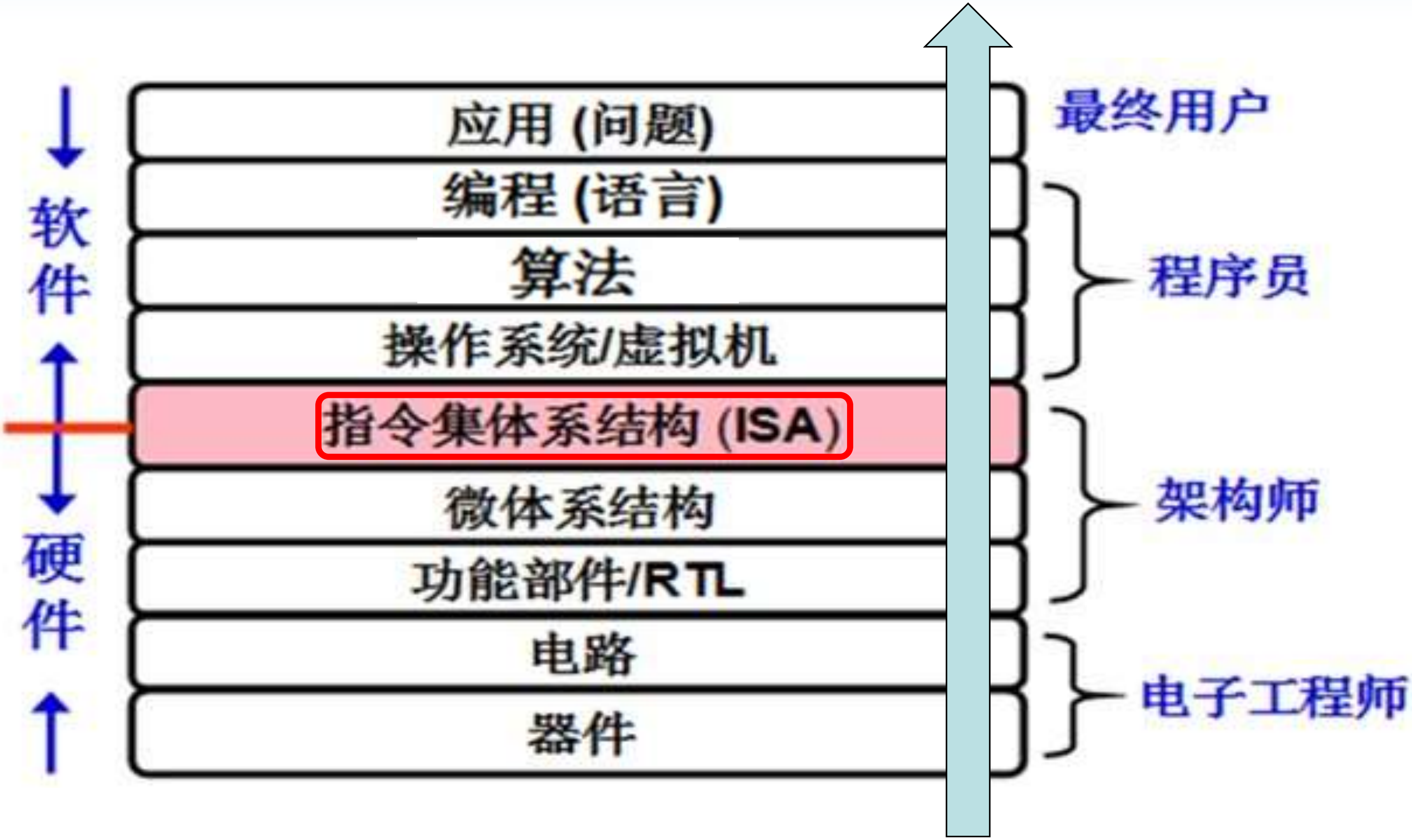


应用 (问题)

编程 (语言)

算法

操作系统/虚拟机

指令集体系结构 (ISA)

微体系结构

功能部件/RTL

电路

器件

软件

硬件

最终用户

程序员

架构师

电子工程师

Chapter 2

Data manipulation

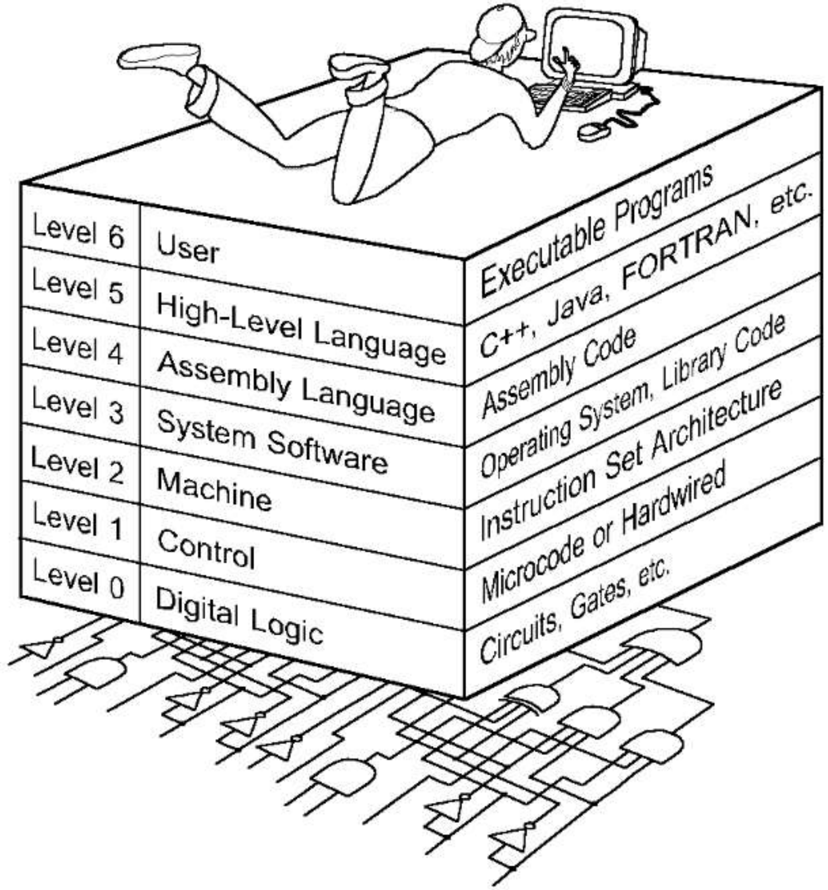
Chapter 2: Data Manipulation

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

For Computers

- Data are stored as 0 and 1
- Instructions are also expressed and stored as 0 and 1

  → **in memory**



Central Processing Unit (CPU)

- An electronic circuit that can execute computer programs
  - Intel
  - AMD
  - ARM

- To understand CPU, we need to know what computer programs are.

Stored Program Concept

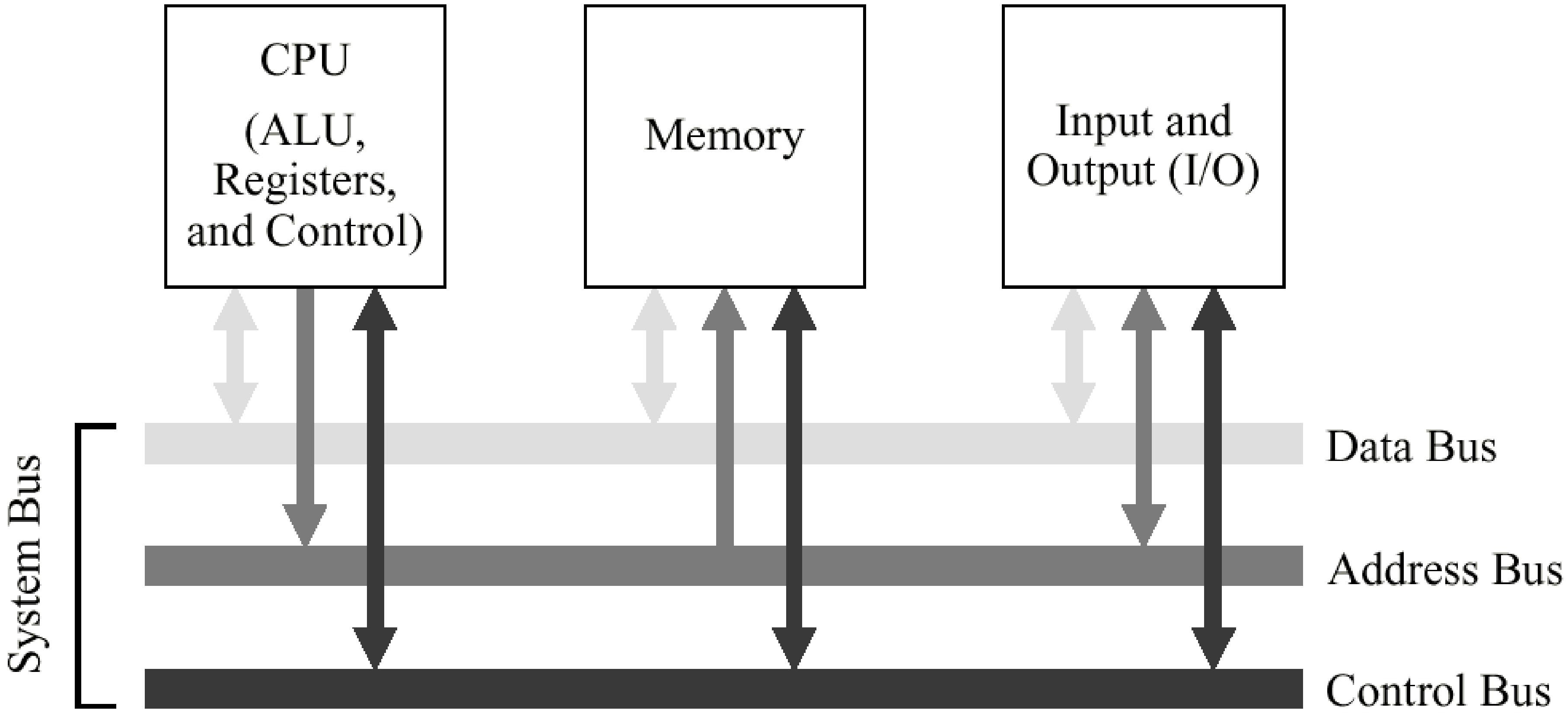
> "*The final major step in the development of the general purpose electronic computer was the idea of a stored program*..."
>
> *Brian Randell*

Stored-Program Concept

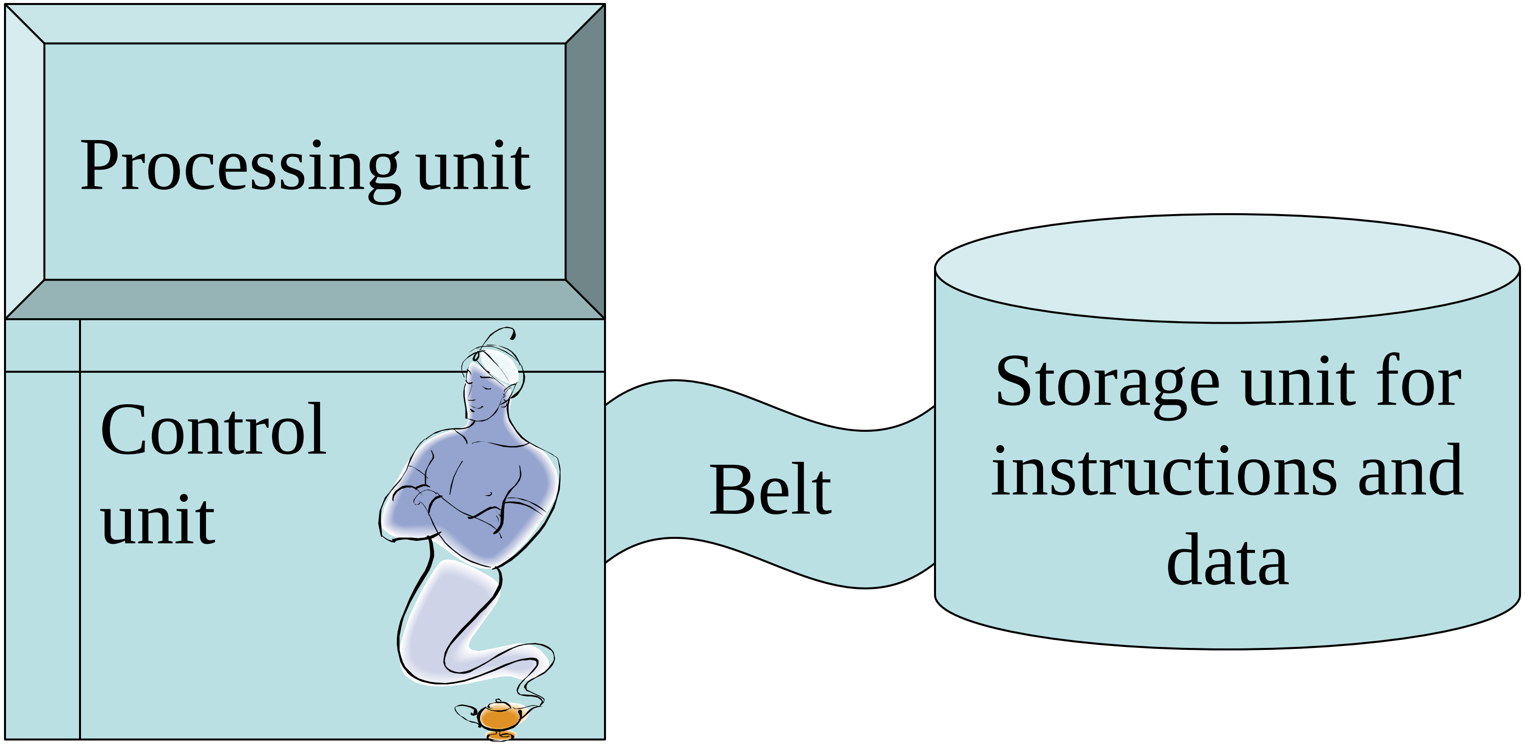
- Program: a sequence of instructions
- *Stored-program concept*:
  - A program can be encoded as bit patterns and stored in main memory, just like data.
  - From there, the CPU can fetch the instructions and execute them.
- Advantage: programmable
  - We can use a single machine to perform different functions by loading different prog.
  - Instead of rewriting the CPU, the program can be altered by changing the contents of Main Memory.

2.1 Computer Architecture

Von-Neumann Model



Outline of the Magic Box

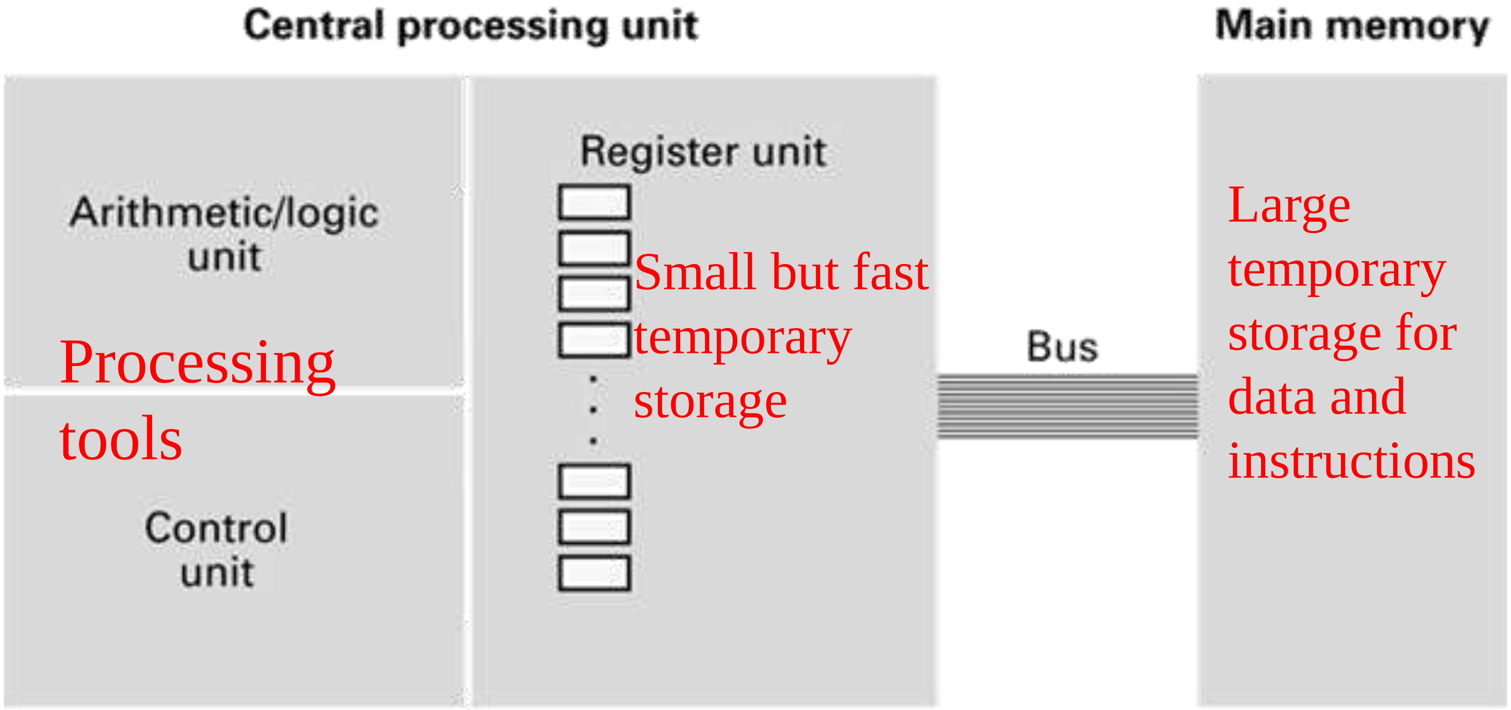


Computer Architecture

- Central Processing Unit (CPU) or processor
  - Arithmetic/Logic unit versus Control unit
  - Registers（寄存器）
    - General purpose（通用）
    - Special purpose（专用）
- Bus
- Motherboard

Von Neumann Architecture

- General purpose electronic computer



Processing tools

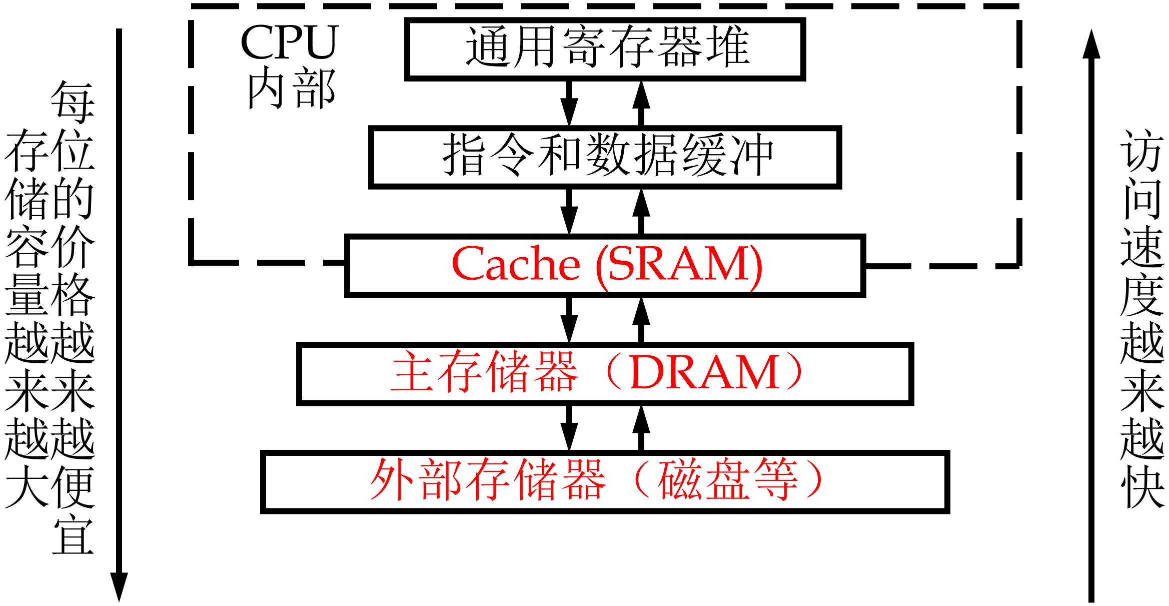
Small but fast temporary storage

Large temporary storage for data and instructions

Fig. 2.1

计算机三级存储系统

三级存储系统图示



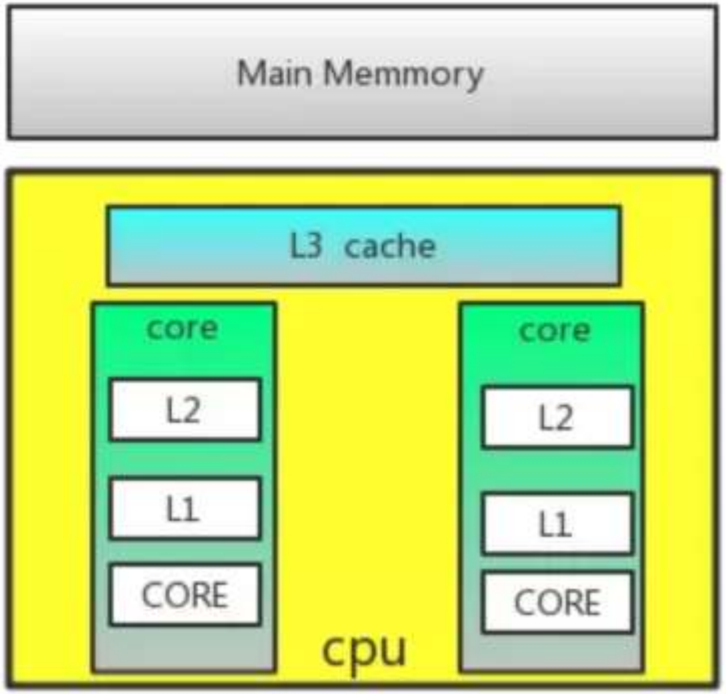
计算机三级存储系统

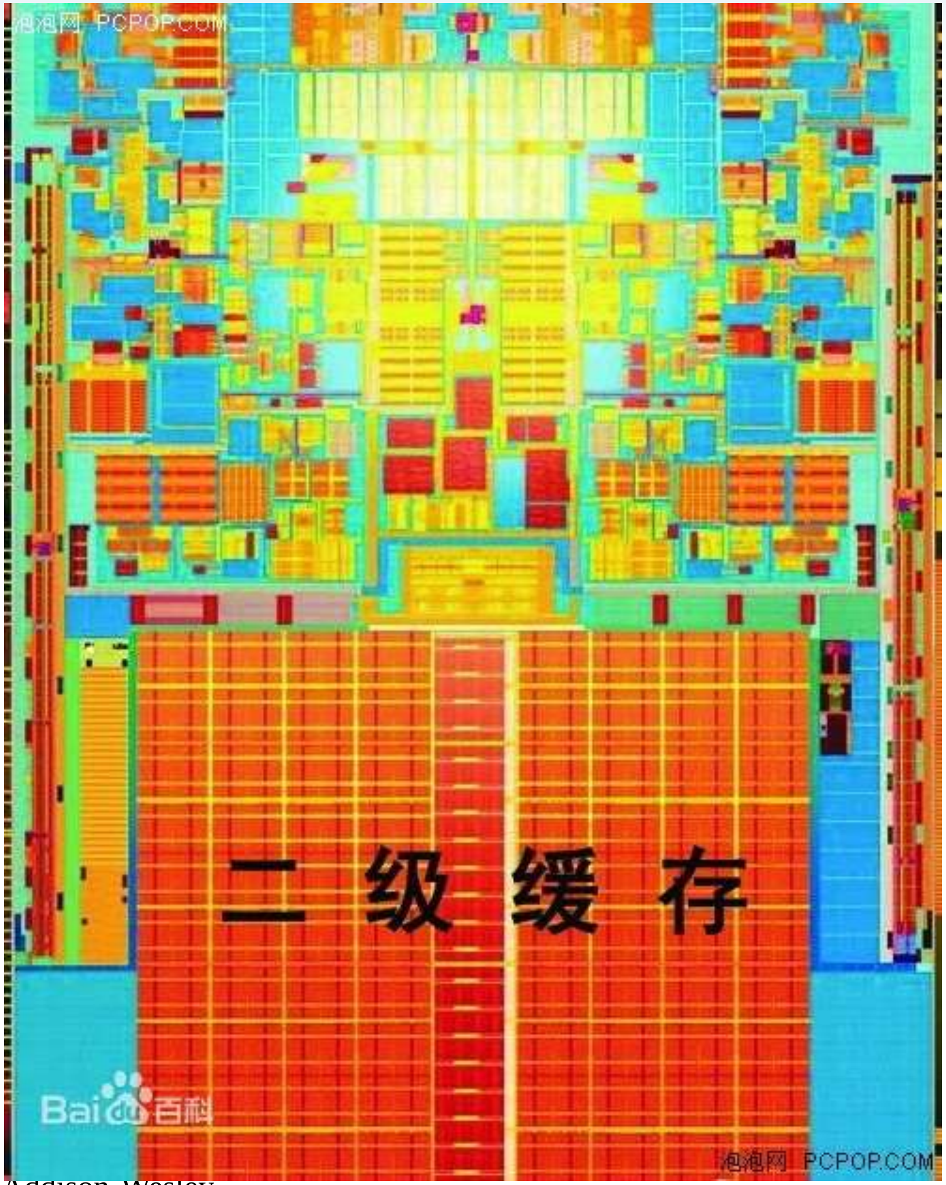
缓存（Cache）

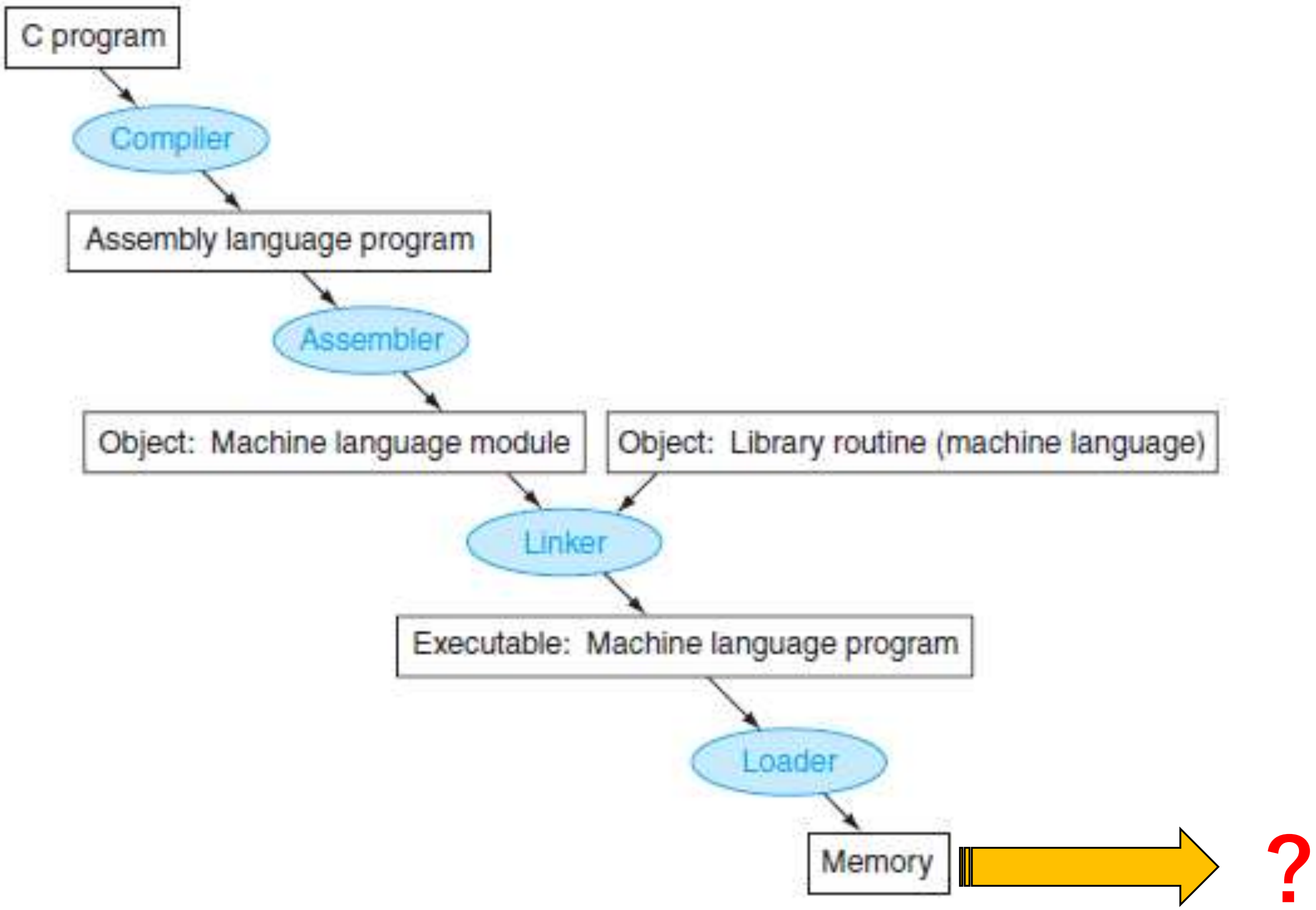
全称为高速缓冲存储器，如 SRAM，主要存放 CPU 在当前一小段时间内多次使用的程序和数据，以缓解 CPU 和主存之间的速度差异；速度极快，容量小。CPU 从主存中取指令、数据，在一定时间内地址范围常局限于内存的某个小区域（**访问局部性原理**），因此可以将正在使用的部分（热点区指令和数据），提前预取并存储到一个高速的、小容量的 Cache 中，使得 CPU 访存 “等价于” CPU 访问 Cache。

多级Cache

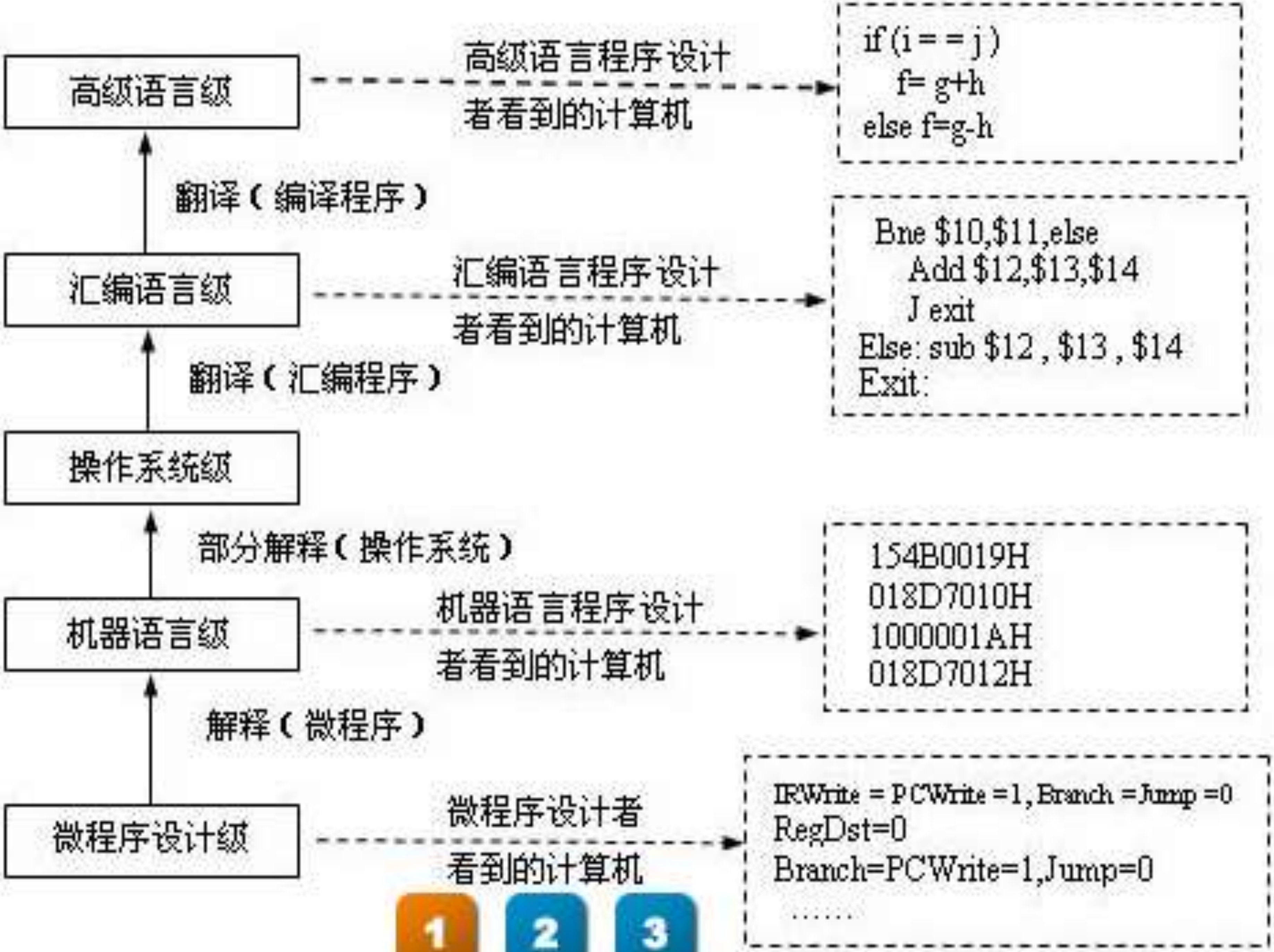
通过在原始Cache和存储器之间增加另一级Cache，第一级Cache可以小到足以跟上飞快的CPU，而第二级Cache能够大到足以捕捉到对主存进行的大多数访问，因而可以有效降低缺失代价。


Main Memmory
L3 cache
core
L2
L1
CORE
cpu
core
L2
L1
CORE


泡泡网 PCPOP.COM
二 级 缓 存
Bai百科
泡泡网 PCPOP.COM


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory
?

2.2 Machine Language



高级语言级

翻译（编译程序）

汇编语言级

翻译（汇编程序）

操作系统级

部分解释（操作系统）

机器语言级

解释（微程序）

微程序设计级

高级语言程序设计者看到的计算机

```
if (i = = j )
    f= g+h
else f=g-h
```

汇编语言程序设计者看到的计算机

```
 Bne $10,$11,else
    Add $12,$13,$14
    J exit
Else: sub $12 , $13 , $14
Exit:
```

机器语言程序设计者看到的计算机

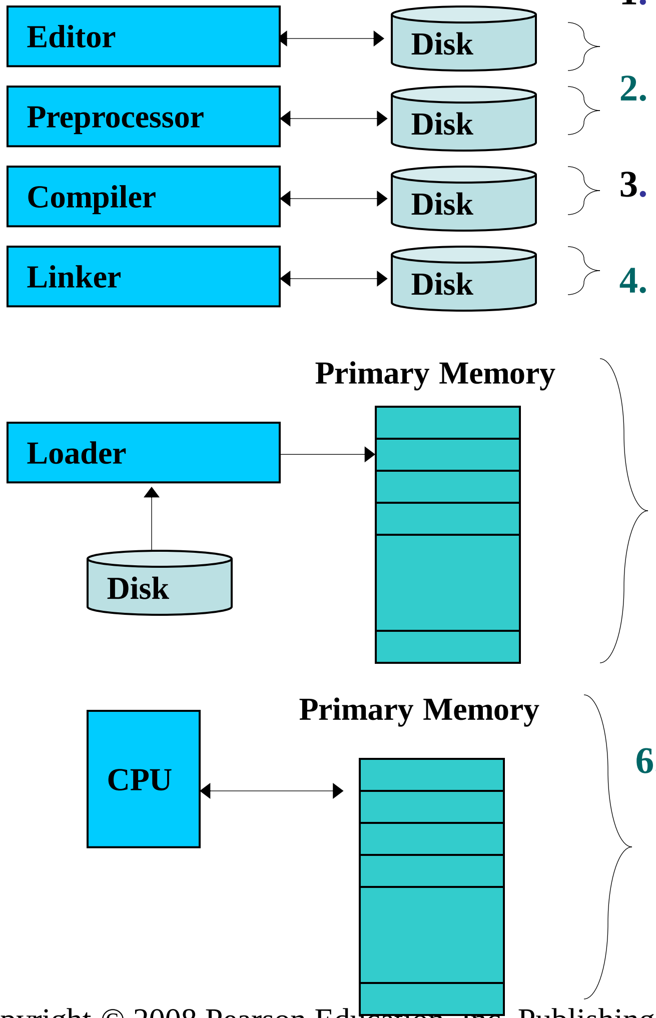
```
154B0019H
018D7010H
1000001AH
018D7012H
```

微程序设计者看到的计算机

```
IRWrite = PCWrite =1, Branch =Jump =0
RegDst=0
Branch=PCWrite=1,Jump=0
  ……
```

A Typical C Program Development Environment

- **Phases of C Programs:**



1. Program is created in the editor and stored on disk
2. Preprocessor program processes the code
3. Compiler creates object code and stores it on disk.
4. Linker links the object code with the libraries
5. Loader puts program in memory.
6. CPU takes each instruction and executes it, possibly storing new data values as the program executes

1. Edit

2. Preprocess

3. Compile

4. Link

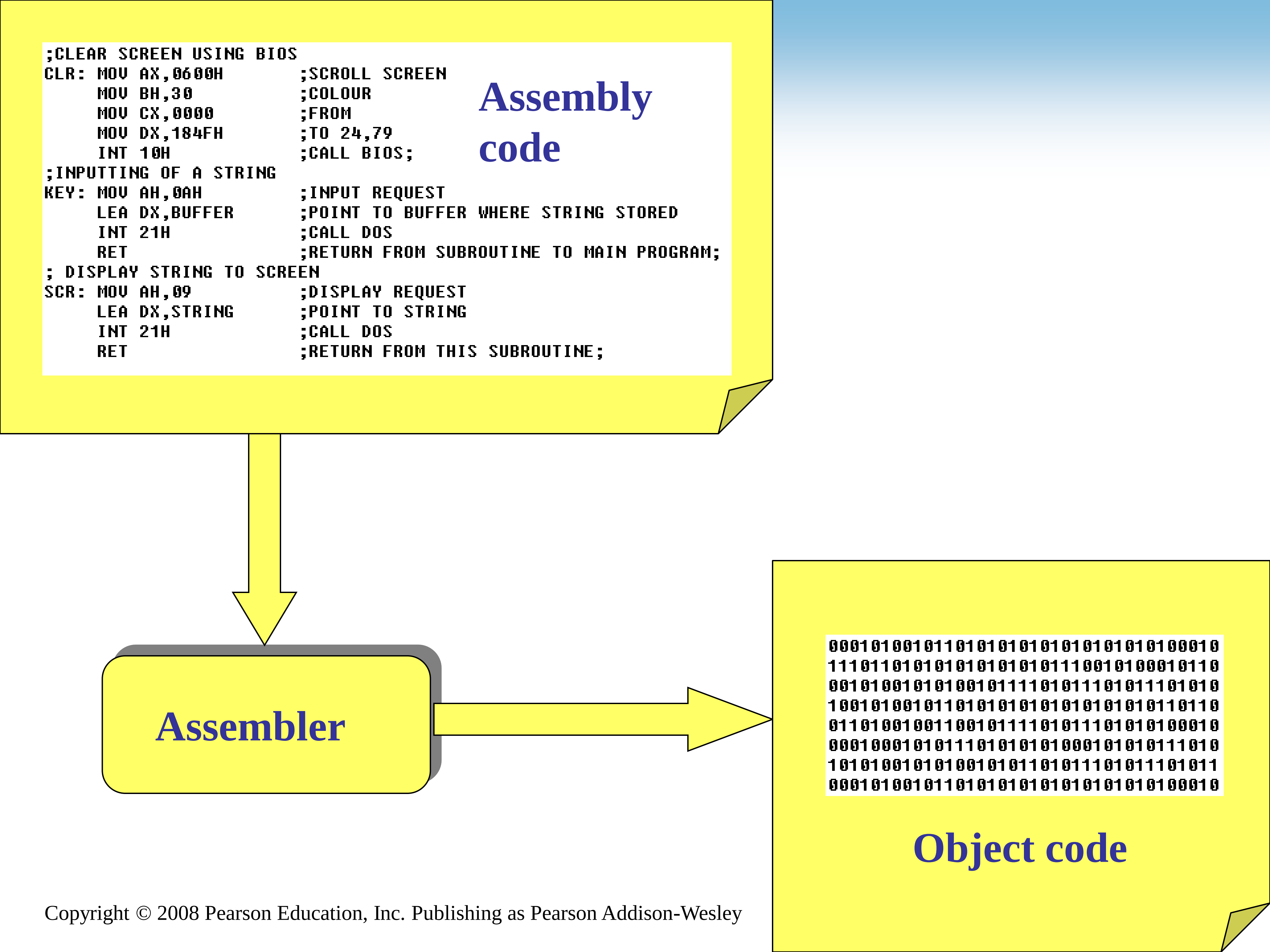
---

5. Load

6. Execute

Some concepts

- Instruction set (指令集): collection of instructions that a processor can execute
- A *compiler* （编译器） translates a high level language, which is architecture independent, into assembly language, which is architecture dependent.
- An *assembler*（汇编器） translates assembly language programs into executable binary codes.



```
;CLEAR SCREEN USING BIOS
CLR: MOV AX,0600H       ;SCROLL SCREEN
     MOV BH,30          ;COLOUR
     MOV CX,0000        ;FROM
     MOV DX,184FH       ;TO 24,79
     INT 10H            ;CALL BIOS;
;INPUTTING OF A STRING
KEY: MOV AH,0AH         ;INPUT REQUEST
     LEA DX,BUFFER      ;POINT TO BUFFER WHERE STRING STORED
     INT 21H            ;CALL DOS
     RET                ;RETURN FROM SUBROUTINE TO MAIN PROGRAM;
; DISPLAY STRING TO SCREEN
SCR: MOV AH,09          ;DISPLAY REQUEST
     LEA DX,STRING      ;POINT TO STRING
     INT 21H            ;CALL DOS
     RET                ;RETURN FROM THIS SUBROUTINE;
```

```
0001010010110101010101010101010100010
1110110101010101010101110010100010110
0010100101010010111101011101011101010
1001010010110101010101010101010110110
0110100100110010111101011101010100010
0001000101011101010101000101010111010
1010100101010010101101011101011101011
0001010010110101010101010101010100010
```

- Different for various processor type (size, kind of operation, type of operands, type of results)
- Incompatibility: in contrast to high-level language (re-compile)
- One exception: Java bytecodes for virtual machine , run on any processor
- Computer systems are often identified by CPU
- Instruction sets: 80x86 CPU (e.g. IBM PC) ~ Motorola PowerPC CPU (e.g. Apple Macintosh)

Terminology

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

What to do

+

Specified information

Machine Language Philosophies

精简指令集

- 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: Pentium from Intel

复杂指令集

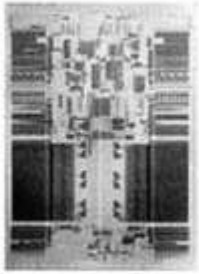
开源指令集RISC-V

David Patterson(RISC-V的开放标准芯片指令的创建者)

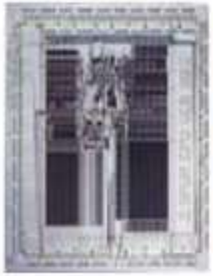
- 2010年，加州大学伯克利分校的一个研究团队为新项目选架构
- Arm、MIPS、SPARC和x86指令集越来越复杂，还有很多法律问题
- RISC-V指令集架构简单、完全开源并且免费，将基准指令和扩展指令分开，可以通过扩展指令做定制化的模块和扩展
- RISC-V已经有多个单位加入，包括谷歌、华为、三星、英伟达、高通、麻省理工学院、普林顿大学、印度理工学院、中科院计算所等。
- RISC-V不仅为我们提供了一个庞大且快速发展的社区，而且还给予了我们对架构进行创新以达到极致能效的机会，无需购买昂贵的架构许可证
- NVIDIA和高通已经在使用RISC-V架构开发自己的物联网处理器和GPU内存控制器

RISC-I
1981

RISC-II
1983



RISC-III (SOAR)
1984



RISC-IV (SPUR)
1988

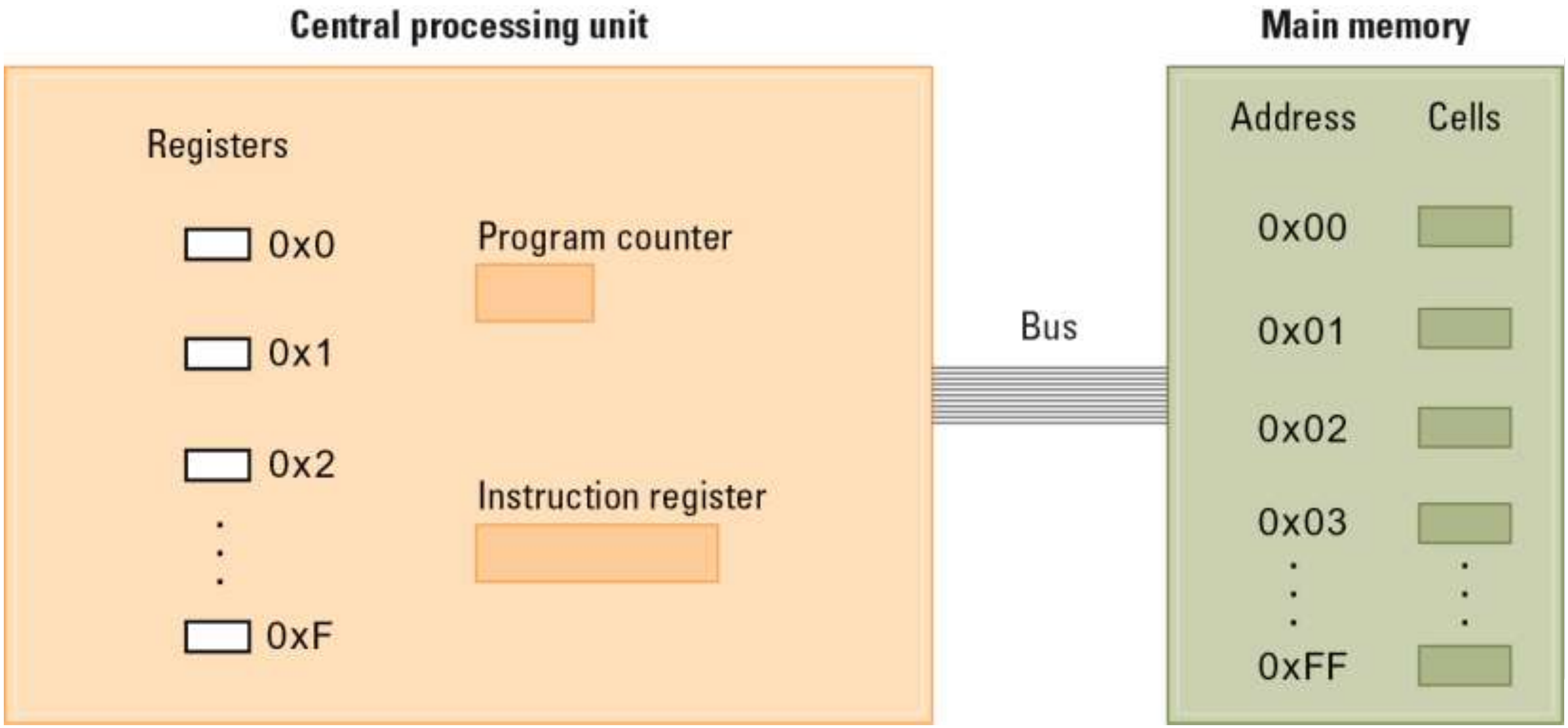
RISC-V
2013

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)

- Programing 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

Figure 2.4 **The architecture of the machine (Vole) described in Appendix C**



Parts of a Machine Instruction

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

Figure 2.5 The composition of a Vole instruction

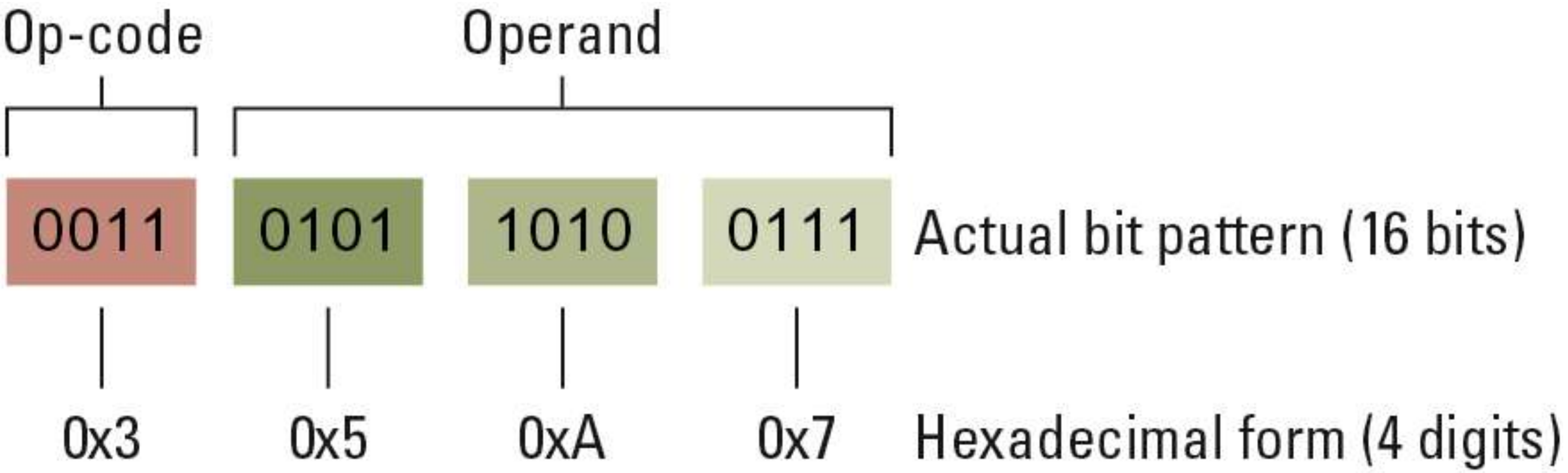
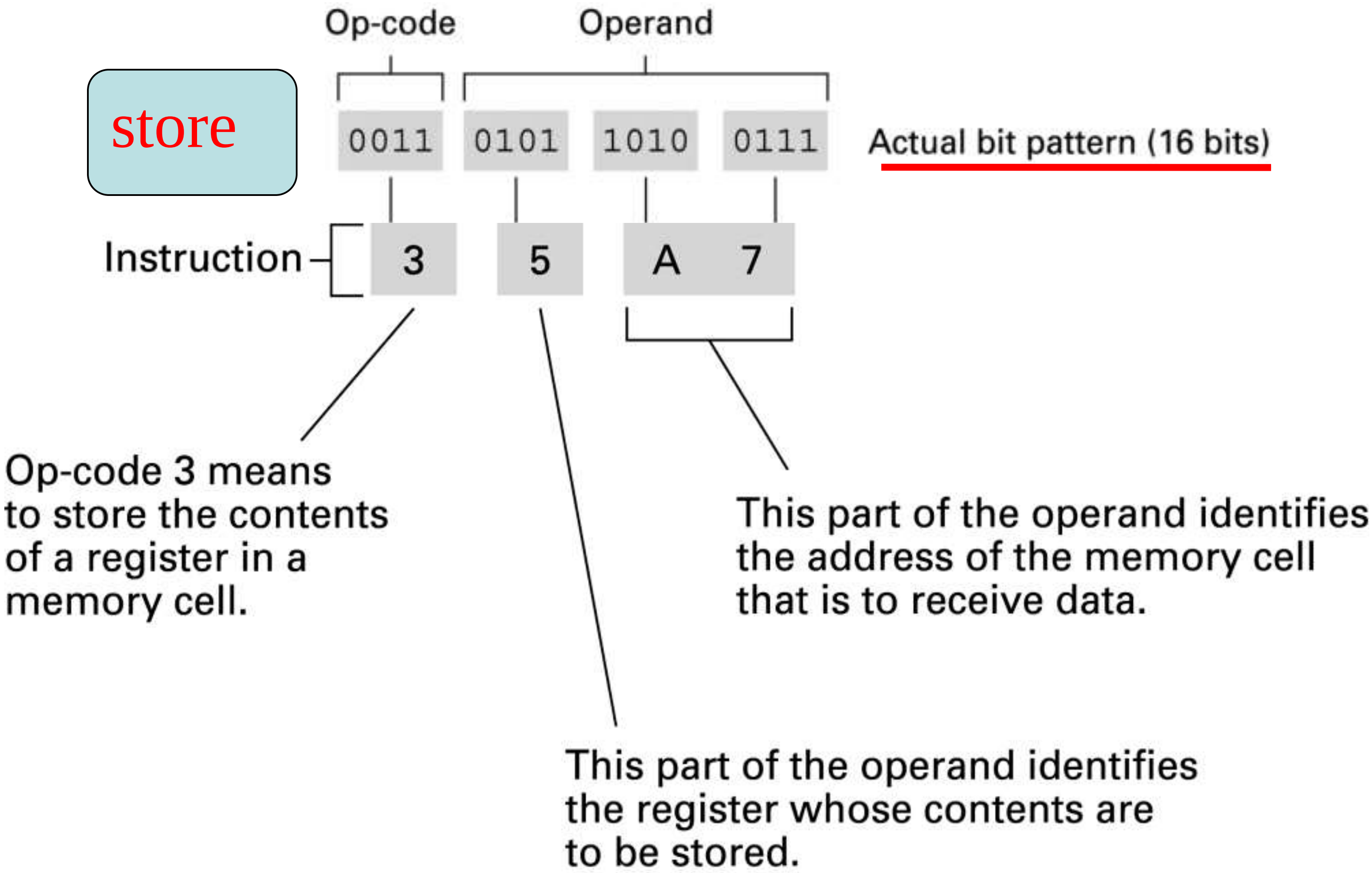


Figure 2.5 and Figure 2.6

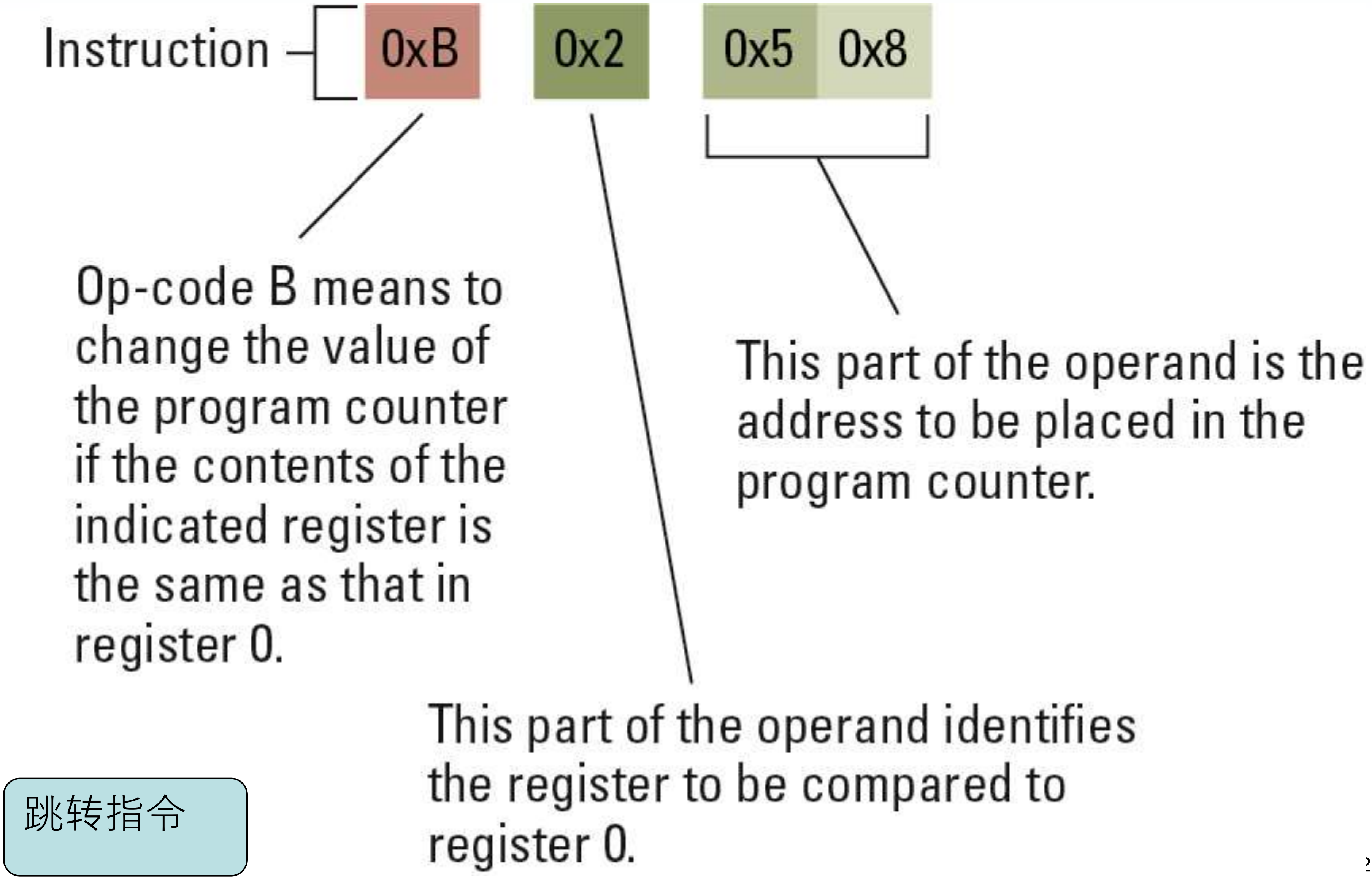


Op-code 3 means to store the contents of a register in a memory cell.

This part of the operand identifies the address of the memory cell that is to receive data.

This part of the operand identifies the register whose contents are to be stored.

Figure 2.9 **Decoding the instruction B258**



Instruction

0xB

0x2

0x5 0x8

Op-code B means to change the value of the program counter if the contents of the indicated register is the same as that in register 0.

This part of the operand is the address to be placed in the program counter.

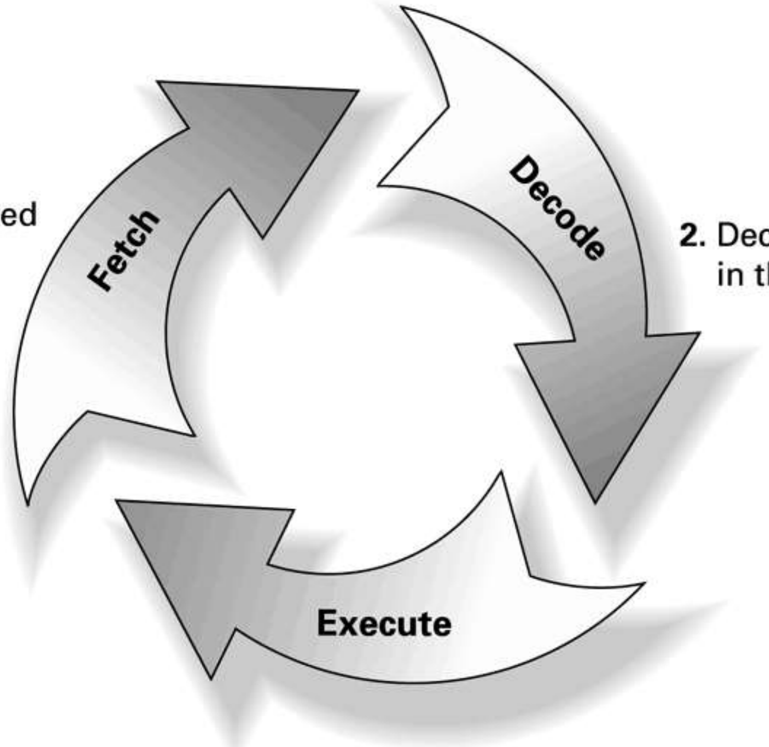
This part of the operand identifies the register to be compared to register 0.

跳转指令

2.3 Program Execution

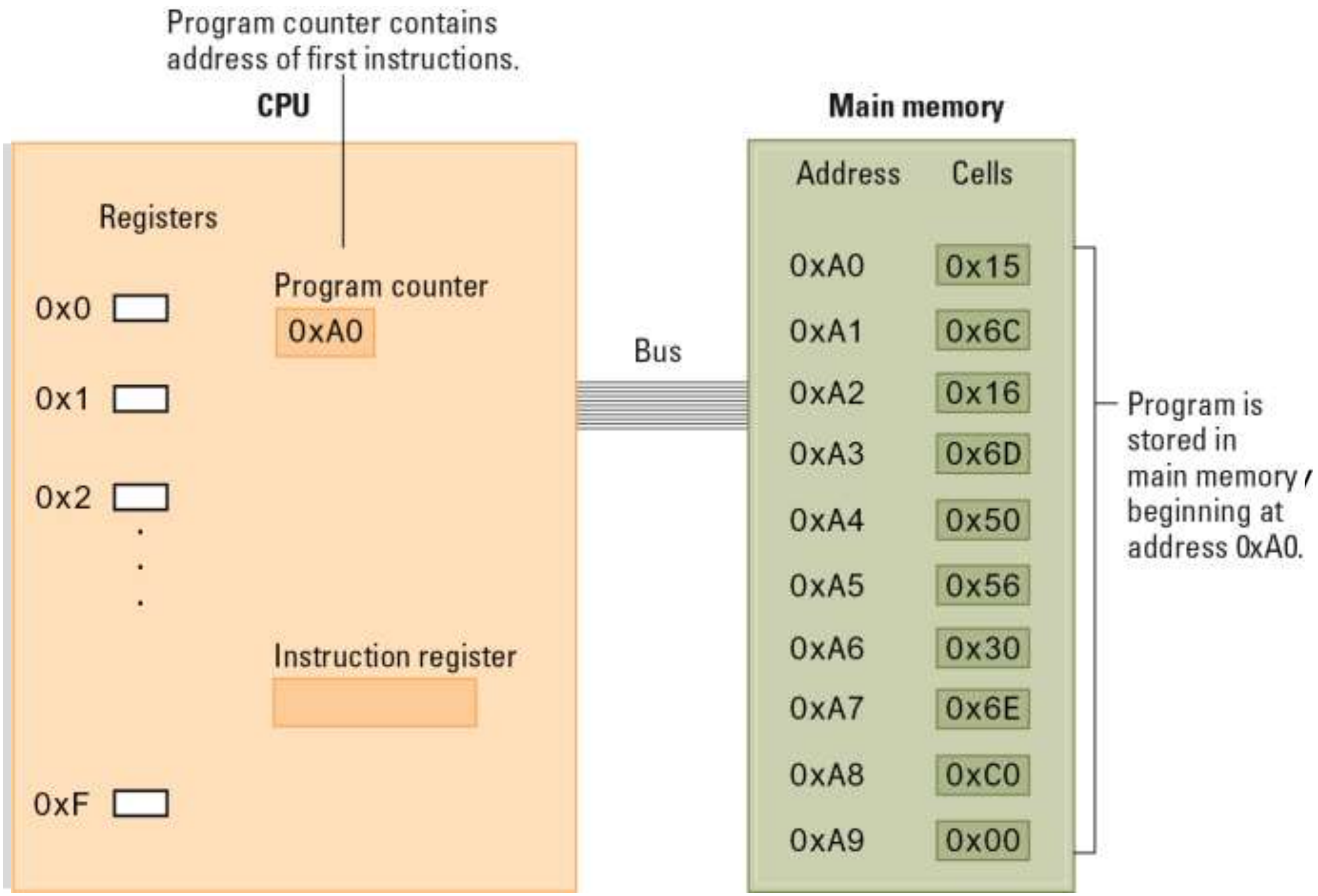
- Controlled by two special-purpose registers
  - Program counter(程序计数器): holds address of next instruction
  - Instruction register（指令寄存器）: holds current instruction
- Machine Cycle (repeat these 3 steps)
  - Fetch（取指）
  - Decode（译码）
  - Execute（执行）

Figure 2.8 **The machine cycle**



1. Retrieve the next instruction from memory (as indicated by the program counter) and then increment the program counter.
2. Decode the bit pattern in the instruction register.
3. Perform the action required by the instruction in the instruction register.

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

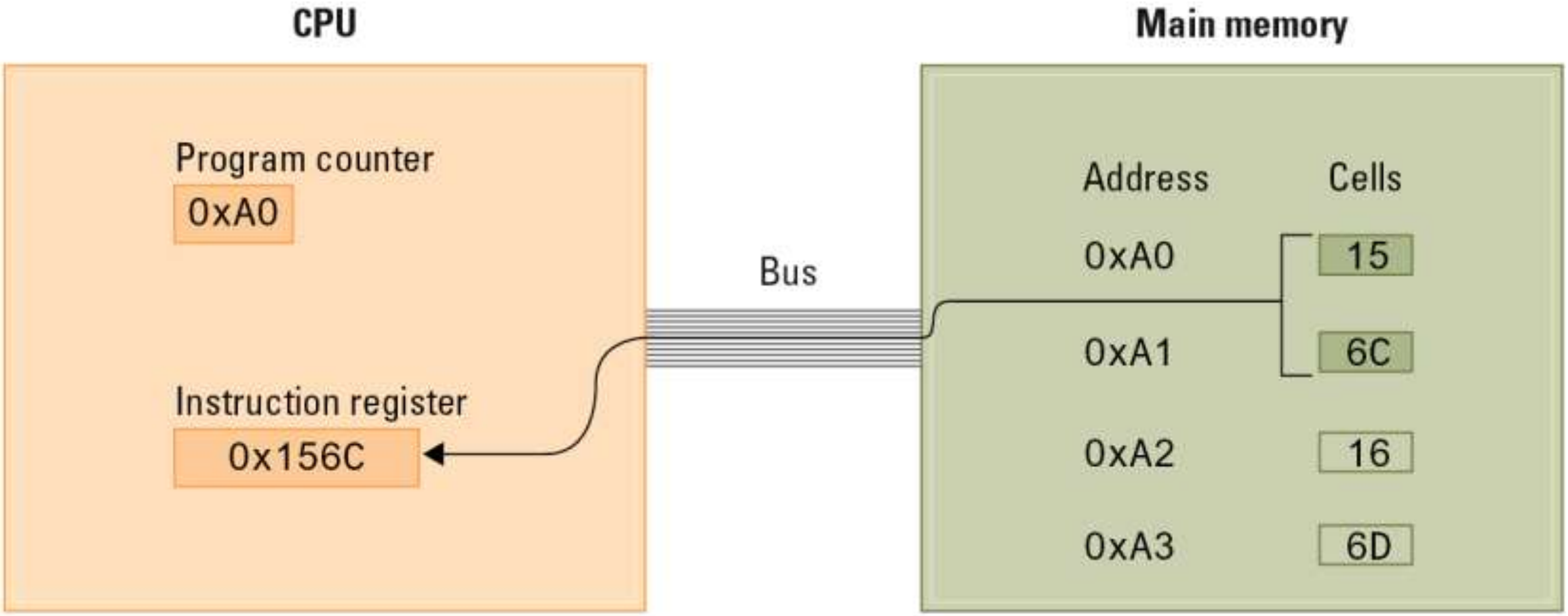


Program counter contains address of first instructions.

Address	Cells
0xA0	0x15
0xA1	0x6C
0xA2	0x16
0xA3	0x6D
0xA4	0x50
0xA5	0x56
0xA6	0x30
0xA7	0x6E
0xA8	0xC0
0xA9	0x00

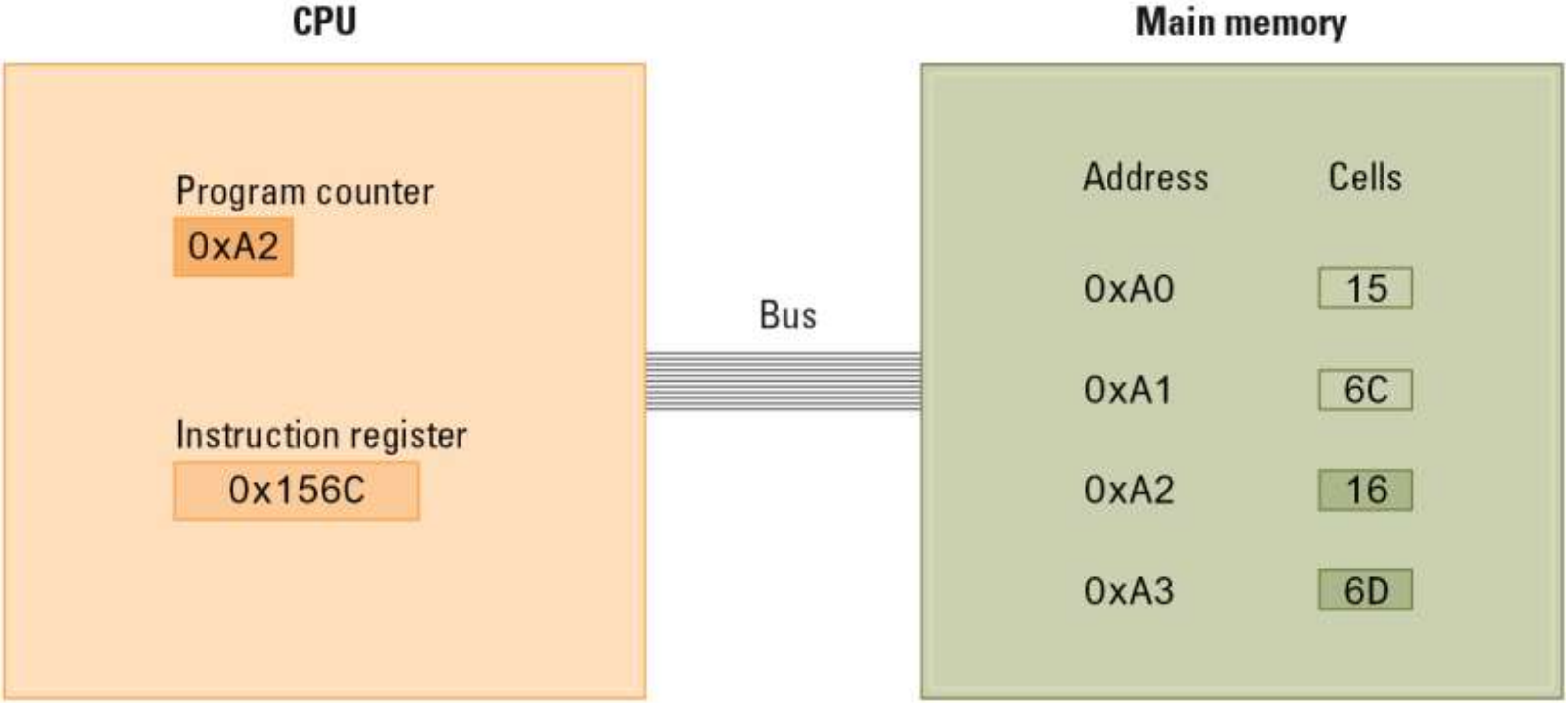
Program is stored in main memory, beginning at address 0xA0.

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 (cont’d)**



Address	Cells
0xA0	15
0xA1	6C
0xA2	16
0xA3	6D

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

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

Figure 2.7 **An encoded version of the instructions in Figure 2.2**

operation	register	address

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.

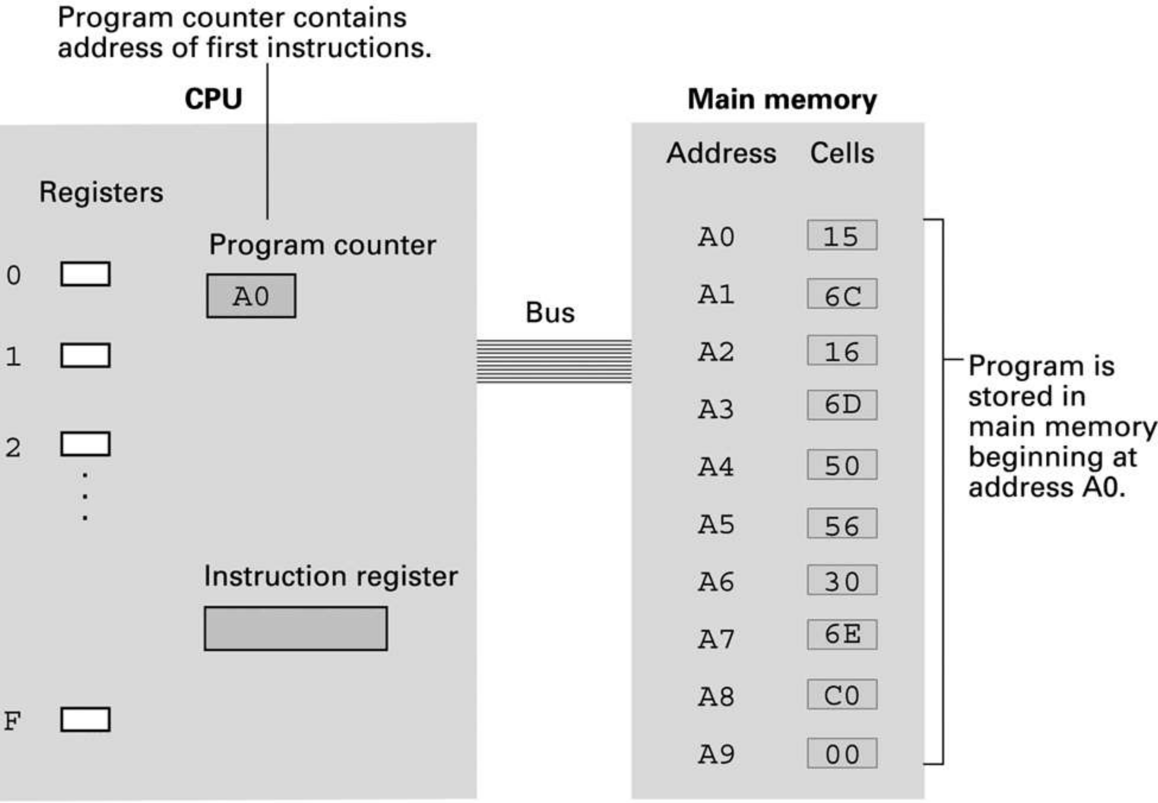
Step 5. Stop.

Encoded instructions	Translation
156C	Load register 5 with the bit pattern found in the memory cell at address 6C.
166D	Load register 6 with the bit pattern found in the memory cell at address 6D.
5056	Add the contents of register 5 and 6 as though they were two's complement representation and leave the result in register 0.
306E	Store the contents of register 0 in the memory cell at address 6E.
C000	Halt.

Operation: 1-load 5-add 3-store

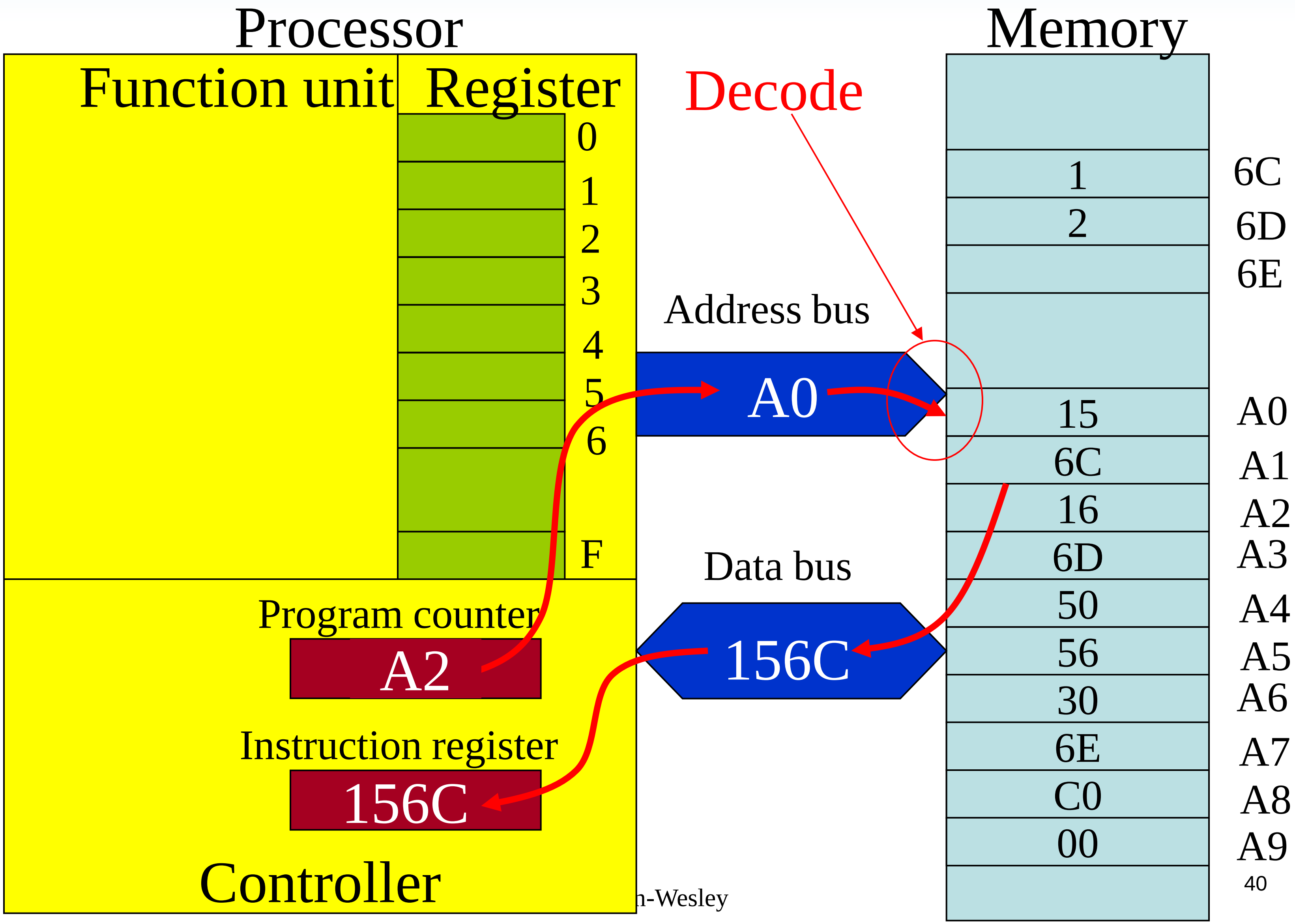
Figure 2.10 **The program from Figure 2.7 stored in main memory ready for execution**

Encoded instructions	Translation
156C	Load register 5 with the bit pattern found in the memory cell at address 6C.
166D	Load register 6 with the bit pattern found in the memory cell at address 6D.
5056	Add the contents of register 5 and 6 as though they were two's complement representation and leave the result in register 0.
306E	Store the contents of register 0 in the memory cell at address 6E.
C000	Halt.


Program counter contains address of first instructions.
CPU
Registers
0
1
2
F
Program counter
A0
Instruction register
Bus
Main memory
Program is stored in main memory beginning at address A0.

Address	Cells
A0	15
A1	6C
A2	16
A3	6D
A4	50
A5	56
A6	30
A7	6E
A8	C0
A9	00

Fetch Instruction 1



Processor

Function unit

Register

0
1
2
3
4
5
6
F

Program counter

A2

Instruction register

156C

Controller

Decode

Address bus

A0

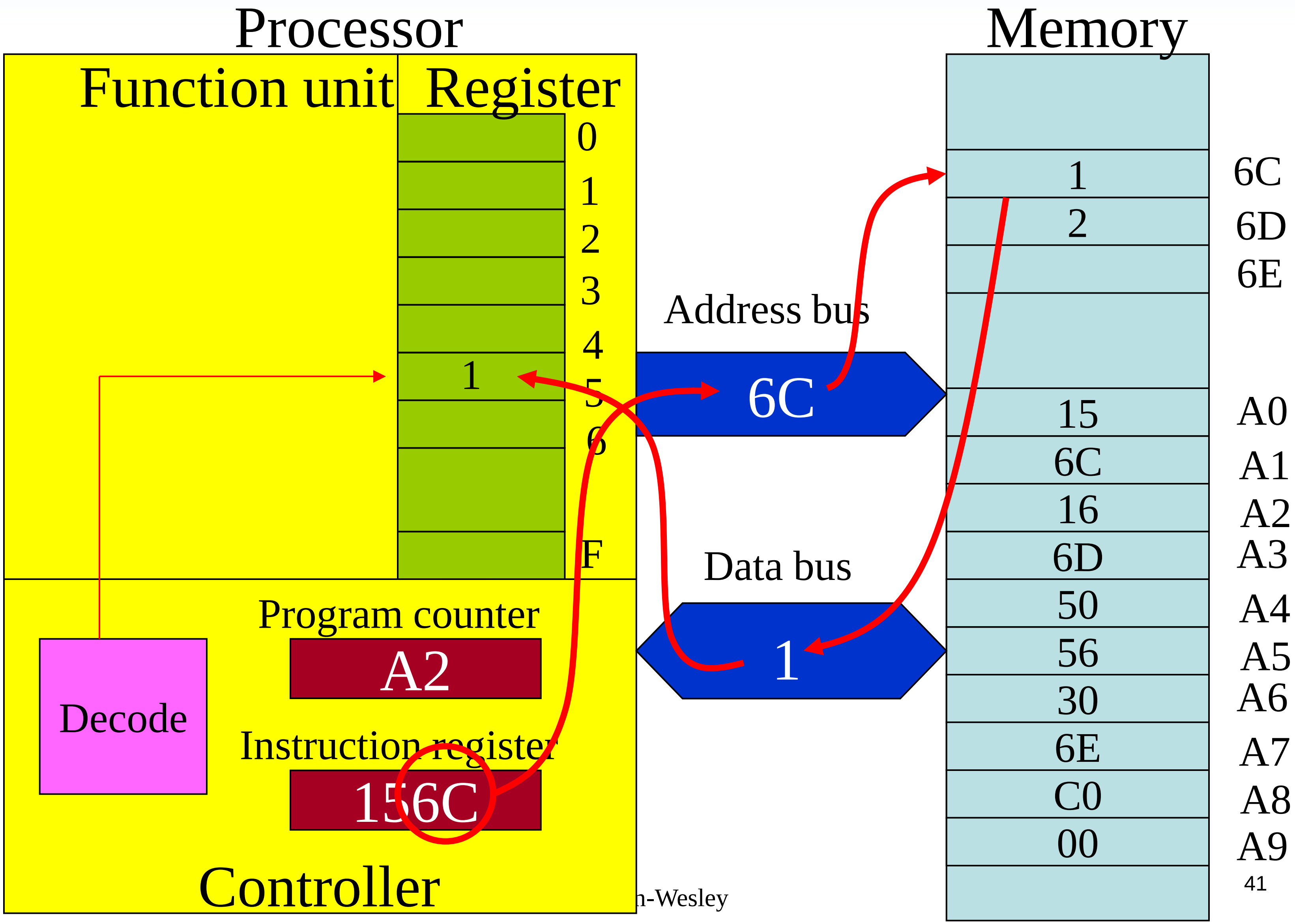
Data bus

156C

Memory

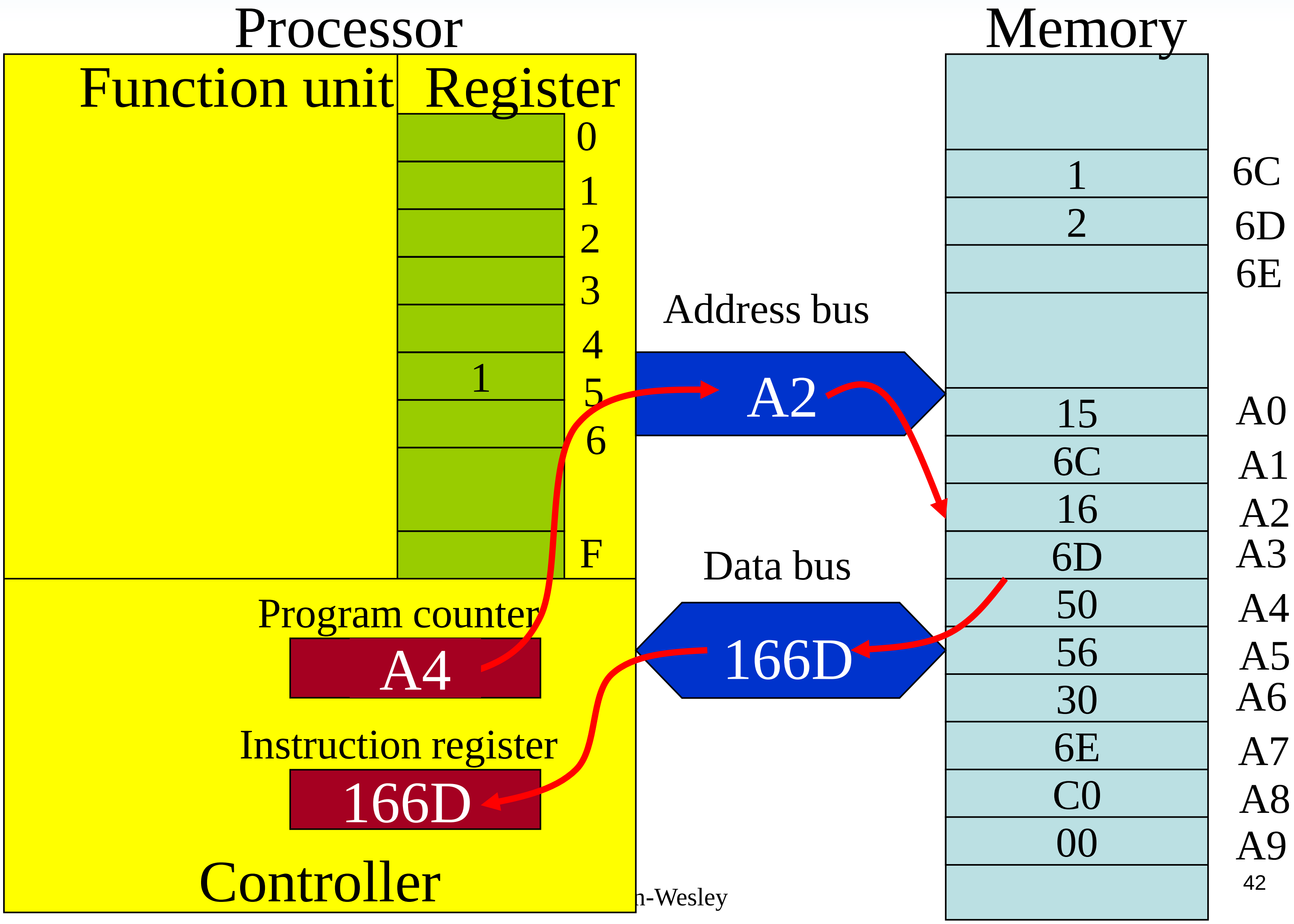
1	6C
2	6D
	6E
15	A0
6C	A1
16	A2
6D	A3
50	A4
56	A5
30	A6
6E	A7
C0	A8
00	A9

Execute Instruction 1

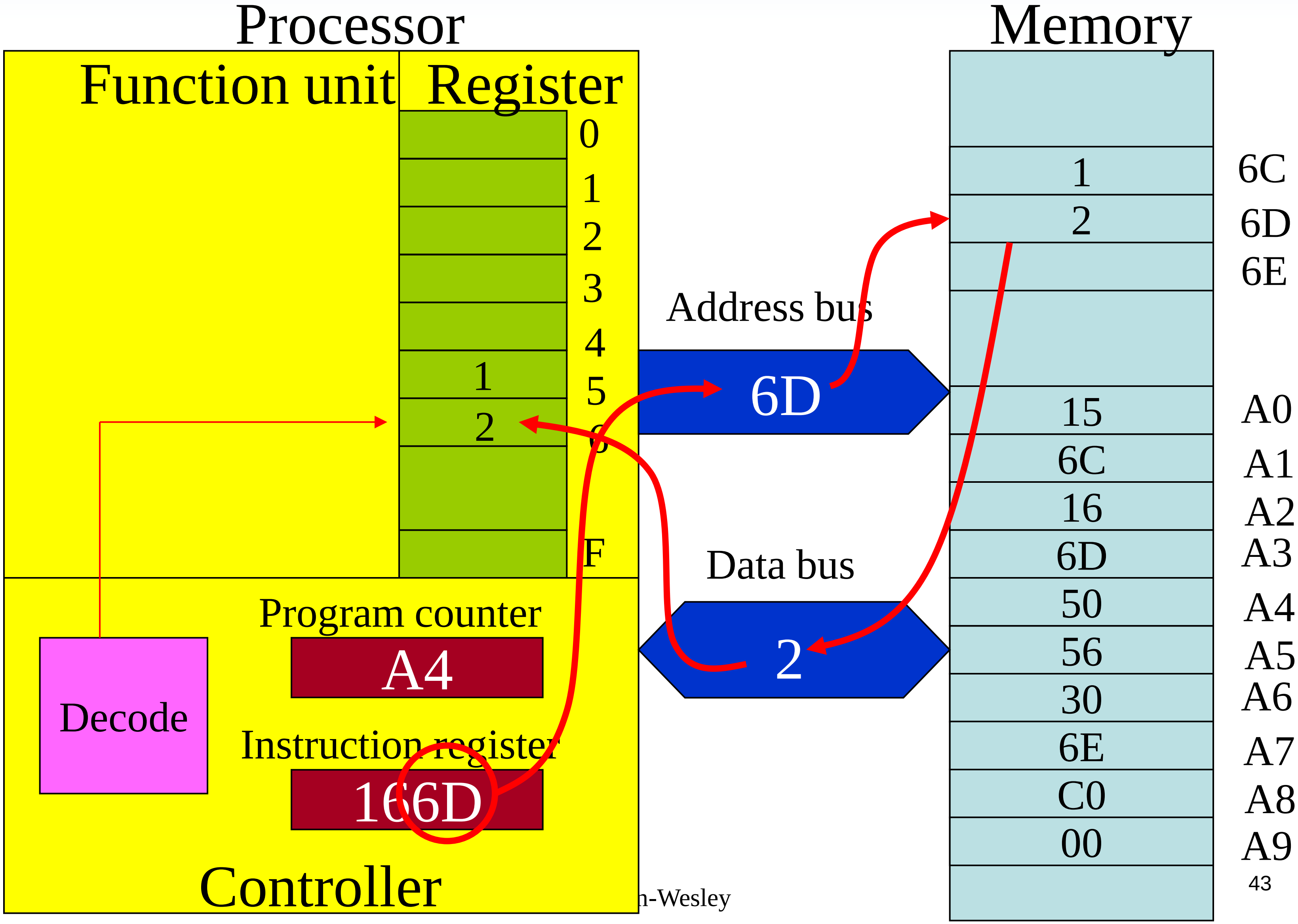


1	6C
2	6D
	6E
15	A0
6C	A1
16	A2
6D	A3
50	A4
56	A5
30	A6
6E	A7
C0	A8
00	A9

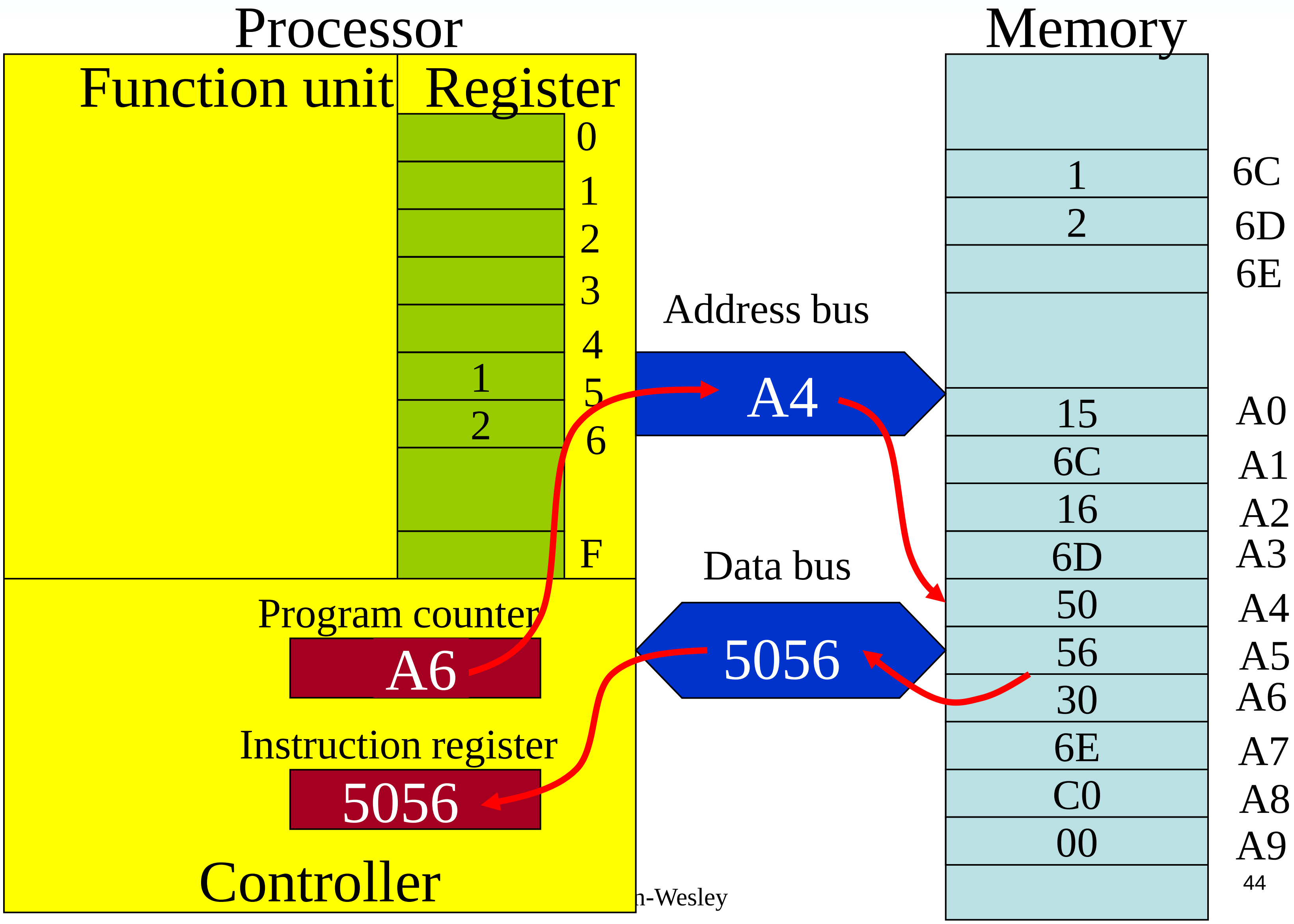
Fetch Instruction 2



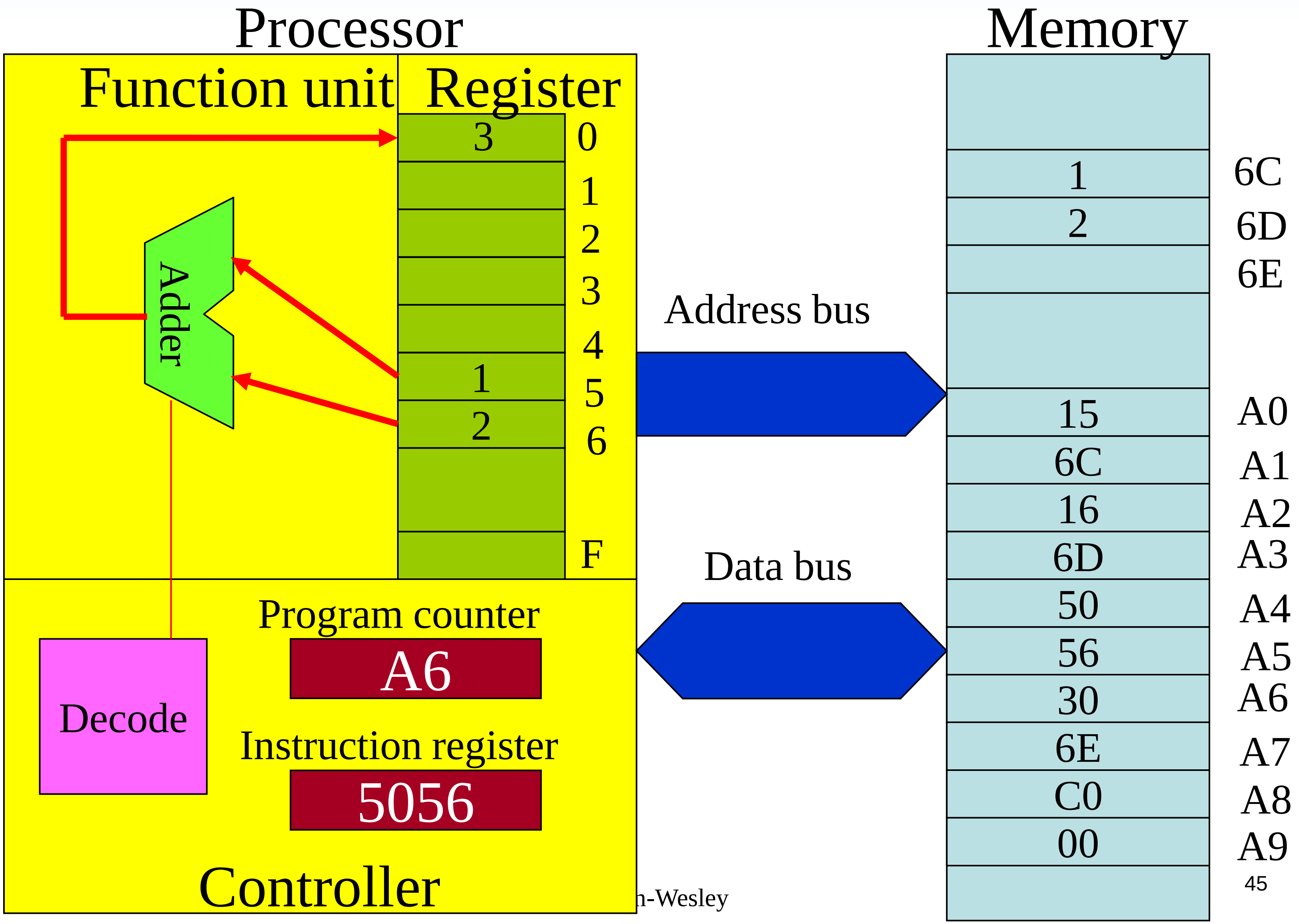
Execute Instruction 2



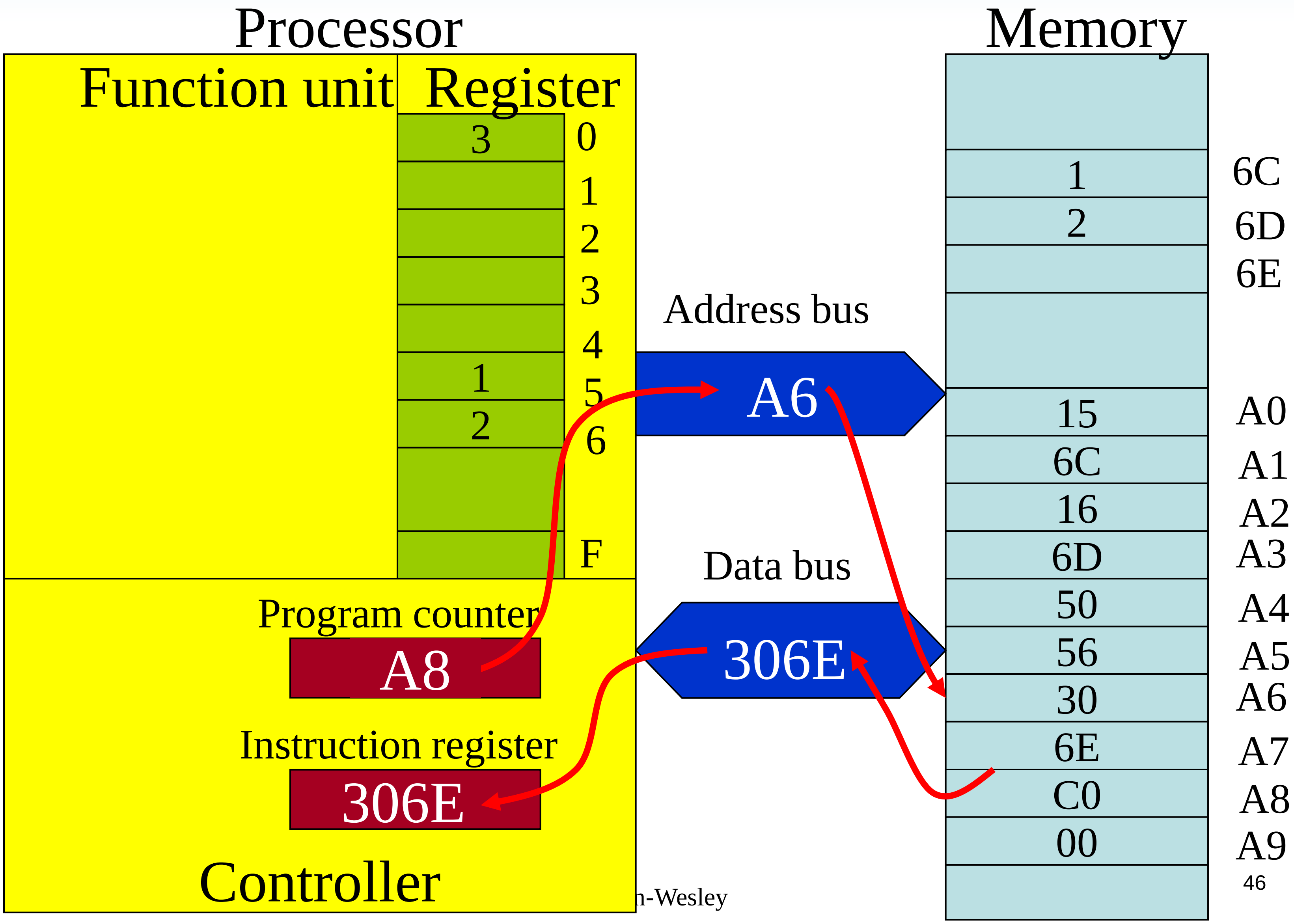
Fetch Instruction 3



Execute Instruction 3



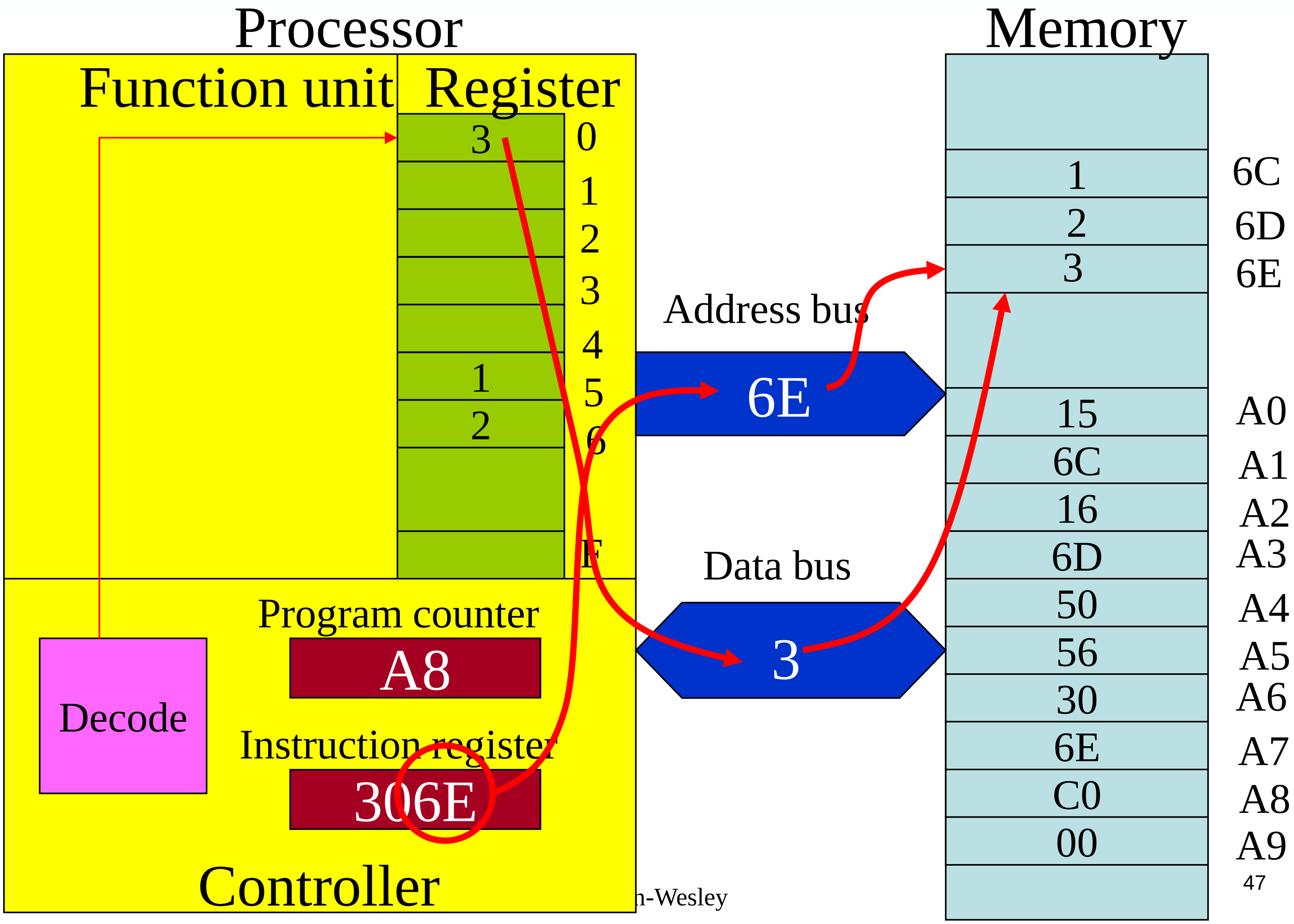
Fetch Instruction 4



Register	Contents
0	3
1	
2	
3	
4	
5	1
6	2
F	

Address	Memory
6C	1
6D	2
6E	
A0	15
A1	6C
A2	16
A3	6D
A4	50
A5	56
A6	30
A7	6E
A8	C0
A9	00

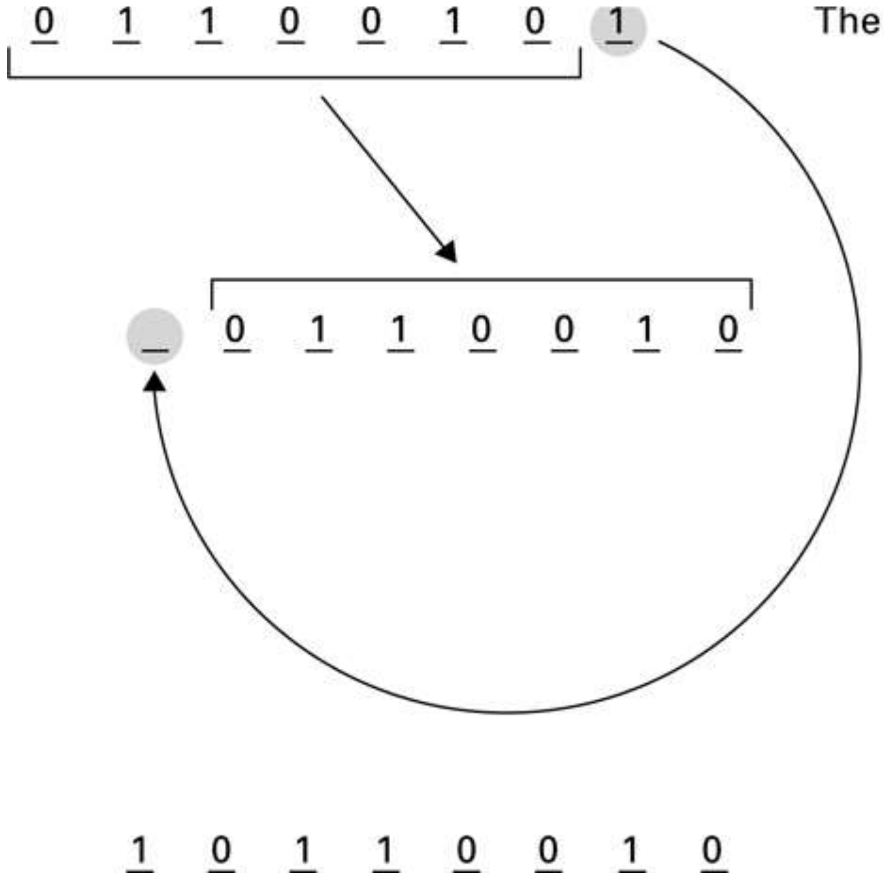
Execute Instruction 4



2.4 Arithmetic/Logic Operations

- Logic: AND, OR, XOR
  - Masking
- Rotate and Shift: circular shift, logical shift, arithmetic shift
- Arithmetic: add, subtract, multiply, divide
  - Precise action depends on how the values are encoded (two's complement versus floating-point).

Figure 2.12 **Rotating the bit pattern 65 (hexadecimal) one bit to the right**

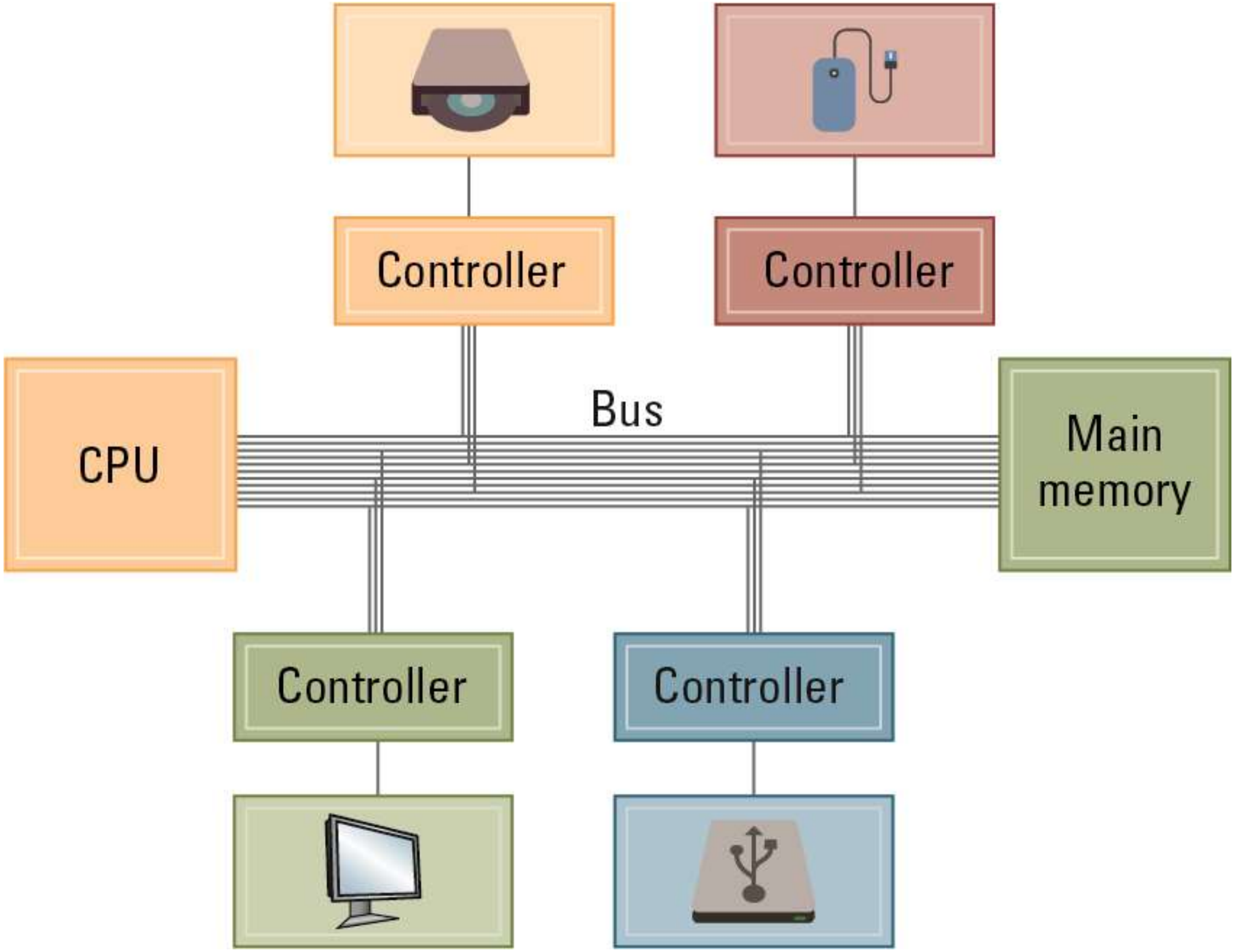


The original bit pattern

The bits move one position to the right. The rightmost bit "falls off" the end and is placed in the hole at the other end.

The final bit pattern

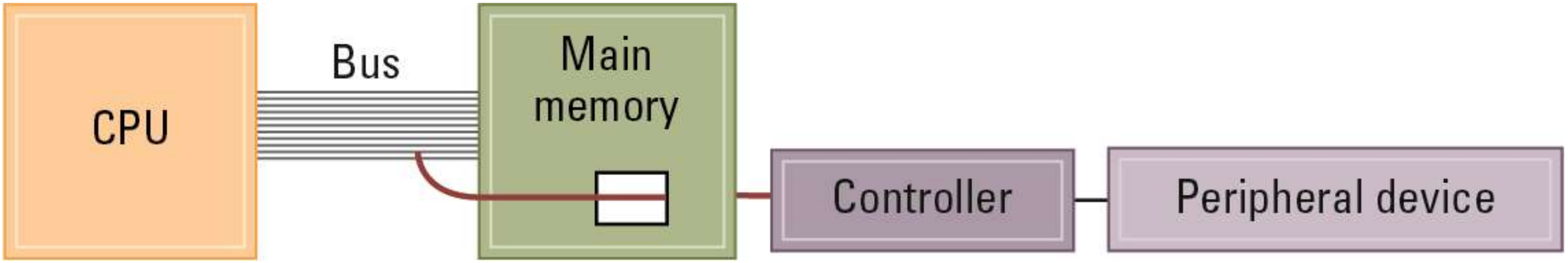
2.5 CPU communicates with Other Devices



CPU communicates with Other Devices

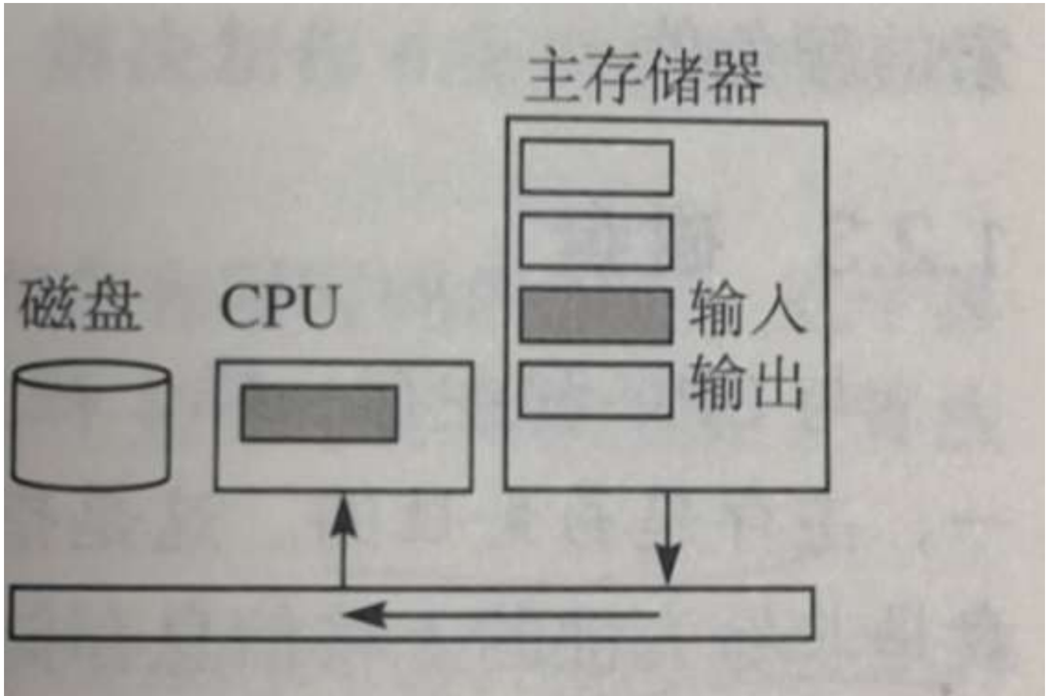
- **Controller:** handles communication between the computer and other devices
  - Specialized controllers for each type of device
  - General purpose controllers (USB and HDMI)
- **Port:** The point at which a device connects to a computer
- **Memory-mapped I/O**内存映射I/O **:** devices appear to the CPU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

Figure 2.14 **A conceptual representation of memory-mapped I/O 内存映射I/O**



Example: keyboard

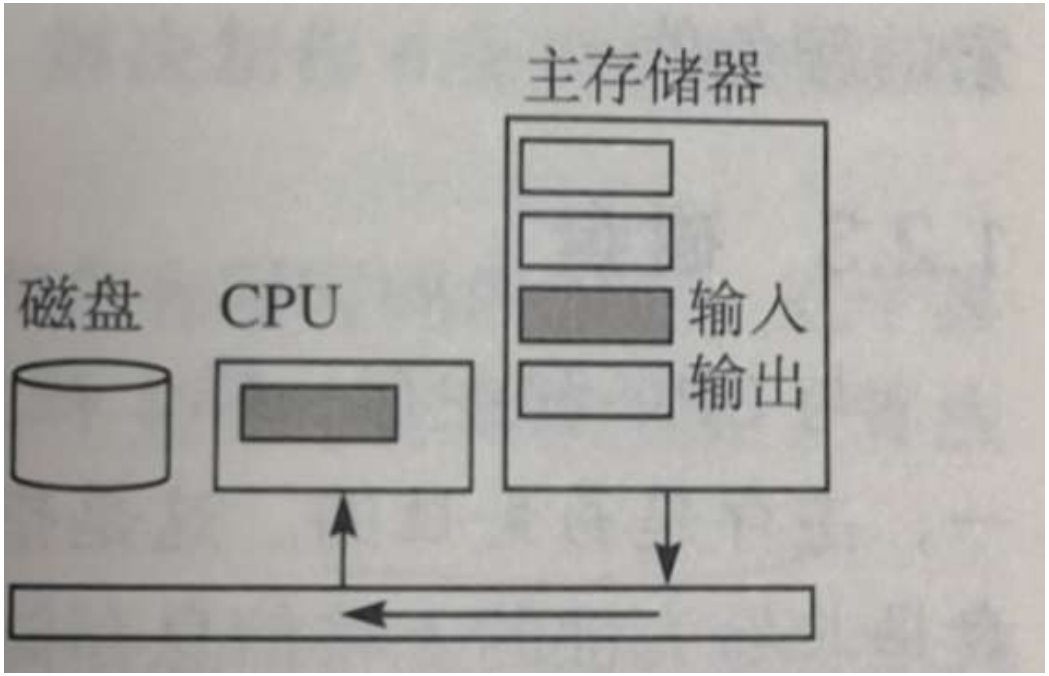
按下键→检测被按哪个键→传送按键信息给主存对应输入单元→键盘发信号给CPU表明有键按下→CPU响应从输入单元读入信息



2 从键盘接收一个字符的信息流

Example: keyboard & mouse

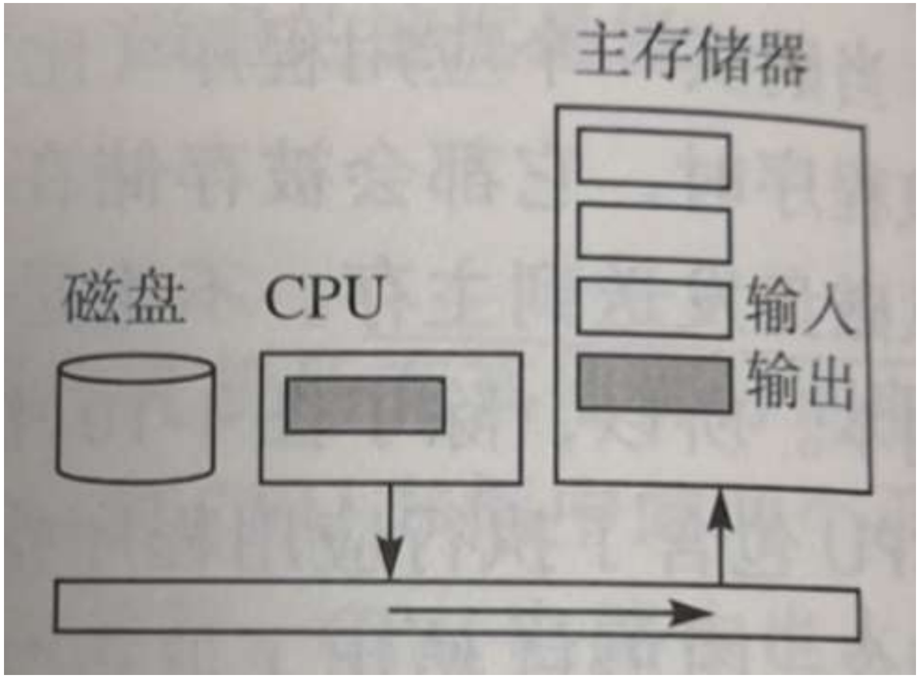
按下键→检测被按哪个键→传送按键信息给主存对应输入单元→键盘发信号给CPU表明有键按下→CPU响应从输入单元读入信息



鼠标处理器传递方向和速度

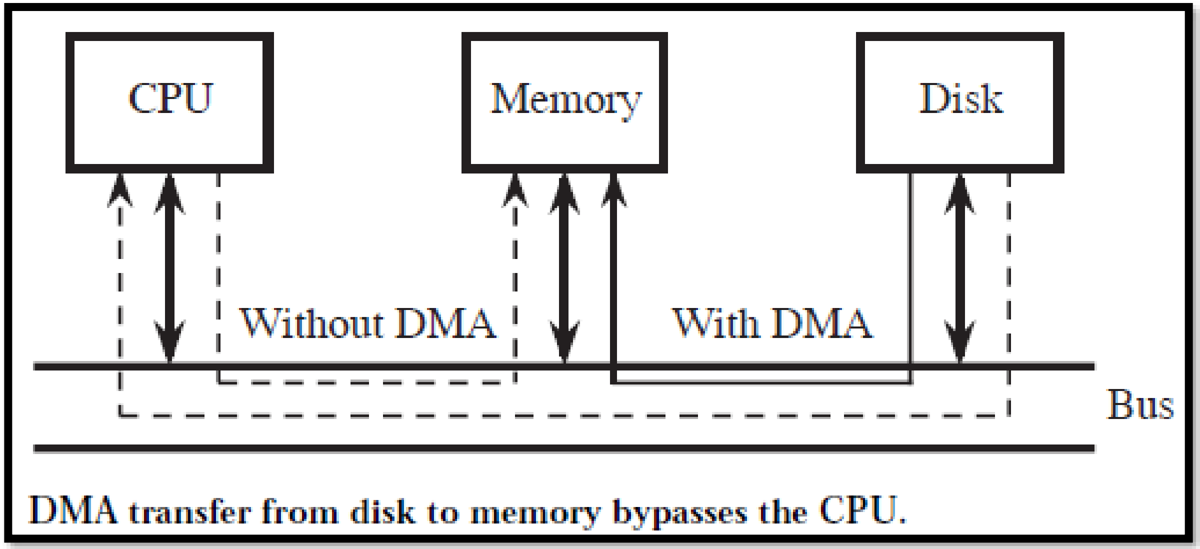
Example: monitor

- 像素显示
- 主存显示输出不直接连到每个像素
- 连接独立的专用处理器的I/O模块



Communicating with disk

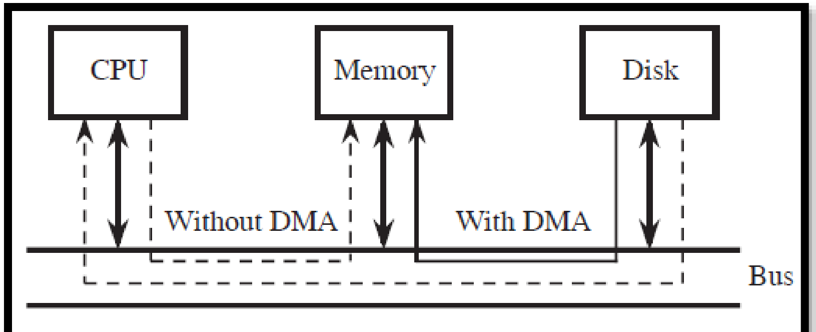
- **Von Neumann Bottleneck:** occurs when the CPU and controllers compete for bus access
- **Direct memory access (DMA直接内存访问):** Main memory access by a controller over the bus
- **Handshaking:** The process of coordinating the transfer of data between components



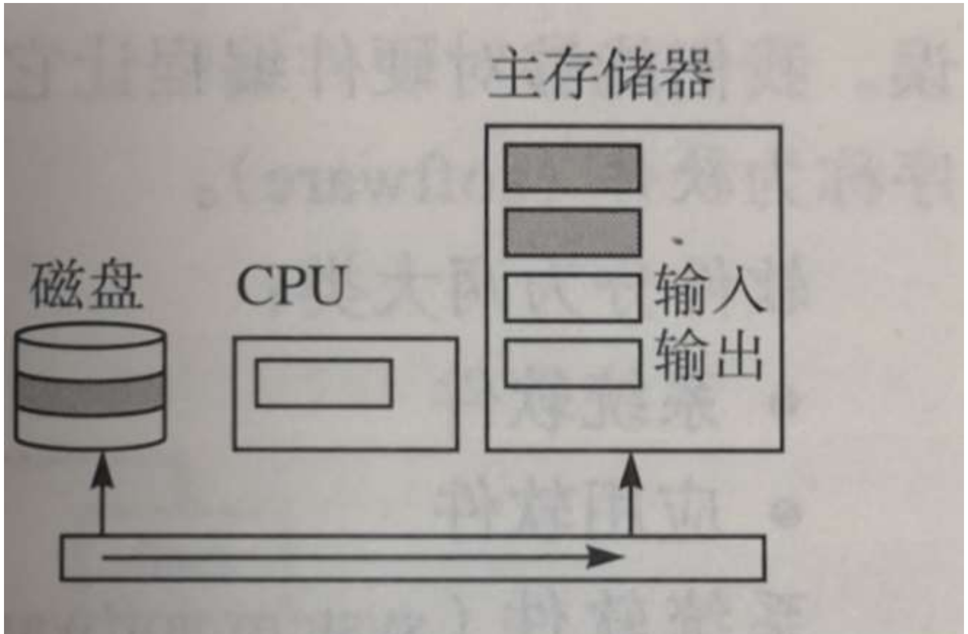
DMA transfer from disk to memory bypasses the CPU.

Communicating with disk

- CPU向DMA控制器发送信号启动传输请求
- 当CPU处理其他事务时，DMA控制器在总线传输信息
- 传输结束后向CPU发送信号



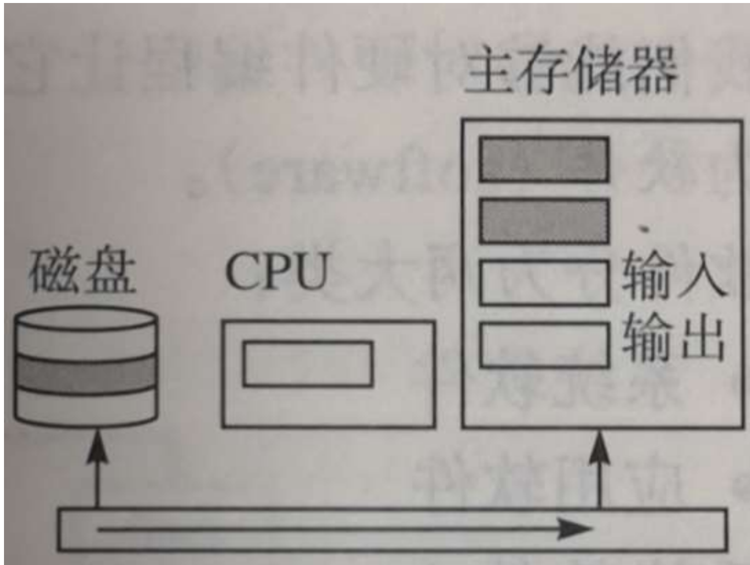
DMA transfer from disk to memory bypasses the CPU.



4　磁盘与主存间的直接内存访问

Example: word processing

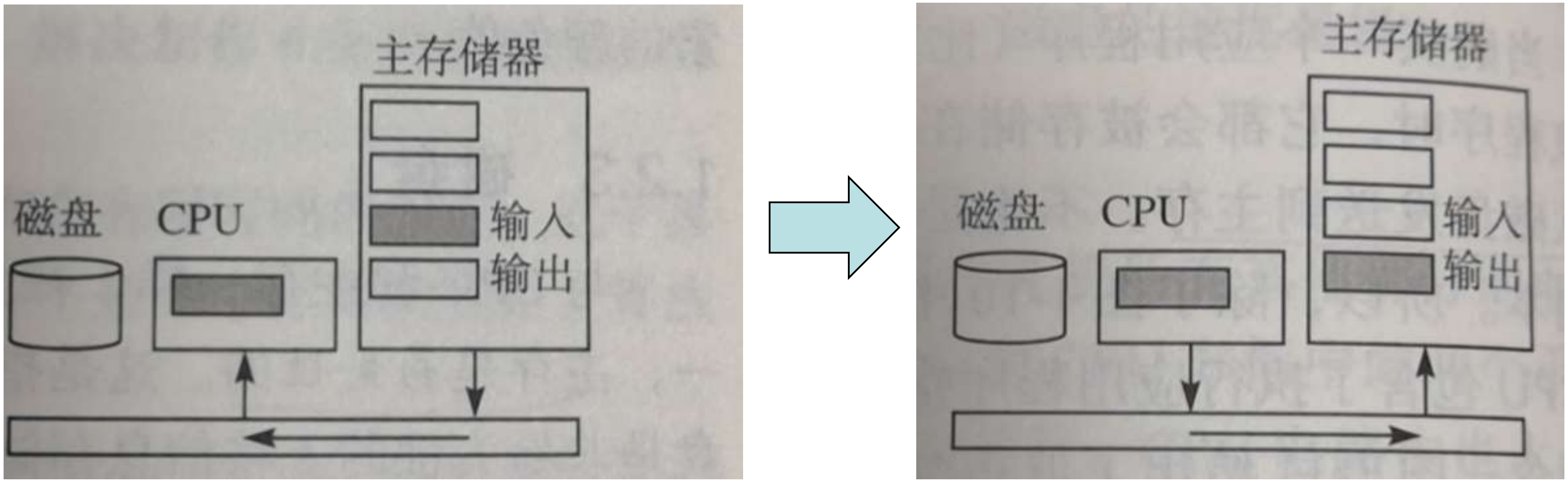
- 文字处理应用程序启动
- 系统用DMA传输，将其副本从磁盘发送到内存
- 打开文档时系统完成同样传输，将文档送入内存



4　磁盘与主存间的直接内存访问

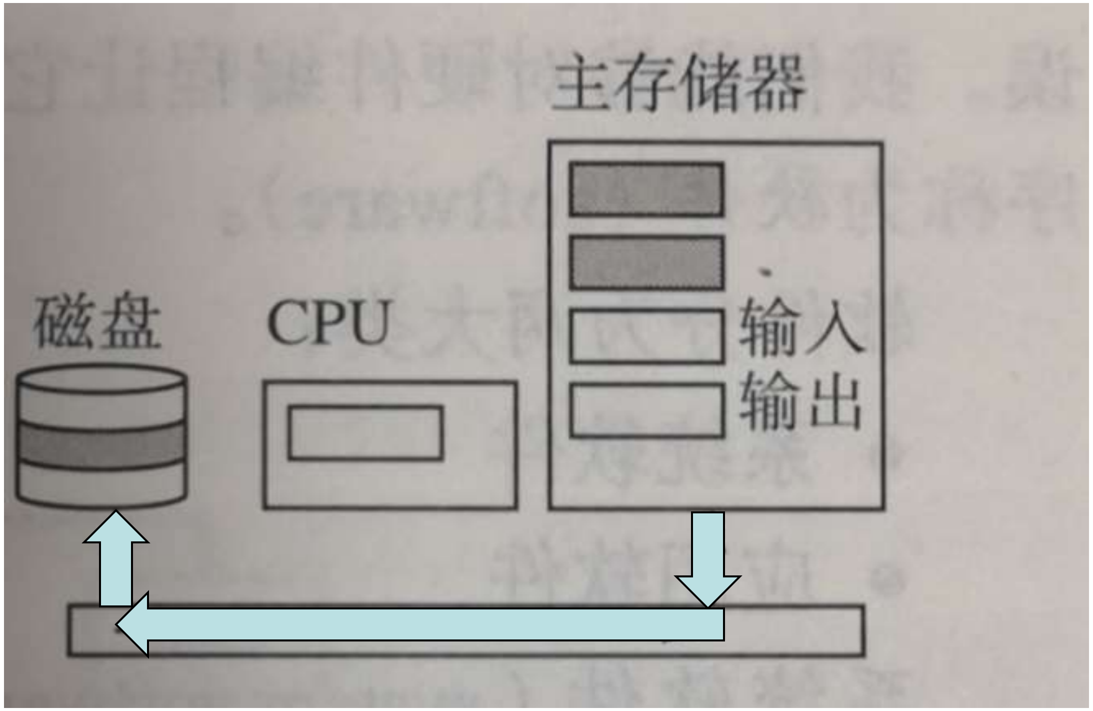
Example: word processing

- 敲键并显示



Example: word processing

- 保存文档时， 系统通过DMA传输将文档从内存传输到磁盘



Communicating with Other Devices (continued)

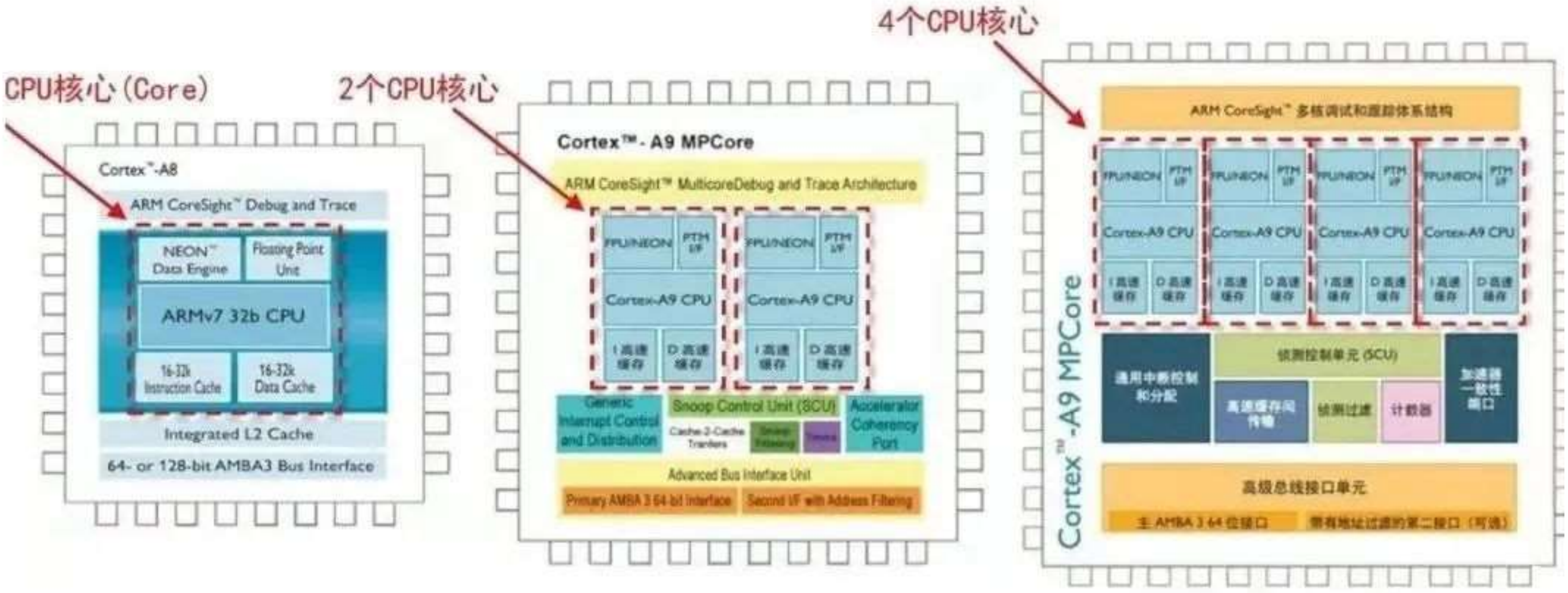
- **Popular Communication Media**
  - **Parallel Communication:** Several signals transferred at the same time, each on a separate “line” (computer’s internal bus)
  - **Serial Communication:** Signals are transferred one after the other over a single “line” (USB, FireWire)

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

The Multi-Core CPU

- 多核CPU是一种包含多个处理器核心（CPU核心）在单个集成电路芯片（或芯片组）中的中央处理器。每个核心基本上是一个独立的处理器，具有自己的执行单元、寄存器和缓存，能够独立执行或解码指令。
- 设计目的是通过并行处理来提高处理能力和效率



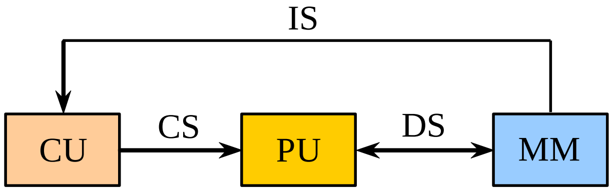
(a) ARM 单核心CPU (b) ARM 2核心CPU (c) ARM 4核心CPU

SoC (system-on-a-chip)

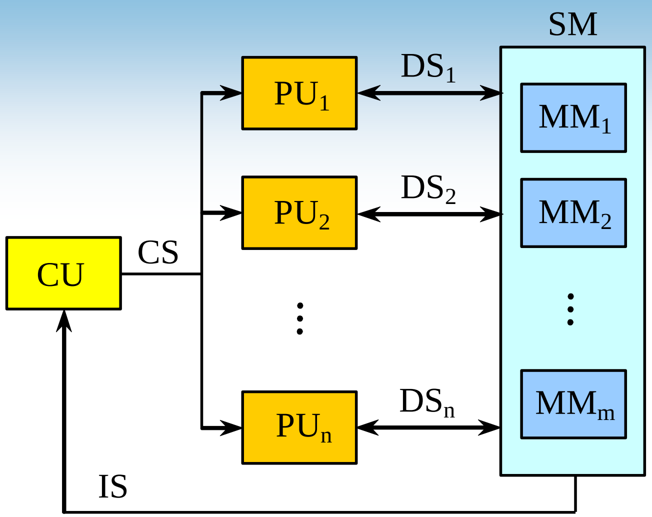
- SoC：称为系统级芯片，也有称片上系统，意指它是一个产品，是一个有专用目标的集成电路，其中包含完整系统并有嵌入软件的全部内容。
- 将计算机或其他电子系统的所有必要组件集成到单个芯片上的一种微芯片
- 计算设备变得更小、更便宜、更快和更低功耗。可用于智能手机、平板电脑、物联网设备、路由器、相机等。
- 可包含：微控制器、微处理器、数字信号处理器、内存功能、用于有线通信协议的外部接口（HDMI，USB）、无线功能（如WiFi或蓝牙）、图形处理器

2.6 Other Architectures

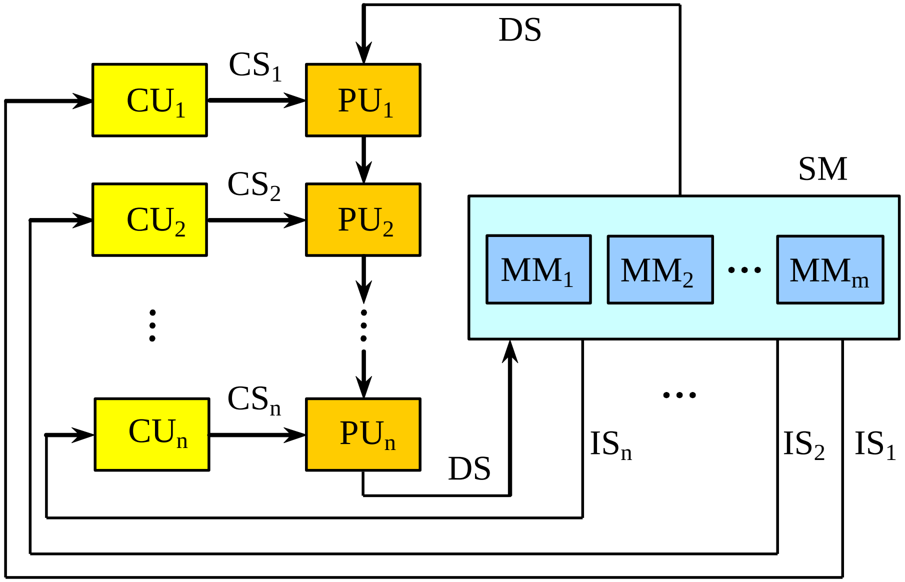
- Technologies to increase throughput:
  - Pipelining: Overlap steps of the machine cycle
  - Parallel Processing: Use multiple processors simultaneously
    - SISD: No parallel processing
    - MIMD: Different programs, different data
    - SIMD: Same program, different data



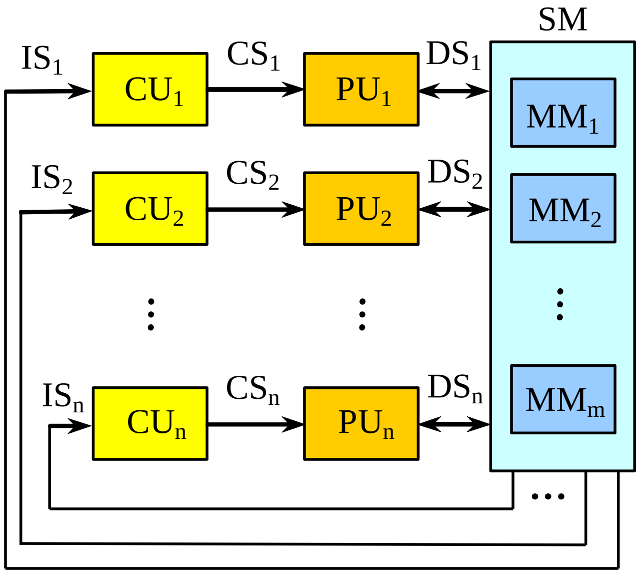
（a）SISD 计算机



（b）SIMD 计算机



（c）MISD 计算机



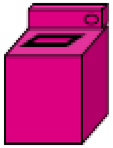
（d）MIMD 计算机

- 单指令流单数据流机器（SISD）
  - SISD机器是一种传统的串行计算机，它的硬件不支持任何形式的并行计算，所有的指令都是串行执行。并且在某个时钟周期内，CPU只能处理一个数据流，早期的计算机都是SISD机器
- 单指令流多数据流机器（SIMD）
  - SIMD是采用一个指令流处理多个数据流，我们现在用的单核计算机基本上都属于SIMD机器。
- 多指令流单数据流机器（MISD）
  - MISD是采用多个指令流来处理单个数据流。由于实际情况中，采用多指令流处理多数据流才是更有效的方法，因此MISD只是作为理论模型出现，没有投入到实际应用之中。
- 多指令流多数据流机器（MIMD）
  - 同时执行多个指令流，这些指令流分别对不同数据进行操作，比如多核计算平台。

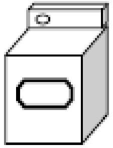
Pipeline

– Laundry Example

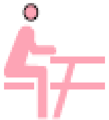
- A, B, C, D each have one load of clothes to wash, dry, and fold



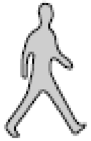
- Wash takes 30 minutes



- Dryer takes 30 minutes

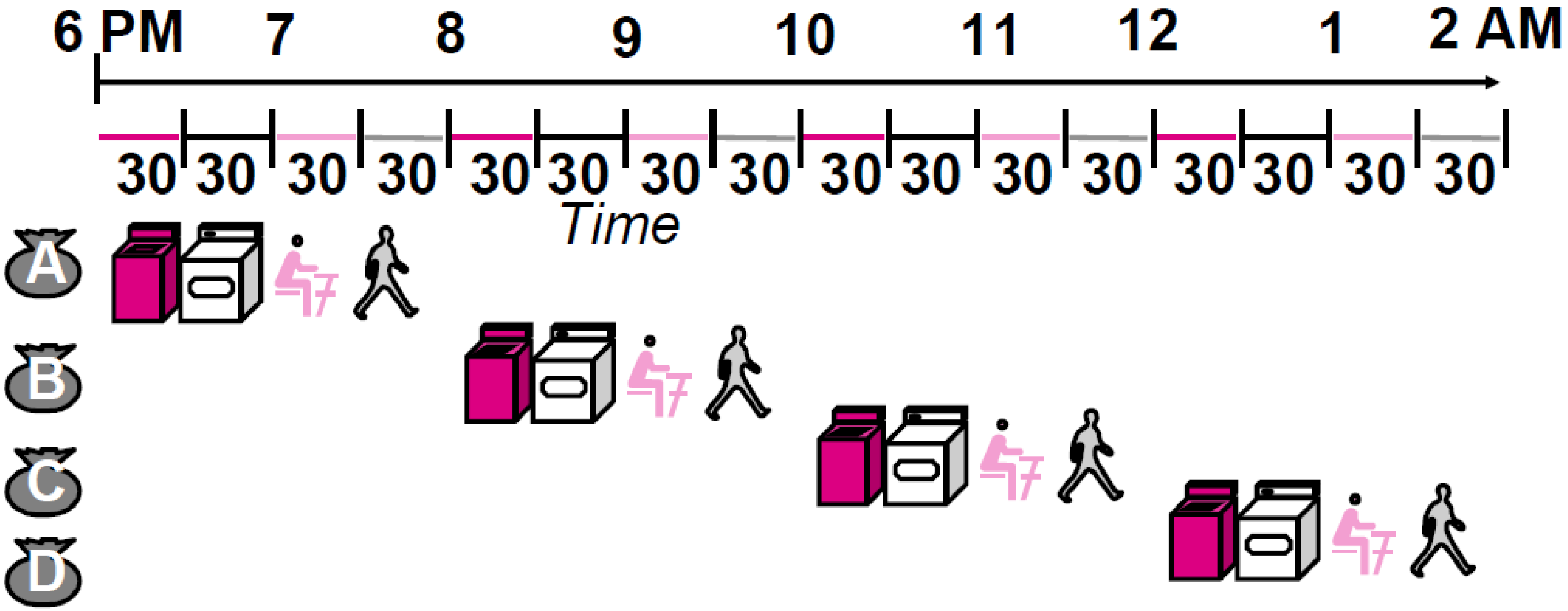


- “Folder” takes 30 minutes



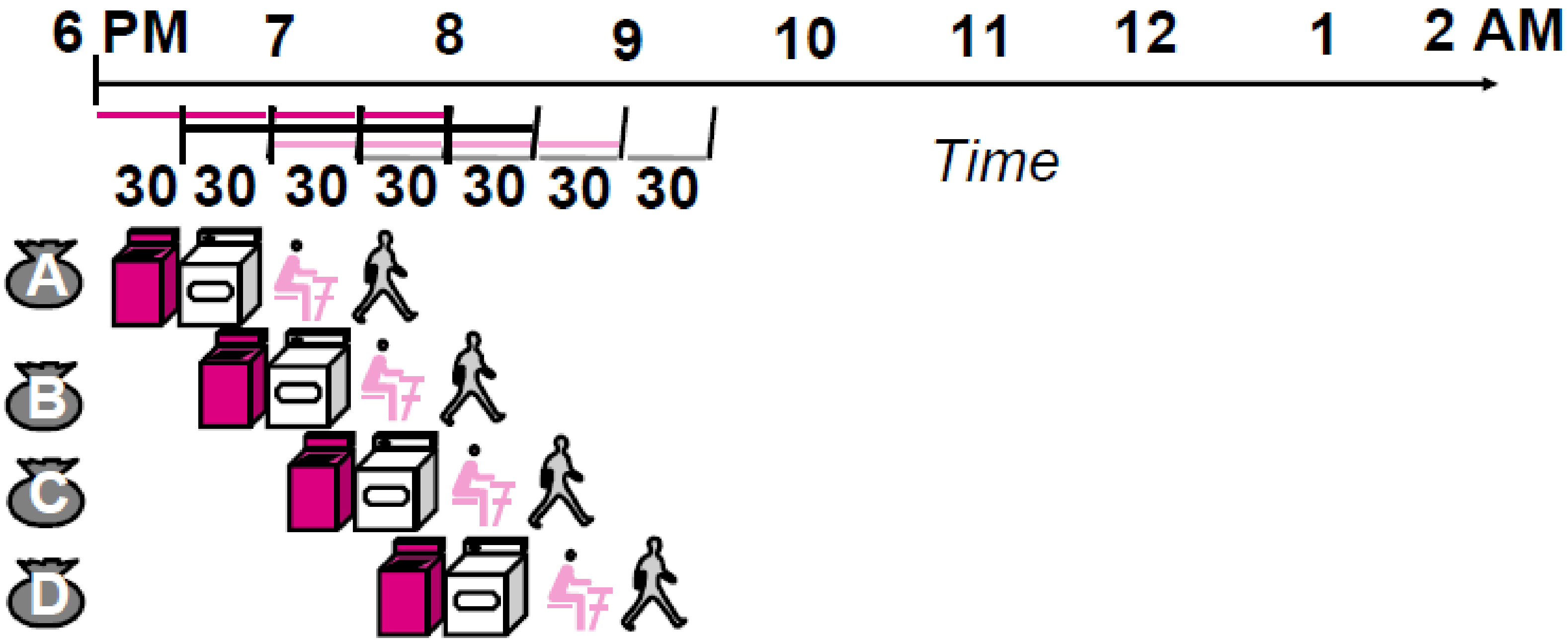
- “Stasher” takes 30 minutes to put clothes into drawers

Pipeline

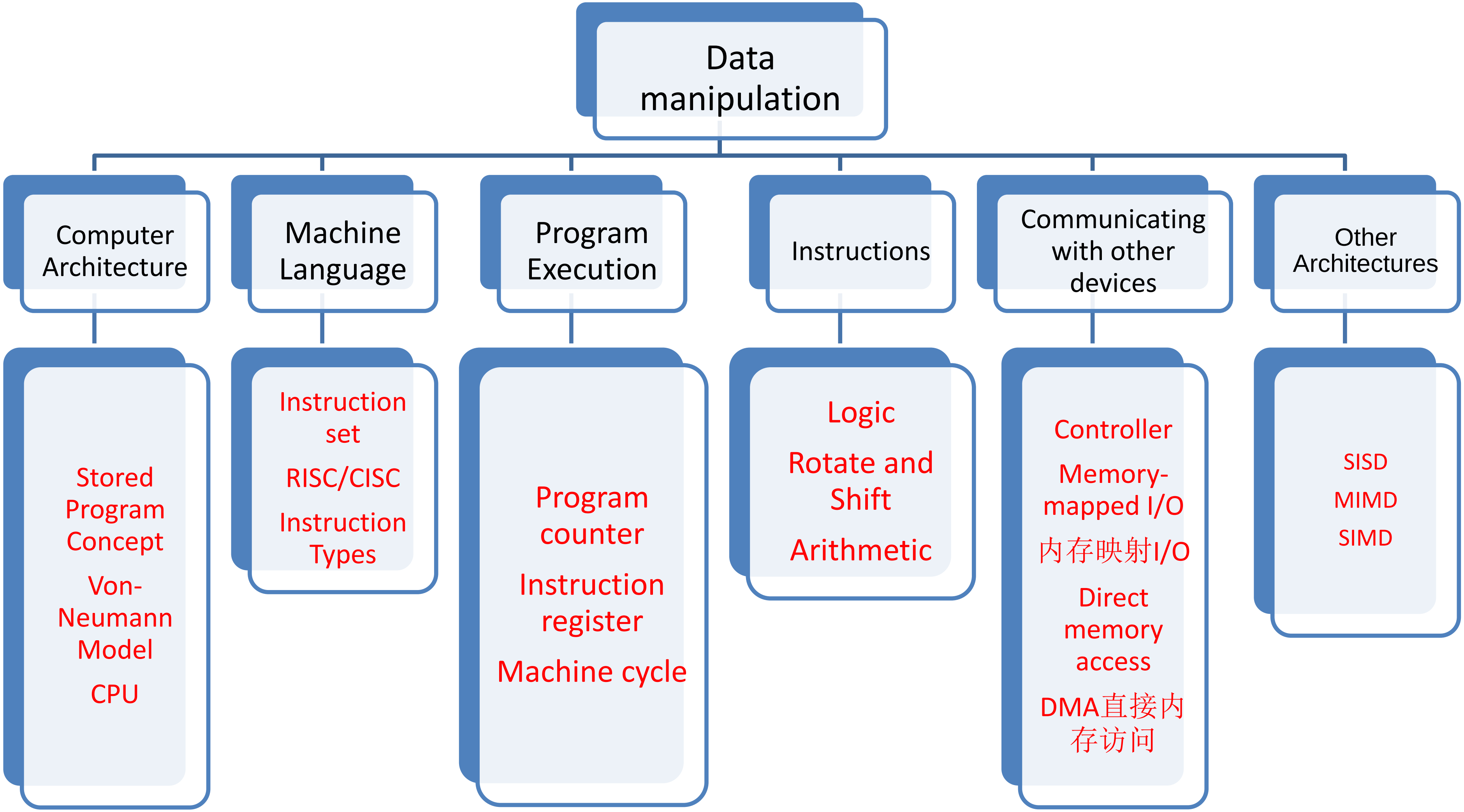


- Sequential laundry takes 8 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipeline



- Pipelined laundry takes 3.5 hours for 4 loads!



Data manipulation

- Computer Architecture
  - Stored Program Concept
  - Von-Neumann Model
  - CPU
- Machine Language
  - Instruction set
  - RISC/CISC
  - Instruction Types
- Program Execution
  - Program counter
  - Instruction register
  - Machine cycle
- Instructions
  - Logic
  - Rotate and Shift
  - Arithmetic
- Communicating with other devices
  - Controller
  - Memory-mapped I/O
  - 内存映射I/O
  - Direct memory access
  - DMA直接内存访问
- Other Architectures
  - SISD
  - MIMD
  - SIMD