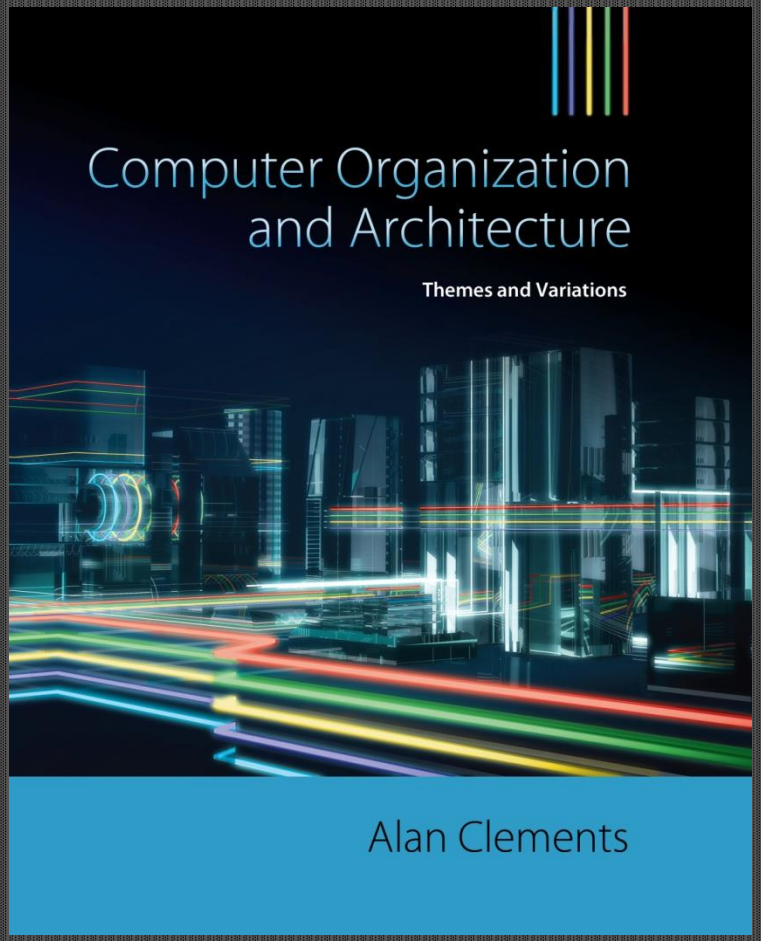
Chapter 1

Computer Systems Architecture



Structure of the Course

Part I ***Foundations*** introduces the concepts, history and underlying technology of digital computers.

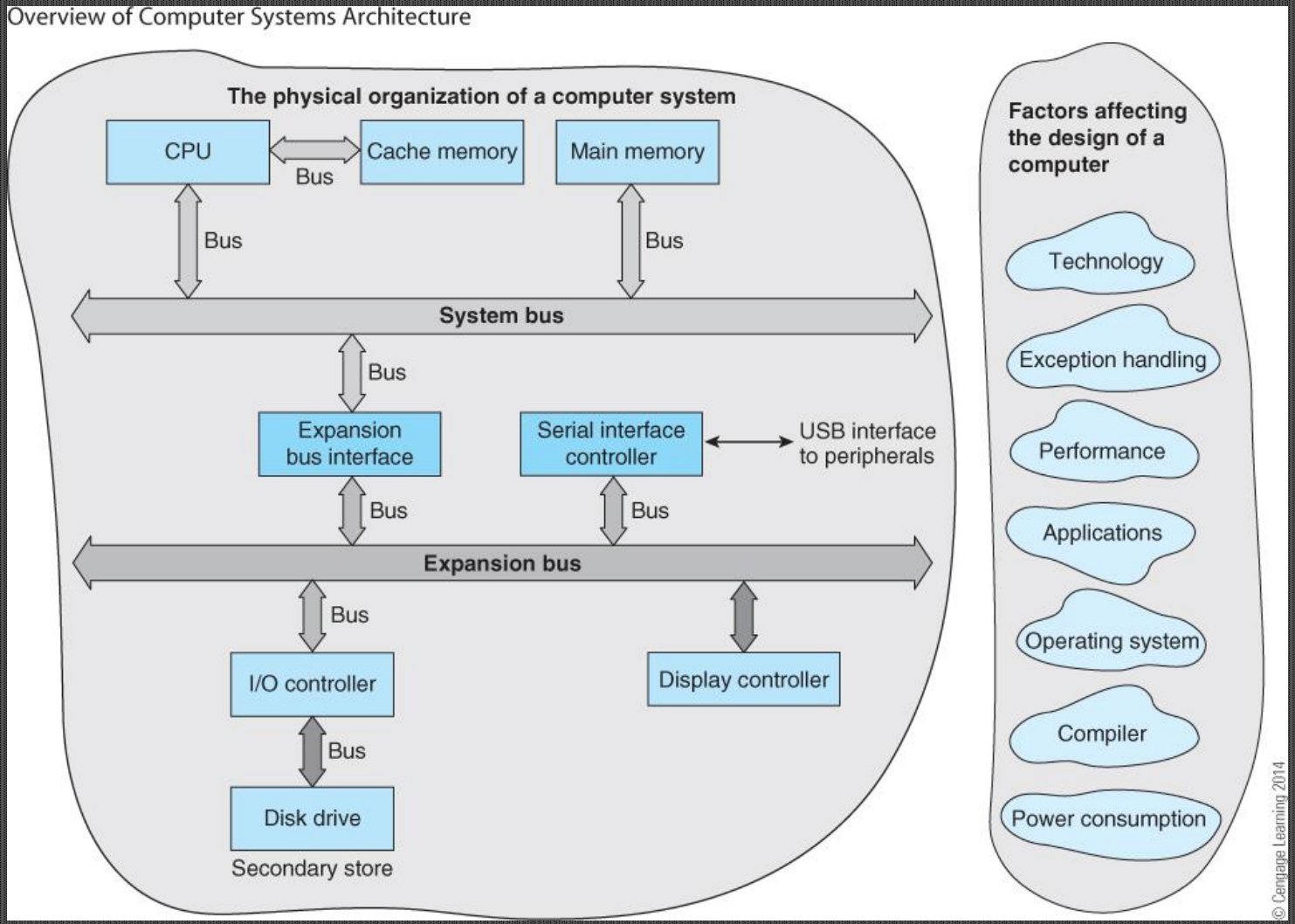
Part II ***Instruction set architectures (ISAs)*** looks at the programming model of a computer. We introduce the register model of a computer, its instruction types, and the addressing modes of a typical microprocessor.

Part III ***Organization and Performance*** describes how we measure the performance of computers.

Part IV ***The Computer System*** covers the other parts of a computer required to convert the microprocessor chip into a complete system; for example, peripheral subsystems and the wide range of memory systems, storage devices, and buses available to the computer systems' designer.

Part V ***Processor Level Parallelism*** goes beyond the single-processor computer and introduces the notion of computers with multiple processors.

Factors affecting a computer's design



Computer Architecture

1. A computer is characterized by its instruction set architecture, ISA
2. An instruction set architecture defines the programming model of a computer
3. An ISA is an abstract entity because it does not consider the specific design or implementation of a computer
4. An ISA is concerned with the computer's register set, instruction set, and addressing modes
5. The computer's assembly language embodies its ISA

Computer Organization

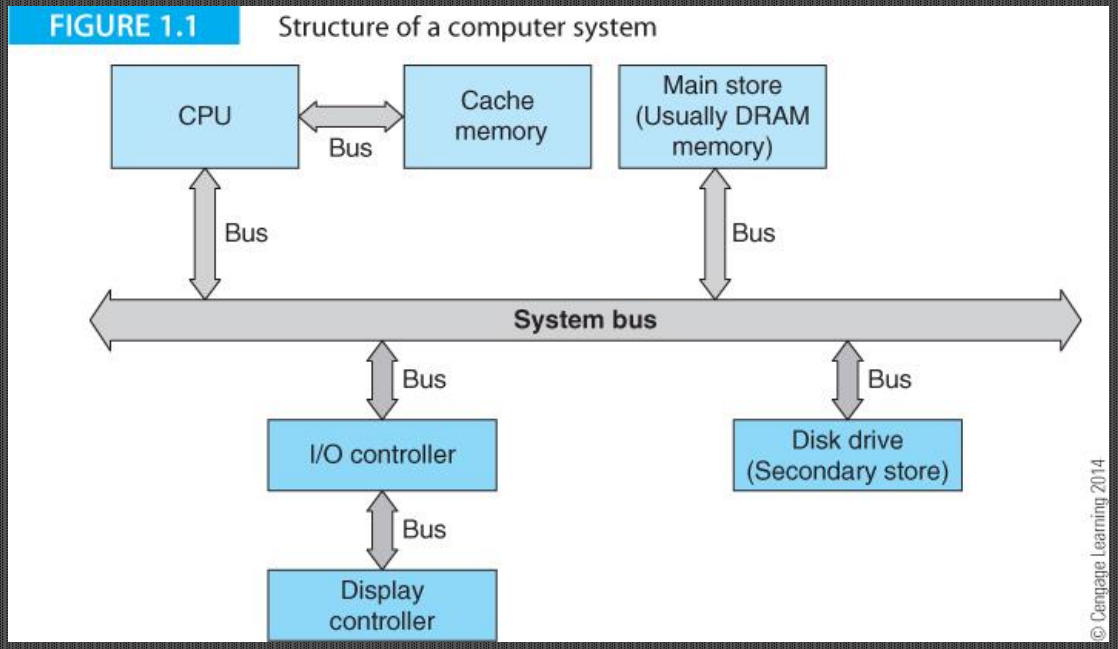
1. Computer organization is concerned with the implementation of an ISA
2. Any given ISA can have many different organizations
3. Computer manufacturers regular modify the architecture of a processor while keeping its ISA essentially constant
4. Today, a computer's organization is often referred to as its microarchitecture
5. In theory, architecture and organization are orthogonal; that is, they are entirely independent
6. You could say that architecture tells you what a computer does and organization tells you how it does it

Computer Structure

1. Figure 1.1 describes the structure of a computer. The term computer describes the entire system. The CPU is the central processing unit that reads instructions and executed them. The CPU is often synonymous with microprocessor.

2. Modern microprocessors usually include cache (high-speed) memory on-chip.

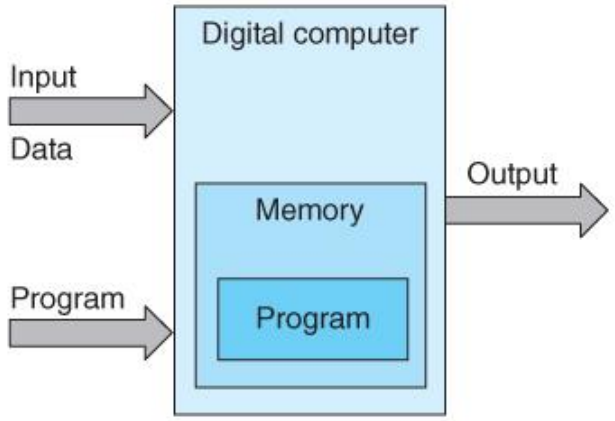
3. A key component of computers is the bus (or family of busses) that moves information round the computer between different functional units.



Computer Types

1. Computers are dedicated or general-purpose. A dedicated computer solves only one class of problems (e.g., a computer in a calculator, a cruise speed control, or washing machine).
2. A general-purpose computer can be programmed to solve any problem.
3. Figure 1.2 describes the structure of a general-purpose computer.
4. A key feature of the general-purpose computer is that the program and its data are held in the same memory.
5. Such a computer is called a **von Neumann machine**.

FIGURE 1.2 The general-purpose computer



General-purpose digital machine. By changing the program, this machine can carry out any task capable of being performed by a computer.

The Register

A register is a memory element that holds a single unit or word of data. A register is specified in terms of the number of bits it holds, which is typically, 8, 16, 32, or 64. Many of the computers we discuss in this text have either 32-bit or 64-bit-wide registers.

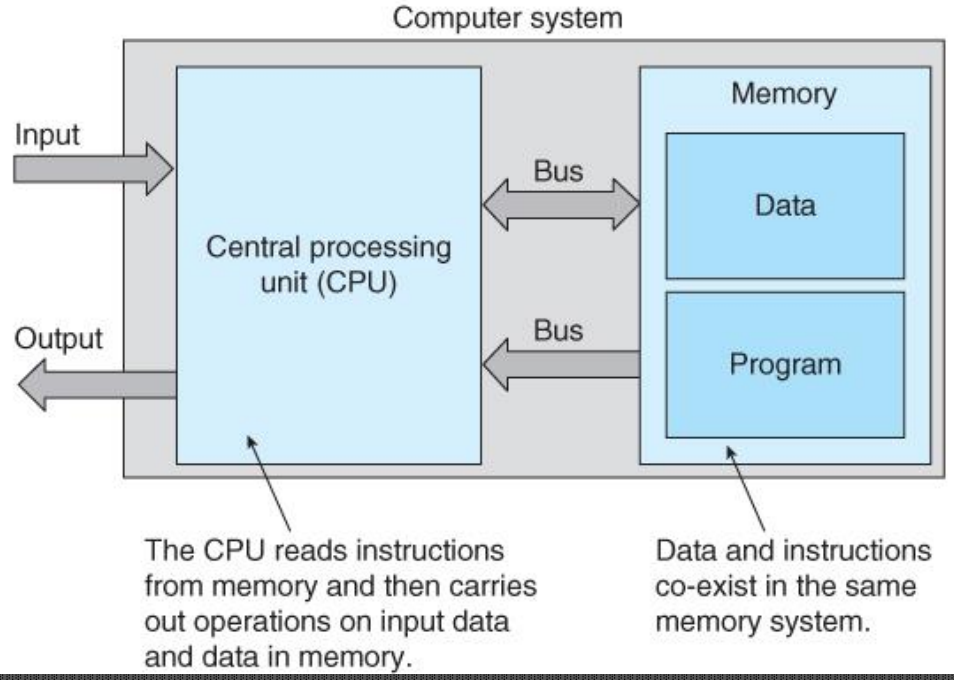
There is no fundamental difference between a register and a word in memory. The practical difference is that registers are located within the CPU and can be accessed more rapidly than external memory.

Stored Program Computer

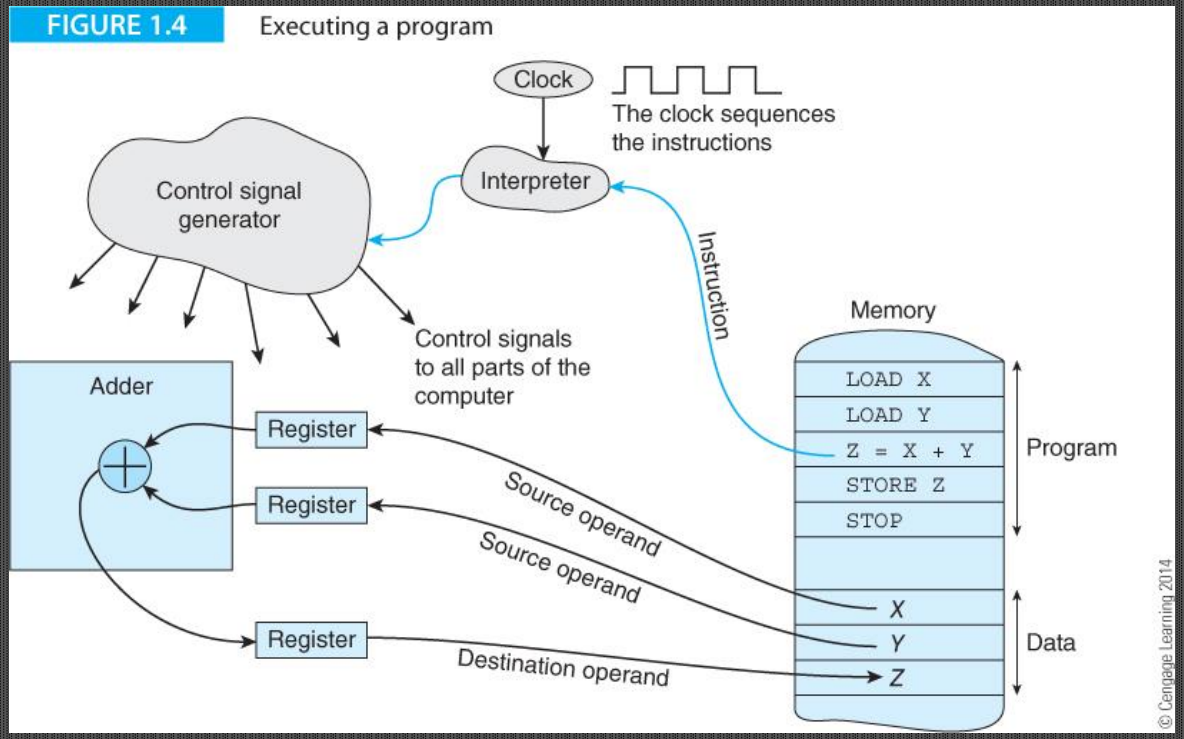
1. Figure 1.3 emphasizes the nature of the stored program computer.

2. A general-purpose computer can be programmed to solve any problem (that is possible of solving).

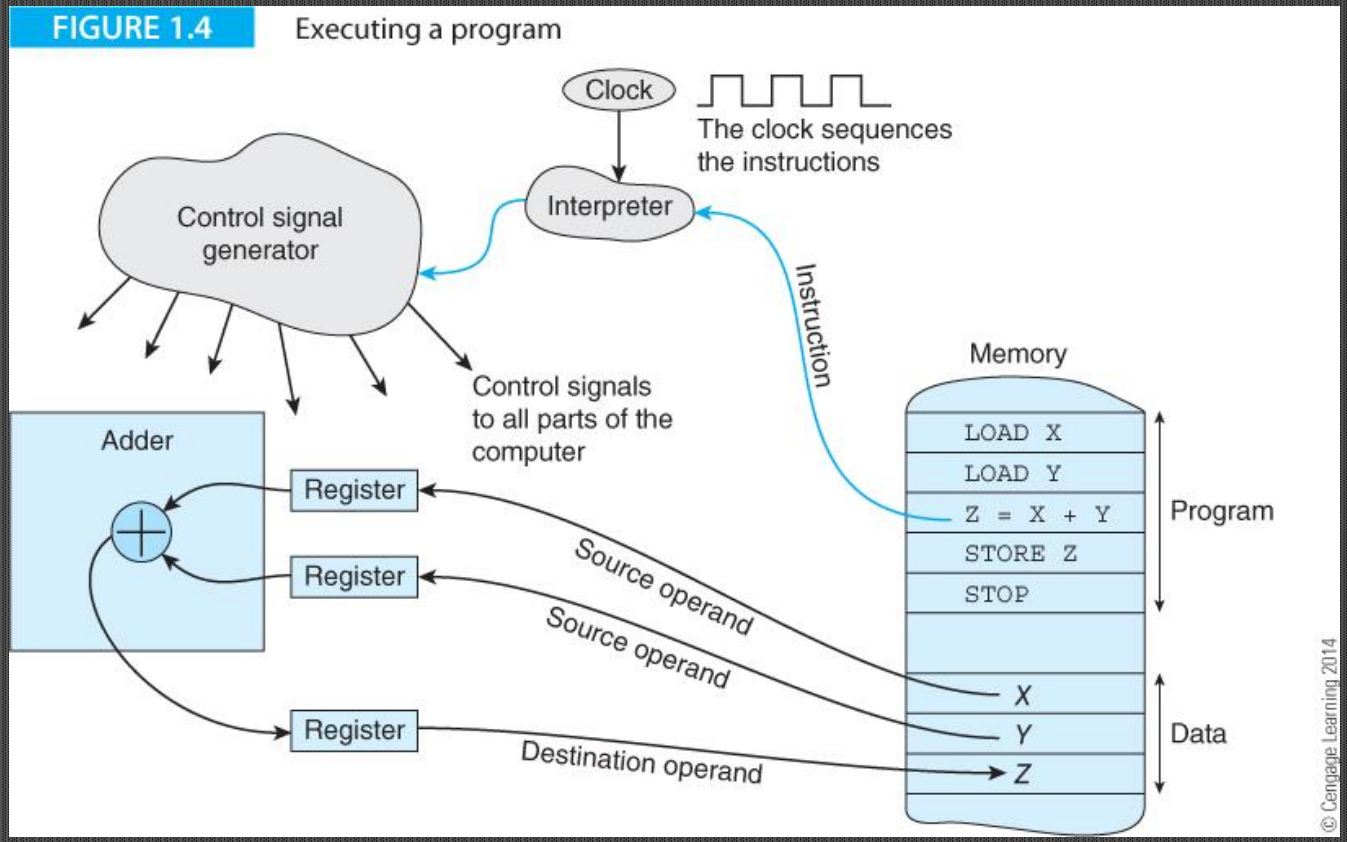
FIGURE 1.3 Structure of the stored program computer



1. Figure 1.4 illustrates the operation of a stored program.
2. A clock (a stream of pulses) sequences all operations in a computer. All events in a computer are triggered by clock pulses. The most primitive events are the movement of data from A to B, or the capture of data in a register, or a simple operation on data such as A + B.
3. Figure 1.4 illustrates how the operation Z = X + Y is read from memory, interpreted and used to add X and Y to create Z.



1. LOAD moves data from memory to a register and STORE moves data from a register to memory.
2. Z = X + Y performs a simple operation on data (addition).
3. Memory is a bottleneck because instructions have to flow from it. Data has to flow from it to take part in operations and back to to store the result.



The Clock

Most digital electronic circuits have a clock that generates a continuous stream of regularly spaced electrical pulses. It's called a clock because the pulses are used to time or sequence all events within the computer; for example, a processor might execute a new instruction each time a clock pulse arrives.

A clock is defined in terms of its *repetition rate* or *frequency*. Typical clock frequencies in computers range from 1 MHz to about 4.5 GHz.

Clocks are also defined in terms of the width of a clock pulse, which is the reciprocal of its frequency; that is f = 1/T; for example a 1 MHz clock has a duration of 1 µs, and a 1 GHz clock has a duration of 1×10^{-9}s or 1 ns.

A 5 GHz clock has a period of 200 ps (ps = picoseconds). Light travels approximately two inches in 200 ps.

Digital circuits whose events are triggered by a clock are called *synchronous*. Some events are asynchronous because they can happen at any time. For example, if you move the mouse, it sends a signal to the computer. That is an asynchronous event. However, the computer may check the status of the mouse at each clock pulse; that is a synchronous event.

Factors determining computer design

1. Figure 1.5 illustrates some of the factors affecting the design of a computer.

2. Figure 1.6 shows how a computer is dependent on several entirely different technologies.

FIGURE 1.5 Factors affecting the computer designer

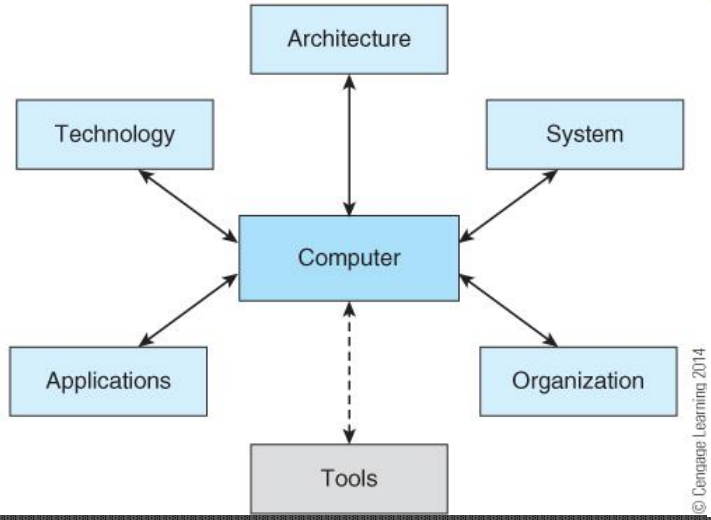
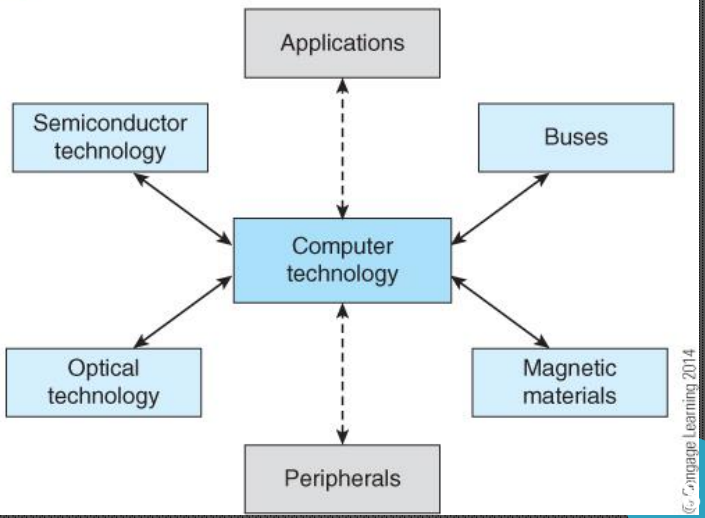


FIGURE 1.6 Computer technologies



Computer History

1. The computer has a very long history – longer than many believe. It has been reported that the Greeks (over 2,000 years ago) had a highly sophisticated mechanical calculator that could track the moon and make predictions about eclipses.

2. In the 1830s Charles Babbage designed the first mechanical computing engine. It was never constructed, but it incorporated the basic features of a computer: program, data, storage, and processing unit.

3. The invention of the telegraph, the telephone network, and the wireless were all precursors to the invention of the computer.

4. The computer as we know it emerged in 1940s and early 1950s. It is both unfair and impossible to name any single person as the inventor of the computer because it was being developed simultaneously by groups in various countries.

The Analog Computer

1. An analog computer simulates a system. There is no program

2. Probably the most famous analog computer is the governor used by steam engines to maintain speed. When the vertical spindle rotates, the two balls swing out. The arms from which they hang controls the flow of steam. If the engine load is decreased, it speeds up, the balls swing out further and reduce the flow of steam to bring the speed down again.

3. This demonstrate that there was a need for automation long before our era and that engineers have been interested in automatic control for a long time.

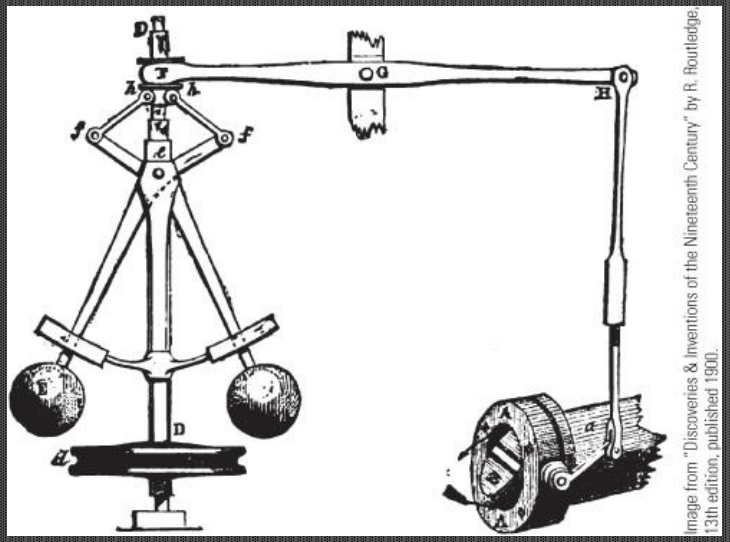
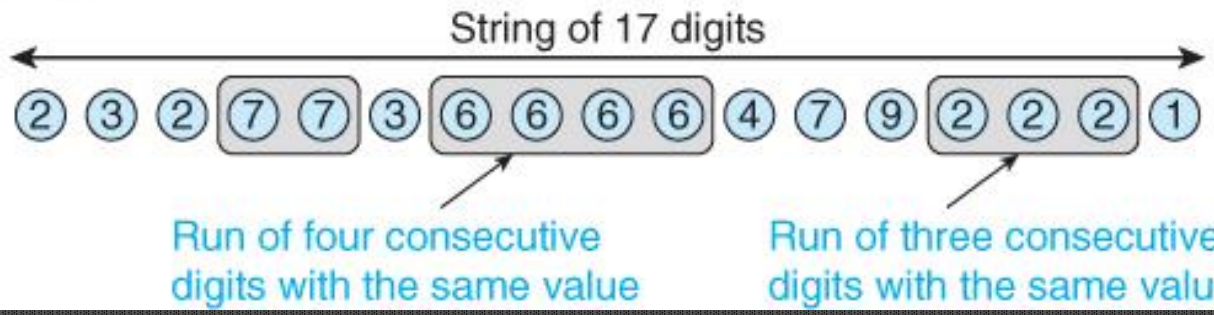


Image from "Discoveries & Inventions of the Nineteenth Century" by R. Routledge, 13th edition, published 1900.

Introducing the Computer by Solving a Problem

1. Before introducing the computer itself, we look at what is needed to solve a simple problem.
2. We want to find the longest sequence of repeated digits in a stream of digits.
3. In figure 1.7 the longest run of repeated digits is four consecutive sixes.
4. How can we automate this? What do we need to do?

FIGURE 1.7 A string of digits



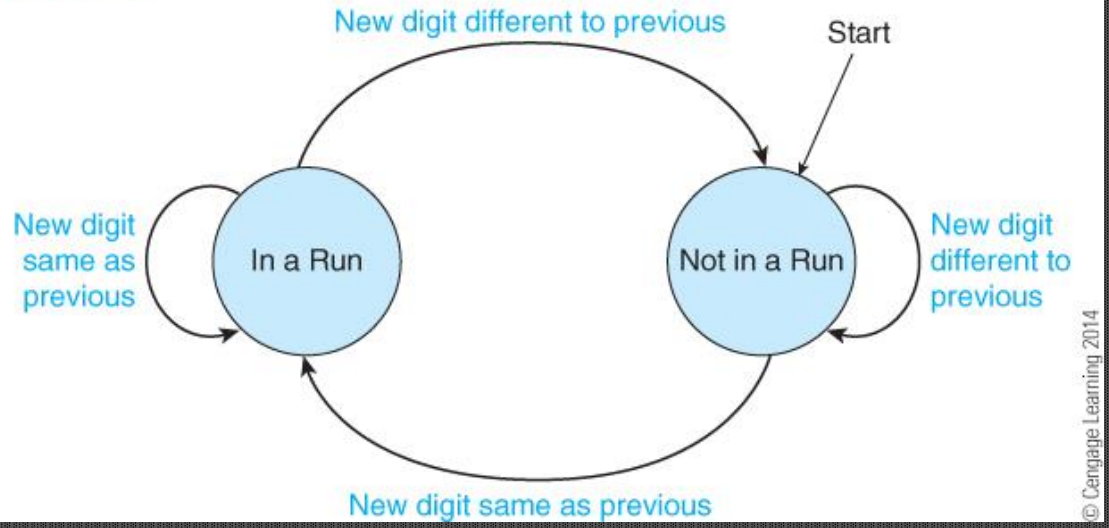
1. We are going to solve this problem sequentially by examining a digit at a time.

2. We could design a parallel machine or a dedicated machine – but we won't.

3. We will take a general-purpose problem solving approach.

4. One way of solving this problem is to note that we are always in one of two states: in a sequence of repeated digits, or at the start of a new sequence. Figure 1.8 demonstrates how we can illustrate this with a state diagram.

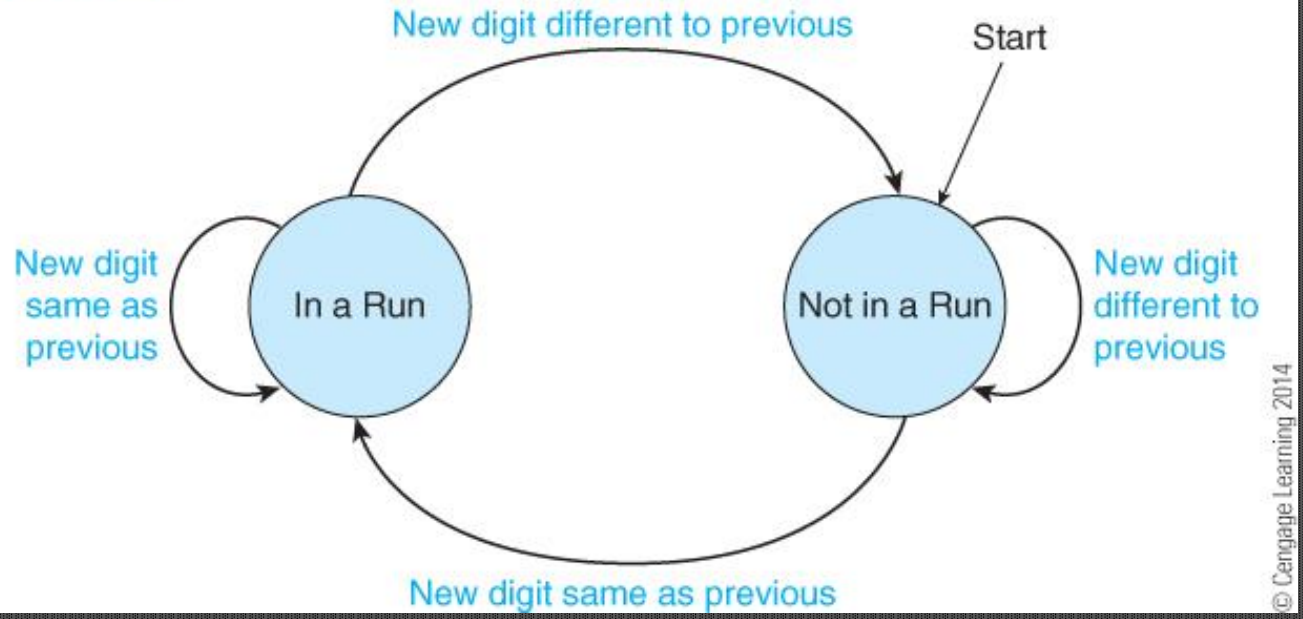
FIGURE 1.8 A state diagram for a run-length counter



State Diagram

1. Each circle represents a possible state
2. There are two states: **In a Run** and **Not in a Run**
3. A state change takes place each time we examine a new digit
4. A state transition can take you from the current state to a new state or keep you in the current state.

FIGURE 1.8 A state diagram for a run-length counter



1. This figure shows the state we are in after picking up each digit
2. We start at the left hand end

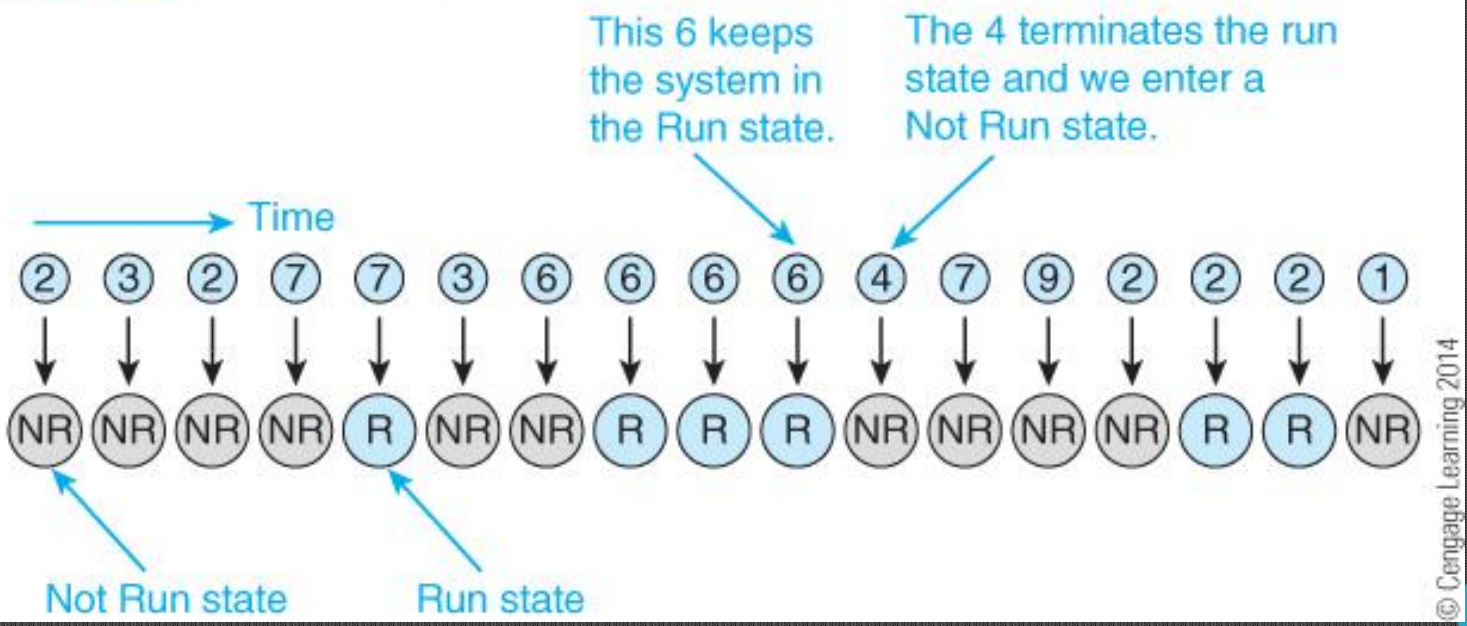


FIGURE 1.9 State changes when reading the string of Figure 1.7

1. Table 1.1 represents the problem in table form
2. The top line gives the position or location of each digit from 1 to 17
3. The second line gives the value of each element (i.e., the string itself)
4. The third line gives the current run value. This is the same as the previous digit.

TABLE 1.1 Turning the String into a Table of Values

Position in String	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Element Value	2	3	2	7	7	3	6	6	6	6	4	7	9	2	2	2	1
Current Run Value	?	2	3	2	7	7	3	6	6	6	6	4	7	9	2	2	2

1. Table 1.2 is an extension of table 1.1
2. We have added a new row at the bottom: the length of the current run

TABLE 1.2 The Current Run Length at Each Position Along the String of Digits

Position in String	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Element Value	2	3	2	7	7	3	6	6	6	6	4	7	9	2	2	2	1
Current Run Value	?	2	3	2	7	7	3	6	6	6	6	4	7	9	2	2	2
Current Run Length	1	1	1	1	2	1	1	2	3	4	1	1	1	1	2	3	1

1. Table 1.3 adds a new bottom line, the length of the longest run found so far
2. We can now look at how we would solve the problem mechanically.

TABLE 1.3 Expanding Table 1.2 to Include the Maximum Run Length

Position in String	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Element Value	2	3	2	7	7	3	6	6	6	6	4	7	9	2	2	2	1
Current Run Value	?	2	3	2	7	7	3	6	6	6	6	4	7	9	2	2	2
Current Run Length	1	1	1	1	2	1	1	2	3	4	1	1	1	1	2	3	1
Maximum Run Length	1	1	1	1	2	2	2	2	3	4	4	4	4	4	4	4	4

The Data

1. We now invent some names for the variables in Table 1.3
2. We can use these names in the algorithm we are to create.

i	The current position in the string
New_Digit	The value of the current digit just read from the string of digits
Current_Run_Value	The value of the elements in the current run
Current_Run_length	The length of the current run
Max_Run	The length of the longest run we've found so far

The Algorithm in Pseudocode

Table 1.3 adds a new bottom line, the length of the longest run found so far. We can now look at how we would solve the problem mechanically.

```
1.    Read the first digit in the string and call it New_Digit
2.    Set the Current_Run_Value to New_Digit
3.    Set the Current_Run_Length to 1
4.    Set the Max_Run to 1
5.    REPEAT
6.        Read the next digit in the sequence (i.e., read New_Digit)
7.        IF its value is the same as Current_Run_Value
8.            THEN Current_Run_Length = Current_Run_Length + 1
9.            ELSE {Current_Run_Length = 1
10.           Current_Run_Value = New_Digit}
11.       IF Current_Run_Length > Max_Run
12.           THEN Max_Run = Current_Run_Length
13.   UNTIL The last digit is read
```

The Naming of Parts

Constant – a value that doesn't change during the execution of a program; for example, if c = 2πr, then both '2' and 'π' are constants.

Variable – a value that can change during the execution of a program. In the previous example, both c and r are variables.

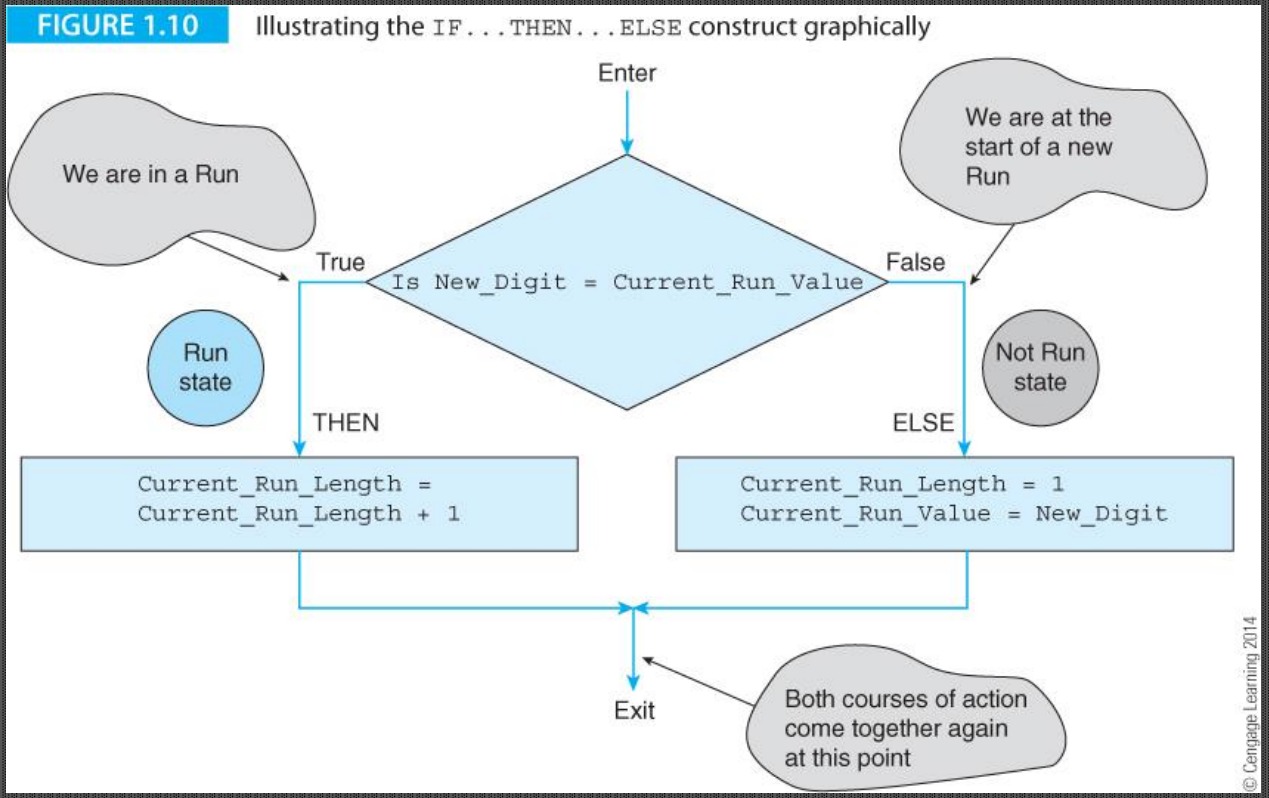
Symbolic name – we often refer to a variable or a constant by a name that makes it easier for us to remember. For example, we give the irrational number 3.1415926 the symbolic name π. When a program is compiled, symbolic names are replaced by actual values.

Address – information in a computer is stored in memory locations and each location has a unique address. Rather than trying to remember actual address locations in memory, we give addresses symbolic names; in this case the address may be called r.

Value and Location – When we write c = 2πr, what is r? We (humans) see r as the symbolic name for the value of the radius, say 5. But, the computer sees r as the symbolic address 1234 which has to be read to provide the value. If we write r = r + 1, do we mean r = 5 + 1 = 6 or do we mean r = 1234 + 1 = 1235? It is very important to distinguish between an address and its contents. This factor becomes significant when we introduce pointers.

Pointer – A pointer is a variable whose value is an address. If you modify a pointer, it points to a different value. In conventional arithmetic we write x_i where *i* is really a pointer; we just call it an index. If you change the pointer (index) we can step through the elements of a table, array or matrix and step x_1, x_2, x_3, x_4.

1. The pseudocode represents a series of actions such as assignments (setting a variable to another variable or to a constant), and conditional constructs.
2. The conditional construct allows us to select one of two courses of action depending on the outcome of a test; for example, IF X = 4 THEN Y = 6 ELSE Y = 7
3. Figure 1.10 illustrates the use of the construct where we test whether we are in a run or not and then either increment the run length or reset it to 1.



Program and Data

1. Figure 1.11 provides a table that includes the operations, the variables, and the string of digits to be tested.

2. This table can be modelled by a memory. The line number 0 to 37 corresponds to an address and the contents of that location represent programs or data.

3. Note that real computer instructions are not exactly like these. But they are very similar.

FIGURE 1.11 Memory map of a program and its data

0	`i = 21`
1	`New_Digit = Memory(i)`
2	`Set Current_Run_Value to New_Digit`
3	`Set the Current_Run_Length to 1`
4	`Set the Max_Run to 1`
5	`REPEAT`
6	`i = i + 1`
7	`  New_Digit = Memory(i)`
8	`  IF New_Digit = Current_Run_Value`
9	`    THEN Current_Run_Length = Current_Run_Length + 1`
10	`    JUMP to 13`
11	`    ELSE Current_Run_Length = 1;`
12	`         Current_Run_Value = New_Digit`
13	`  IF Current_Run_Length > Max_Run`
14	`    THEN Max_Run = Current_Run_Length`
15	`UNTIL i = 37`
16	`Stop`
17	**`New_Digit`**
18	**`Current_Run_Value`**
19	**`Current_Run_Length`**
20	**`Max_Run`**
21	**`2 (the first digit in the string)`**
22	**`3`**
23	**`2`**
23	**`7`**
…	…
37	**`1 (the last digit in the string)`**

Register Transfer Language (RTP) Notation

RTL is a *notation* used to define operations. Square brackets indicate the *contents* of a memory location. The expression `[15] = Max_Run` means "the contents of memory location 15 contains the value of `Max_Run`".

The backward arrow symbol, ←, indicates a *data transfer*.

For example, [15] ← [15] + 1 is interpreted as "the contents of memory location 15 are increased by 1 and the result put in memory location 15". Consider:

a. [20] = 5
b. [20] ← 6
c. [20] ← [6]

(a) states that the contents of memory location 20 are equal to the number 5.
(b) states that the number 6 is put into memory location 20.
(c) indicates that the contents of location 6 are copied into location 20.

The Stored Program Concept

The following pseudocode expresses the fundamental action of a stored program machine.

```
Stored_program_machine
    Point to the first instruction in memory
    REPEAT
        Read the instruction at the memory location pointed at
        Point to the next instruction
        Decode the instruction read from memory
        Execute the instruction
    FOREVER
End
```

This pseudocode sequence tells us that a *memory reference* (i.e., a memory read) is required to fetch each instruction from memory. We can expand the action Execute the instruction to give

```
Execute the instruction
    IF the instruction requires data
        THEN fetch the data from memory
    END_IF
    Perform the operation defined by the instruction
    IF the instruction requires data to be stored in memory
        THEN store the data in memory
    END_IF
End
```

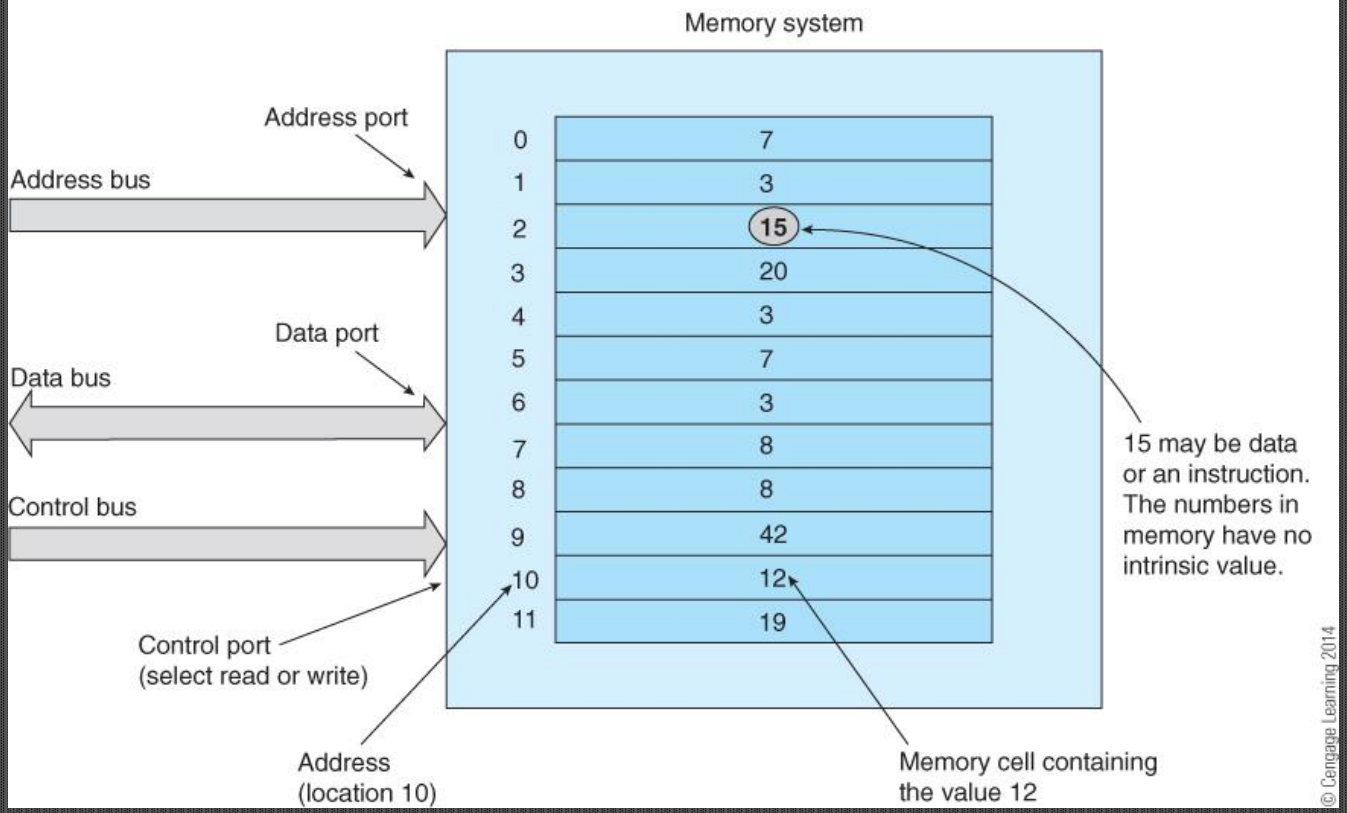
We can also express this sequence of actions in C as follows:

```
InstructionPointer = 0;
do
{ instruction = memory[InstructionPointer];        /* read the instruction
*/
   decode(instruction);                            /* decode the instruction  */
   fetch(operands);                                /* fetch data required     */
   execute;                                        /* execute the instruction */
   store(results);                                 /* store the result        */
} while (instruction != stop);
```

1. A key component of a computer is the memory that holds the program (instructions) and data.

2. Figure 1.12 illustrates the elements of a computer's immediate access store.

FIGURE 1.12 The memory system



Address Formats

Consider the three-address format:
Operation **Address1**,Address2,Address3 where Operation specifies the action the instruction, and Address1, Address2, and Address3, are locations of the three operands in memory.

The operands are the *addresses* of data and not the data itself.

We use bold font to indicate the address that is the *destination* of data.

ADD **P**,Q,R, is three-operand instruction where P, Q, and R are the symbolic names of the addresses of three memory locations.

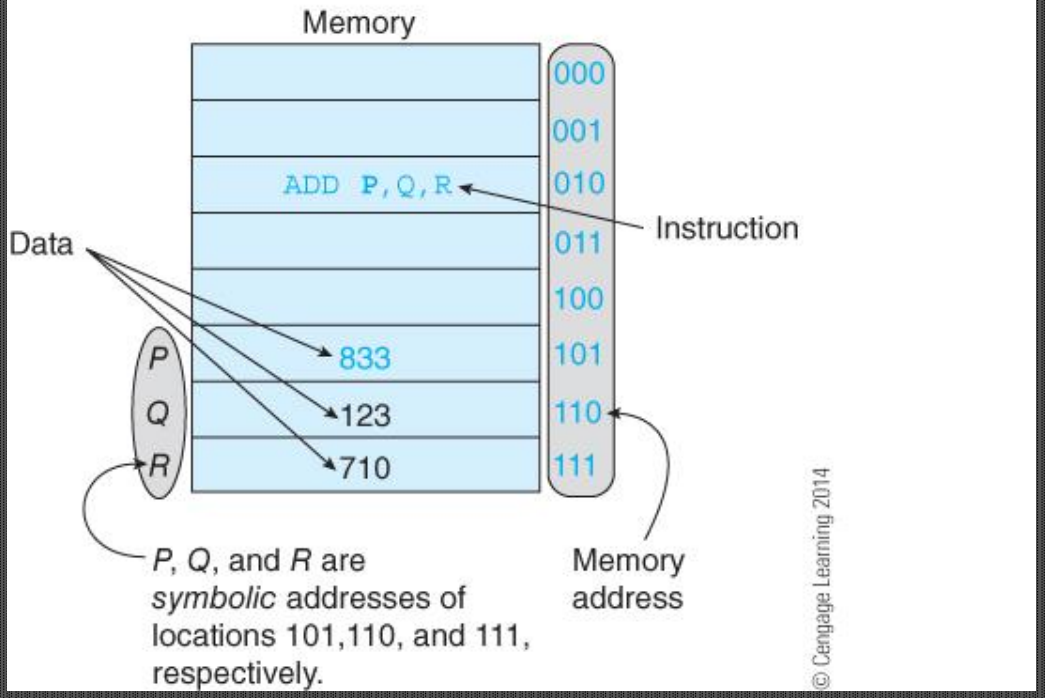
The three-operand format can be expressed in RTL notation as:

[Address1] ← [Address2] Operation [Address3]

The contents of the memory locations specified by Address2 and Address3 are operated on by the operation (e.g., add, subtract), and the result placed in the memory location specified by Address1.

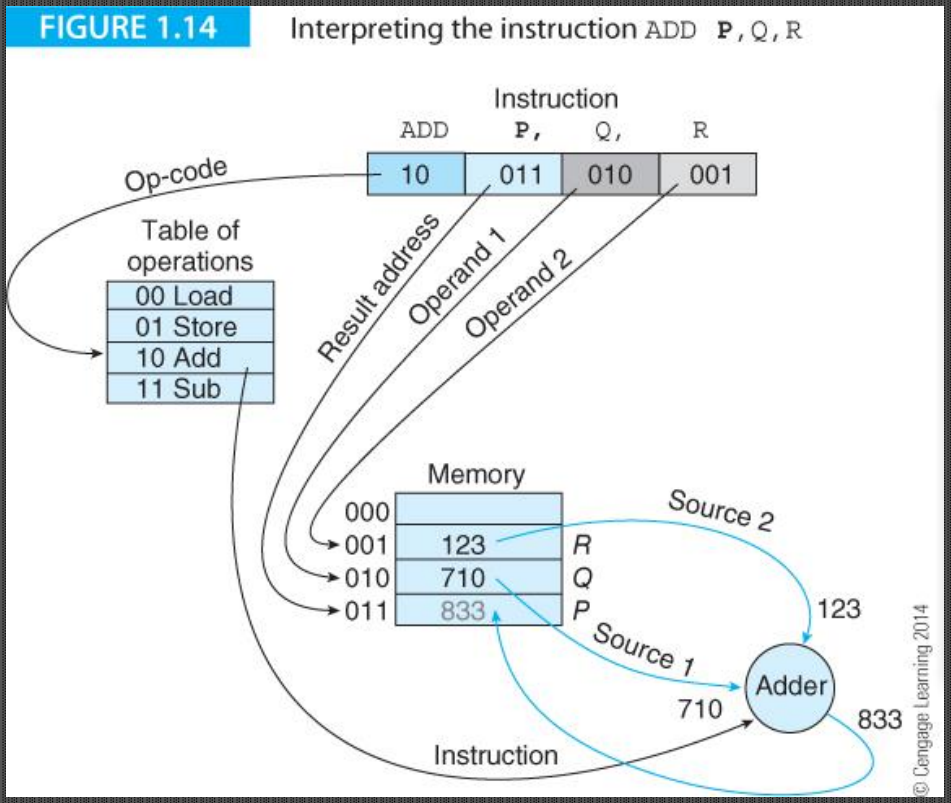
1. Figure 1.13 illustrates the features of a memory that we will need in the rest of the text.
2. Although memory addresses are numeric (in this case we use binary numbers 000 to 111), we normally use symbolic names because they are easier for us to remember. If you write P in a program, that is automatically translated to address 101.

FIGURE 1.13 Relationship between instruction and operands



1. Figure 1.14 Applies to a hypothetical computer that has an instruction with three addresses; for example, ADD P,Q,R which implements P = Q + R. Here P, Q, and R are the symbolic names of there locations in memory.

2. The purpose of this figure is to show the flow of information when an instruction is executed and to demonstrate the possible structure of an instruction.



Two Address Instructions

Some computers implement a two-address instruction format of the form

Operation **Address1**,Address2

where Address2 is a source operand and Address1 is *both* a source and a destination operand. This operand is accessed, operated on, and the result placed in the same location. The definition of ADD **P**,Q, is [P] ← [P] + [Q]

A two-address instruction *destroys* one of the operands; that is, source operand P is replaced (overwritten) by the result.

Practical computers do not generally allow you to use two memory addresses in the same instruction. Computers like the Core i7 processors specify one address in memory and a second address which is a register.

A register is a single storage element in the computer with a name like r0, r1, r2 … or r31 and is used to hold temporary data during calculations.

One Address Instructions

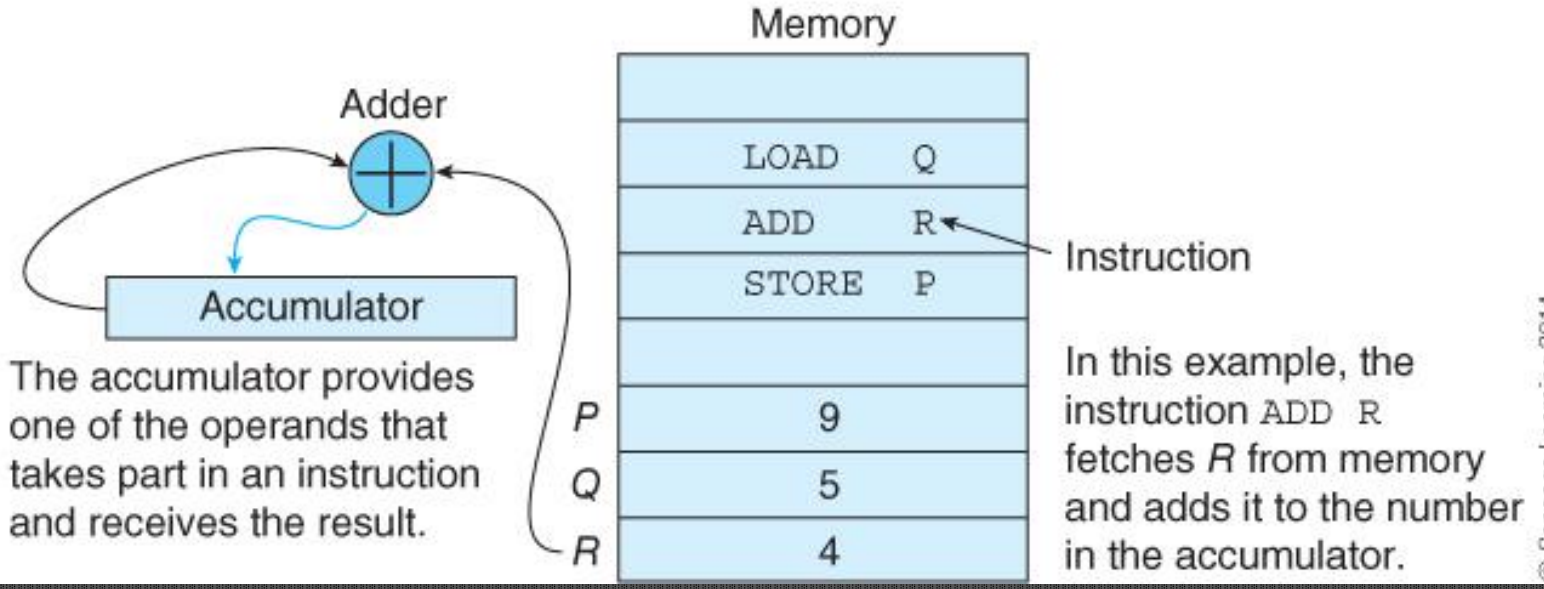
Typically, 8-bit first-generation microprocessors implemented a single address instruction of the form **Operation address**

The processor has to use a second operand that doesn't require an explicit address; that is, the second operand comes from a *register* once called the *accumulator* within the CPU.

A *register* behaves like a memory location except that it is located within the CPU. The term *accumulator* is little used today because most microprocessors now have several on-chip registers. Figure 1.16 demonstrates the flow of information during the execution of a single-operand instruction. The result remains in a register until another instruction transfers it to memory. Such a computer is hardly elegant as the following sequence that implements P = Q + R demonstrates

```
LOAD  Q      ; Read Q  into the accumulator
ADD   R      ; Add R  to the accumulator
STORE P      ; Store the accumulator in P
```

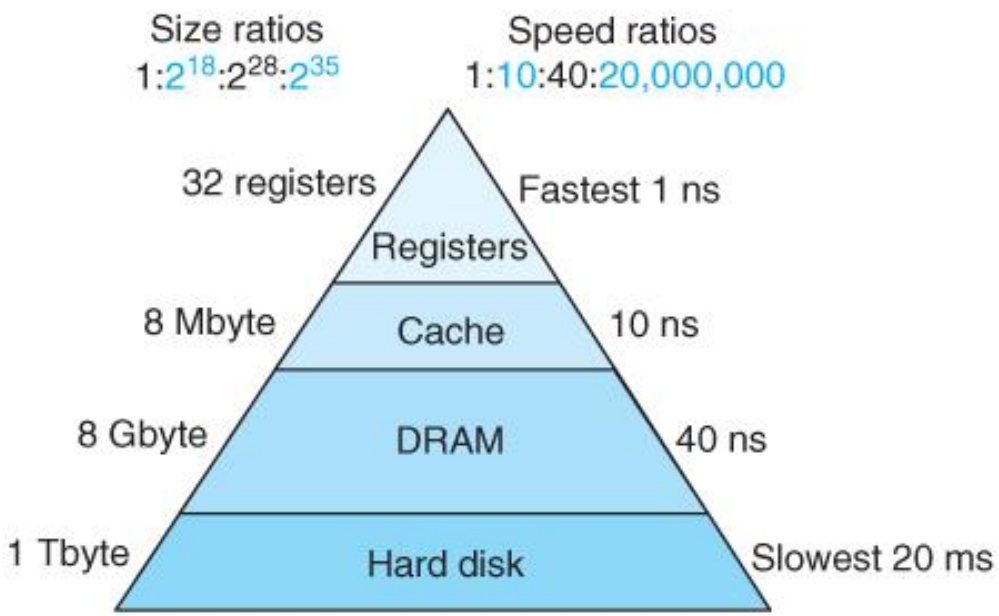
FIGURE 1.15 Single-operand instructions



Memory Hierarchy

1. An important characteristic of modern computers is the wide range of technologies used to implement computers.
2. Figure 1.16 illustrates memory hierarchy that covers the memory system of a typical computer.
3. At the top are small amounts of on-chip register memory. At the bottom are the large quantities of storage provides by hard disks.

FIGURE 1.16 Memory hierarchy



The Bus

The **bus** links together two or more functional parts of a computer and allows the exchange of data; for example, the bus between the CPU and its graphics card.

Buses also link computers to external peripherals; for example, the USB bus that connects a printer to a computer.

Figure 1.17 illustrates the structure of a hypothetical system without a bus. Imagine that the blue circles are processing units that have to communicate with each other.

In this example some units communicate directly with only one other unit, whereas other units have to communicate with several devices.

FIGURE 1.17 An arbitrary interconnect structure—life without the bus

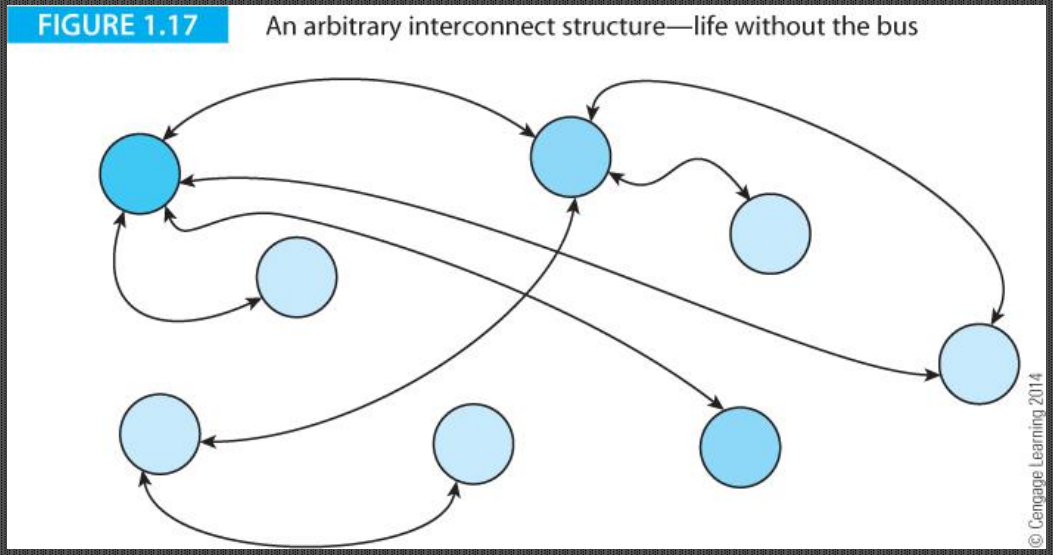
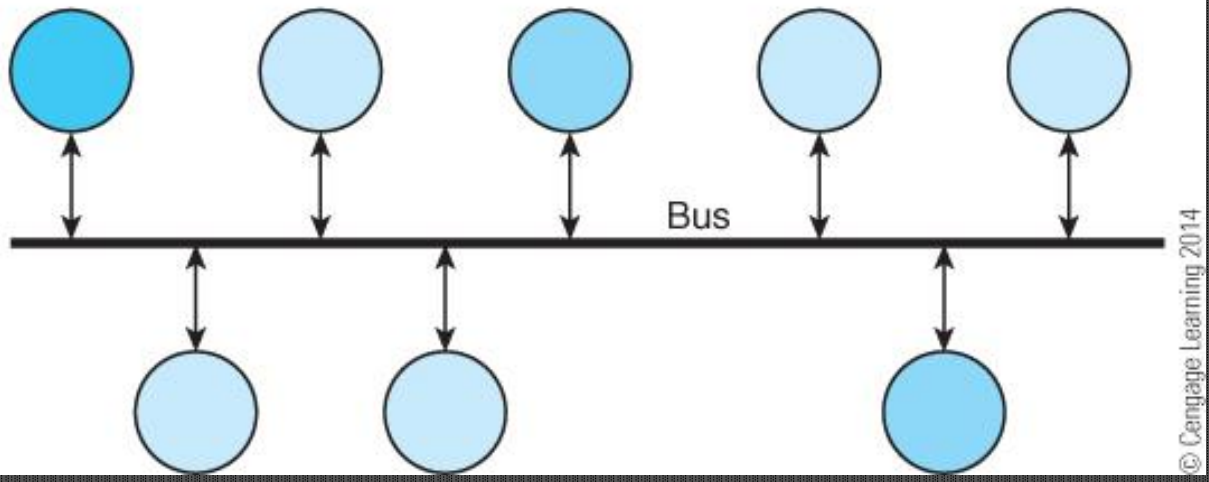


Figure 1.18 illustrates the structure of a system with a bus.

Functional units may request the bus, use it to communicate with other units and then relinquish the bus.

Internal buses (within the CPU or on the motherboard) and external buses (USB, FireWire) are vital components of the computer system and contribute to its overall performance.

FIGURE 1.18 A common bus connecting all units

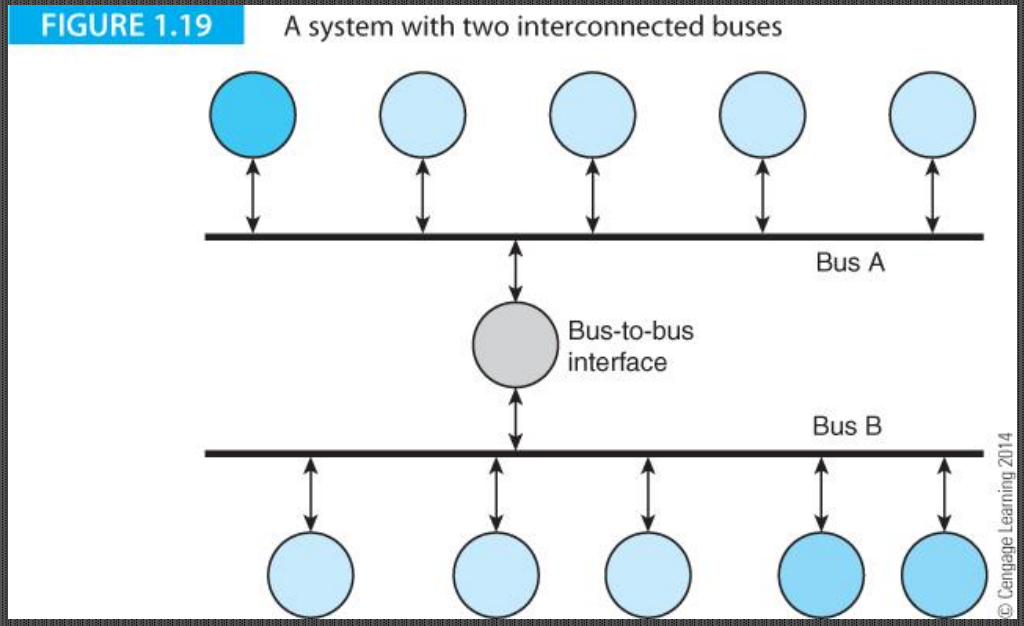


The Bus

Figure 1.19 illustrates the structure of a system with two buses. Such a system might be found in a PC.

Multiple buses permit parallel operation because transactions on each bus can take place simultaneously.

Each bus may be optimized for its specific application (e.g., a high speed bus for graphics and a lower speed bus for peripherals).



Bus Terminology

Width The width of a bus is defined as the number of parallel data paths. A 64-bit bus can carry 64-bits (8 bytes) of information at a time. However, the same term can also be used to indicate the total number of wires (connections) that make up a bus. For example, a bus may have 50 information paths of which 32 of them carry data (the rest may be paths for control signals or even power lines).

Bandwidth The bandwidth of a bus is a measure of the rate at which information can be transported across the bus. The bandwidth is expressed in either bytes per second or bits per second. Increasing the width of a bus while keeping the data rate constant increases the bandwidth.

Latency Latency is the waiting period between a data transfer request and the actual data transmission. Typically, a bus's latency includes the time taken to arbitrate for the bus before transmission can take place.