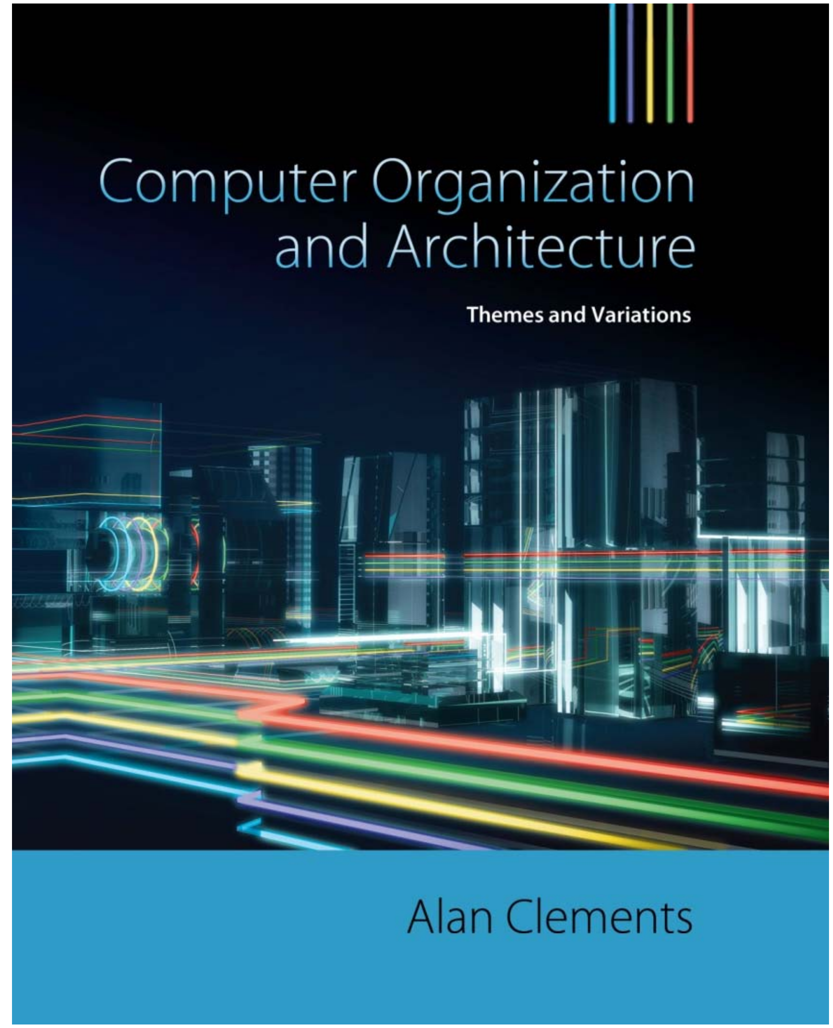
Part 2

CHAPTER 1

Computer Systems Architecture

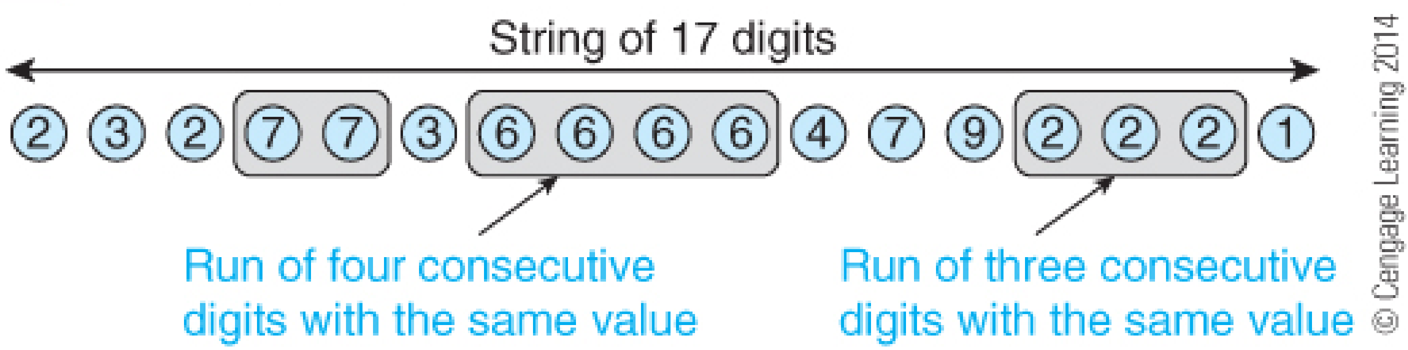


CENGAGE Learning

Introducing the Computer by Solving a Problem

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

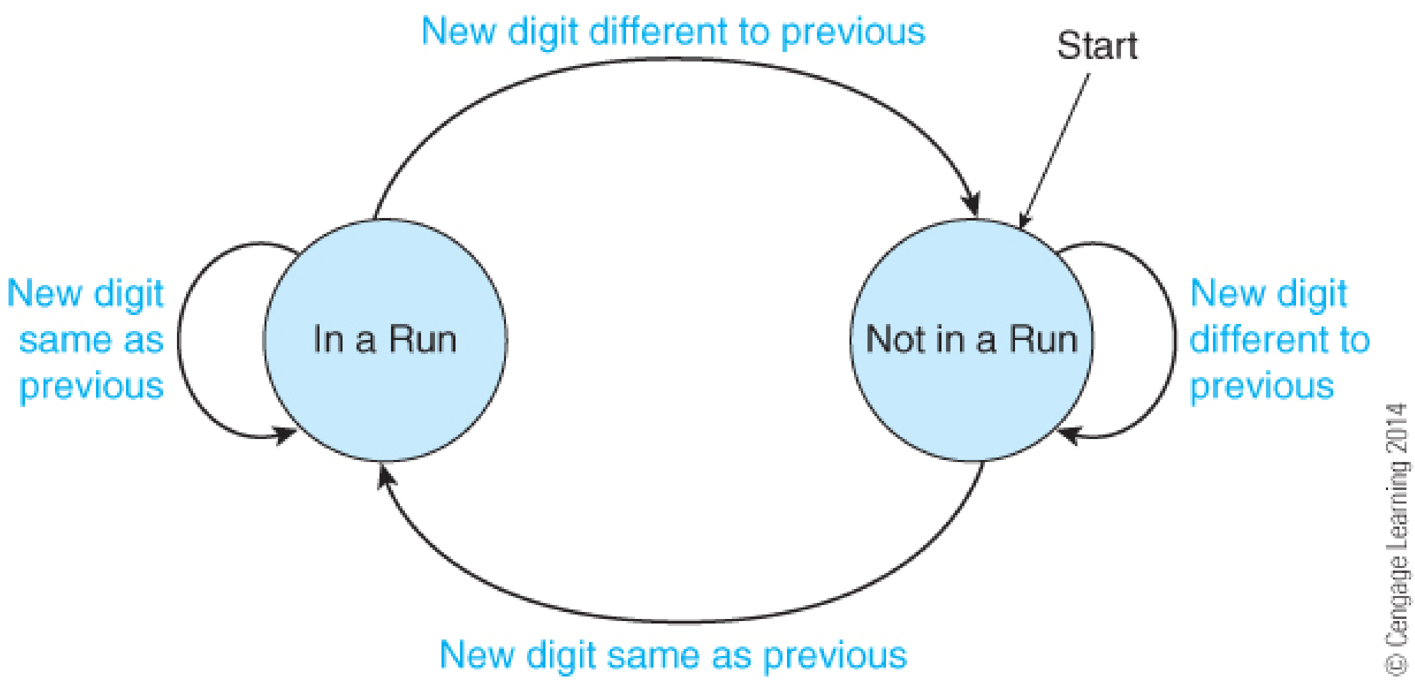
FIGURE 1.7 A string of digits



Introducing the Computer by Solving a Problem

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

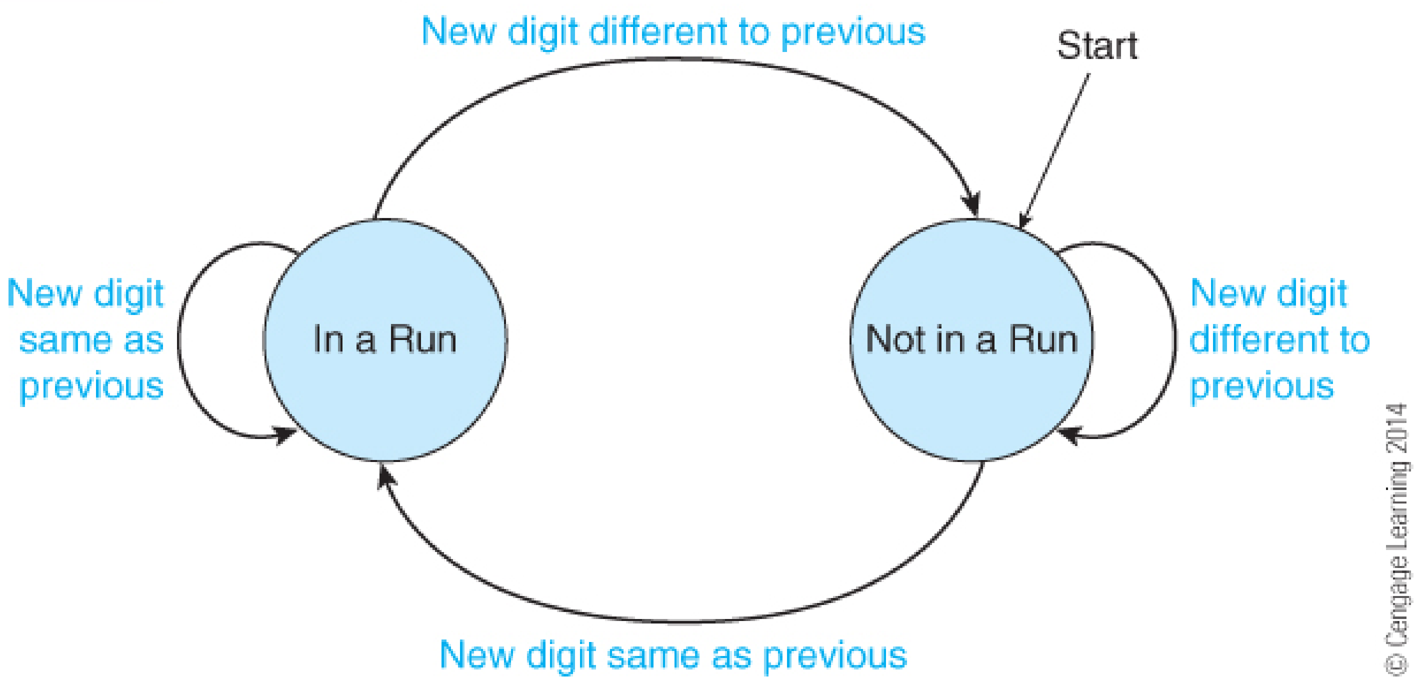
FIGURE 1.8 A state diagram for a run-length counter



State Diagram

- Each circle represents a possible state
- There are two states: **In a Run** and **Not in a Run**
- A state change takes place each time we examine a new digit
- 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



State Diagram

- Figure 1.9 shows the state we are in after picking up each digit
- We start at the left hand end

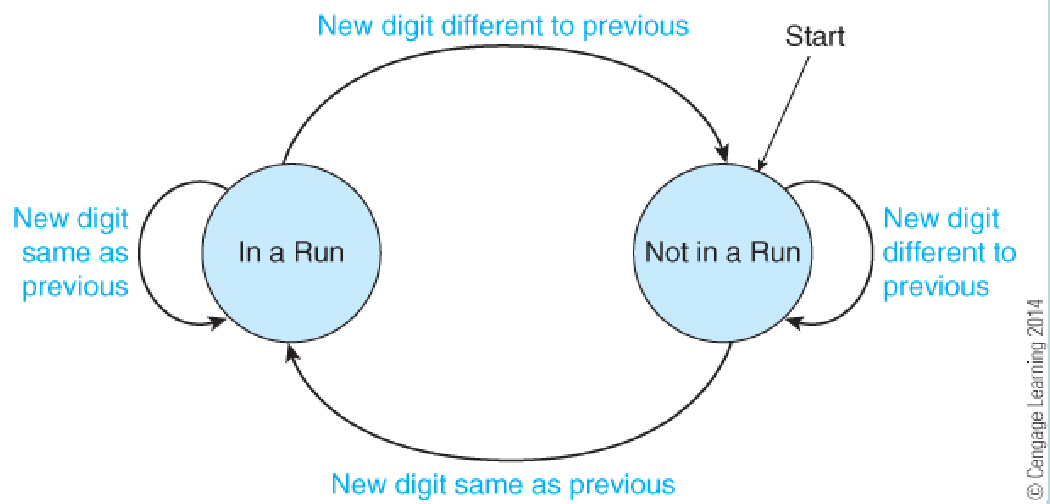
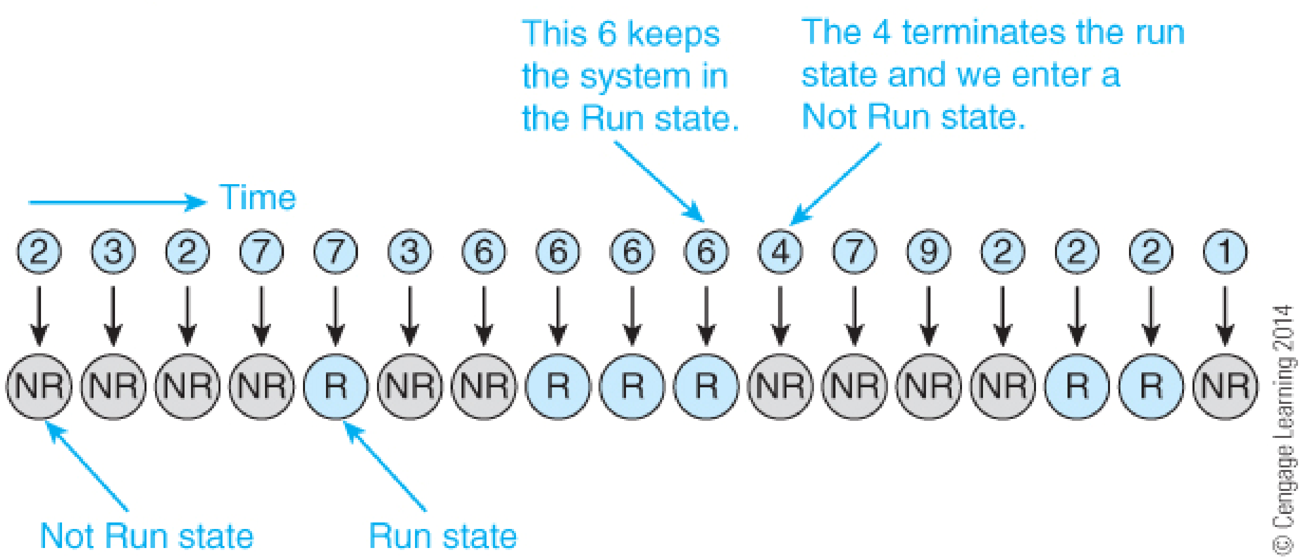


FIGURE 1.8 A state diagram for a run-length counter

FIGURE 1.9 State changes when reading the string of Figure 1.7



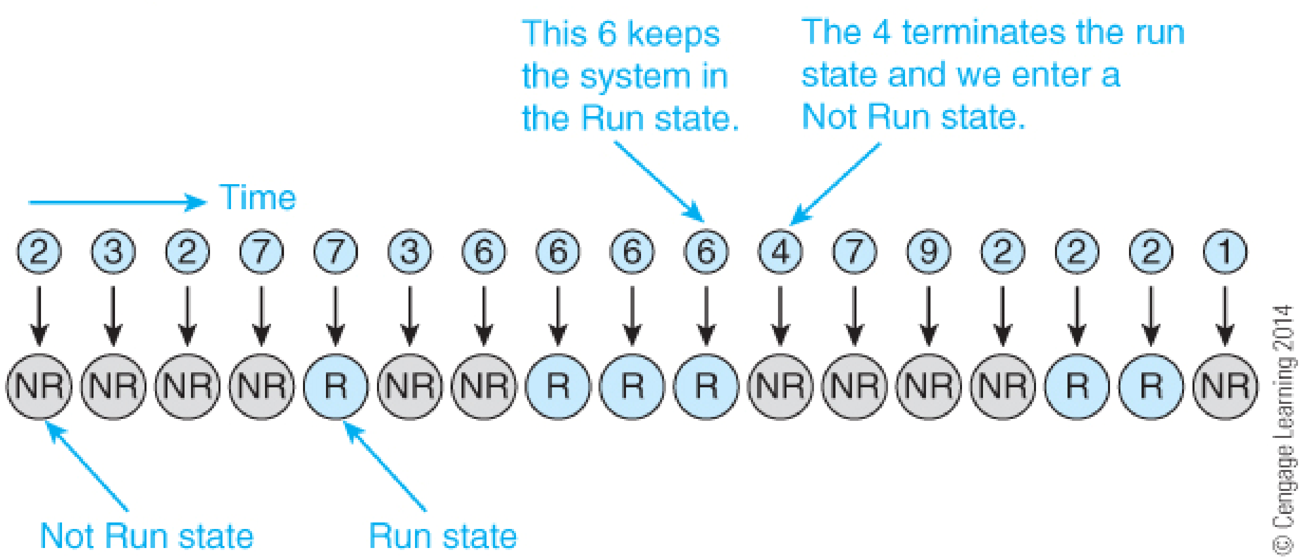
State Diagram

- Table 1.1 represents the problem in a table form
- The top line gives the position or location of each digit from 1 to 17
- The second line gives the value of each element (i.e., the string itself)
- 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

FIGURE 1.9 State changes when reading the string of Figure 1.7



State Diagram

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

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

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

State Diagram

- 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.

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

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

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

- We now invent some names for the variables in Table 1.3

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

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 Algorithm in Pseudo-code

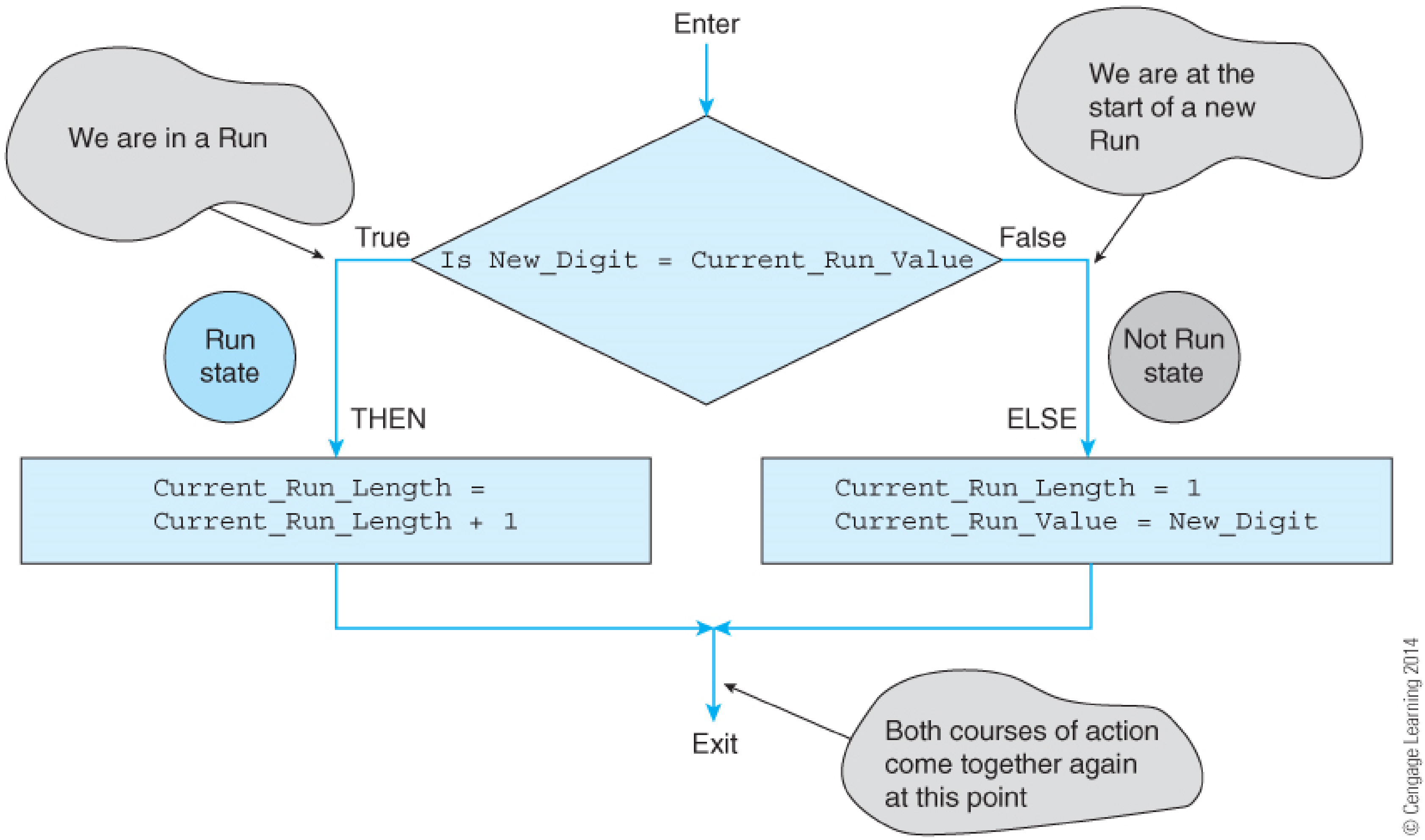
- We can now look at how we would solve the problem.

```
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 a 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 Algorithm in Pseudo-code

- Figure 1.10 illustrates the use of the IF…THEN…ELSE construct where we test whether we are in a run or not to either increment the run length or reset it to 1.

FIGURE 1.10 Illustrating the IF...THEN...ELSE construct graphically



Program and Data

- Figure 1.11 provides a table that includes
  - the operations,
  - the variables, and
  - the string of digits to be tested.
- This table can be modelled as a memory array.
  - The line-number 0 to 37 corresponds to an address
  - The contents of each location represent either
    - A program instruction or
    - data.
- Note that real computer instructions are not exactly like these.
  But they are very similar.
- ***From the bit pattern point of view, there is no way to differentiate between encoded data and encoded program instruction.***

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)

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. *Think of computer memory as if it is an array and the index of this array is the address of the memory locations.*

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 = 2π**r**, 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 the value of 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, i.e., x_1, x_2, x_3, x_4.

Register Transfer Language (RTL) Notation

- In RTL *notation*, **square brackets** indicate the ***contents* of a memory location**.
  - For example, The expression `[15] = Max_Run` means ***"the content of memory location 15 is equal to the value of*** `Max_Run` ***"***, *I.e., it is a sort of initialization not an assignment.*
- The **backward arrow symbol**, ←, indicates a ***data transfer***.
  - For example, **[15] ← [15] + 1** means **"the content of memory location 15 is increased by 1"** and **"the result is put in memory location 15".**
- Consider:

  a. [20] = 6
  b. [20] ← 6
  c. [20] ← [6]

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

The Stored Program Concept

- The following pseudo-code expresses the fundamental action of a stored program machine.

```
Stored_program_machine
    Point to the first instruction in memory
    REPEAT
        1. Read the instruction at the memory location pointed at
        2. Point to the next instruction
        3. Decode the instruction just read from memory
        4. Execute the instruction
    FOREVER
End
```

- This pseudo-code sequence tells us that a *memory reference* (i.e., a memory read) is required to fetch each instruction from memory.

The Stored Program Concept

- 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

- As you see, **Execute the instruction** may require a ***memory read*** ***and/or*** a ***memory write***

The Stored Program Concept

- We can also express this sequence of actions in **C** or **Java** as follows:

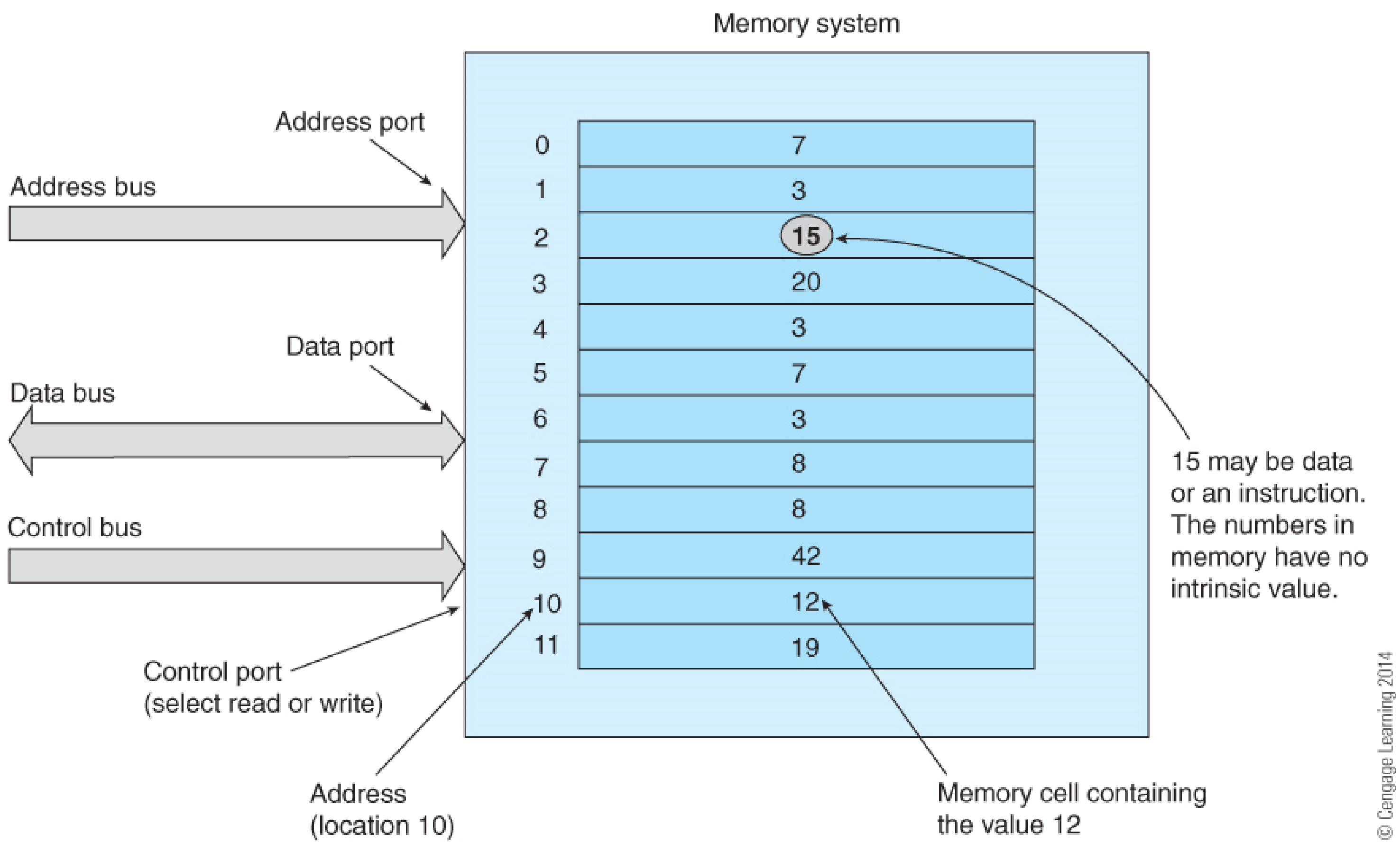
In the book, the post increment is missing

```
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);
```

The Stored Program Concept

- A key component of a computer is the ***memory*** that holds ***the program (instructions)*** and ***data***.
- Figure 1.12 illustrates the elements of a computer's *memory system*.

FIGURE 1.12 The memory system



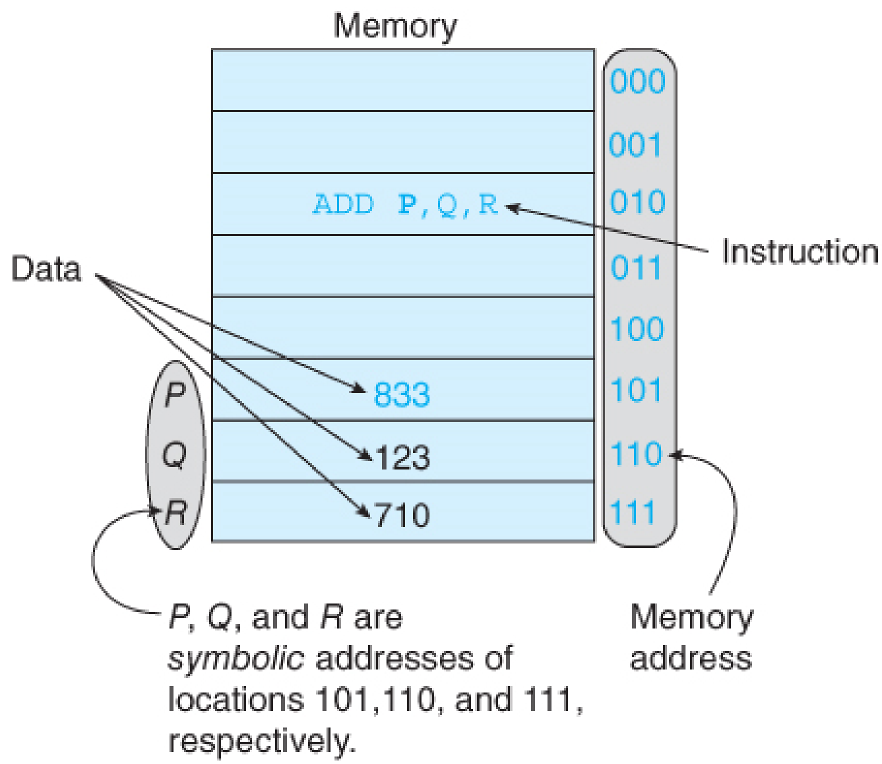
Three Address Instructions

- Consider the **three-address** format:
  - ***Operation*** ***Address1,*** *Address2, Address3*
    where ***Operation*** specifies the action of the instruction, whereas ***Address1***, *Address2*, and *Address3*, are locations of the three operands in memory.
  - We use ***bold*** font to indicate the address that is the destination of data.
- In this example, the operands are the addresses of data and not the data itself.
- ***ADD*** ***P,*** *Q, R*, is a three-operand instruction
  - ***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]*** $\leftarrow$ *[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 is placed in the memory location specified by ***Address1***.

Three Address Instructions

- 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, it is automatically translated to address ***101***.

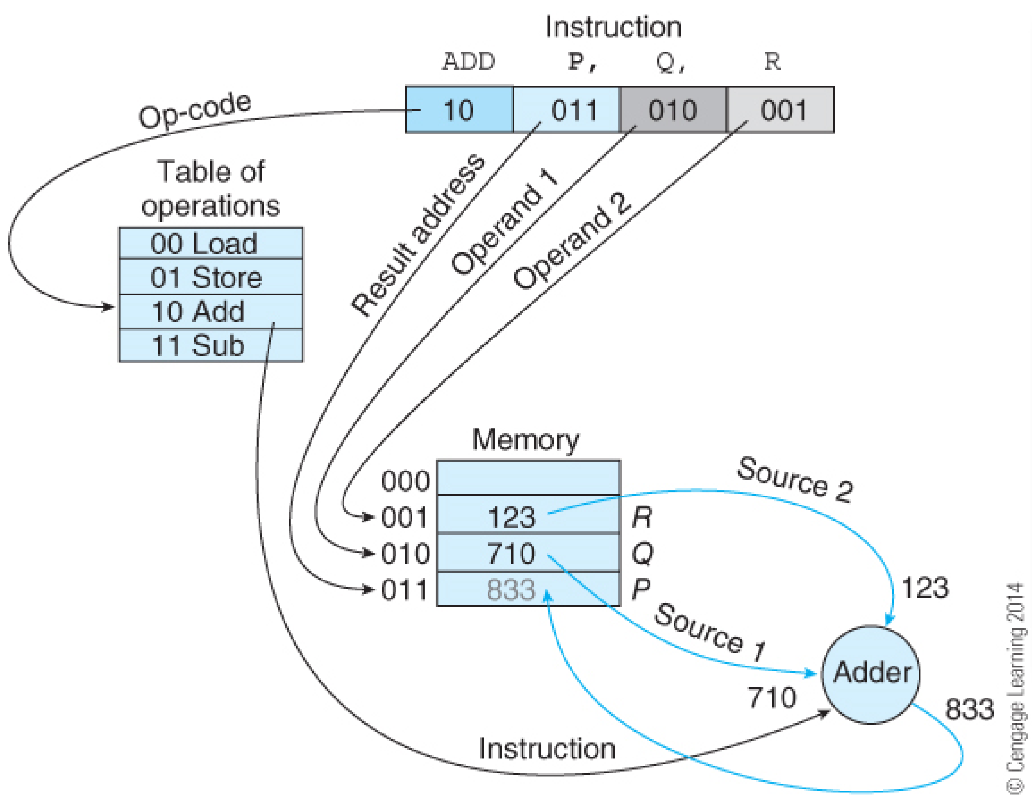
FIGURE 1.13 Relationship between instruction and operands



Three Address Instructions

- Figure 1.14 shows 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 their locations in memory.
- 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.

FIGURE 1.14 Interpreting the instruction `ADD P,Q,R`



Two Address Instructions

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

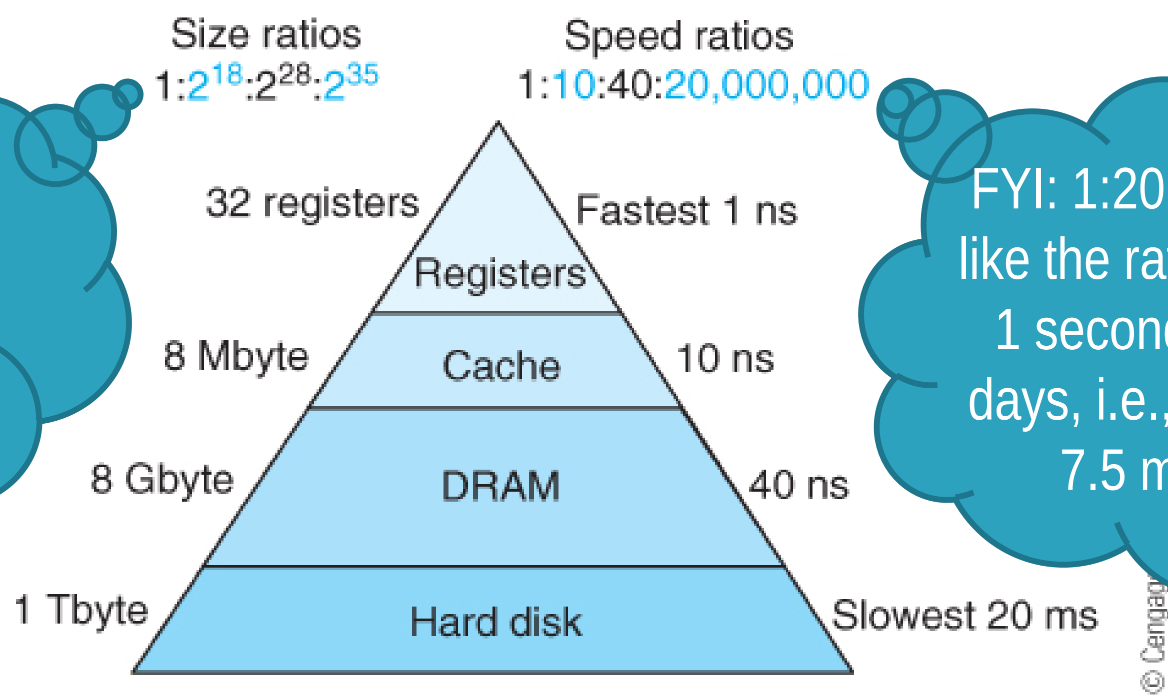
  ***Operation Address1**, Address2*

  - 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 three or two **main** memory addresses in the same instruction*.
  - Computers like the Core ***i7*** processors specify *one address in memory* and *a second address 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.
- A *register* behaves like *a memory location* except that it is located within the CPU.

Memory Hierarchy

- An important characteristic of modern computers is the wide range of technologies used to implement computers.
- Figure 1.16 illustrates memory hierarchy that covers the memory system of a typical computer.
- 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



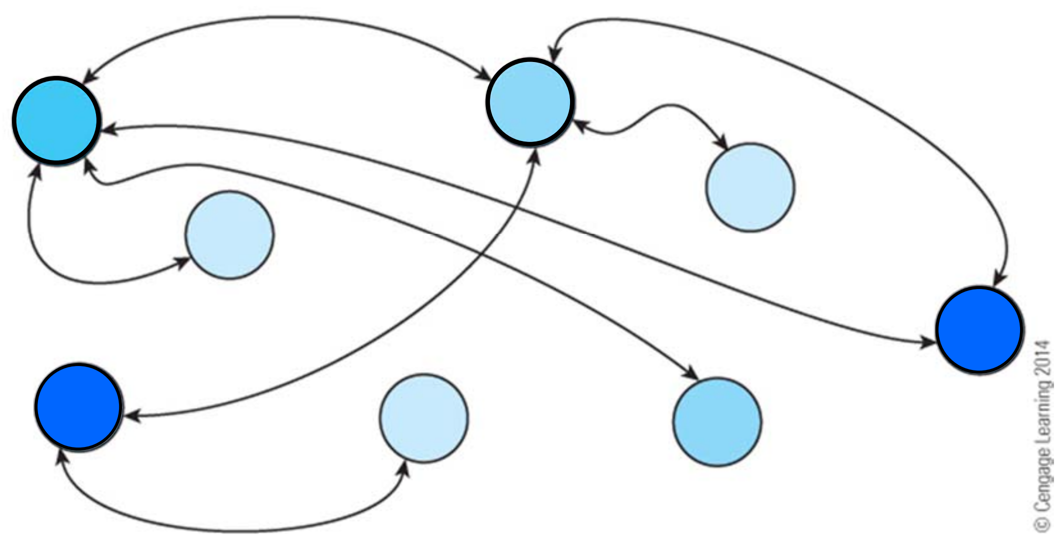
This ratio is based on number of units, not based on number of bytes.

FYI: 1:20,000,000 is like the ratio between 1 second to 231.5 days, i.e., more than 7.5 months.

The Bus

- Buses **link** together two or more **functional parts** of a computer and allow 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.

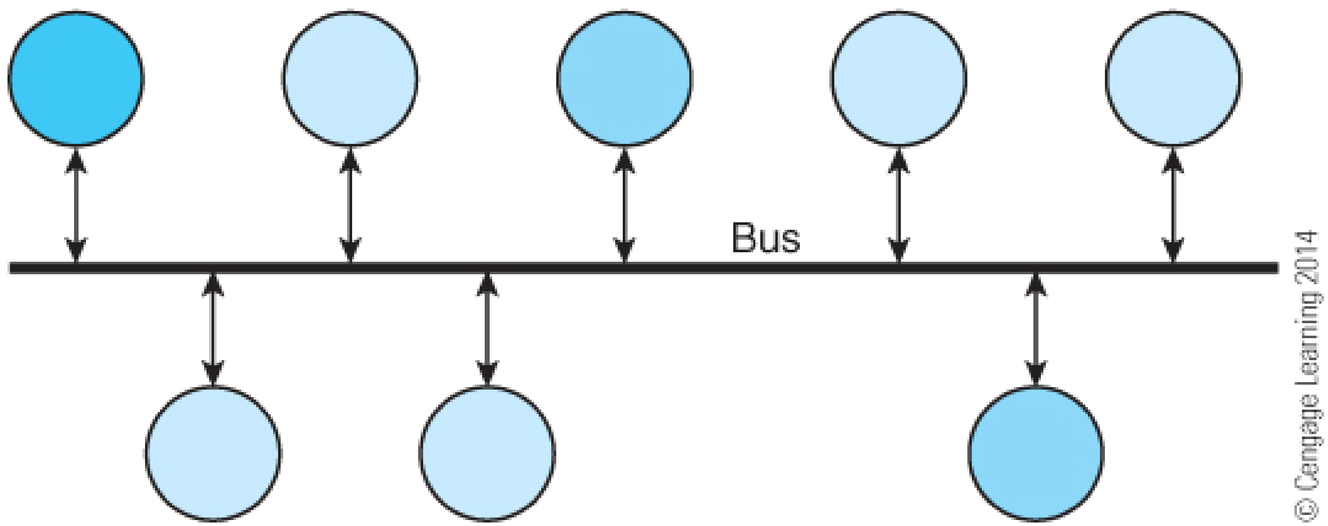
FIGURE 1.17 An arbitrary interconnect structure—life without the bus



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.

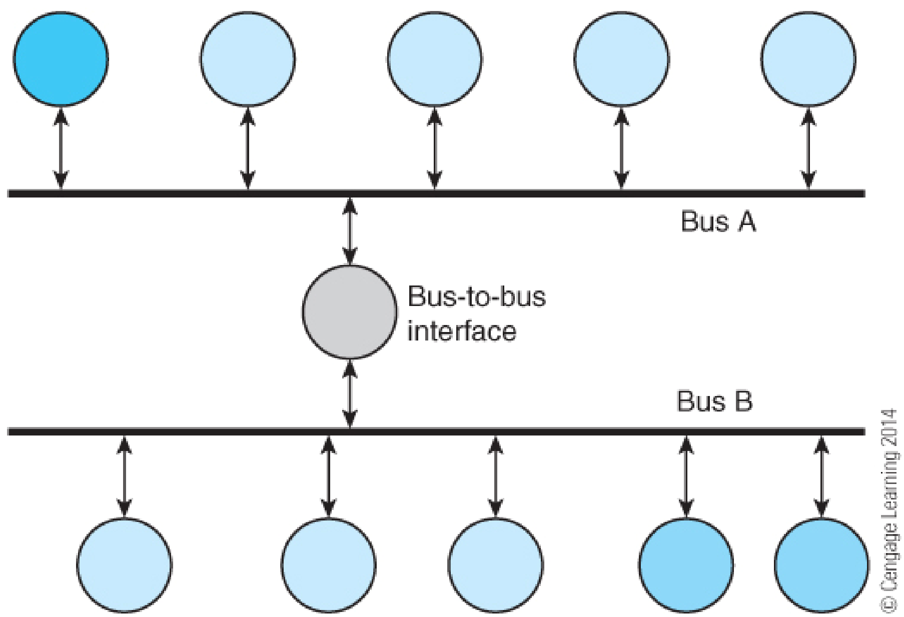
FIGURE 1.18 A common bus connecting all units



The Bus

- Figure 1.19 illustrates the structure of *a system with two buses*.
- 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).

FIGURE 1.19 A system with two interconnected buses



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 ***data*** 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 per wire constant *increases the bandwidth*.

Latency Latency is the *waiting period between a data transfer request and the actual data transmission completed*.

Typically, a bus's latency includes the time taken to arbitrate for the bus before transmission can take place.