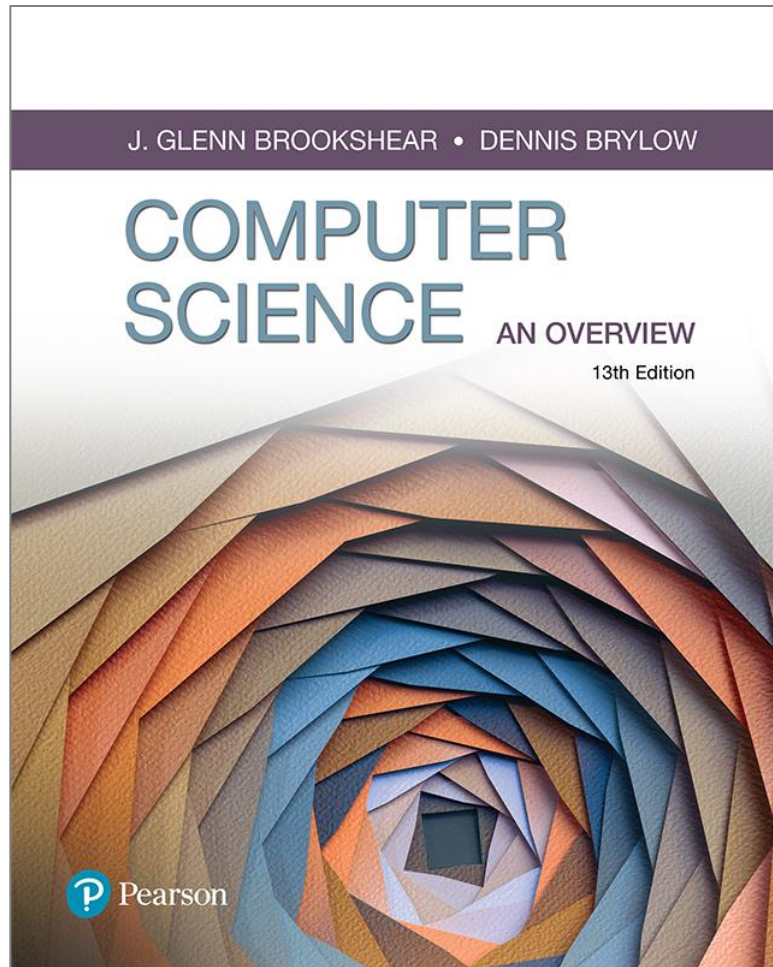


Computer Science An Overview

13th Edition



Chapter 2

Data Manipulation

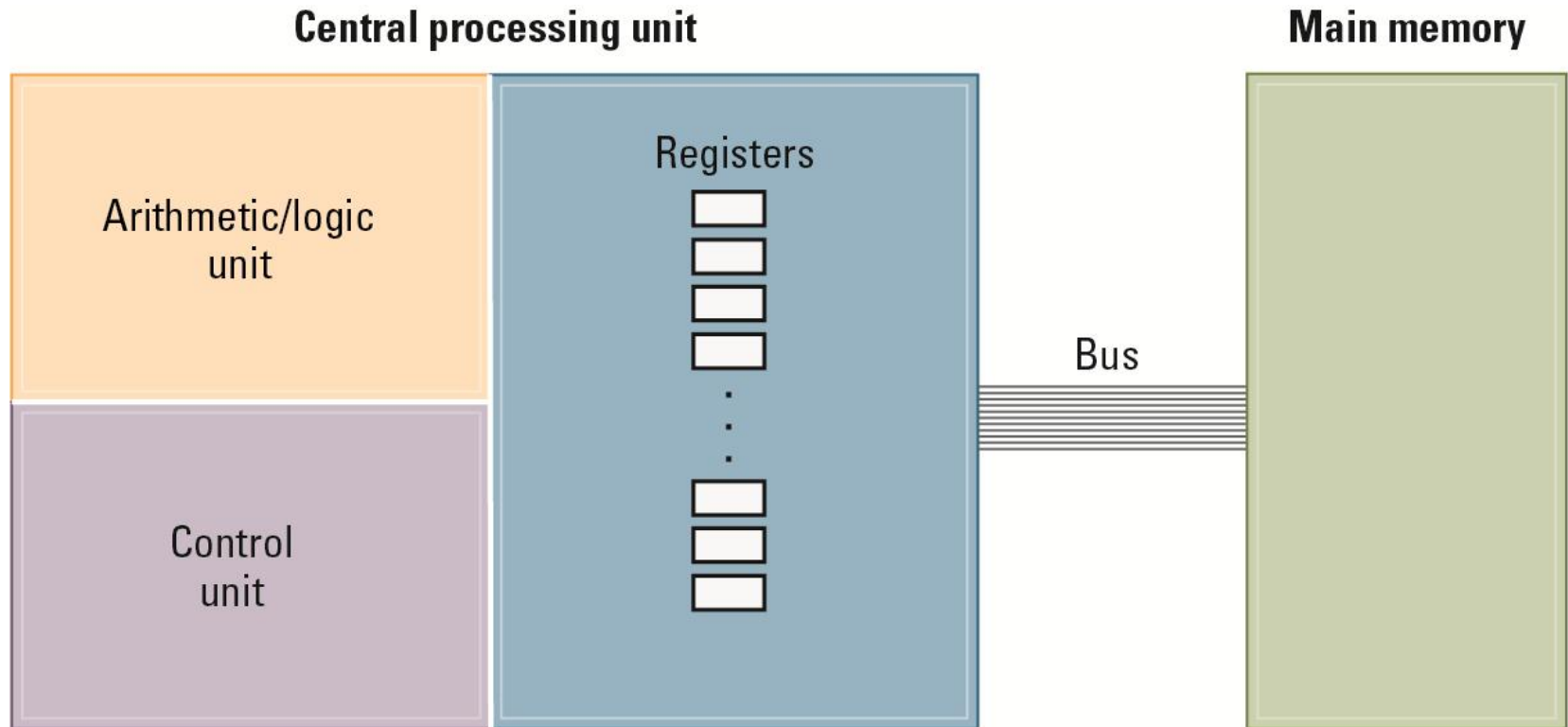
Chapter 2: Data Manipulation

- 2.1 Computer Architecture
- 2.2 Machine Language
- 2.3 Program Execution
- 2.4 Arithmetic/Logic
- 2.5 Communicating with Other Devices
- 2.6 Program Data Manipulation
- 2.7 Other Architectures

2.1 Computer Architecture

- Central Processing Unit (CPU)
 - Arithmetic/Logic Unit
 - Control Unit
 - Register Unit
 - General purpose registers
 - Special purpose registers
- Bus
- Main Memory

Figure 2.1 CPU and main memory connected via a bus



Stored Program Concept

A program can be encoded as bit patterns and stored in Main Memory. From there, the Control Unit can extract, decode, and execute instructions.

Instead of rewiring the CPU, the program can be altered by changing the contents of Main Memory.

2.2 Machine Language

- **Machine instruction:** An instruction encoded as a bit pattern recognizable by the CPU
- **Machine language:** The set of all instructions recognized by a machine

Machine Language Philosophies

- Reduced Instruction Set Computing (RISC)
 - Few, simple, efficient, and fast instructions
 - Examples: PowerPC from Apple/IBM/Motorola and ARM
- Complex Instruction Set Computing (CISC)
 - Many, convenient, and powerful instructions
 - Example: Intel

Machine Instruction Types

- Data Transfer: copy data from one location to another (e.g. LOAD, STORE)
- Arithmetic/Logic: operations on bit patterns (e.g. +, -, *, /, AND, OR, SHIFT, ROTATE)
- Control: direct the execution of the program (e.g. JUMP, BRANCH)

Figure 2.2 Adding values stored in memory

Step 1. Get one of the values to be added from memory and place it in a register.

Step 2. Get the other value to be added from memory and place it in another register.

Step 3. Activate the addition circuitry with the registers used in Steps 1 and 2 as inputs and another register designated to hold the result.

Step 4. Store the result in memory.

Step 5. Stop.

Figure 2.3 Dividing values stored in memory

Step 1. LOAD a register with a value from memory.

Step 2. LOAD another register with another value from memory.

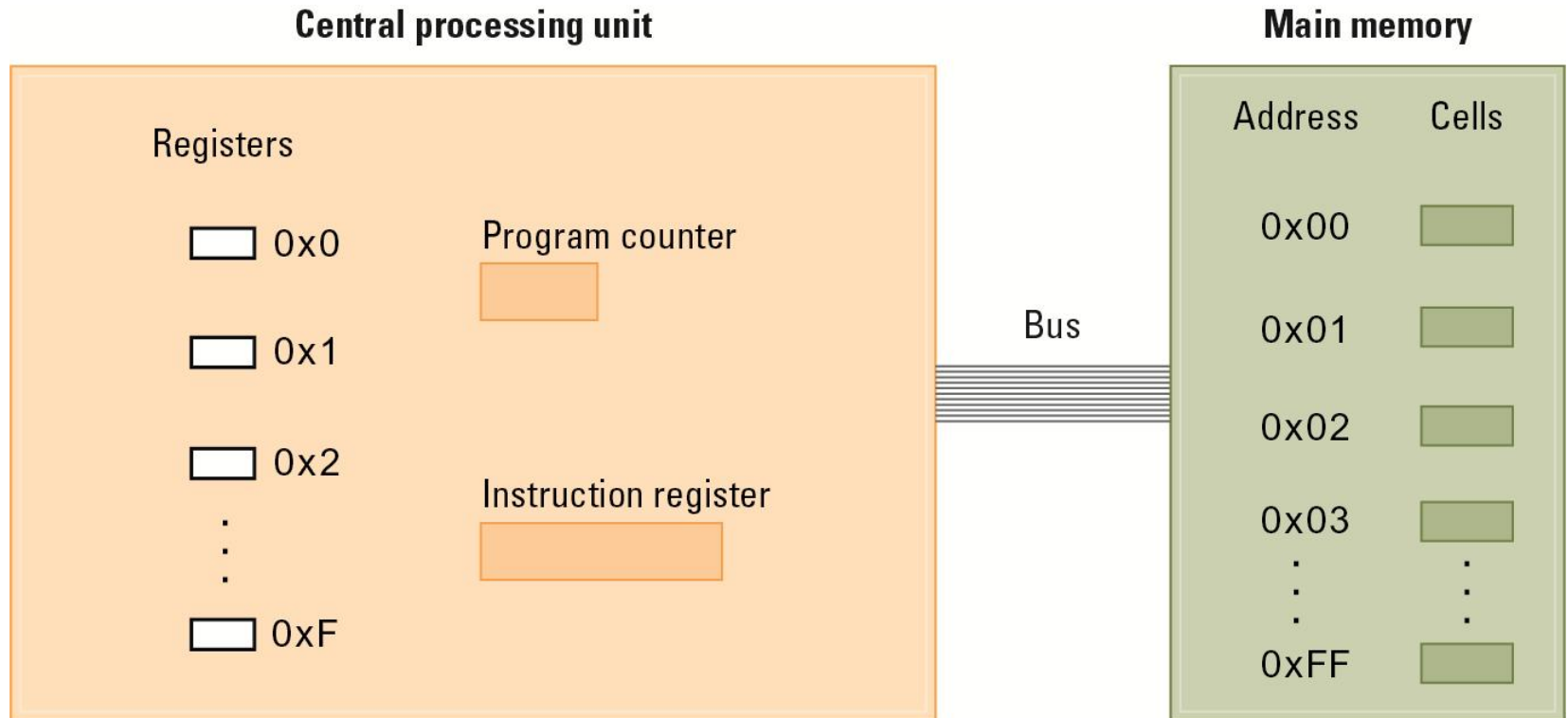
Step 3. If this second value is zero, JUMP to Step 6.

Step 4. Divide the contents of the first register by the second register and leave the result in a third register.

Step 5. STORE the contents of the third register in memory.

Step 6. STOP.

Figure 2.4 The architecture of the Vole, as described in Appendix C



Parts of a Machine Instruction

- **Op-code:** Specifies which operation to execute
- **Operand:** Gives more detailed information about the operation
 - Interpretation of operand varies depending on op-code

Figure 2.5 The composition of a Vole instruction

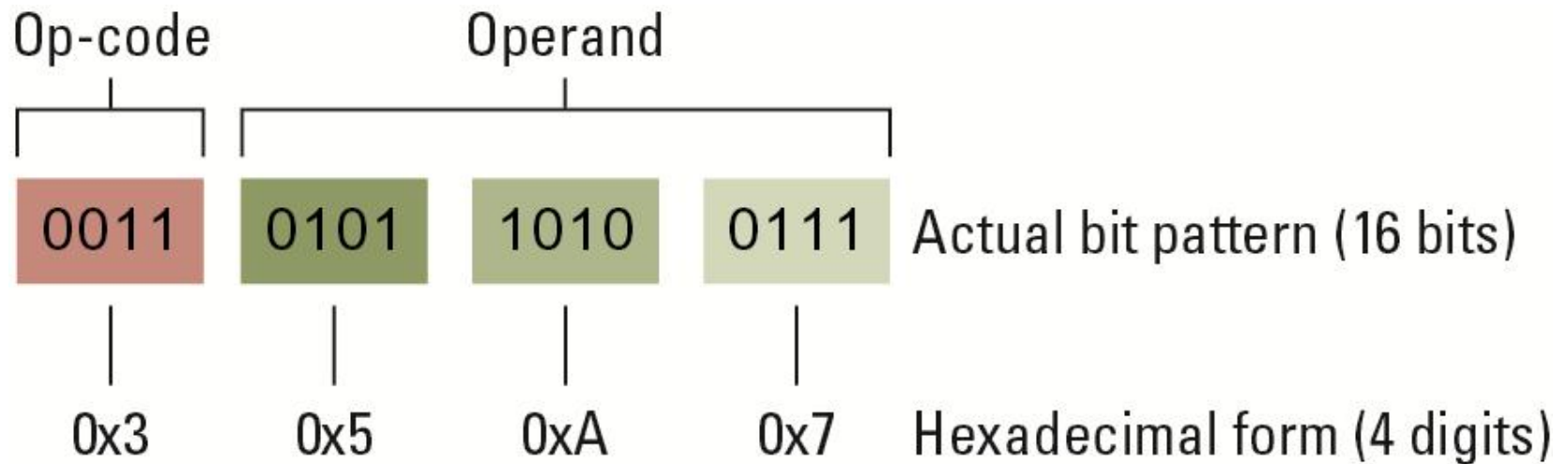


Figure 2.6 Decoding the instruction 0x35A7

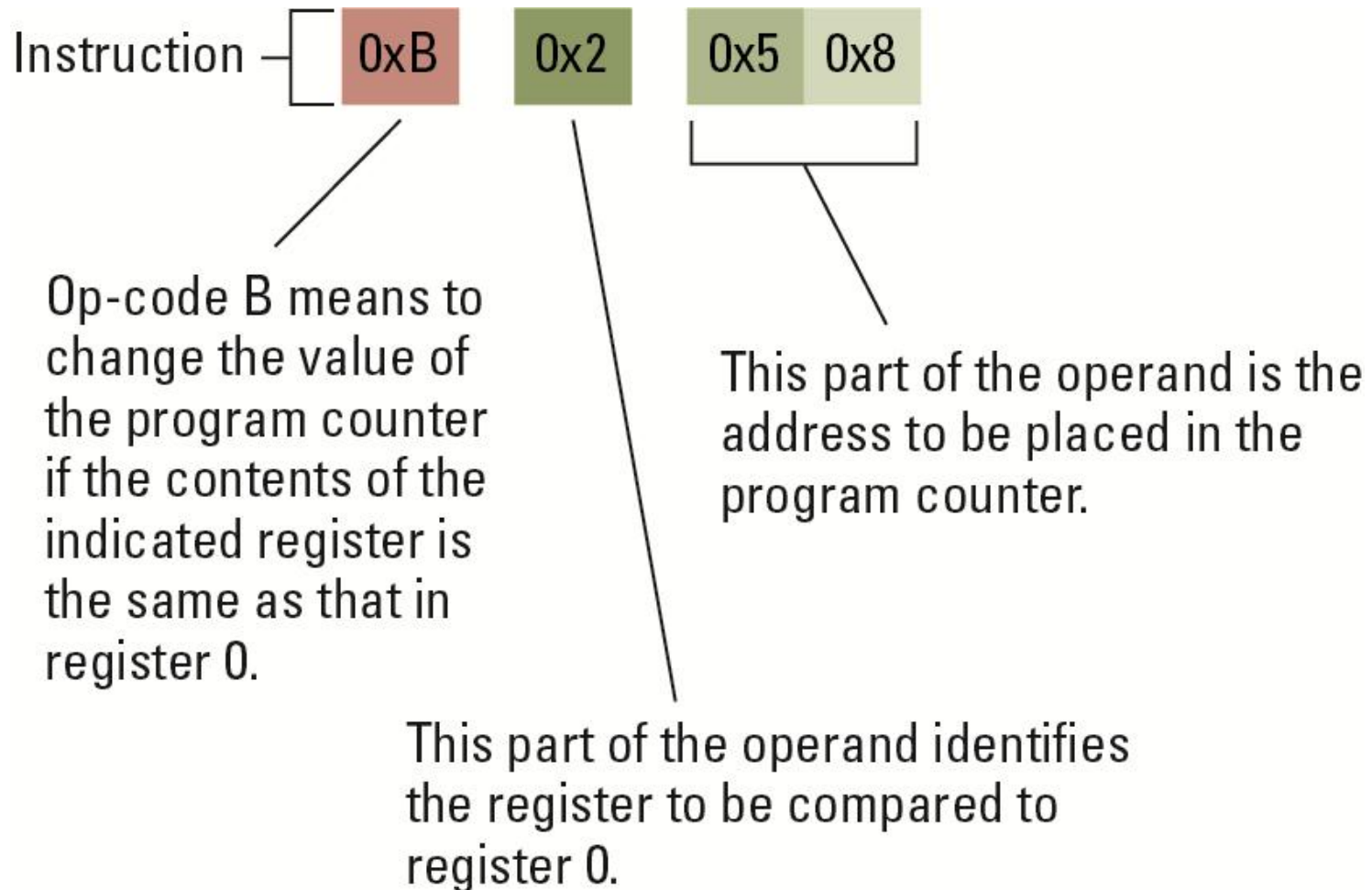


Figure 2.7 An encoded version of the instructions in Figure 2.2

Encoded instructions	Translation
0x156C	Load register 0x5 with the bit pattern found in the memory cell at address 0x6C.
0x166D	Load register 0x6 with the bit pattern found in the memory cell at address 0x6D.
0x5056	Add the contents of register 0x5 and 0x6 as though they were two's complement representation and leave the result in register 0x0.
0x306E	Store the contents of register 0x0 in the memory cell at address 0x6E.
0xC000	Halt.

2.3 Program Execution

- Controlled by two special purpose registers
 - Instruction register
 - holds current instruction
 - Program counter
 - holds address of next instruction
- Machine Cycle: (repeat these 3 steps)
 - Fetch, Decode, Execute

Figure 2.8 The machine cycle

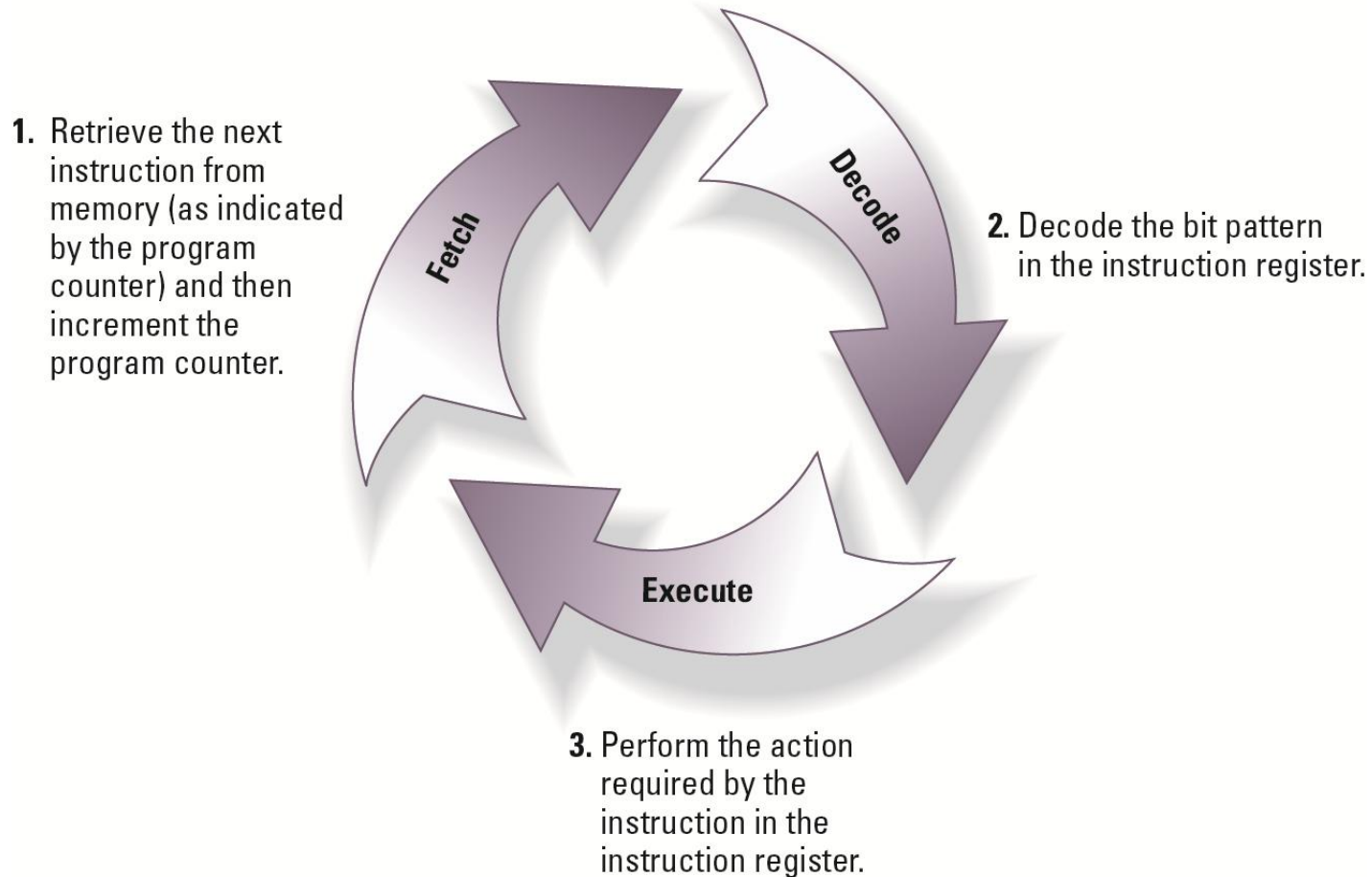


Figure 2.9 Decoding the instruction B258

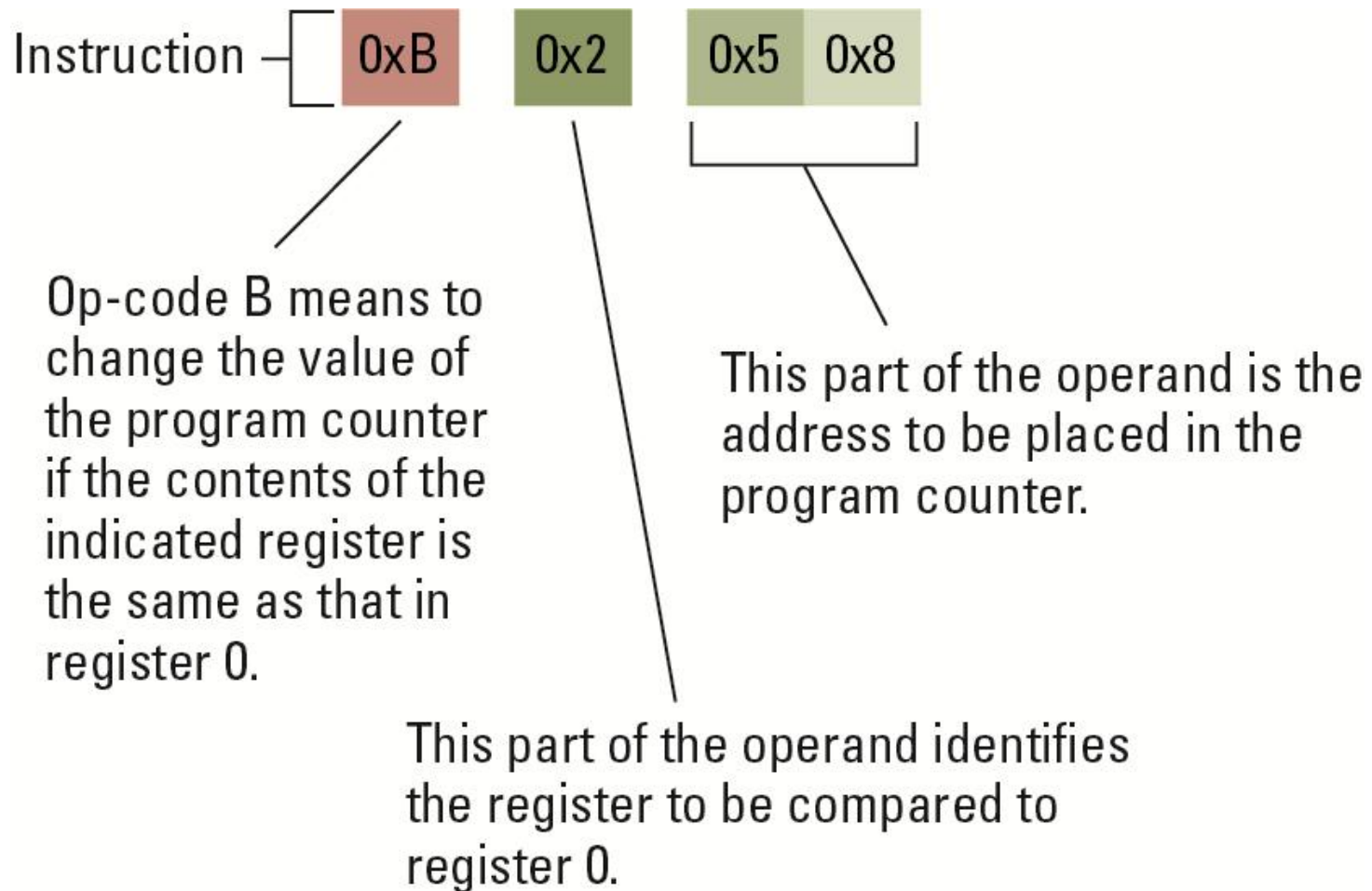


Figure 2.10 The program from Figure 2.7 stored in main memory ready for execution

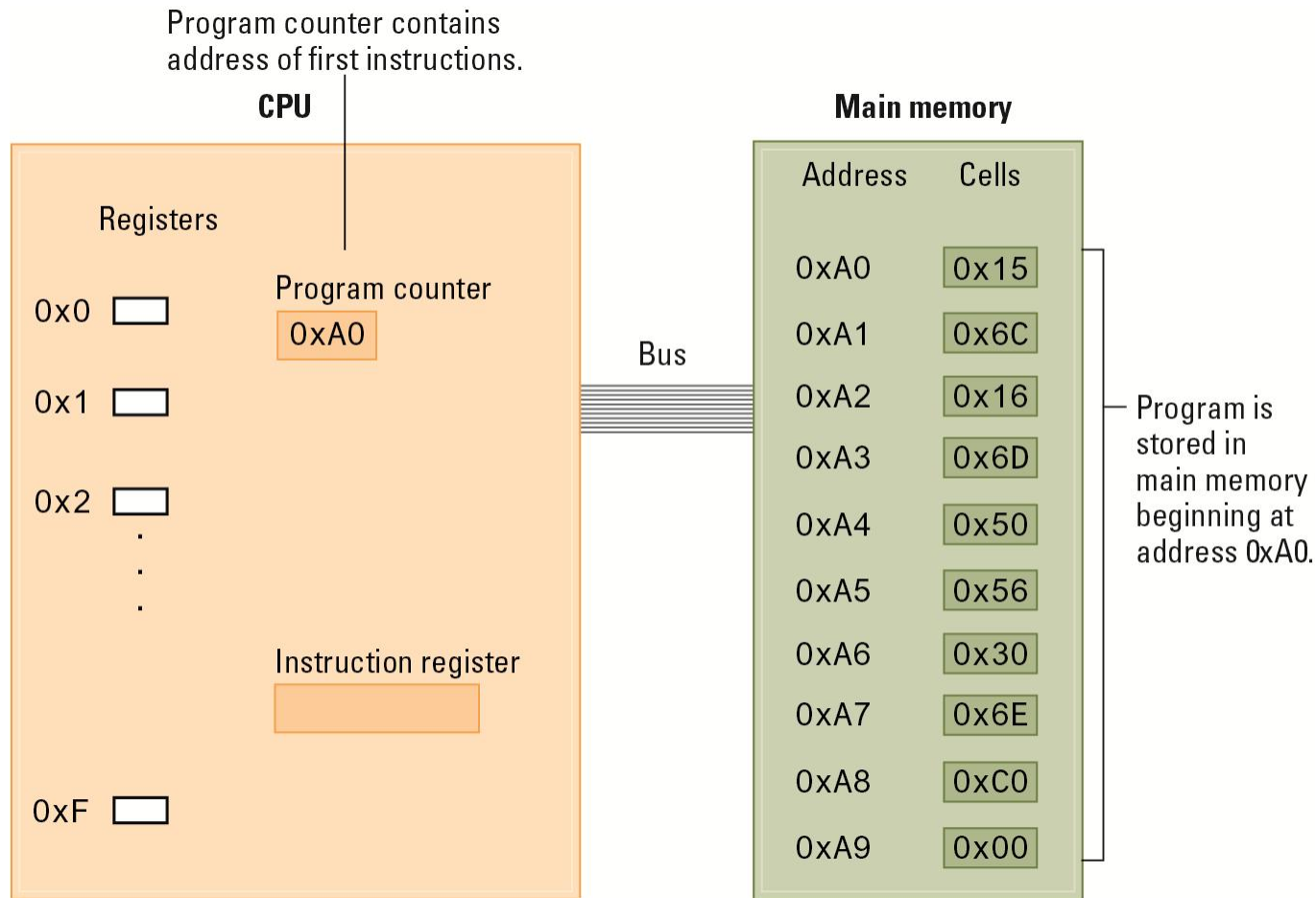
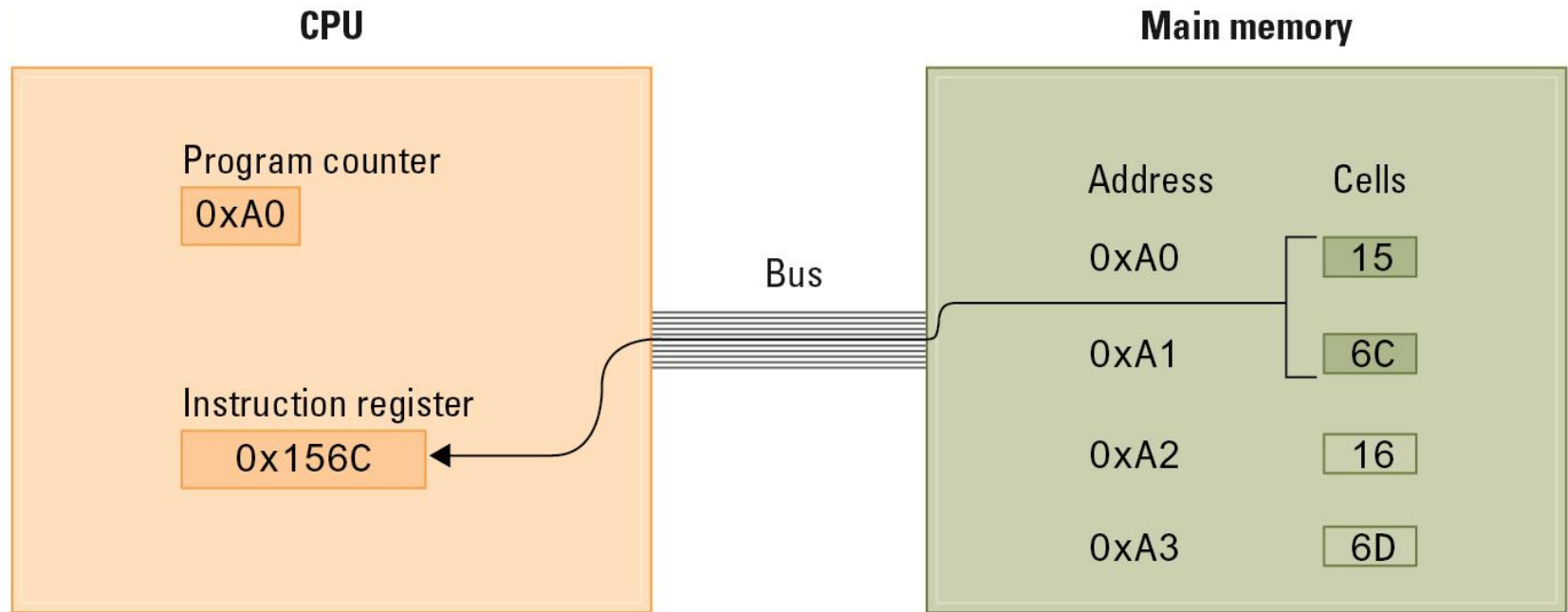
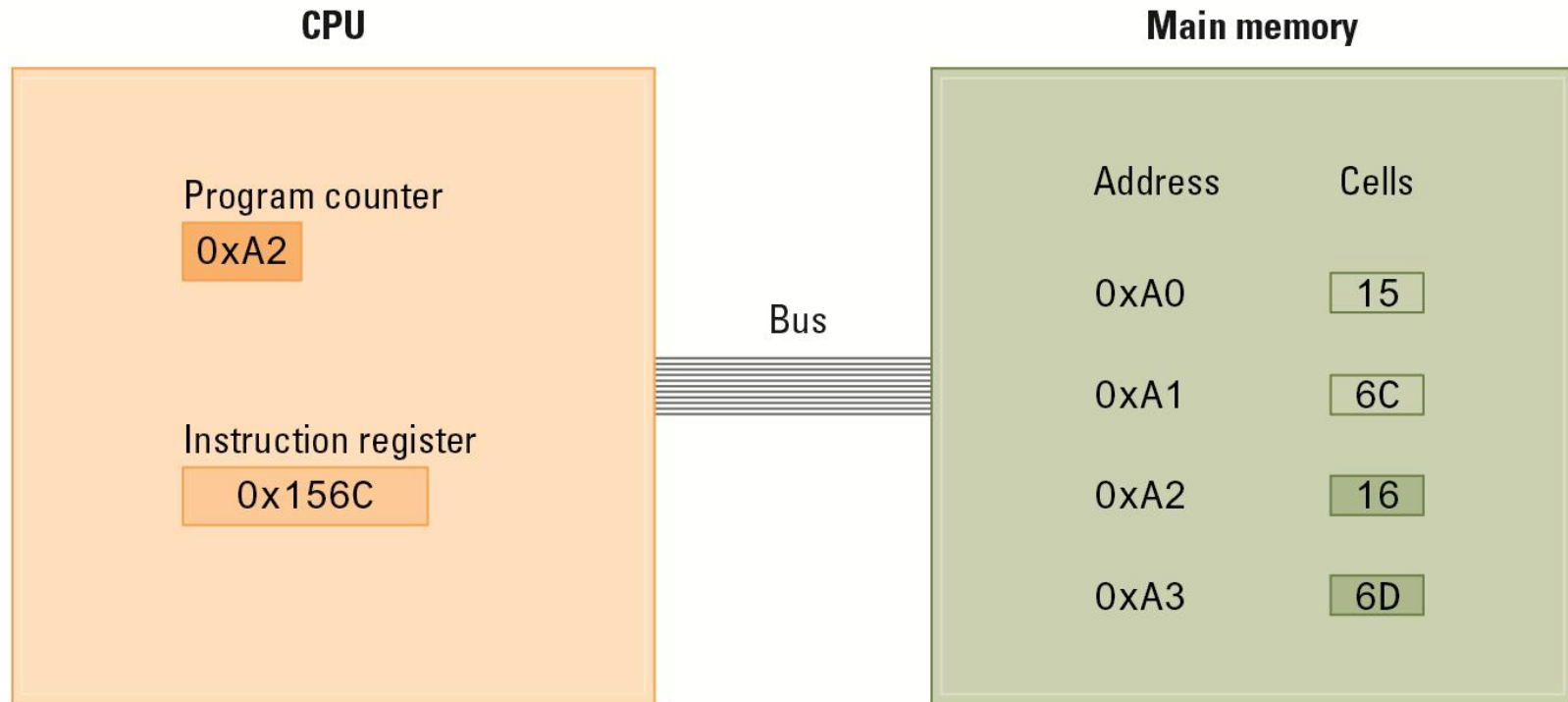


Figure 2.11 Performing the fetch step of the machine cycle



- a. At the beginning of the fetch step, the instruction starting at address 0xA0 is retrieved from memory and placed in the instruction register.

Figure 2.11 Performing the fetch step of the machine cycle (continued)

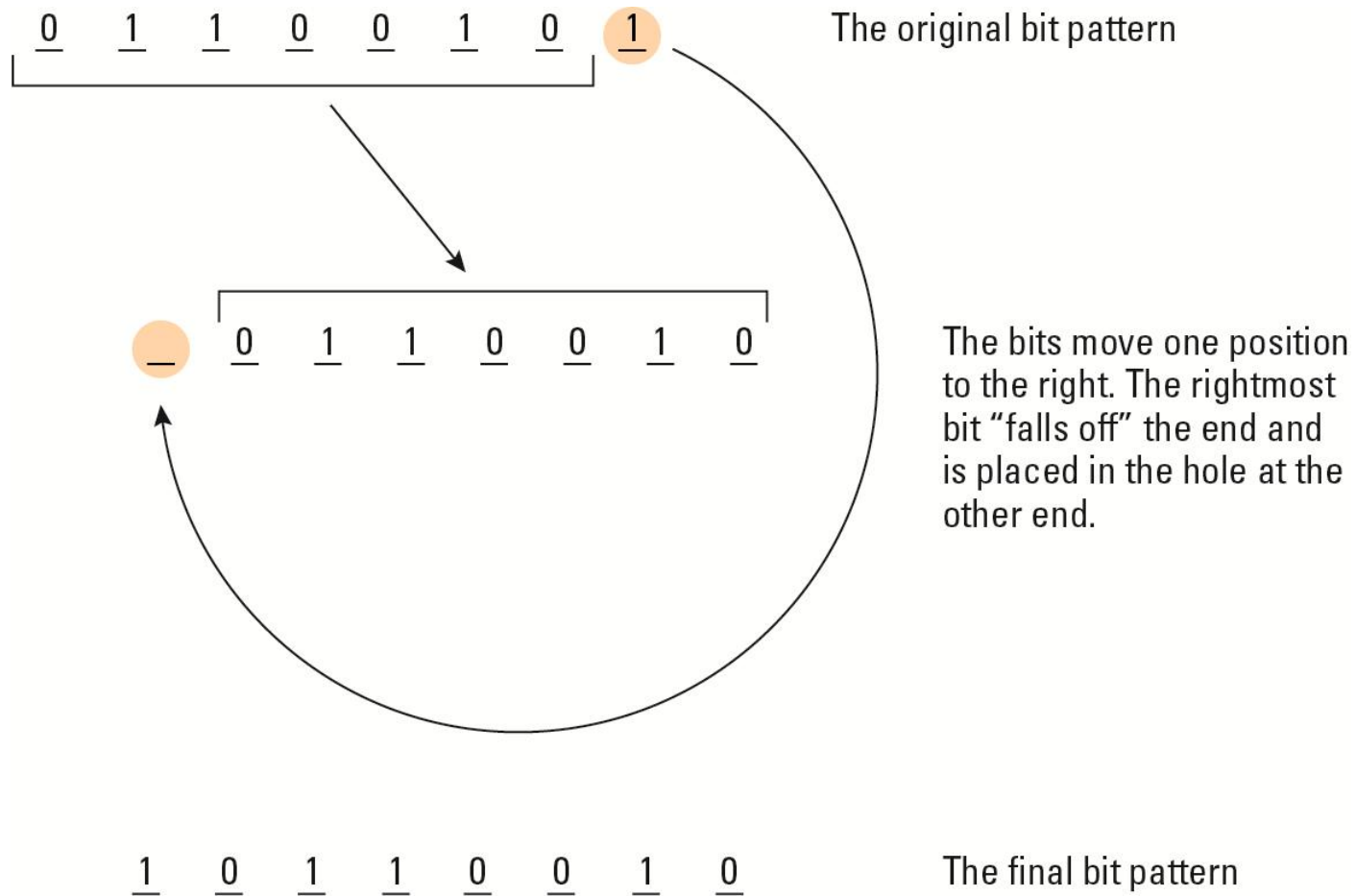


b. Then the program counter is incremented so that it points to the next instruction.

2.4 Arithmetic/Logic Instructions

- Logic Operations:
 - AND, OR, XOR
 - often used to mask an operand
- Rotation and Shift Operations:
 - circular shift, logical shift, arithmetic shift
- Arithmetic Operations:
 - add, subtract, multiply, divide
 - two's complement versus floating-point

Figure 2.12 Rotating the bit pattern 0x65 one bit to the right



2.5 Communicating with Other Devices

- **Controller:** handles communication between the computer and other devices
 - Specialized (by type of device)
 - General purpose (USB, HDMI)
- **Port:** The point at which a device connects to a computer
- **Memory-mapped I/O:** devices appear to the CPU as though they were memory locations

Figure 2.13 Controllers attached to a machine's bus

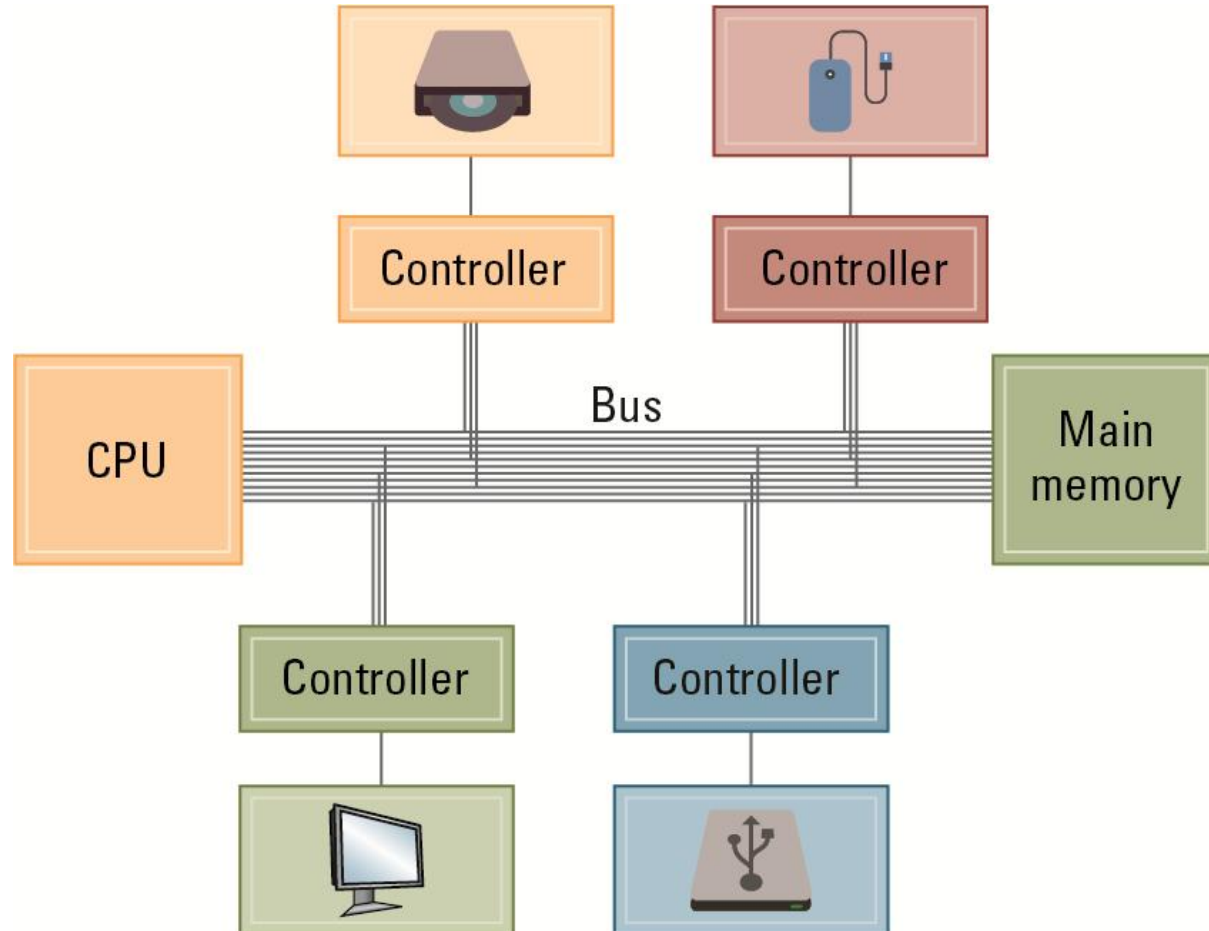
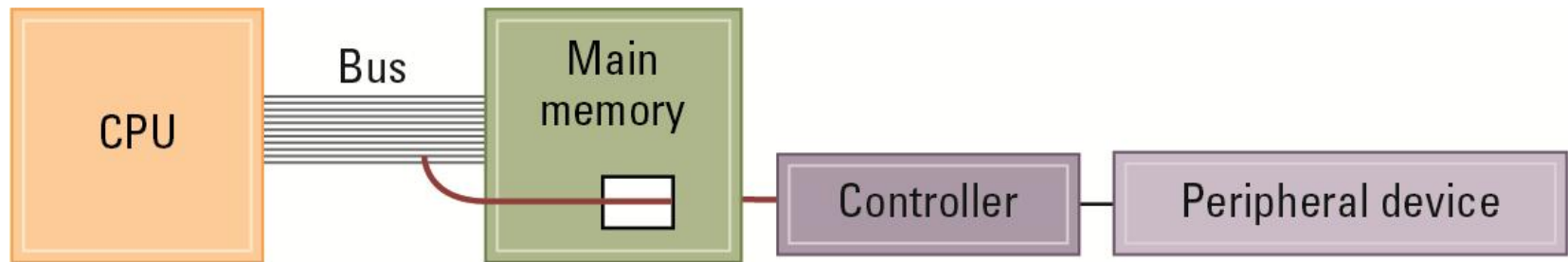


Figure 2.14 A conceptual representation of memory-mapped I/O



Communicating with Other Devices (continued)

- **Direct memory access (DMA):** Main memory access by a controller over the bus
 - **Von Neumann Bottleneck:** occurs when the CPU and controllers compete for bus access
- **Handshaking:** the process of coordinating the transfer of data between the computer and the peripheral device

Communicating with Other Devices (continued)

- **Popular Communication Media**
 - **Parallel Communication:** Several signals transferred at the same time, each on a separate “line” (computer’s internal bus)
 - **Serial Communication:** Signals are transferred one after the other over a single “line” (USB, FireWire)

Data Communication Rates

- Measurement units
 - bps: bits per second
 - Kbps: Kilo-bps (1,000 bps)
 - Mbps: Mega-bps (1,000,000 bps)
 - Gbps: Giga-bps (1,000,000,000 bps)
- Bandwidth: Maximum available rate

2.6 Programming Data Manipulation

- Programming languages shields users from details of the machine:
 - A single Python statement might map to one, tens, or hundreds of machine instructions
 - Programmer does not need to know if the processor is RISC or CISC
 - Assigning variables surely involves LOAD, STORE, and MOVE op-codes

Bitwise Problems as Python Code

```
print(bin(0b10011010 & 0b11001001))  
# Prints '0b10001000'
```

```
print(bin(0b10011010 | 0b11001001))  
# Prints '0b11011011'
```

```
print(bin(0b10011010 ^ 0b11001001))  
# Prints '0b1010011'
```

Control Structures

- If statement:

```
if (water_temp > 140):  
    print('Bath water too hot!')
```

- While statement:

```
while (n < 10):  
    print(n)  
    n = n + 1
```

Functions

- **Function:** A name for a series of operations that should be performed on the given parameter or parameters
- **Function call:** Appearance of a function in an expression or statement

```
x = 1034
y = 1056
z = 2078
biggest = max(x, y, z)
print(biggest)    # Prints '2078'
```

Functions (continued)

- **Argument Value:** A value plugged into a parameter
- **Fruitful functions** **return** a value
- **void functions**, or **procedures**, do not return a value

```
sideA = 3.0
```

```
sideB = 4.0
```

```
# Calculate third side via Pythagorean Theorem
```

```
hypotenuse = math.sqrt(sideA**2 + sideB**2)
```

```
print(hypotenuse)
```

Input / Output

```
# Calculates the hypotenuse of a right triangle

import math

# Inputting the side lengths, first try
sideA = int(input('Length of side A? '))
sideB = int(input('Length of side B? '))

# Calculate third side via Pythagorean Theorem
hypotenuse = math.sqrt(sideA**2 + sideB**2)

print(hypotenuse)
```

Marathon Training Assistant

```
# Marathon training assistant.
```

```
import math
```

```
# This function converts a number of minutes and  
# seconds into just seconds.
```

```
def total_seconds(min, sec):  
    return min * 60 + sec
```

```
# This function calculates a speed in miles per hour  
given
```

```
# a time (in seconds) to run a single mile.
```

```
def speed(time):  
    return 3600 / time
```

Marathon Training Assistant (continued)

```
# Prompt user for pace and mileage.
pace_minutes = int(input('Minutes per mile? '))
pace_seconds = int(input('Seconds per mile? '))
miles = int(input('Total miles? '))

# Calculate and print speed.
mph = speed(total_seconds(pace_minutes, pace_seconds))
print('Your speed is ' + str(mph) + ' mph')

# Calculate elapsed time for planned workout.
total = miles * total_seconds(pace_minutes, pace_seconds)
elapsed_minutes = total // 60
elapsed_seconds = total % 60

print('Your elapsed time is ' + str(elapsed_minutes) +
      ' mins ' + str(elapsed_seconds) + ' secs')
```

Figure 2.15 Example Marathon Training Data

Time Per Mile				Total Elapsed Time	
Minutes	Seconds	Miles	Speed (mph)	Minutes	Seconds
9	14	5	6.49819494584	46	10
8	0	3	7.5	24	0
7	45	6	7.74193548387	46	30
7	25	1	8.08988764044	7	25

2.7 Other Architectures

- Technologies to increase throughput:
 - Pipelining: Overlap steps of the machine cycle
 - Parallel Processing: Use multiple processors simultaneously
 - SISD: Single Instruction, Single Data
 - No parallel processing
 - MIMD: Multiple Instruction, Multiple Data
 - Different programs, different data
 - SIMD: Single Instruction, Multiple Data
 - Same program, different data

Copyright



This work is protected by United States copyright laws and is provided solely for the use of instructions in teaching their courses and assessing student learning. dissemination or sale of any part of this work (including on the World Wide Web) will destroy the integrity of the work and is not permitted. The work and materials from it should never be made available to students except by instructors using the accompanying text in their classes. All recipients of this work are expected to abide by these restrictions and to honor the intended pedagogical purposes and the needs of other instructors who rely on these materials.