COMPUTER ARCHITECTURE AND IMPLEMENTATION

HARVEY G. CRAGON
University of Texas at Austin

CAMBRIDGE
UNIVERSITY PRESS

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK http://www.cup.cam.ac.uk
40 West 20th Street, New York, NY 10011-4211, USA http://www.cup.org
10 Stamford Road, Oakleigh, Melbourne 3166, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain

First published 2000

Printed in the United States of America

Typeface Stone Serif 9.25/13 pt. *System* LaTeX 2$_\varepsilon$ [TB]

A catalog record for this book is available from the British Library.

Library of Congress Cataloging-in-Publication Data

Cragon, Harvey G.
Computer architecture and implementation / Harvey G. Cragon.
p. cm.
ISBN 0-521-65168-9 (hard). – ISBN 0-521-65705-9 (pbk.)
1. Computer architecture. I. Title.
QA76.9.A73C74 2000
004.2′2–dc21 99-16243
CIP

ISBN 0 521 65168 9 hardback

CONTENTS

ONE

COMPUTER OVERVIEW

1.0 INTRODUCTION

The general-purpose computer has assumed a dominant role in our world-wide society. From controlling the ignition of automobiles to maintaining the records of Olympic Games, computers are truly everywhere. In this book a one-semester course is provided for undergraduates that introduces the basic concepts of computers without focusing on distinctions between particular implementations such as mainframe, server, workstation, PC, or embedded controller. Instead the interest lies in conveying principles, with illustrations from specific processors.

In the modern world, the role of computers is multifaceted. They can store information such as a personnel database, record and transform information as with a word processor system, and generate information as when preparing tax tables. Computers must also have the ability to search for information on the World Wide Web.

The all-pervasive use of computers today is due to the fact that they are general purpose. That is, the computer hardware can be transformed by means of a stored program to be a vast number of different machines. An example of this power to become a special machine is found in word processing. For many years, Wang Laboratories was the dominant supplier of word processing machines based on a special-purpose processor with wired-in commands. Unable to see the revolution that was coming with the PC, Wang held to a losing product strategy and eventually was forced out of the business.

Another example of the general-purpose nature of computers is found in the electronic control of the automobile ignition. When electronic control was forced on the auto industry because of pollution problems, Ford took a different direction from that of Chrysler and General Motors. Chrysler and General Motors were relatively well off financially and opted to design special-purpose electronic controls for their ignition systems. Ford on the other hand was in severe financial difficulties and decided to use a microprocessor that cost a little more in production but did not require the development costs of the special-purpose circuits. With a microprocessor, Ford could, at relatively low cost, customize the controller for various engine configurations by changing the read-only memory (ROM) holding the program. Chrysler and General Motors, however, found that they had to have a unique controller for each configuration of

auto – a very expensive design burden. Microprocessor control, as first practiced by Ford, is now accepted by all and is the industry design standard.

A number of special-purpose computers were designed and built before the era of the general-purpose computer. These include the Babbage difference engine (circa 1835), the Anatasoff–Berry Computer (ABC) at Iowa State University in the late 1930s, and the Z3 of Konrad Zuse, also in the late 1930s. Other computers are the Colossus at Benchly Park (used for breaking German codes in World War II) and the ENIAC (which stands for electronic numerical integrater and computer, a plug-board-programmed machine at The University of Pennsylvania). These computers are discussed in Subsection 1.3.1.

Each of the early computers noted above was a one-of-a-kind machine. What was lacking was a design standard that would unify the basic architecture of a computer and allow the designers of future machines to simplify their designs around a common theme. This simplification is found in the von Neumann Model.

1.1 VON NEUMANN MODEL

The von Neumann model of computer architecture was first described in 1946 in the famous paper by Burks, Goldstein, and von Neumann (1946). A number of very early computers or computerlike devices had been built, starting with the work of Charles Babbage, but the simple structure of a stored-program computer was first described in this landmark paper. The authors pointed out that instructions and data consist of bits with no distinguishing characteristics. Thus a common memory can be used to store both instructions and data. The differentiation between these two is made by the accessing mechanism and context; the program counter accesses instructions while the effective address register accesses data. If by some chance, such as a programming error, instructions and data are exchanged in memory, the performance of the program is indeterminate. Before von Neumann posited the single address space architecture, a number of computers were built that had disjoint instruction and data memories. One of these machines was built by Howard Aiken at Harvard University, leading to this design style being called a Harvard architecture.[1]

A variation on the von Neumann architecture that is widely used for implementing calculators today is called a tagged architecture. With these machines, each data type in memory has an associated tag that describes the data type: instruction, floating-point value (engineering notation), integer, etc. When the calculator is commanded to add a floating-point number to an integer, the tags are compared; the integer is converted to floating point, the addition is performed, and the result is displayed in floating point. You can try this yourself with your scientific calculator.

All variations of the von Neumann that have been designed since 1946 confirm that the von Neumann architecture is classical and enduring. This architecture can be embellished but its underlying simplicity remains. In this section the von Neumann

[1] The von Neumann architecture is also known as a Princeton architecture, as compared with a Harvard architecture.

architecture is described in terms of a set of nested state machines. Subsection 1.2.1 explores the details of the von Neumann architecture.

We should not underestimate the impact of the von Neumann architecture, which has been the unifying concept in all computer designs since 1950. This design permits an orderly design methodology and interface to the programmer. One can look at the description of any modern computer or microprocessor and immediately identify the major components: memory, central processing unit (CPU), control, and input/output (I/O).

The programmer interface with the von Neumann architecture is orderly. The programmer knows that the instructions will be executed one at a time and will be completed in the order issued. For concurrent processors, discussed in Chapter 6, order is not preserved, but as far as the programmer is concerned order is preserved.

A number of computer architectures that differ from the von Neumann architecture have been proposed over the years. However, the simplicity and the order of the von Neumann architecture have prevented these proposals from taking hold; none of these proposed machines has been built commercially.

State Machine Equivalent

A computer is defined as the combination of the memory, the processor, and the I/O system. Because of the centrality of memory, Chapter 4 discusses memory before Chapters 5 and 6 discuss the processor.

The three components of a computer can be viewed as a set of nested state machines. Fundamentally, the memory holds instructions and data. The instructions and the data flow to the logic, then the data (and in some designs the instructions) are modified by the processor logic and returned to the memory. This flow is represented as a state machine, shown in Figure 1.1.

The information in memory is called the process state. Inputs into the computer are routed to memory and become part of the process state. Outputs from the computer are provided from the process state in the memory.

The next level of abstraction is illustrated in Figure 1.2. The logic block of Figure 1.1 is replaced with another state machine. This second state machine has for its memory the processor registers. These registers, discussed in Chapter 3, include the program counter, general-purpose registers, and various dedicated registers. The logic consists of the arithmetic and logic unit (ALU) plus the logic required to support the interpretation of instructions.

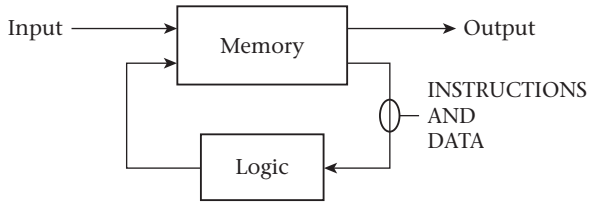


Figure 1.1 State machine

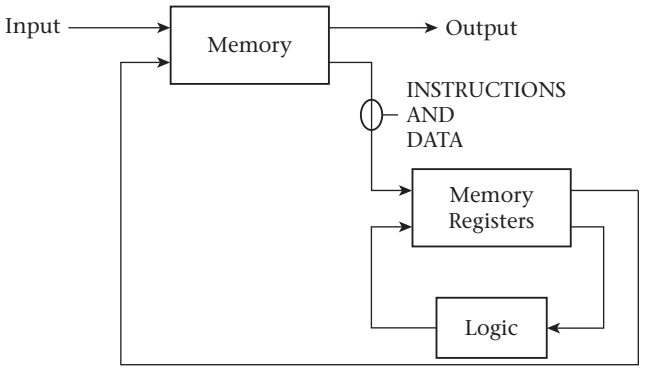


Figure 1.2 State machine II

The information contained in the registers is called the processor state. The processor state consists of (1) the information needed to interpret an instruction, and (2) information carried forward from instruction to instruction such as the program counter value and various tags and flags. When there is a processor context switch, it is the processor state that is saved, so the interrupted processor state can be restored.[2]

When microprogramming is discussed in Chapter 5, we will see that the logic block of Figure 1.2 can also be implemented as a state machine. For these implementations, there are three levels of state machine: process state, processor state, and micromachine processor state.

We now examine the major components of a computer, starting with the memory. As discussed in the preceding paragraphs, the memory is the space that holds the process state, consisting of instructions and data. The instruction space is not only for the program in execution but also for the operating system, compilers, interpreters, and other system software.

The processor reads instructions and data, processes the data, and returns results to memory, where the process state is updated. Thus a primary requirement for memory is that it be fast; that is, reads and writes must be accomplished with a small latency.

In addition, there are two conflicting requirements for memory: memory should be both very large and very fast. Memory cost is always a factor, with low cost being very desirable. These requirements lead to the concept of hierarchical memory. The memory closest to the processor is relatively small but is very fast and relatively expensive. The memory most distant from the processor is disk memory that is very slow but very low cost. Hierarchical memory systems have performance that approaches the fast memory while the cost approaches that of the low-cost disk memory. This characteristic is the result of the concept of locality, discussed in Chapter 4. Locality of programs and data results in a high probability that a request by the processor for either an instruction or a datum will be served in the memory closest to the processor.

The processor, sometimes called the CPU, is the realization of the logic and registers of Figure 1.2. This portion of the system fetches instructions, decodes these

[2] A context switch saves the processor state and restores a previously saved processor state.

instructions, finds operands, performs the operation, and returns the result to memory. The complexity of the CPU is determined by (1) the complexity of the instruction set, and (2) the amount of hardware concurrency provided for performance enhancement.

As shown in Figure 1.2, a computer must have some method of moving input data and instructions into the memory system and moving results from the memory to the outside world. This movement is the responsibility of the I/O system. Input and output devices can have differing bandwidths and latency. For example, keyboards are low-bandwidth devices whereas color display units have high bandwidth. In between we find such devices as disks, modems, and scanners.

The control of I/O can take a number of forms. At one extreme, each transfer can be performed by the CPU. Fully autonomous systems, such as direct memory access (DMA), however, provide high-bandwidth transfers with little CPU involvement. I/O systems are discussed in Chapter 7.

The formal specification of a processor, its interaction with memory, and its I/O capabilities are found in its instruction set architecture (ISA). The ISA is the programmer's view of the computer. The details of how the ISA is implemented in hardware, details that affect performance, are known as the implementation of the ISA.

1.2 THE VON NEUMANN ARCHITECTURE

The von Neumann ISA is described in this section. Except for the I/O, this architecture is complete and represents a starting point for the discussion in the following chapters. The features found in this architecture can be found in any of today's architectures; thus a thorough understanding of the von Neumann architecture is a good starting point for a general study of computer architecture. This architecture, of which a number were actually built, is used in this book for a simple example rather than the presentation of a contrived example of a simple architecture. When a von Neumann computer was actually completed at Princeton University in 1952, it was named the Institute for Advanced Studies (IAS) computer.

The von Neumann architecture consists of three major subsystems: instruction processing, arithmetic unit, and memory, as shown in Figure 1.3. A key feature of this architecture is that instructions and data share the same address space. Thus there is one source of addresses, the instruction processing unit, to the memory. The output of the memory is routed to either the Instruction Processing Unit or the Arithmetic Unit

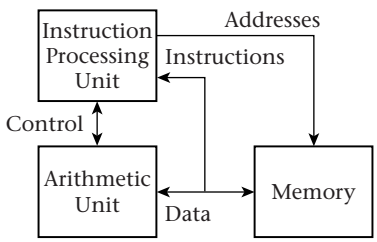


Figure 1.3 von Neumann architecture

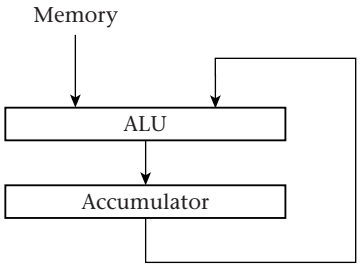


Figure 1.4 Accumulator local storage

depending upon whether an instruction or a datum is being fetched. A corollary to the key feature is that instructions can be processed as data. As will be discussed in later chapters, processing instructions as data can be viewed as either a blessing or a curse.

1.2.1 THE VON NEUMANN INSTRUCTION SET ARCHITECTURE

The von Neumann ISA is quite simple, having only 21 instructions. In fact, this ISA could be called an early reduced instruction set computer (RISC) processor.[3] As with any ISA, there are three components: addresses, data types, and operations. The taxonomy of these three components is developed further in Chapter 3; the three components of the von Neumann ISA are discussed below.

Addresses

The addresses of an ISA establish the architectural style – the organization of memory and how operands are referenced and results are stored. Being a simple ISA, there are only two memories addressed: the main memory and the accumulator.

The main memory of the von Neumann ISA is linear random access and is equivalent to the dynamic random-access memory (DRAM) found in today's processors. The technology of the 1940s restricted random-access memory (RAM) to very small sizes; thus the memory is addressed by a 12-bit direct address allocated to the 20-bit instructions.[4] There are no modifications to the address, such as base register relative or indexing. The formats of the instructions are described below in the subsection on data types.

Local storage in the processor is a single accumulator, as shown in Figure 1.4. An accumulator register receives results from the ALU that has two inputs, a datum from memory, and the datum held in the accumulator. Thus only a memory address is needed in the instruction as the accumulator is implicitly addressed.

Data Types

The von Neumann ISA has two data types: fractions and instructions. Instructions are considered to be a data type since the instructions can be operated on as data, a feature called self-modifying code. Today, the use of self-modifying code is considered

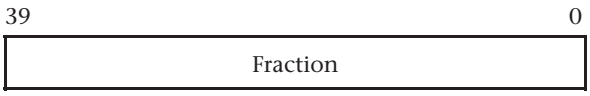
[3] An instruction set design posited by Van der Poel in 1956 has only one instruction.
[4] The Princeton IAS designers had so much difficulty with memory that only 1K words were installed with a 10-bit address.

to be poor programming practice. However, architectures such as the Intel x86 family must support this feature because of legacy software such as MSDOS.

Memory is organized with 4096 words with 40 bits per word; one fraction or two instructions are stored in one memory word.

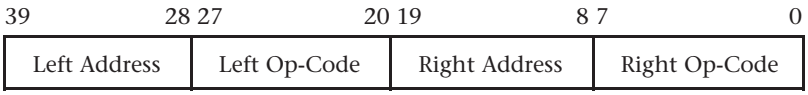
FRACTIONS

The 40-bit word is typed as a 2's complement fraction; the range is $-1 \leq f < +1$:



INSTRUCTIONS

Two 20-bit instructions are allocated to the 40-bit memory word. An 8-bit operation code, or op-code, and a 12-bit address are allocated to each of the instructions. Note that, with only 21 instructions, fewer op-code bits and more address bits could have been allocated. The direct memory address is allocated to the 12 most significant bits (MSBs) of each instruction. The address and the op-code pairs are referred to in terms of left and right:



Registers

A block diagram of the von Neumann computer is shown in Figure 1.5. Note that I/O connections are not shown. Although only sketchily described in the original paper on this architecture, I/O was added to all implementations of this design.

The von Neumann processor has seven registers that support the interpretation of the instructions fetched from memory. These registers and their functions are listed in Table 1.1. Note that two of the registers are explicitly addressed by the instructions and defined in the ISA (called architected registers) while the other six are not defined

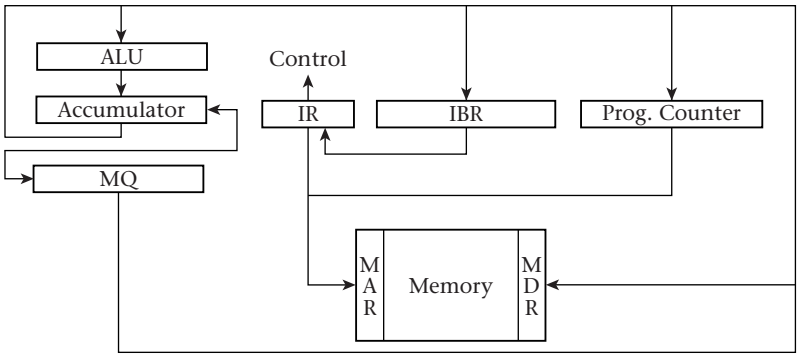


Figure 1.5 Block diagram of the von Neumann architecture: MQ, multiplier quotient register; IR, instruction register; IBR, instruction buffer register; MAR, memory address register; MDR, memory data register

TABLE 1.1 VON NEUMANN ISA REGISTERS

Name	Function
Architected Registers	
Accumulator, AC, 40 bits	Holds the output of the ALU after an arithmetic operation, a datum loaded from memory, the most-significant digits of a product, and the divisor for division.
Multiplier quotient register, MQ, 40 bits	Holds a temporary data value such as the multiplier, the least-significant bits of the product as multiplication proceeds, and the quotient from division.
Implemented Registers	
Program counter, PC, 12 bits *	Holds the pointer to memory. The PC contains the address of the instruction pair to be fetched next.
Instruction buffer register, IBR, 40 bits	Holds the instruction pair when fetched from the memory.
Instruction register, IR, 20 bits	Holds the active instruction while it is decoded in the control unit.
Memory address register, MAR, 12 bits	Holds the memory address while the memory is being cycled (read or write). The MAR receives input from the program counter for an instruction fetch and from the address field of an instruction for a datum read or write.
Memory data register, MDR, 40 bits	Holds the datum (instruction or data) for a memory read or write cycle.

* The program counter is a special case. The PC can be loaded with a value by a branch instruction, making it architected, but cannot be read and stored, making it implemented.

but are used by the control for moving bits during the execution of an instruction (called implemented registers).

Operations

The operations of the von Neumann ISA are of three types:

- moves between the accumulator, multiplier quotient register, and memory
- ALU operations such as add, subtract, multiply, and divide
- Unconditional and conditional branch instructions that redirect program flow.[5]

The von Neumann ISA consists of 21 instructions, shown in Table 1.2, which are sufficient to program any algorithm. However, the number of instructions that must

[5] Many computer historians credit the von Neumann ISA with the first use of conditional branching with a stored program computer. No prior computer possessed this feature and subprograms were incorporated as in-line code.

TABLE 1.2 THE VON NEUMANN ISA

	Move Instructions	
1.	$AC \leftarrow MQ$	Move the number held in the MQ into the accumulator.
2.	$M(x) \leftarrow AC$	Move the number in the accumulator to location x in memory. The memory address x is found in the 12 least-significant bits of the instruction.
3.*	$M(x,28{:}39) \leftarrow AC(28{:}39)$	Replace the left-hand 12 bits of the left-hand instruction located at position x in the memory with the left-hand 12 bits in the accumulator.†
4.*	$M(x,8{:}19) \leftarrow AC(28{:}39)$	Replace the left-hand 12 bits of the right-hand instruction in location x in the memory with the left-hand 12 bits in the accumulator.
	ALU Instructions	
5.	$AC_c \leftarrow M(x)$	Clear the accumulator and add the number from location x in the memory.
6.	$AC \leftarrow AC_c - M(x)$	Clear the accumulator and subtract the number at location x in the memory.
7.	$AC \leftarrow AC_c + \lvert M(x)\rvert$	Clear the accumulator and add the absolute value of the number at location x in the memory.
8.	$AC \leftarrow AC_c - \lvert M(x)\rvert$	Clear the accumulator and subtract the absolute value of the number at location x in the memory.
9.	$AC \leftarrow AC + M(x)$	Add the number at location x in the memory into the accumulator.
10.	$AC \leftarrow AC - M(x)$	Subtract the number at location x in the memory from the accumulator.
11.	$AC \leftarrow AC + \lvert M(x)\rvert$	Add the absolute value of the number at location x in the memory to the accumulator.
12.	$AC \leftarrow AC - \lvert M(x)\rvert$	Subtract the absolute value of the number at location position x in the memory into the accumulator.
13.	$MQ_c \leftarrow M(x)$	Clear the MQ register and add the number at location x in the memory into it.
14.	AC_c, $MQ \leftarrow M(x) \times MQ$	Clear the accumulator and multiply the number at location x in the memory by the number in the MQ, placing the most-significant 39 bits of the answer in the accumulator and the least-significant 39 bits of the answer in the MQ.
15.	MQ_c, $AC \leftarrow AC \div M(x)$	Clear the register and divide the number in the accumulator by the number at location x of the memory, leaving the remainder in the accumulator and placing the quotient in MQ.
16.	$AC \leftarrow AC \times 2$	Multiply the number in the accumulator by 2.
17.	$AC \leftarrow AC \div 2$	Divide the number in the accumulator by 2.
	Control Instructions	
18.	Go to $M(x,2\ 0{:}39)$	Shift the control to the left-hand instruction of the pair in $M(x)$.
19.	Go to $M(x,0{:}19)$	Shift the control to the right-hand instruction of the pair in $M(x)$.
20.	If $AC \geq 0$, then $PC \leftarrow M(x,0{:}19)$	If the number in the accumulator is ≥ 0, go to the right-hand instruction in $M(x)$.
21.	If $AC \geq 0$, then $PC \leftarrow M(x,20{:}39)$	If the number in the accumulator is ≥ 0, go to the left-hand instruction in $M(x)$.

* These instructions move the address portion of an instruction between memory and the accumulator. These instructions are required to support address modification. Indexing, common today in all computer's ISAs, had not yet been invented.

† The notation $M(x,0{:}19)$ means the right-hand 20 bits of location $M(x)$; $M(x,20{:}39)$ means the left-hand 20 bits, and so on.

be executed is considerably greater than that required by more modern ISAs. The instructions are grouped into three groups: move, ALU, and control. This grouping is typical of all computers based on the von Neumann architecture. A more modern terminology, not the terminology of von Neumann, is used in Table 1.2.

1.2.2 INSTRUCTION INTERPRETATION CYCLE

Interpretation of an instruction proceeds in three steps or cycles. The instruction is fetched, decoded, and executed. These three steps are discussed in the following subsections.

Instruction Fetch

A partial flow chart for the instruction fetch cycle is shown in Figure 1.6. Because two instructions are fetched at once, the first step is to determine if a fetch from memory is required. This test is made by testing the least-significant bit (LSB) of the program counter. Thus, an instruction fetch from memory occurs only on every other state of the PC or if the previous instruction is a taken branch. The fetch from memory places a left (L) and a right (R) instruction in the instruction buffer register (IBR).

Instructions are executed, except for the case of a branch instruction, left, right, left, right, etc. For example, consider that an R instruction has just been completed.

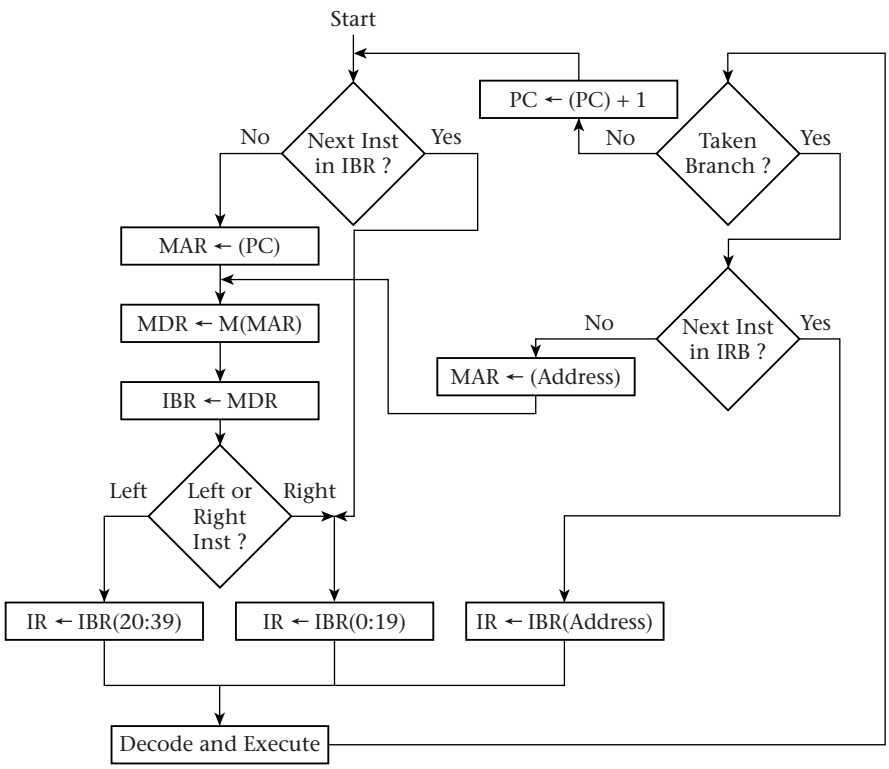


Figure 1.6 Instruction fetch cycle

There is no instruction in the IBR and a reference is made to memory to fetch an instruction pair.

Normally, the L instruction is then executed. The path follows to the left, placing the instruction into the instruction register (IR). The R instruction remains in the IBR for use on the next cycle, thereby saving a memory cycle to fetch the next instruction.

If the prior instruction had been a branch to the R instruction of the instruction pair, the L instruction is not required, and the R instruction is moved to the IR. In summary, the instruction sequence is as follows:

Sequence	*Action*
L followed by R	No memory access required
R followed by L	Increment PC, access memory, use L instruction
L branch to L	Memory access required and L instruction used
R branch to R	Memory access required and R instruction used
L branch to R	If in same computer word, memory access not required
R branch to L	If in same computer word, memory access not required

After the instruction is decoded and executed, the PC is incremented for the next instruction and control returns to the start point.

Decode and Execute

Instruction decode is only indicated in Figure 1.6. However, the instruction has been placed in the IR. As shown in Figure 1.7, combinatorial logic in the control unit decodes the op-code and decides which of the instructions will be executed. In other words, decoding is similar to the CASE statement of many programming languages. The flow charts for two instruction executions are shown in Figure 1.7: numbers 21 and 6. After an instruction is executed, control returns to the instruction fetch cycle, shown in Figure 1.6.

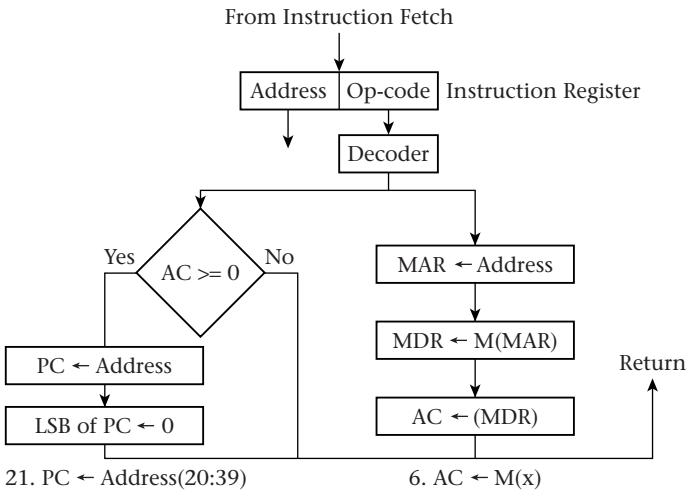


Figure 1.7 Decode and execute

The sequencing of the instruction interpretation cycle is controlled by a hardwired state machine, discussed in Chapter 5. Each of the states is identified in flowchart form, flip flops are assigned to represent each state, and the logic is designed to sequence through the states. After the invention of microprogramming, the flow chart is reduced to a series of instructions that are executed on the micromachine. In other words, a second computer, rather than a hardwired state machine, provides the control.

Example Program

Without indexing, the complexity of programming the von Neumann ISA is illustrated with the following example shown in Table 1.3. We wish to compute the vector add of two vectors of length 1000:

$$\mathbf{C}i = \mathbf{A}i + \mathbf{B}i.$$

Vector **A** is stored in locations 1001–2000, vector **B** in locations 2001–3000, and vector **C** in locations 3001–4000. The first steps of the program initialize three memory

TABLE 1.3 VECTOR ADD PROGRAM

Location	Datum/Instruction	Comments
0	999	Count
1	1	Constant
2	1000	Constant
	Inner Loop for Each Add	
3L	AC ← M(3000)	Load **B***i*
3R	AC ← AC + M(2000)	**B***i* + **A***i*
4L	M(4000) ← AC	Store AC
	Loop Test and Continue/Terminate	
4R	AC ← M(0)	Load count
5L	AC ← AC − M(1)	Decrement count
5R	If AC ≥ 0, go to M(6,0:19)	Test count
6L	Go to M(6,20:39)	Halt
6R	M(0) ← AC	Store count
	Address Adjustment (Decrement)	
7L	AC ← AC + M(1)	Increment count
7R	AC ← AC + M(2)	Add constant
8L	M(3,8:19) ← AC(28:39)	Store modified address in 3R
8R	AC ← AC + M(2)	Add constant
9L	M(3,28:39) ← AC	Store modified address in 3L
9R	AC ← AC + M(2)	Add constant
10L	M(4,28:39) ← AC	Store modified address in 4L
10R	Go to M(3,20:39)	Unconditional branch to 3L

locations with the count 999, the constant 1000 (for testing the number of times the operation is performed), and the constant 1 (for a decrement value).

1.2.3 LIMITATIONS OF THE VON NEUMANN INSTRUCTION SET ARCHITECTURE

There are a number of major limitations of the von Neumann ISA highlighted by the vector add program of Table 1.3. The first limitation has been noted in Subsection 1.2.2: there are no facilities for automatic address modification as with modern processors. Thus the addresses in the instructions must be modified by other instructions to index through an array. This is the self-modifying code that is very prone to programming error.

In addition, modular programming was unknown at the time of the von Neumann ISA development. Thus the architecture provides no base register mode to assist in partitioning instructions and data.

Another major limitation can be found in the vector add program of Table 1.3. With this architecture, the program counter is an implemented register. All modern processors have an architected program counter; thus the PC can be stored and restored, thereby enabling the programming concepts of subroutines and procedure calls. These concepts cannot be used on the von Neumann ISA with its implemented program counter.

Finally, as mentioned in Subsection 1.2.1, the I/O was only briefly mentioned in the original paper on the von Neumann ISA. The implementation of the I/O on this and other computers will be discussed in Chapter 7.

1.3 HISTORICAL NOTES

Indexing: Apparently, the first incorporation of indexing to an ISA was with the Mark I, developed by Kilburn and Williams at The University of Manchester, 1946–1949, and produced by Ferranti Corp. The first Ferranti machine was installed in 1951. Although this machine is well known for the development and the use of virtual memory, the pioneering work regarding indexing is equally important. Indexing was provided as an adjunct function, called the B lines or B box.

The IBM 704, announced in 1954, has three index registers. These registers, along with floating point, provided the hardware support for the development of FORTRAN (Blaauw and Brooks 1997). It is interesting to note that the IBM 701, first installed in 1953, required program base address modification, as did the von Neumann ISA.

Subroutines: A subroutine is a program that executes only when called by the main program. After execution, the main program is then restarted. Because subroutines require a return to the main program, a necessary condition for subroutine support is that the program counter must be saved. As the von Neumann ISA had no provisions for saving the program counter, the ISA cannot execute subroutines.

This problem was first solved by Wheeler for the EDSAC at Cambridge University under the direction of Maurice Wilkes (Wilkes, Wheeler, and Gill 1951). The program

counter became architected, permitting its contents to be saved and restored. A further discussion of subroutines is found in Chapter 3.

1.3.1 PRECURSORS TO THE VON NEUMANN INSTRUCTION SET ARCHITECTURE

There were a number of precursor computers to the von Neumann ISA. Noteworthy are the machines of Babbage, Atanasoff, Zuse, Aiken, and Eckert-Mauchly. Even before these machines there had been several centuries of study and design of calculating aids and mathematics, particularly the invention of logarithms and advances in numerical analysis. These topics are outside the scope of this textbook but reference can be made to the papers of Chase (1980) and de Solla Price (1984) for information. For additional information on the early computers, the reader is referred to Goldstein (1972) and Randell (1973). Extensive coverage can be found in the *IEEE Annals of the History of Computing*, now in its 21st volume. The reprint series of the Charles Babbage Institute (Cambell-Kelly and Williams 1985) provides insight into the writings of the early computer pioneers.

The early computers were designed to solve two types of problems. First, there was a problem in preparing tables of functions; the machines of Babbage, Aiken, and Eckert-Mauchly were designed for this purpose. The second problem class was the solution of systems of linear equations; the machines of Atanasoff and Zuse were designed for these problems. For both problem types, the amount of labor required for human solutions to anything but trivial cases was excessive, and, as the problem became larger, errors would be made even when mechanical calculators were used for the arithmetic. Thus very large problems were almost intractable; they took too long and were too full of errors.[6]

Charles Babbage (1792–1871) can be called the father of modern computers. At the beginning of the 19th century, a critical need existed for computationally reliable printed tables of functions. Babbage posited that a difference engine could meet this need. He built a small model during 1820–1822 of a difference engine having six significant digits for functions with a constant second difference; a replica is shown in Figure 1.8. The method of differences was adopted as it requires only addition and subtraction. The technology for these operations had been long demonstrated, starting with Blasé Pascal's mechanical adding machine, which used rotating decimal wheels.

In 1836, efforts to make a larger, 26-decimal-digit, sixth-difference version of the difference engine eventually failed because of the difficulty of manufacturing the parts and the withdrawal of British government financial aid. A reproduction of this larger machine, built to Babbage's drawings, was completed in 1991 and can be found in the London Science Museum. Even though Babbage never completed the difference

[6] During the 1930s, the Works Progress Administration hired young women to compute tables by hand. Calculators were not used, as one of the goals of the program was to put as many people to work as possible.

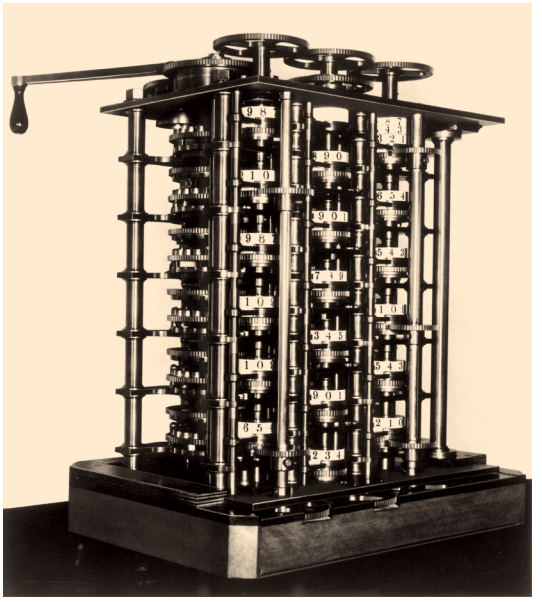


Figure 1.8 Replica of Babbage difference engine (courtesy International Business Machines Corporation)

engine, a machine based on his ideas was completed by Edvard Scheutz in 1843 in Stockholm, Sweden.

With the failure of the difference engine design effort, Babbage turned his attention to an "analytical engine." He attempted to design a machine that could perform any sequence of operations and that would not be a slave to the numerical analysis technique of differences. The analytical engine consisted of three parts: the store is the memory of a modern computer, the mill is the arithmetic unit of a modern computer, and the third part is the control. The store and the mill would be sequenced by punched cards, a technology perfected by the Jacquard loom. Thus Babbage spelled out the three major components of a modern computer.

John Atanasoff (1903–1995), Professor of Mathematics and Physics at Iowa State College, made the first machine that brought computer technology out of the mechanical and relay ages into the electronics age. Atanasoff saw the need for an apparatus for solving systems of linear algebraic equations. He estimated that the time in hours for using mechanical calculators to solve n simultaneous linear equations with n unknowns was

$$\frac{n^3}{64} \text{ h.}$$

Thus, large systems of equations, such as $n = 30$, would require over 50 8-h days, with the opportunity for many errors. In the spirit of physics researchers, he proceeded to design and construct an apparatus for the solution of these systems of equations. This apparatus is a special-purpose digital computer known as the Atanasoff–Berry Computer (ABC). Berry was a research assistant to Atanasoff. Unfortunately, Atanasoff's work did not become widely known until 1967–1969 when patent litigation over the ENIAC patent between Sperry-Rand and Honeywell revealed that John Mauchly

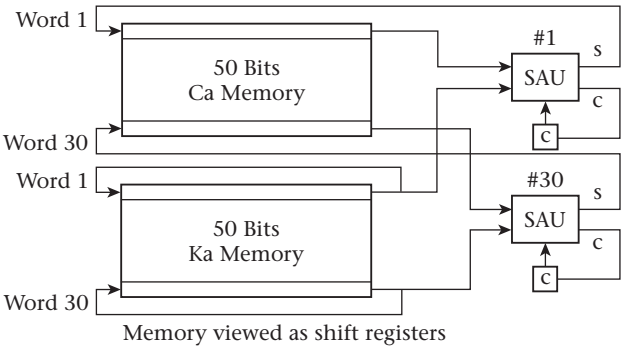


Figure 1.9 Organization of the ABC memory and SAUs

(one of the designers of ENIAC) visited with Atanasoff in 1940 or 1941 and received information on the ABC.

In addition to the decision to construct an electronic computer, Atanasoff made a number of other design decisions that are noteworthy. First, arithmetic would be performed by logical operations, not enumeration or counting as with Babbage. Second, the data type would be a 50-bit signed, binary integer that uses 1's complement (see Chapter 3 for a discussion of 1's complement arithmetic). Third, memory would be provided by a motor-driven drum of rotating capacitors: the first DRAM. The rotation time is 1s; the capacitors are refreshed each rotation.

The organization of the memory and the serial arithmetic units (SAUs) is shown in Figure 1.9. There are two memory banks, Ca and Ka. Each bank is 30 words of 50 bits. The coefficients and constant of one equation are loaded into Ca and the coefficients and constant of another are loaded into Ka. Thirty full adder/subtractors add or subtract the bits of Ka and Ca, saving the carry, and replacing Ca with the sum, bit by bit. With 30 words, the system limit is 29 equations with 29 unknowns.

The coefficients and the constant of one equation are read from the card reader into the Ca memory and the other equation into the Ka memory. By Gaussian elimination, one of the coefficients in Ca is reduced to zero. The technique iteratively subtracted the 30 coefficients in Ka from Ca, reducing one coefficient of Ca to zero (the coefficient to be eliminated). The process consists of successively dividing the coefficients by two in each cycle. When the coefficient becomes negative, subtraction is changed to addition; when the coefficient becomes positive, addition is changed to subtraction – a process akin to nonrestoring division (a topic discussed in Chapter 3). After one coefficient of Ca is reduced to zero, the results are punched into an output card and two new equations are introduced by the card reader; the problem has been reduced by one unknown. This process of input, compute, and output continues until all of the coefficients of the system are available.

Figure 1.10 (Atanasoff 1984) shows an artist's conception of the final version of the ABC tested in the spring of 1942. Atanasoff reported difficulty with reliable punching of the output cards. Because of wartime demands on Anatasoff and his students' leaving school for the military or war work, no further work was accomplished, and the machine was eventually scrapped. Today, elements of the machine can be seen at The

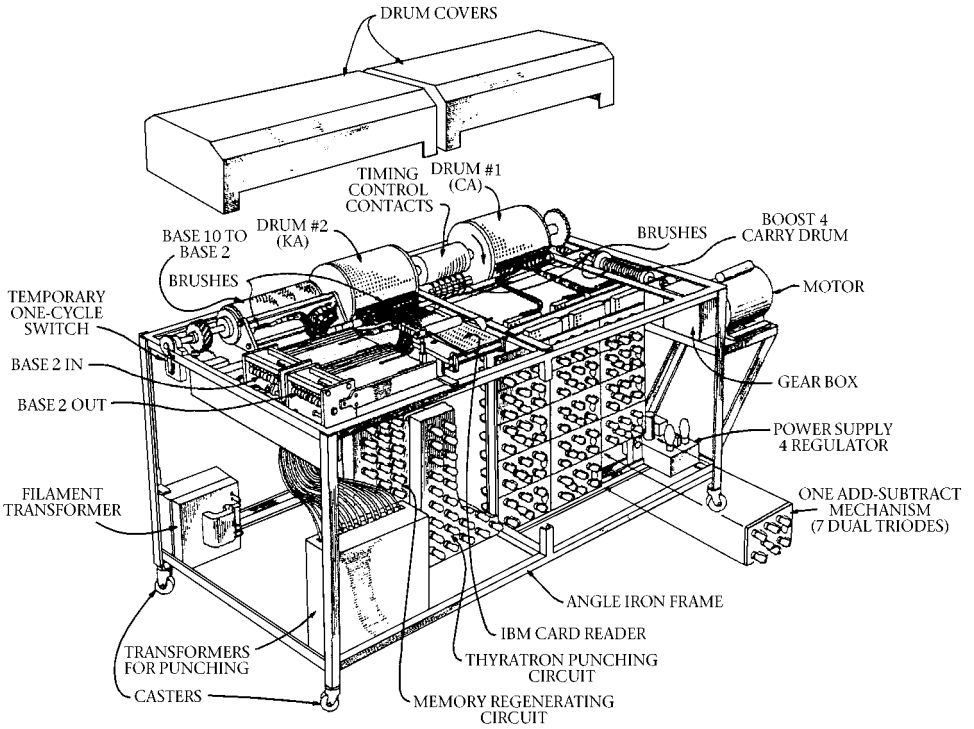


Figure 1.10 The Atanasoff–Berry computer (© 1984 IEEE)

Computer Museum in Boston. A working replica of the ABC has been constructed at Iowa State University and is on permanent display there.

Konrad Zuse (1910–1995) worked in Germany and designed and built a number of machines during the period 1934–1944: Z1, Z2, and Z3. The primary motivation for these machines was the solution of systems of linear equations. Because he had a degree in civil engineering, Zuse was interested in the problem of static engineering of load-bearing structures. This problem required the solution of a system of linear equations, one example requiring 30 equations with 30 unknowns. Several months were required for solving this problem with mechanical calculators (Ceruzzi 1981, Rojas 1997).

Zuse's work seems to have been supported by his family and occasional small grants from the Henschel Aircraft Company. Unfortunately, the Z3 was destroyed in a bombing raid on Berlin, but a reproduction was built in 1961–1963 from the original plans.

The Z3 is a relay machine, with approximately 2600 relays. A block diagram of the Z3 is shown in Figure 1.11 (Ceruzzi 1981). Note that the primary data type is floating point, base 2, and the program input is from previously used punched 35-mm movie film. The main memory (64 words of 22 bits plus sign) is directly addressed and implemented with relays. Output used lamp displays, and no provisions for printing results were provided. Input is by means of a keyboard. With disjoint program and data memories, the Z3 is classified as a Harvard architecture. The minimum provision

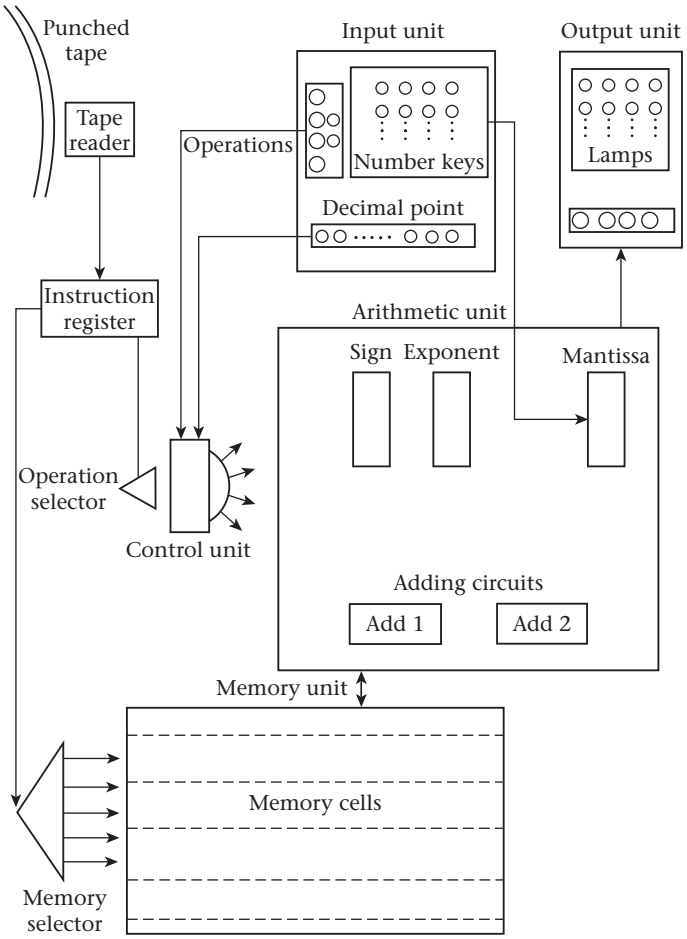


Figure 1.11 Zuse Z3 computer (© 1981 IEEE)

for input and output is not surprising, given that solving large systems of simultaneous linear equations is compute bound: few inputs, a lot of computing, and few outputs.

The internal architecture is a basic register file of two registers. Instructions were of the form

$$R2 \leftarrow R1 \text{ op } R2.$$

The instruction set consisted of 11 operations–everything we expect in a modern computer except for conditional branching, the ability to use subroutines, and indexing. Looping could be provided when a loop was made of the punched film.

The instruction set and the number of cycles required per instruction are given in Table 1.4. The cycle time is estimated to be 5.33 Hz. Finally, a rudimentary form of pipelining and concurrency is used. The tape reader is two instructions ahead of the instruction in execution. This allows two instructions to be reversed in execution,