- In order to increase the traffic reliability, CAN protocol include extensive error checking mechanisms.
- CAN system is able to transmit up to 7600 8-byte messages per seconds.
- The devices that are connected by a CAN network are typically sensors, actuators and control devices. A CAN message never reaches these devices directly, but instead a host-processor and a CAN Controller is needed between these devices and the bus.

9.6.2 CAN Architecture: CAN and OSI Model

The Controller Area Network specification defines only two layers of the OSI model: the Physical and Data-Link Layers, Figure 9.32. ISO 11898 defines the Physical Layer. Based on the levels of abstraction, it is possible to say that the CAN protocol can be described in terms of four levels: application, object, data-link and physical levels. CAN specifications describe the physical and data-link layers while all the remaining layers are managed by the microprocessor in software as appropriate. The four layers are:

- Application Layer:
  Many applications of CAN require services that are beyond the basic functionality specified by the Data-Link Layer but which may be

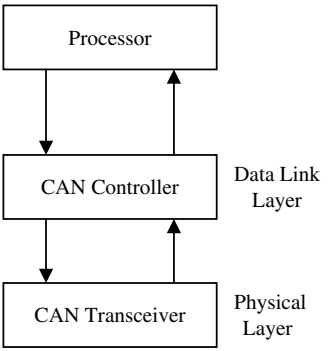


Figure 9.32 CAN Bus Hierarchy.

implemented at the Application Layer. To meet such needs, several organizations have developed Application Layers.

- Object Layer:
  This includes the following functions:
  - Message Filtering
  - Message and Status Handling
- Data-Link Layer:

This layer is the kernel of the protocol. It recognizes and understands the format of messages. It presents messages received to the object layer and accepts messages to be transmitted from the object layer. It constructs the messages to be sent to the Physical Layer, and decodes messages received from the Physical Layer. The data-link layer is responsible for:

- Fault Confinement
- Error Detection
- Message Validation
- Acknowledgment
- Arbitration
- Message Framing
- Transfer Rate and Bit Timing
- Information Routing

In CAN controllers, hardware is usually used to implement the Data-Link Layer. The functions, operations and implementation of the Data-Link layer are considered latter.

- Physical Layer

The physical layer is responsible for defining how the signals are actually transmitted. Some of the tasks of the physical layer are:

- Transmission Medium: The physical layer specifies the physical and electrical characteristics of the bus.
- Signal Level and Bit Representation: This includes the hardware required to convert the characters of a message into electrical signals for transmitted messages and electrical signals into characters for received messages.

The Physical Layer is all the time a "real" hardware. The Physical Layer will be discussed next.

9.6.3 The CAN Physical Layer

CAN Bus Description

The CAN bus is a differential balanced 2-wire interface running over either a Shielded Twisted Pair (STP), Un-shielded Twisted Pair (UTP), or Ribbon cable. Each node uses a Male 9-pin D connector. For data communication on the differential wire bus, CAN uses for bit encoding the "Non Return to Zero (NRZ)" encoding technique (with bit-stuffing). NRZ encoding is used to ensure compact messages that have a minimum number of transitions and high resilience to external disturbance.

A basic system is configured as shown in Figure 9.33. Unlike the 12C bus, the CAN bus does not require that one of the nodes is a master. Nodes can be added to the network at any time; it is not necessary to power the system down. The standard specifies that up to 30 nodes can be added to the bus.

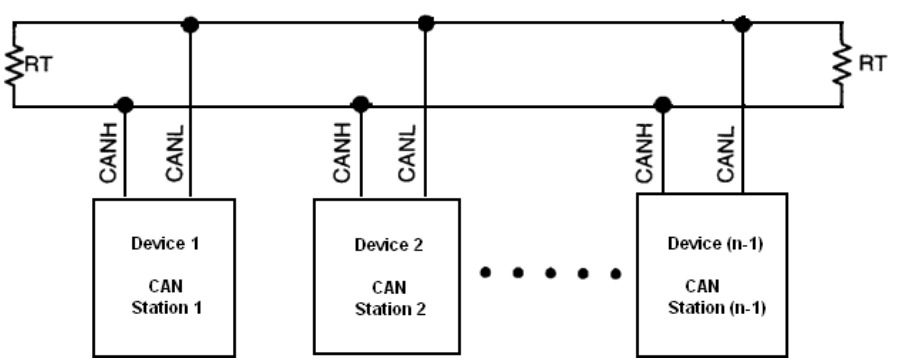


Figure 9.33 Basic CAN bus connection.

The communication between nodes in Controller Area Network protocol is a synchronous. This means that all the nodes connected to the bus while receiving or transmitting must use the same clock rate and that all the clock rates in the network are based on the same reference (single) point. The use of synchronous serial communication results in more efficient data transmission. On the other hand, to keep any two clocks synchronized for long time represents a difficult task and needs some sort of fixed reference signal. With time, it is very common that the clocks lose their synchronization due to oscillator drift, propagation delays, and phase errors.

Electrical Consideration

Electrically, the CAN bus utilizes balanced differential signaling; the current in each signal line is equal and opposite. Such a scheme significantly enhances noise immunity, increases common mode noise rejection, and improves fault tolerance over single-ended drive models. For bus interfacing, CAN uses the circuit shown in Figure 9.34.

The two serial lines that make up the CAN Bus are designated CANH and CANL. The bus signaling protocol defines two states, *dominant* and *recessive*. Similar to the wired AND configuration in the I2C bus, *dominant* will override *recessive*. When the transmitted data, TXD, is a logical 0, the bus is in the dominate state; the CANH signal line can be taken to voltage between 2.75 and 4.5 volts (e.g. 3.5 V), and the CANL is taken to voltage between 0.5 to 2.25 volts (e.g. 1.5 V), giving normally a difference of 2.0 volts. When the transmitted data is a logical 1, the bus is in the *recessive* (quiescent) state, which is set to 2.5 V.

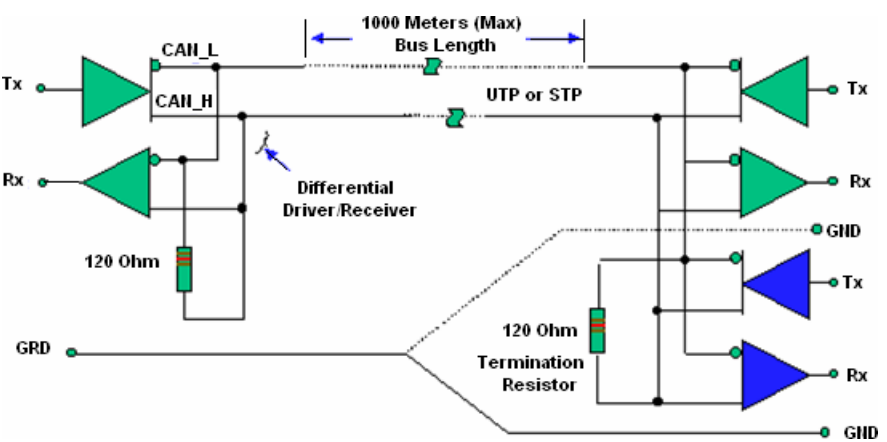


Figure 9.34 CAN Bus electrical interface circuit (Ref. www.interfacebus.com).

9.6.4 Synchronization Mechanisms used in CAN

Any node receiving a frame has to be synchronized with the transmitter. CAN system has no explicit clock signal that can be used by the receiver and the transmitter as reference to achieve synchronization. CAN nodes use two different mechanisms to synchronize their clocks; hard synchronization and resynchronization:

- *Hard synchronization: Hard synchronization* occurs only once during a message transmission. It takes place at Start-of-Frame (SOF). The Start of Frame is the first bit of the frame and it is transmitted dominantly. Before it the CAN bus is in a recessive state. All the nodes connected to the bus automatically synchronize its clock using the transition created by the SOF bit. As we shall see while discussing "bit timing" the transition occurs in the "Synchronization Segment" of the bit time.
- *Resynchronization:* Hard synchronization mechanism achieves the synchronization at the beginning of the message but it is not enough to compensate for oscillator drift, and phase differences between transmitter and receiver oscillators during the entire time of the frame. This makes the clocks to be not able to remain synchronized throughout the entire frame. To keep the clock continually resynchronize, CAN uses an additional synchronization mechanism: resynchronization
  - Resynchronization occurs every time the bus transitions from recessive to dominant. The bit stuffing used in CAN (see latter) guarantee that a transition will occur in the bit stream even if the original message is a string of "0"s or "1"s. Bit stuffing does not allow the contents of the frame to has more than five consecutive identical bit levels. If it happened, the transmitter will automatically add (stuff) a bit of the opposite polarity into the bit stream.
  - Resynchronization is automatically invoked if the system does not recognize the occurrence of an edge at the "Synchronization Segment" of any of the bit that follow the SOF bit of any received frame. These results in either shorten or lengthen the current bit time depending on where the edge occurs. The maximum amount by which the bit time is lengthened or shortened is determined by a user-programmable number of time quanta known as the *Synchronization Jump Width* (SJW).

Bit Rates and Bus Lengths

The rate of data transmission depends on the total overall length of the bus (the total actual length of the wiring in the network) and the delays associated

with the transceivers. This is a physical limitation and not set by CAN protocol. The length of the bus is limited by two factors:

- The propagation delay time: is the time period necessary for a signal to go from one end of the bus to the other back again before the next signal is transmitted.
- Time needed by the electronic circuitry to transmit and receive these signals.

Increasing the length of the bus increases the sum of the propagation delay and the time needed by the transmitting and receiving devices. This, in turn, increases the nominal bit time and, accordingly, decreases the possible transmission bit rate. As a result, every CAN system must trade bus length for bit speed. For example, to have 1Mbit/sec speed, the maximum possible bus length is specified as 25 meters, for longer bus lengths it is necessary to reduce the bit rate. Table 9.3 gives some indication of the bit rates and the corresponding maximum bus length.

The bit rate can always be slower than the maximum possible speed for a given bus length. Conversely, the bus length can be shorter than the maximum possible bus length for a given transmission speed. The CAN system designer needs to keep in mind that transmissions tend to become more reliable with a slower bit speed and shorter bus lengths.

9.6.5 CAN Data Link Layer

As mentioned earlier, Data-Link Layer is responsible implementing many functions: bit timing and synchronization, message framing, arbitration, acknowledgment, error detection and signaling, and fault confinement. Synchronization and acknowledgement are discussed before. In the following we are discussing in detail the rest of the functions: bus arbitration, message framing, error detection and signaling.

Table 9.3 Maximum Bit Rate vs. Bus Length.

Bit Rate	Bus Length
1 MBit/s	25 m
800 kBit/s	50 m
500 kBit/s	100 m
250 kBit/s	250 m
125 kBit/s	500 m
50 kBit/s	1000 m
20 kBit/s	2500 m
10 kBit/s	5000 m

a. Bus Arbitration

One of the subfunctions of the Data Link layer is called Medium Access Control (MAC). The main function of MAC is to prevent conflicts on the network. If two or more transmitters seek to send messages across the network at the same time, MAC will make sure that each one of them will be given an opportunity to transmit one at a time so that the messages do not interfere with each other. As in any communication network, the use of medium access control has a significant impact on the performance of the network.
Access control methods have two varieties, Determined and Random:

- ***Determined access control***: In case of determined access control, the right to access the bus is defined before a node tries to access the bus, guaranteeing that no conflicts will occur. Determined access control requires either a central entity to manage network access or a decentralized agreement between nodes (such as token passing). Centrally controlled access methods are more vulnerable to system failures. One of the reasons is that if the central entity fails then the entire network fails. Decentralized determined access methods are more complex than centralized ones. For decentralized methods, it also becomes difficult to dynamically assign priority to nodes.
- ***Random access control***: In case of random access control any node can access the bus as soon as it is idle. Most random access control methods are based on a mechanism called "Carrier-Sense Multiple Access" (CSMA). In CSMA all nodes monitor the network and wait for it to become idle. Once the network is idle all of the nodes that have a message to transmit will attempt to access the network at the same time. Of course only one node is able to transmit at a time, so a method must be found to sort out which node has priority. Two mechanisms can be used; Collision Avoidance method and Collision Detect method:
  - **Collision Avoidance method:** In this case the CSMA is set up to limit or prevent collisions between messages.
  - **Collision Detection method:** In this case the network is set up to allow for message conflicts, but then intervenes to detect and clean up these conflicts. With the Collision Detection method each node will check to see if the bus is clear before transmitting. If two or more nodes transmit simultaneously, the nodes that are transmitting will detect the conflict, stop transmitting, and then try to re-transmit at a randomly determined time in the future. Because none of the two nodes has priority, so whichever node retries first will gain

advantage. If the nodes clash again or clash with a third node, there will be further delay.

The primary problem with the Collision Detection method occurs when there is a lot of contention on the network. Frames are constantly aborted and retransmitted which wastes bandwidth and creates long delays. Consider as an example a desktop network uses Ethernet as technology that applies the collision detection methods just mentioned; a well-managed Ethernet network is operated well below full capacity, keeping such clashes to a minimum, but still leaving us with a nondeterministic component in our communications. Since the original clashing messages were both destroyed, this situation is sometimes referred to as ***destructive arbitration***.

Non-Destructive Bit-Wise Arbitration

Controller Area Network systems takes a different approach; it uses "Carrier-Sense Multiple Access with Collision Avoidance" (CSMA/CA). The CAN protocol calls this "***Non-Destructive Bit Wise Arbitration***." It is not centralized, and it grants nodes access to the bus based on priority.

The CAN protocol controls bus traffic by allowing high-priority messages access to the bus over lower-priority messages. To achieve that, CAN specification does not specify the actual layout and structure of the physical bus itself, instead, it requires that a device connected to the CAN bus to be able to transmit, or detect, one of two signals called dominant or recessive. The dominant is a logical 0 and recessive is logical 1. Any node wants to send a message, it must attach with it a field called "Arbitration field". The Arbitration field comes at the first beginning of the message and is used to identify the message and to determine its priority. Whenever a node transmits the arbitration field, it listens to the bus at the same time. If the node that is transmitting a recessive bit ("logic 1") detects a dominant bit (logic "0") on the bus, it automatically stops transmitting (i.e. it lost arbitration) and becomes a receiver of the dominant message (see Figure 9.35). (We remind the reader here that as the value of the identifier is less, the message has higher priority.) At the end of transmitting the identifier field all nodes but one have backed off, and the highest priority message gets through unimpeded. As in case of I2C, this is achieved through the physical implementation of the bus. The bus uses an open collector transistors to act as wired-AND.

This is nondestructive bus arbitration, since the highest priority message does not get destroyed. In fact, the node transmitting that message does not even know that a collision happened. The only way for a node to know there

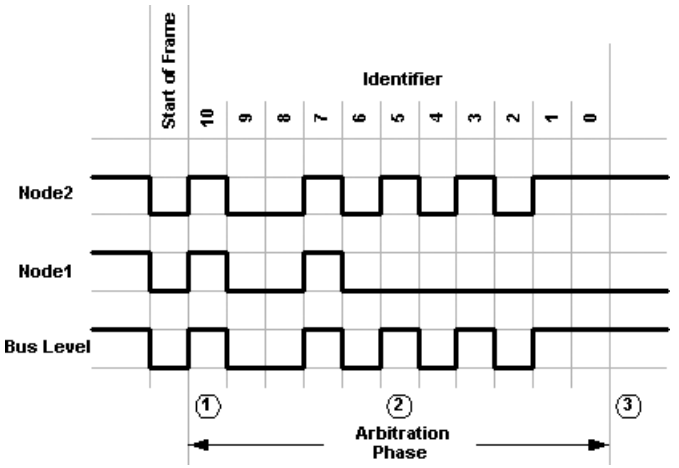


Figure 9.35 Bit Wise Arbitration: Nodes 1 & 2 start arbitration at point 1. Node 1 yields to Node 2 at point 2, and stops transmitting. Only Node 2 can continue to transmit over the bus.

is a collision is for the node to see something on the bus that is different from what it transmitted. So the successful node and any other listening nodes never see any evidence of a collision on the bus.

The highest priority message always gets through, but at the expense of the lower-priority messages. Thus, CAN's real-time properties are analogous to the properties of a preemptive real-time kernel on a single processor. In both cases, the goal is to ensure that the highest-priority work gets completed as soon as possible. It is still possible to miss a hard real-time deadline, but there should never be a case where a high priority job misses its deadline because it was waiting for a lower-priority task to complete.

Once the more dominant message has been transmitted and the bus becomes idle, the less dominant nodes are able to try again. The result is that no bandwidth is wasted. Note that every node in the network will look at every message transmitted on the bus. Most of the time any given node will ignore most of the messages it sees.

The arbitration field used to define the priority of the message can be 11 or 29 bits long, depending which variation of the protocol is used. It is possible to use the first few bits for priority and the remaining bits to identify the message type. The CAN standard does not dictate what meaning you attach to those bits, but the many higher-level protocols that sit on top of CAN do define them. For example, the J1939 standard allows one portion of the bits to be a destination address, since the CAN protocol itself specifies a source

address for all packets, but does not mandate a destination address. This is quite reasonable since much of the traffic on an automotive bus consists of broadcasts of measured information, which is not destined for one specific node.

Benefits of using Non-Destructive bitwise arbitration:

The use of Non-destructive bitwise arbitration achieves some benefits for the network:

- It provides bus allocation on the basis of need, and delivers efficiency benefits that cannot be gained from either fixed time schedule allocation (e.g. Token ring) or destructive bus allocation (e.g. Ethernet.)
- With only the maximum capacity of the bus as a speed limiting factor, CAN will not collapse or lock up. Outstanding transmission requests are dealt with in their order of priority, with minimum delay, and with maximum possible utilization of the available capacity of the bus.

9.6.6 Frame Types and Frame Format

As any serial communication system, the data exchange takes the form of frames. Controller Area Network systems have four kinds of frames:

- **Data Frame**. A standard message used to transmit data over the network.
- **Remote Frame**. A message sent by a receiver to request data from another node on the network.
- **Error Frame**. A message sent out by a receiver to destroy a frame that contains errors. The Error Frame tells the transmitter to send the message again.
- **Overload Frame**. An Overload Frame is similar to an error frame. A receiver would typically send out an Overload Frame to ask a transmitter to delay the next message sent.

A. Data Frames

CAN systems use Data Frames to transmit data over the network. A Data Frame contains an identifier and various pieces of control information, and can hold up to eight bytes of data. CAN systems provide two versions of the Data Frame, the Base Format and the Extended Format. The Extended Data Format Frame has been introduced in the early 1990's as part of CAN Specification 2.0B to avoid the shortage in the Base format due to the length of the identifier. By the beginning of 1990's the number of the different messages created by transmitters on the network was more than the number of the unique identifier

Field name	Length (bits)	Purpose
Start-of-frame	1	Denotes the start of frame transmission
Identifier	11	A (unique) identifier for the data
Remote transmission request (RTR)	1	Dominant (0) (see Remote Frame below)
Identifier extension bit (IDE)	1	Must be dominant (0)Optional
Reserved bit (r0)	1	Reserved bit (it must be set to dominant (0), but accepted as either dominant or recessive)
Data length code (DLC)*	4	Number of bytes of data (0-8 bytes)
Data field	0-8 bytes	Data to be transmitted (length dictated by DLC field)
CRC	15	Cyclic Redundancy Check
CRC delimiter	1	Must be recessive (1)
ACK slot	1	Transmitter sends recessive (1) and any receiver can assert a dominant (0)
ACK delimiter	1	Must be recessive (1)
End-of-frame (EOF)	7	Must be recessive (1)

Figure 9.36 Base frame format.

codes that can be created using the 11 bits assigned to the identifier field of the base format. The Extended Data Format adds more bits to the identifier field. The identifier of the extended format has 29 bits that allows the CAN system to create as many as 512 million different unique messages and priorities.

Base Format Data Frame (2.0A Format)

The Base Format Data Frame (2.0A Format) is shown in Figure 9.36.

Extended Format Data Frame (2.0B Format)

The Extended Format Data Frame is nearly identical to the Base Format Data Frame. The only differences between these two formats are found in the Identifier Extension (IDE) bit in the Control Field, and the size and arrangement of the Arbitration Field. Extended format data frame (2.0B) controllers are completely compatible with base format (2.0A) controllers and can transmit and receive messages in either format. This means that both the Base Format and the Extended Format can coexist in the same CAN system. The rule is that Base Format frames always have priority over Extended Format frames, but the Extended Format is designed to be backwards compatible with the Base Format.

B. Remote Frame

Generally data transmission is performed on an autonomous basis with the data source node (e.g. a sensor) sending out a Data Frame. It is also possible for a destination to request the data from the source by sending a Remote Frame. In other words Remote Frames are used for receivers to request information from another node. They are often sent out on a regular schedule,

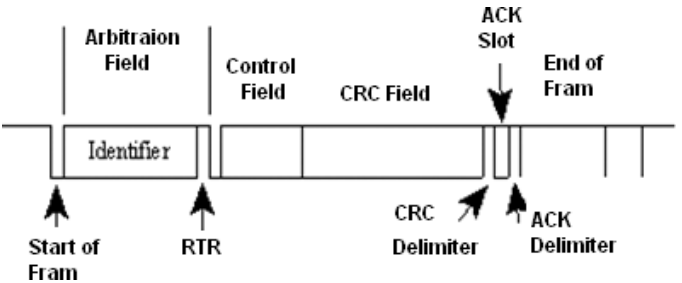


Figure 9.37 A remote frame (2.0A type).

to draw updates from sensors. The format for a Remote Frame is identical to that of a Data Frame (Figure 9.37). Both frame types also feature base and extended formats, and both have a single Remote Transmission Request bit at the end of the Arbitration Field. With a Remote Frame, this bit is transmitted recessively identify it as a Remote Frame. For Data Frames the RTR bit is always transmitted dominantly, i.e.

$$\begin{aligned} \text{RTR} = 0; &\quad \text{DOMINANT in data frame} \\ \text{RTR} = 1; &\quad \text{RECESSIVE in remote frame} \end{aligned}$$

The RTR bit is considered part of the bitwise arbitration so a Data Frame will always dominate over a Remote Frame with the same identifier. This is reasonable since a request for data should not take precedence over the data being requested.

It can happen in very rare events that a Data Frame and Remote Frame will be transmitted at the same time and with the same identifier. In such unlikely events, the Data Frame wins arbitration. This is due to the RTR bit; RTR is dominant bit in case of Data Frame. In this way, the node that transmitted the Remote Frame receives the desired data immediately.

C. Error Frame

Receivers send out Error Frames whenever they detect that a frame contains an error. Sending an error frame by one node forces all other nodes in the network to send an error frame as well; the original transmitter then resend the message. The Error Frame can be sent during any point in a transmission and is always sent before a Data Frame or Remote Frame has completed successfully. The transmitter constantly monitors the bus while it is transmitting. When the transmitter detects the Error Frame, it aborts the current frame and prepares to resend once the bus becomes idle again.

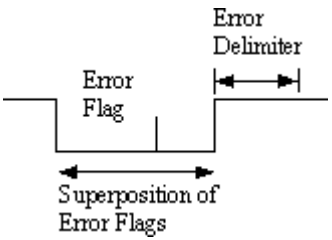


Figure 9.38 12 error frame.

The Error Frame contains the following fields (Figure 9.38):

- **Error Flag Field**. The Error Frame deliberately violates the bit stuffing rule by sending either six recessive bits or six dominant bits in the Error Flag Field. The determination of whether the Error Flag is recessive or dominant depends on the Error State of the node.
- **Error Delimiter Field**. The Error Delimiter is a sequence of eight recessive bits and indicates the end of the frame. The bus will then enter an idle state or a new Data Frame or Remote Frame will start.

Note that not every node in a network is permitted to send out Error Frames. The CAN protocol has a method of keeping faulty nodes from participating on the bus.

D. Overload Frame

The Overload Frame can be considered a special form of Error Frame. It is used in support of data flow. It is used to ask a transmitter to delay further frames or to signal problems with the Intermission Field. The Overload Frame has the same format as the Error Frame but unlike the Error Frame it does not cause the retransmission of the previous frame. If the Overload Frame is being used to delay further transmissions, then no more than two Overload Frames can be generated successively.

An Overload Frame includes a Flag Field that contains a sequence of six dominant bits followed by a Delimiter Field, a series of eight recessive bits. When one node sends out an Overload flag all of the nodes on the network detect it and send out their own overload flags, effectively stopping all message traffic on the CAN system. Then, all of the nodes listen for the sequence of eight recessive bits. The maximum amount of time needed to recover in a CAN system after an Overload Frame is 31 bit times.

The Data Link Layer in Controller Area Network systems is very effective in detecting errors. Every frame is simultaneously accepted by every node in

the network or rejected by every node. If a node detects an error it transmits an error flag to every other node in the network and destroys the transmitted frame.

CAN implements five error detection mechanisms; three at the message level and two at the bit level.

a. At the message level:

- **Frame Check**. To check the integrity of the contents of a message, CAN defines some positions within any message and predefines values for these bits. For example, the first bit in every frame must always be a dominant bit. Every node monitors these fixed fields in every frame they receive. A node will detect a frame error if a fixed bit field contains an illegal bit value. Errors of this type are called Form Errors (also known as a Format Error).
- **Cyclic Redundancy Check**. Cyclic redundancy checking is a method of looking for errors in a network by applying a polynomial equation to a block of transmitted data.
- **Acknowledgement Check**. Every node that transmits a message listens for an acknowledgement in the ACK slot from at least one other node in the network. If the transmitter does not receive a response then this is considered an acknowledgement error. The node will continue to retransmit the frame until it receives and acknowledgement.

b. At the bit level:

- **Stuff Rule Check (Bit Stuffing)**. CAN uses a technique known as bit stuffing as a check on communication integrity. After five consecutive identical bit levels have been transmitted, the transmitter will automatically inject (stuff) a bit of the opposite polarity into the bit stream. Receivers of the message will automatically delete (de-stuff) such bits before processing the message in any way. Because of the bit stuffing rule, if any receiving node detects more than five consecutive bits of the same level, a stuff error is flagged and an Error Frame is sent.
- **Bit Check (Bit Monitoring)**. Any transmitter automatically monitors and compares the actual bit level on the bus with the level that it transmitted. If the two are not the same, i.e. if it sends out a dominant bit when it was supposed to transmit a recessive bit, or a recessive bit when the message called for a dominant bit, a bit error is flagged.

9.6.7 Using CAN Bus

Today there is increasing support for and application of the CAN bus outside of the automotive industries. The bus and protocol are specified and controlled by the international ISO standard, ISO 11898. A number of semiconductor manufacturers offer integrated implementations of the CAN Transceiver and the Controller.

9.7 Serial Communication Using SPI

The UART described in Section 9.3 has a few drawbacks. UART is only *half duplex* (also called *simplex*) which means that sending data can take place in only one direction on one line. Connecting the TXD pin on one device connected to RXD pin of another supports data transfer in one direction only, namely TXD to RXD. The Serial Peripheral Interface (SPI), introduced in this section, offers *full duplex, i.e.,* the ability to send data in both directions at the same time. UART as well as the I2C and CAN protocols belong to serial asynchronous communication. SPI is a synchronous mode of communication. This means that all the relevant devices are connected to a common clock, so that they can all be in synch, and operate at a high speed.

Normally the SPI bus is used for very short distance communication with peripherals or other microcontrollers that are located on the same circuit board or at least within the same piece of hardware. This is different from the UART, which is used for communication over long distances, such as between units or between a microcontroller and PC. The SPI bus was developed by Motorola to provide relatively high-speed, short distance communication using a minimum number of microcontroller pins. SPI besides using it for a synchronous serial communication of host processor and peripheral, it can be used equally to connect two processors. A nearly identical standard called “Microwire" is a restricted subset of SPI, trademarked by National Semiconductor. Both have the same functionality. There are also the extensions QSPI (Queued Serial Peripheral Interface) and MicrowirePLUS.

We start the next section by giving a brief idea on synchronous serial transmission followed by the details of SPI bus.

9.7.1 Synchronous Serial Transmission

Synchronous serial transmission requires that the sender and receiver share a clock with one another, or that the sender provide a strobe or other timing signal so that the receiver knows when to “read” the next bit of the data.

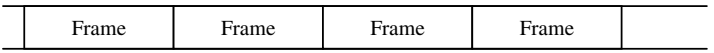


Figure 9.39 Self-clocking synchronous serial transfer. Frames Sent without start or stop bits.

There are two common kinds of synchronous serial data transfer, one related to data communications and the other to serial peripheral ICs. In data communications, synchronous serial transfers are self clocking, and have no shared clock signal. In self-clocking synchronous transmission, one frame is sent immediately after the other, Figure 9.39. Frames do not have start and stop bits for synchronization. Frame synchronization is usually derived from bit or character synchronization. Whenever another data character is not immediately ready for transmission, the transmitter repeatedly sends a special SYNC character until it can transmit the next data character. To achieve bit synchronization, a two-step process is typically used:

- Encoding the clock (transmitter clock) in the data.
- Re-drive the clock from the data (at the receiving end).

By this way the transmitter and receiver may have independent clocks, but the receiver can synchronize its clock from the re-derived clock using the transitions in the received signals. Since start and stop bits are not required, this approach has low overhead and is typically used for transferring large blocks of data at high speeds. Since this form of synchronous serial transfer is not prevalent in embedded systems, it is not considered further in this text.

The second kind of synchronous serial transfer is commonly used with serial peripheral ICs. A separate line is used to carry a common clock signal used by the transmitter and receiver for synchronization (Figure 9.40). This form of synchronous serial transfer is important in embedded systems. In some versions of this scheme, the clock signal is generated by the transmitter, and in others it is generated by the receiver.

This form of synchronous transmission is used with printers and fixed disk devices in that the data is sent on one set of wires while a clock or strobe is sent on a different wire. Printers and fixed disk devices are not normally serial devices because most fixed disk interface standards send an entire word of data for each clock or strobe signal by using a separate wire for each bit of the word. In the PC industry, these are known as Parallel devices.

Compared with asynchronous, synchronous communication is usually more efficient because only data bits are transmitted between sender and receiver, and synchronous communication can be more costly if extra wiring and circuits are required to share a clock signal between the sender and receiver.

Clocked Synchronous Serial Peripheral ICs

Clocked synchronous serial transfer is often used for transfers between a microprocessor and ICs with serial interfaces. The distance over which the data is transferred is usually very short. Typically, the serial peripheral ICs are on the same printed circuit board as the microprocessor.

There are several different protocols available for clocked synchronous serial transfer. These interfaces typically involve two or three wires and may be simplex or half-duplex channels. In a two-wire scheme (Figure 9.40(a)), one wire is the serial clock, which defines the bit times, and the other is the serial data. The third wire for signal ground is not included in the count. If the serial peripheral is the receiver and provides the clock, it is said to be self-clocked or internally docked. If the clock is provided by the transmitter, the receiver is said to be externally clocked. Two-wire schemes can also provide bidirectional data transfer. In such cases, the single data line is half-duplex.

A three-wire scheme has two data lines and the serial clock (Figure 9.40(b)). However, data transfer usually occurs in only one direction at a time. Examples of clocked synchronous serial transfers are DACs and ADCs.

9.7.2 Serial Peripheral Interface (SPI)

Signals Used in SPI Communication

SPI communication involves a master and a slave. Both of them send and receive data simultaneously, but the master is responsible for providing the

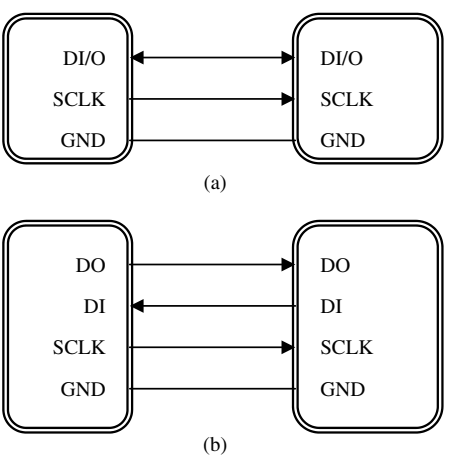


Figure 9.40 Synchronous serial transfer with a common clock: (a) 2-wire scheme; (b) 3-wire scheme.

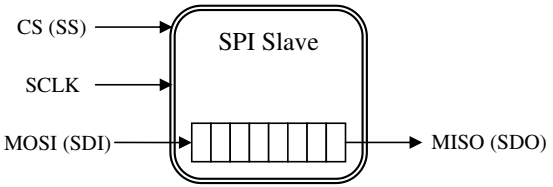


Figure 9.41 SPI Slave.

clock signal for the data transfer. It also determines the state of the chip select lines, i.e. it activates the SLAVE it wants to communicate with. In this way the master has control of the speed of data transfer and is, therefore, in control of the data transfer.

The SPI bus specifies four logic signals: two control and two data lines. The names of the four signals on the pins of SPI port depends on the manufacturer. Figure 9.41 illustrates the names as given by Motorola. The SPI device can be a simple shift register as shown in Figure 9.43 and it can extend to be an independent subsystem, but the basic principle of a shift register must always be present. The most common names of the four signals are (Note: The first names are used by Motorola):

- SCLK — Serial Clock (output from master).
- SS — Slave Select (active low; output from master). This signal is also called: Chip select (CS), Slave Transmit Enable (STE).
- MOSI/SIMO — Master Output- Slave Input (output from master). This signal is sometimes called: Serial Data In (SDI), Data In (DI) and Serial In (SI)
- MISO/SOMI — Master Input- Slave Output (output from slave). Also called: Serial Data Out (SDO), Data Out (DO) and Serial Out (SO)

The SDO/SDI convention requires that SDO on the master be connected to SDI on the slave, and vice-versa.

The MISO (the slave data output SDO) serves the reading back of data. It also offers the possibility to cascade several devices. The data output of the preceding device then forms the data input for the next IC.

SPI communication, as mentioned before, involves a master and a slave. Multiple slave devices are allowed with individual slave select (chip select) lines. Figure 9.42 shows the four signals in a single-slave configuration. The chip select pin is mostly active-low. If multiple slave devices exist, the master generates a separate slave select signal for each slave. In the unselected state the MISO lines are in high-impedance (hi-Z) state and therefore inactive.

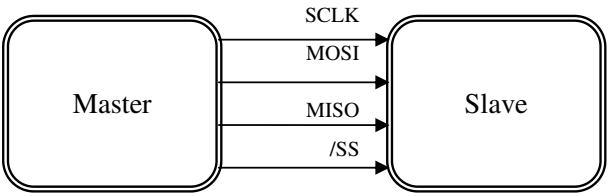


Figure 9.42 Single master, single slave SPI implementation.

This prevents the unselected device to interfere with the currently activated devices. This arrangement also permits several devices to talk to a single input. The master decides with which peripheral device it wants to communicate. The clock line SCLK is brought to the device whether it is selected or not. The clock serves as synchronization of the data communication.

More Possible Signals: Interrupt Signal

SPI devices sometimes use another signal line to send an interrupt signal to a host CPU. Examples include pen-down interrupts from touchscreen sensors, thermal limit alerts from temperature sensors, alarms issued by real time clock chips, SDIO, and headset jack insertions from the sound codec in a cell phone. Interrupts are not covered by the SPI standard; their usage is neither forbidden nor specified by the standard.

9.7.3 Basic Data Transmission

As mentioned before, the basic principle of a shift register is always present in any SPI device. Accordingly, communication between two devices normally involve two shift registers of some given word size, such as eight bits, one in the master and one in the slave. The two shift registers are connected in a ring i.e. they form an inter-chip circular buffer as shown in Figure 9.43. Command codes as well as data values are serially transferred, pumped into a shift register and are then internally available for parallel processing.

Data transmission takes place in the following manner (See Figure 9.44):

1. To begin a communication, the master first configures the clock, using a frequency less than or equal to the maximum frequency the slave device supports. The master then pulls the slave select (SS) low for the desired chip. If a waiting period is required (such as for analogue-to-digital conversion) then the master must wait for at least that period of time before starting to issue clock cycles.

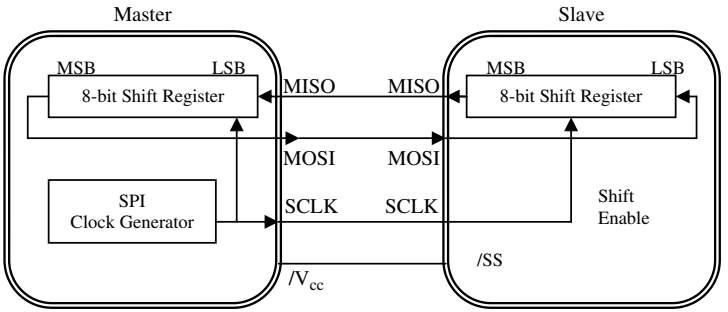


Figure 9.43 SPI Communication Scheme.

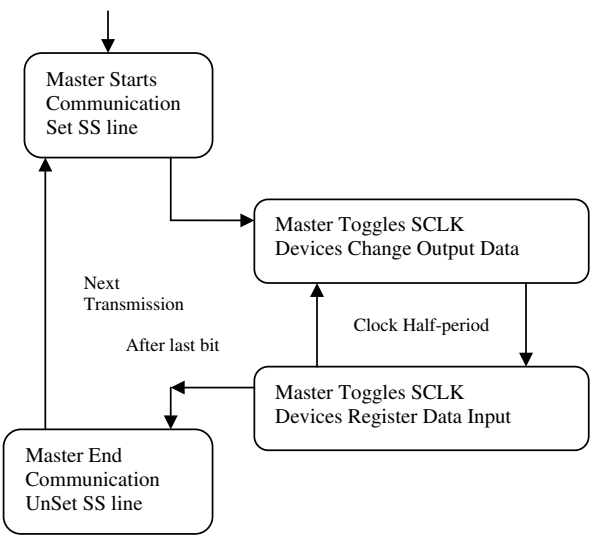


Figure 9.44 Statechart of the basic SPI transmission.

2. Once the selected SS is low, one edge (rising or falling) of the SCLK signals the devices (Master and Slave) to toggle the MOSI and MISO to the correct bit of data being transmitted.
3. The other edge of the SCLK line (rising or falling) signals the devices to register the bits on the MOSI and MISO, effectively reading the bit into the device.

   Steps 2 and 3 mean that the master and the slave are communicating in a full duplex data transmission mode:

   - the master sends a bit on the MOSI line; the slave reads it from that same line

- the slave sends a bit on the MISO line; the master reads it from that same line

4. The transmission continues in this fashion until the devices have exchanged the specified number of bits (usually 8, 16, or 32).

This means that the master supplies the slave with eight bits (16 or 32 bits) of data, which are shifted out of the master-out-slave-in (MOSI) pin. The same eight (16 or 32) bits are shifted into the slave unit, one bit per clock pulse, on its MOSI line. The slave simultaneously uses its own master-in-slave-out connected to the MISO pin of the master to shift eight bits into the master. In other words, at the end of the eight shift cycles, eight bits be shifted into the slave from the master and eight bits shifted into the master from the slave. SPI communication, then, is essentially a circle in which eight bits flow from the master to the slave and a different set of eight bits flows from the slave to the master. In this way the master and a slave can exchange data in a single communication.

5. After the transmission is over the Master pulls the SS line for the slave back high and either goes to another slave on the network or reinitiates the transmission with the same slave by pulling the corresponding SS line back to low.

Data is usually shifted out with the most significant bit first, while shifting a new least significant bit into the same register. After that register has been shifted out, the master and slave have exchanged register values. Then each device takes that value and does something with it, such as writing it to memory. If there is more data to exchange, the shift registers are loaded with new data and the process repeats.

While the above discussion means that the amount of data sent and the amount of data received must be equal, it is possible for the side that does not have any data to send to provide dummy data. In fact, in the majority of applications of SPI the data only goes in one direction and the opposite direction always passes a dummy value. In many SPI applications, the slave is the source of the data. As mentioned before, the frequency and the instances at which the slave has to transmit is controlled by the master. Accordingly, in such applications, the slave must always have a valid byte ready to send; the slave does not have any control over when the master is sending the next clock pulse requesting more data. If the slave device is dedicated to a single job, this may not be difficult. Consider, for example, a thermistor that communicates as an SPI slave. It could provide a single buffer of one byte that is always

contain the last temperature reading. Whenever clock pulses appear, that byte is transmitted and the master gets a reading.

Keeping in mind that SPI does not specify the length of the message or its contents, failing in updating the buffer can cause a problem; if the transmitted message ended by a checksum, repeating one character during transmission will let the checksum fail. The master and slave are no longer synchronized, and some recovery must take place.

It is possible to give the slave a longer time to update the contents of the buffer by letting the master to pause after each byte is transmitted. This will help to solve partially the problem of the slave since it still has a deadline to update the buffer.

One of the possible solutions of this timing issue is by avoiding transmitting and receiving at the same time. It is possible to provide an extra signal that the slave asserts when it wants to transmit. When the master sees this signal, it knows that the slave has a byte ready, and the master then provides the clock to fetch that byte. When the master has something to send, it checks that the slave is not sending before clocking out its own byte; anything simultaneously received from the slave is ignored. Sending the messages in this way, in turn, means that a large fraction of the potential bandwidth is lost. In exchange, one gets a more reliable and flexible software

Every slave on the bus that hasn't been activated using its slave select line must disregard the input clock and MOSI signals, and must not drive MISO. The master must select only one slave at a time.

Some devices use this technique of SPI implementation (clock data out as data is clocked in) to implement an efficient, high-speed full-duplex data stream for applications such as digital audio, digital signal processing, or full-duplex telecommunications channels.

On many devices, the "clocked-out" data is the data last used to programme the device. Read-back is a helpful built-in-self-test, often used for high-reliability systems such as avionics or medical systems. At this point it is important to mention that SPI does not have an acknowledgement mechanism to confirm receipt of data. In fact, without a communication protocol, the SPI master has no knowledge of whether a slave even exists. SPI also offers no flow control. If the application needs hardware flow control, the user might need to do something outside of SPI.

9.7.4 Connecting Devices on SPI Bus

It is possible to connect many different devices (masters and slaves) together using a SPI bus, because all of the MOSI pins and all of the MISO pins can

be hooked together. For such a network, two protocol variants are possible. The first protocol deals with the case when the SPI network consists of only one master and several slaves. The slaves can be microcontrollers. The second protocol any microcontroller (or processor) connected to the network can take the role of master. The selection of the slave can take place either by hardware or software. When using the hardware variant, the master uses the chip select to select the targeted slave. In case of software variant, the software assigns an ID for each slave. The selection of the slave takes place by attaching its ID into the frames. Only the selected slave drives its output, all other slaves are in high-impedance state. The output remains active as long as the slave is selected by its address.

First variant: Single-master protocol

The single-master protocol resembles the normal master-slave communication. The system can contain any number of microcontrollers, but only one of them will be configured as master. The microcontroller configured as a slave behaves like a normal peripheral device.

There are two types of connection of single master and multiple slave devices: slave devices are connected in cascade and slaves are working independently.

a. Cascading Several SPI devices: Daisy Chain SPI Configuration

Figure 9.45 shows the type of connection for cascading several devices. The slaves are connected in a daisy chain configuration and all of them are connected to the same SS line coming from the master. The first slave output

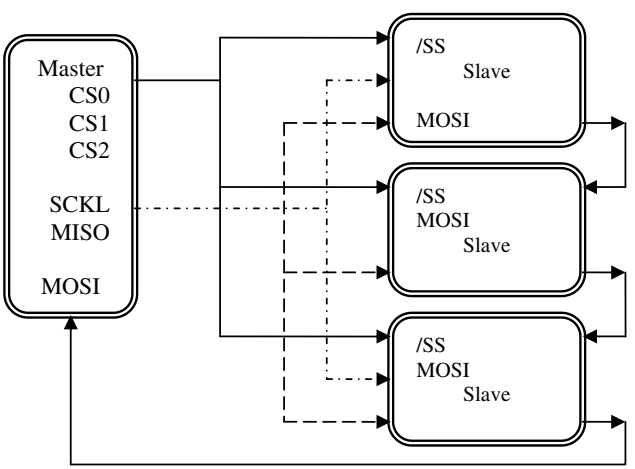


Figure 9.45 Cascading several SPI devices- Daisy Chain Configuration.

is connected to the second slave input, etc., thus forming a wider shift register. In other words, the whole chain acts as an SPI communication shift register; daisy chaining is often done with shift registers to provide a bank of inputs or outputs through SPI.

b. Multiple Independent slave SPI Configuration

If independent slaves are to be connected to a master, the master generates a separate slave select signal for each slave. The connection in this case is shown in Figure 9.46. The clock and the MOSI data lines are brought to each slave from the master. The MISO data lines of the different slaves are tied together and go back to the master. Only the chip selects are separately brought to each SPI device. The master uses the pins of its general purpose input/output ports to generate the slave select signals. Accordingly, the number of slaves that can be connected to one master is limited by the available chip select pins.

Second variant: Multiple Masters

The bus in this variant allows any device connected on the network to be master; simply by deciding to transmit data. The protocol in this case is called

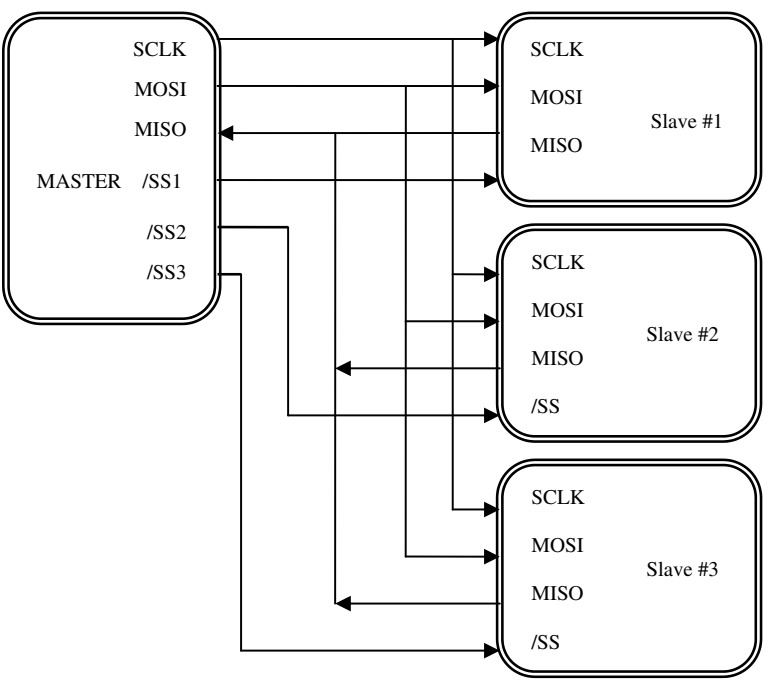


Figure 9.46 Single master, multiple slave SPI implementation.

multi-master protocol. On the other hand, a device on the network becomes slave when its slave select pin is grounded. In this configuration, one controller must permanently provide clock signal for all the devices.

Two types of system error may happen in SPI with multiple masters. The first occurs if several SPI devices want to become master at the same time. The other is a collision error that occurs for example when SPI devices work with different polarities. Some microcontrollers (e.g. MC68HC11) provide a hardware error recognition.

Clock polarity and phase: SPI Modes

There are no general rules for transitions where the data should be latched. Then the master, in addition to setting the clock frequency, it must also configure the clock polarity and phase with respect to the data. Most vendors have adopted a pair of parameters called clock polarity (CPOL) and clock phase (CPHA) that used to determine the edges of the clock signal on which the data are driven and sampled. Each of the two parameters has two possible states (0 and 1), which allows for four possible combinations, or equivalently four possible modes of operation. Table 9.4 gives the four modes and Figure 9.47 represents the timing diagram.

SPI Interface

SPI Interface consists of more than one register. In case of VAR microcontroller three registers are used for interfacing: Data Register (SPDR), Control

Table 9.4

SPI Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

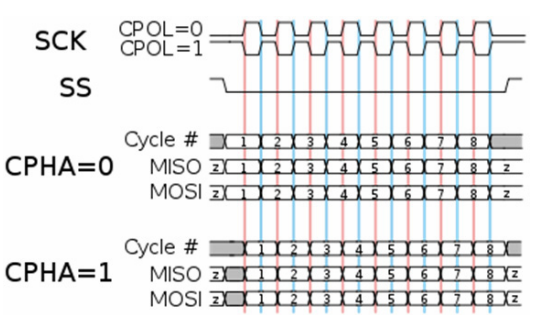


Figure 9.47 A timing diagram showing clock polarity and phase.

Register (SPCR) and Status Register (SPSR). Many microcontrollers are using the same registers with the same names.

SPI Data Register:

The SPI Data Register (Figure 9.48) is a read/write register used for data transfer between the Register File and the SPI Shift Register. Writing to the register initiates data transmission. Reading the register causes the Shift Register Receive buffer to be read.

SPI Control Register

Figure 9.49 gives an example of the SPI control register and the bit definitions. (AVR uses the same control register and bit definitions)

SPI Status Register

Normally, the status register contains two bits (Figure 9.50a).

Bit	7	6	5	4	3	2	1	0	
	MBS							LSB	SPDR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial value	X	X	X	X	X	X	X	X	Undefined

Figure 9.48 SPI Data Register (SPDR).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

Bit	Description
SPIE	SPI Interrupt Mask bit
SPE SPI	Enable bit.
DORD Dat	a Order bit Cleared causes the MSB of the data order to be transmitted first
MSTR	Master/Slave Select bit
CPOL	Clock Polarity bit
CPHA	Clock Phase bit
SPR1	SPI Clock Rate bits
SPR0	

Relationship Between SCK and the Oscillator Frequency

SPR1	SPR0	SCK Frequency
0 0		System Clock/4
0 1		System Clock/16
1 0		System Clock/64
1 1		System Clock/128

Figure 9.49 SPI control register and bit definitions.

Bit	7	6	5	4	3	2	1	0
$0E($2E)	SPIF	WCOL						
Read/Write	R	R	R	R	R	R	R	R
Initial value	0	0	0	0	0	0	0	0

(a)

Bit	7	6	5	4	3	2	1	0	
	SPIF	WCOL						SPI2X	SPSR
Read/Write	R	R	R	R	R	R	R	R/W	
Initial value	0	0	0	0	0	0	0	0	

(b)

Figure 9.50 SPI status register SPSR (a) AT90S series (b) AVR mega series.

SPIF: SPI Interrupt Flag

When a serial transfer is complete, the SPIF Flag is set. An interrupt is generated if SPIE in SPCR is set and global interrupts are enabled. If SS is an input and is driven low when the SPI is in Master mode, this will also set the SPIF Flag. SPIF is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, the SPIF bit is cleared by first reading the SPI Status Register with SPIF set, then accessing the SPI Data Register (SPDR).

WCOL: Write COLlision Flag

The WCOL bit is set if the SPI Data Register (SPDR) is written during a data transfer. The WCOL bit (and the SPIF bit) are cleared by first reading the SPI Status Register with WCOL set, and then accessing the SPI Data Register.

The AVR Mega series uses the least significant bit of the status register as a control signal (SPI2X)

SPI2X: Double SPI Speed Bit

When this bit is written logic one the SPI speed (SCK Frequency) will be doubled when the SPI is in Master mode (see Table 9.3). This means that the minimum SCK period will be two CPU clock periods. When the SPI is configured as Slave, the SPI is only guaranteed to work at $f_{osc}/4$ or lower.

Valid SPI communications

The control register as shown in Figure 9.49 does not contain any information about the expected response from the slave in case if it receives a number of clock pulses greater than specified. Some manufacturers design the slave devices to ignore any SPI communications in which the number of clock pulses is greater than the specified number. Other slaves are designed to ignore

all the extra inputs and continuing to shift the same output bit. It is common for different devices to use SPI communications with different lengths, as, for example, when SPI is used to access the scan chain of a digital IC by issuing a command word of one size (32 bits) and then getting a response of a different size (153 bits, one for each pin in that scan chain).

9.7.5 SPI Applications

SPI is used to talk to a variety of peripherals, such as:

- Sensors: temperature, pressure, ADC, touchscreens
- Control devices: audio codecs, digital potentiometers, DAC
- Camera lenses: Canon EF lens mount
- Communications: Ethernet, USB, USART, CAN, IEEE 802.15.4, IEEE 802.11
- Memory: flash and EEPROM
- Real-time clocks
- LCD displays, sometimes even for managing image data
- Any MultiMediaCard (MMC) MMC or Secure Digital (SD) SD card (including Secure Digital Input Output (SDIO) variant)

9.7.6 Strengths and Weaknesses of SPI

Strengths of SPI

- Full duplex communication. This capability makes SPI very simple and efficient for single master/single slave applications.
- Higher throughput than I2C or SMBus
- Complete protocol flexibility for the bits transferred
  - Not limited to 8-bit words
  - Arbitrary choice of message size, content, and purpose
- Extremely simple hardware interfacing
  - Typically no external circuitry required (like pullup resistors for I2C)
  - Typically lower power requirements than I2C or SMBus due to less circuitry
- No arbitration or associated failure modes
  - Slaves use the master's clock, and don't need precision oscillators
  - Slaves don't need a unique address — unlike I^2C or GPIB or SCSI
  - Transceivers are not needed

- Uses many fewer pins on IC packages, and wires in board layouts or connectors, than parallel interfaces
- At most one “unique" bus signal per device (chip-select); all others are shared by multiple devices
- Signals are unidirectional allowing for easy Galvanic isolation
- Addressing not needed (decreases complexity and helps throughput by not sending an address for each communication)
- Mostly shared lines for multiple devices (except the separate SS lines for each device)

Weaknesses of SPI

- No standards body governs SPI as an official protocol.
  - without a formal standard, validating conformance is not possible
- Requires more pins on IC packages than I2C, even in the “3-Wire" variant: The more devices you have the more pins and connections necessary
- No in-band addressing; out-of-band chip select signals are required on shared buses
- No hardware flow control
- Unlike I2C, SPI has no acknowledgement mechanism or flow control. This prevents the SPI master from knowing whether a slave received a data byte correctly or even whether it is connected to the bus.
- Supports only one master device; Does not support a multi-master architecture
- Only handles short distances compared to RS-232, RS-485, or CAN-bus

9.7.7 Differences between SPI and I2C

SPI and I2C protocols are the main two serial communications protocol. Both systems have their unique advantages and disadvantages which make them more or less suitable for a given application. The differences between the two protocols are:

- SPI supports full duplex communication with higher throughput than I2C.
- SPI is not limited to 8-bit words so it is possible to send any message sizes with arbitrary content and purpose.
- The SPI interface does not require pull-up resistors, which translates to lower power consumption.
- I2C is simpler by having fewer lines which means fewer pins are required to interface to an IC.

- When communicating with more than one slave, I2C has the advantage of in-band addressing as opposed to have a chip select line for each slave.
- I2C also supports slave acknowledgment which means that the sender can be absolutely sure that it is actually communicating with something. With SPI, a master can be sending data to nothing at all and have no way to know that.
- In general SPI is better suited for applications that deal with longer data streams and not just words like address locations. Mostly longer data streams exist in applications where you're working with a digital signal processor or analogue-to digital converter. For example, SPI would be perfect for playing back some audio stored in an EEPROM and played through a digital to analogue converter DAC.
- SPI can support significantly higher data rates comparing to I2C, mostly due to its duplex capability, accordingly it is far more suited for higher speed applications reaching tens of megahertz.
- Since there is no device addressing involved in SPI the protocol is a lot harder to use in multiple slave systems. This means when dealing with more than one node, generally I2C is the way to go.
- SPI has higher clocking speed than I2C is better when used in high speed applications

9.7.8 Examples of Using SPI

Example 9.6: Use of SPI

Figures 9.51 and 9.52 show the hardware and software, respectively, for a simple SPI demonstration system. The system reads the data from port D and sends it out by means of the SPI bus. The data received by the SPI is then sent out to display on port C. To demonstrate that, the MOSI and MISO pins are connected so that whatever is transmitted is also received. This really has no practical application; however, two such systems could work together to communicate data back and forth — one would have to be modified to be the slave, but its programme would be almost identical. This would form a simple high-speed network.

The SPI demonstration software is relatively simple. Port B is initialized so that the MOST, SCLK, and SS pins are functioning as outputs, and the M1SO pin, an input, has a pull-up activated. Ports C and D are initialized for output and input respectively. The SPI system is enabled, along with its interrupt, and set to function as the master.

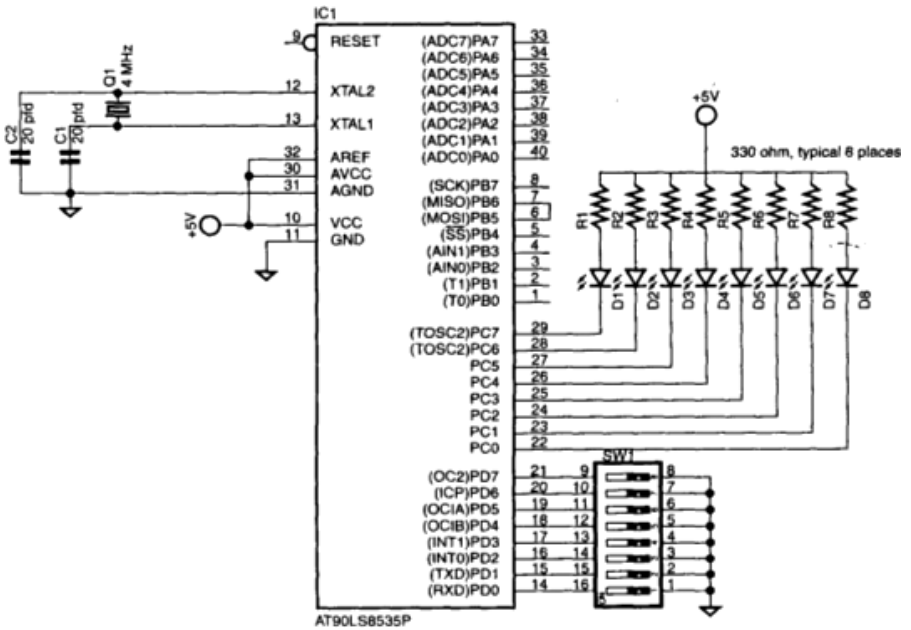


Figure 9.51 SPI example hardware.

Following the initializations, there is a short assembly code routine automatically inserted by CodeVisionAVR that makes sure that the SPI interrupt is cleared. This is an important step, to be sure that the SPI interrupt service routine does not execute until you intend it to. Finally, 0x00 is written to SPDR to start the SPI communication process.

Once started, the entire function is handled in the SPI TSR. The TSR reads the data received, writes it to the output port, the LEDs. Then it reads the input port, the DIP switch, and starts a new communication cycle by writing the data from the input port to SPDR. And the cycle repeats forever.

The SPI bus is often used to form a high-speed network within a device. The connections can be paralleled, with all the MOSI pins tied together and all the MISO pins tied together, or, as mentioned before, in a large serial loop, where all of the data bits travel through all the devices in turn in what amounts to a large circle of shift registers. The parallel method requires a method of selecting the slave device, and only two devices share the data flow, while in the serial loop all of the data flows through all of the devices. In either method, multiple CPUs and slave devices can exist on a SPI-based network to share data.

```
# include <90S8535.h>
# define SPI_output_data PORTC
#define SPI_input_data PIND

//SPI interrupt service routine
interrupt [SPI_STC] void spi_isr (void)
{
       SPI_output_data = SPDR;       //read out new data received
       SPDR = SPI_input_data;        //Load new data to start SPI transmission
}
void main(void)
{
       PORTB = 0x40;                 //pull up on MISO
DDRB = 0xB0;                         //sclk, MOSI, SS = output
       DDRC = 0xFF;                  //all output
       PORTD = 0xFF;                 //pull ups for dip switch
       SPCR = 0xD0;                  //enable SPI, and its interrupt
              // Provide by CodeVisionAVR to clear the SPI interrupt flag

       #asm
         in    r30, spsr
         in    r30, spdr
       #endasm

       #asm("sei")                   // Global enable interrupts
       SPDR = 0x00;                  //start SPI communication
       While (1)
          ;
}
```

Figure 9.52 SPI example system software.

Example 9.7 Using the UART for receiving messages from the PC:

This example explains how to use the UART of another Atmel microcontroller, the AT90S2313, to receive and transmit serial data via RS232 or COM ports of PC's. In this LED example only use the data reception feature. To receive data from a PC a level converter, MAX232, is used. A level converter is needed because the COMmunication port of a PC switches the data between approx. −9.23 to 9.23 Volt, −9.23 Volt corresponds with a logical '0' (lo), 9.23 Volt corresponds with a logical '1' (hi), both on TTL level of 5 Volt, so can be connected directly to the i/o's of an AVR. The MAX232 can convert at a maximum speed of 120 kbit/sec. Figure 9.53 shows a simple diagram of an RS232 converter (receive data only):

Not much components needed, only five electrolytic capacitors and a MAX232. For the PC cable use shielded data cable. The UART (software) setup for the AVR is also very simple. First set the baud rate and enable the UART, like this:

```
sbi UCR, RXEN ;enable UART (receive mode)
ldi temp, 25 ;set baud rate at 9600
```

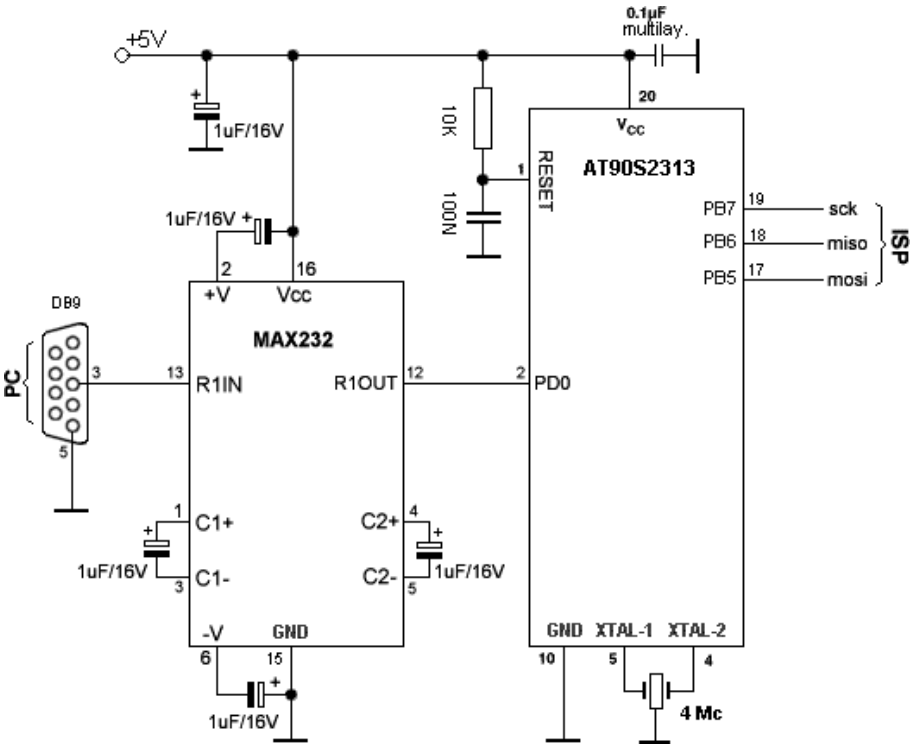


Figure 9.53 RS232 level converter MAX232 for receiving message-data for the moving message sign.

```
out UBRR, temp
```

Now the UART has been setup, now you can use the data in the software. The number 25 must be calculated, with the following formula (explained in detail in the AT90S2313 datasheet):

```
UBRR = XTAL/(baud rate*16) - 1
UBRR = 4.000.000/(9600*16) - 1
UBRR = (4.000.000/153600) - 1
         = (approx. round number down) 26 - 1 = 25.
```

Another way of calculating this number is, let the assembler do it for you. Simply put the numbers in the code, as follows:

```
    ldi temp, 4000000/(9600*16)-1 ;set baud rate
```

Now the assembler does the calculation. Much easier to change the baud rate. You can even use variables 'BAUD' and 'XTAL', like this:

```
.equ BAUD = 9600
.equ XTAL = 4000000
ldi temp, XTAL/(BAUD*16)-1 ;set baud rate
```

This way you can change the parameters of the programme from a list with variables. The advantage is when you have a huge programme, you don't have to fill in each parameter by hand, change just one variable and all others change with that variable. Figure 9.54 shows a low-cost UART interface for RS232 with low-cost standard components.

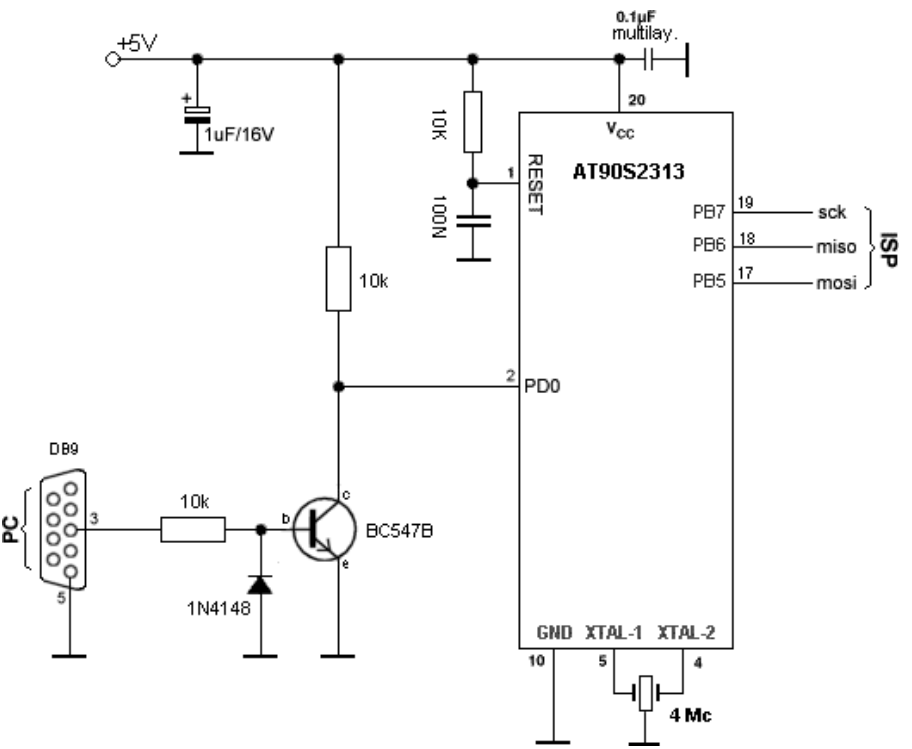


Figure 9.54 RS232 level converter with standard low-cost components for receiving message-data for the moving message sign.

9.8 Summary of the Chapter

I/O Devices and timing devices are essential in any system. The following is a summary of the important points raised in this chapter.

— An embedded system connects to external devices like keypad, multi-line display unit, printer or modem through ports.
— A device connects and accesses from and to the system processor through either a parallel or serial I/O port. A device port may be full-duplex or half-duplex. Each port has an assigned port address. The processor accesses that address in a memory-mapped I/O, as if it accesses a memory address. A decoder takes the address bus signals as the input and generates a chip select signal, CS, for the port address selection. It is activated when the processor initiates a read or write cycle from the port. This device can use the handshaking signals before storing the bits at the port buffer or before accepting the bits from the port buffer.
— Bits are received at the receiver according to the clock phases of the transmitter at the synchronous serial input and output.
— Bits are received at the receiver independent of the clock phases at the UART (asynchronous serial input and output port) transmitter. UART (in microcontrollers) usually sends a byte in a 10-bit or 11-bit format.
— We studied three networking protocols:
  – 12C bus is used between multiple ICs for inter-integrated circuit communication.
  – CAN bus is used in centrally controlled networks in automobile electronics.
  – SPI bus for high speed short distance communication.

9.9 Review Questions

9.1 The chapter introduces four common network-based architectures. What are they?
9.2 What was the originally intended purpose of the RS-232 (now EIA-232) interface standard?
9.3 What are the primary signals that comprise the EIA-232 interface standard? Give a brief description of each and its function in a data exchange.
9.4 Why is information exchange over the EIA-232 network described as asynchronous?

9.5 In an EIA-232 exchange, what is the purpose of the start and stop bits in a data character?
9.6 What are the two primary signalling lines on the 12C bus? What is the function of each?
9.7 Describe the steps involved in a bus master read operation on the 12C bus.
9.8 Describe the steps involved in a bus master write operation on the 12C bus.
9.9 With multiple masters, how is a bus contention situation resolved?
9.10 What are some of the more significant differences between the CAN bus and the 12C, USB, and EIA-232 signalling?
9.11 What are the two primary signalling lines on the CAN bus? What is the function of each?
9.12 The CAN bus signalling protocol specifies two states, dominant and recessive. What do these terms mean in this context?
9.13 Describe the format for message that is sent over the CAN bus?
9.14 What are the bit times for data transferred at the following rates:
a. 4800 baud
b. 14.4 kbaud
c. 38.4 kbaud
d. 115 kbaud
9.15 Draw the 10-bit frame for the asynchronous transfer of the following ASCII characters with even parity.
a. S
b. H
c. E
d. CR (carriage return)
9.16 Compare and contrast the transport mechanisms used for the four bus architectures discussed in the chapter. What are the strengths and weaknesses of each?
9.17 What kinds of devices can be placed on the CAN bus? Give some examples.
9.18 What kinds of message types can be exchanged over the CAN bus? Explain the purpose and meaning of each of the different types.
9.19 How are errors managed on the CAN bus? Compare the CAN strategy with that of the other three busses that we have studied in this chapter.
9.20 Typically, a Universal Asynchronous Receiver Transmitter (UART) is used to manage data flow over an EIA-232 network. Without using a UART, design a logic block that will accept 7 data bit characters, in

parallel, from your microcontroller, add an odd parity bit over the 7 bits, convert each to a 10-bit serial EIA-232 compatible character, and transmit the character over a 9600 baud bit stream.

9.21 A new generation automobile has about 100 embedded systems. How do the bus arbitration bits, control bits for address and data length, data bits, CRC check bits, acknowledgement bits and ending bits in the CAN bus help the networking devices distributed in an automobile embedded system?

9.22 Search the Internet and design a table that gives the features of the following latest generation serial buses. (i) IEEE 802.3-2000 [1 Gbps bandwidth Gigabit Ethernet MAC (Media Access Control)] for 125 MHz performance, (ii) lEE P802.3oe draft 4.1 [10 Gbps Ethernet MAC] for 156.25 MHz dual direction performance, (iii) lEE P802.3oe draft 4.1 [12.5 Gbps Ethernet MACI for four channel 3.125 Gbps per channel transreceiver performance, (iv) XAUI (10 Gigabit Attachment Unit), (v) XSBI (10 Gigabit Serial Bus Interchange), (vi) SONET OC-48, OC-192 and OC-768 and (vii) ATM OC-12/461192.

9.23 Refer to the material in this chapter and use a Web search. Design a table that compares the maximum operational speeds and bus lengths and give two example of uses of each of the followings serial devices: (i) UART, (ii) 1-wire CAN, (iii) Industrial 12C, (iv) SM 12C Bus, (v) SPI of 68 Series Motorola Microcontrollers, (vi) Fault tolerant CAN, (vii) Standard Serial Port, (viii) MicroWire, (ix) 12C, (x) High Speed CAN, (xi) IEEE 1284, (xii) High Speed I2C, (xiii) USB 1.1 Low Speed Channel and High Speed Channel, (xiv) SCSI parallel, (xv) Fast SCSI, (xvi) Ultra SCSI-3, (xvii) FireWire/IEEE 1394 and (xviii) High Speed USB 2.0.

9.24 Refer to the material in this chapter and use a Web search. Design a table that compares the maximum operational speeds and bus lengths and give two example of uses of each of the following parallel devices: (i) ISA, (ii) EISA, (iii) PCI, (iv) PCI-X, (v) COMPACT PCI, (vi) GMII, (Gigabit Ethernet MAC Interchange Interface), (vii) XGMI (10 Gigabit Ethernet MAC Interchange).

References

[1] Scott I. Mackenzi, The 8051 Microcontroller, Prentice Hall, 1999.

[2] Claus Kuhnel, AVR RISC Microcontroller Handbook, Newnes, Boston, 1998.

[3] Arnold S. Berger, Embedded Systems Design- An Introduction to Processes, Tools and Techniques, CMP Books, Nov. 2001.

[4] Harvey G.Cragon, Computer Architecture and Implementation, Cambridge University Press, 2000.
[5] David J. Lilja, Measuring Computer Performance, Cambridge University Press, 2000.
[6] Barry Kauler, Flow Design for Embedded Systems — A simple unified Object Oriented Methodology, CMP Books, Feb. 1999.
[7] Bob Zeidman, Designing with FPGAs and CPLDs, CMP Books, Sept. 2002.
[8] D. Lewis, Fundamentals of Embedded Software: Where C and Assemblymeet, Prentice Hall, Feb. 2002.
[9] Arnold S. Berger, Embedded System Design, An introduction to Processors, Tools & Techniques, CMPBooks, 2002.
[10] Frank Vahid and Tony Givargis, Embedded System - A unified Hardware! Software Introduction, John Wiley and Sons, Inc. 2002.
[11] David E. Simon, An Embedded Software Primer, Addison-Wesley, 1999.
[12] Daniel Tabak, Advanced Microprocessors, McGraw-Hill, USA 1995.
[13] Cady F. M., Microcontrollers and Microcomputers — Principles of Software and Hardware Engineering, Oxford University Press, New York, 1997.
[14] Gajski, Daniel D., Frank Vahid, Sanjiv Narayan and Jie Gong, Specification and Design of Embedded Systems, Englewood Cliffs, NJ, Prentice Hall, 1994.
[15] Peatman J.B., Design with Microcontrollers and Microcomputers, McGraw-Hill, 1988.
[16] Stewart J.W., The 8051 Microcontroller — Hardware, Software and Interfacing, Prentice Hall, 1993.
[17] Jean J Labrosse, Embedded Systems Building Blocks, Edition, CMP Books, Dec. 1999.
[18] Mazidi M. Mi and J.G. Mazidi, The 8051 Microcontroller and Embedded Systems, Pearson Education, 2000, First Indian Reprint, 2002.
[19] Kirk Zurell, C Programming for Embedded Systems, CMP Books, Feb. 2002.
[20] Calcutt M.C., F.J.Cowan, and G.H.Parchizadeh, 8051 Microcontrollers — Hardware, Software and Applications, Arnold (and also by John Wiley), 1998.
[21] AVR Instruction Set Manual 0856C-09/01 and 0856D-08/02.
[22] Intel 8051 Instruction Set: wareseeker.com/free-intel-8501-instruction set/
[23] 8051 Instruction Set Manual: http://www.keil.com
[24] http://www.maxim-ic.com

Index

RIVER PUBLISHERS SERIES IN SIGNAL, IMAGE & SPEECH PROCESSING

Other books in this series:

Volume 1
An Introduction to Digital Signal Processing
A Focus on Implementation
Stanley Henry Mneney
ISBN: 978-87-92329-12-7