# Chapter 4

# Computer Organization

# Program Specification

Source

```
int a, b, c, d;
. . .
a = b + c;
d = a - 100;
```

Assembly Language

```
; Code for a = b + c
    load      R3,b
    load      R4,c
    add       R3,R4
    store     R3,a

; Code for d = a - 100
    load      R4,=100
    subtract  R3,R4
    store     R3,d
```

# Machine Language

## Assembly Language

```
; Code for a = b + c
    load      R3,b
    load      R4,c
    add       R3,R4
    store     R3,a

; Code for d = a - 100
    load      R4,=100
    subtract  R3,R4
    store     R3,d
```

## Machine Language

```
10111001001100…1
10111001010000…0
10100111001100…0
10111010001100…1
10111001010000…0
10100110001100…0
10111001101100…1
```

# Von Neumann Concept

- Stored program concept
- General purpose computational device driven by internally stored program
- Data and instructions look same i.e. binary
- Operation being executed determined by HOW we look at the sequence of bits
    - Fetch
    - Decode
    - Execute

View bits as instruction

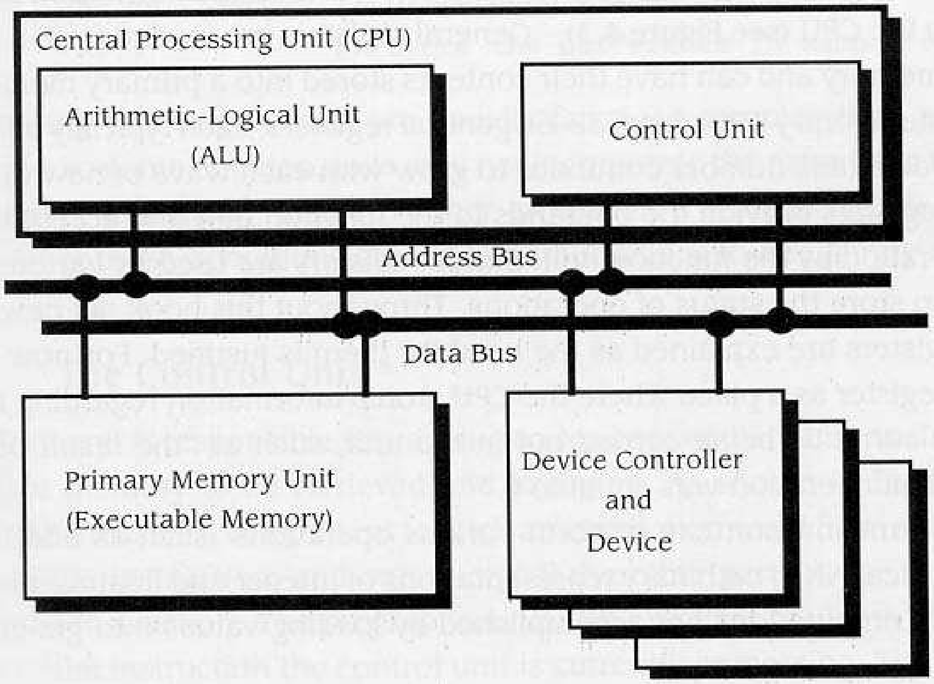
**<u>Data</u>** might be fetched as a result of execution

# Von Neumann Architecture

- CPU
  - ALU
  - Control Unit
- I/O Buses
- Memory Unit
- Devices

**FIGURE 4.2**

**The von Neumann Machine Architecture**



# Von Neumann Machine Architecture

**CPU = ALU + Cntrl Unit**

ALU

- Functional Unit
  - + Instruction set
  - + Arithmetic & Logic
- Registers
  - + Intermediate storage

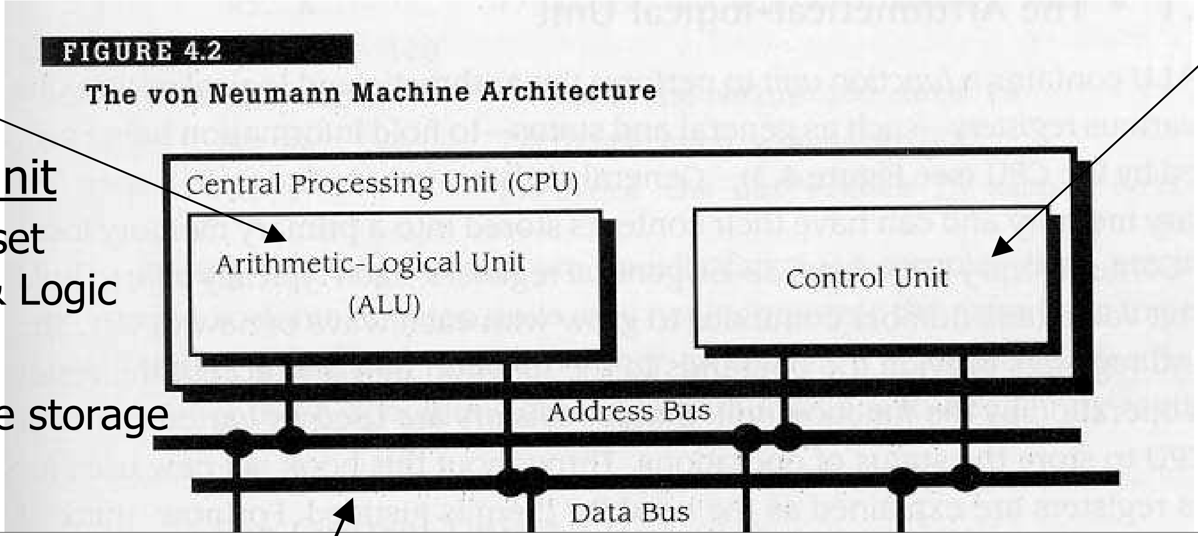


**FIGURE 4.2**

**The von Neumann Machine Architecture**

**Cntrl Unit**

- fetch
- decode
- execute

⇳ ALU

**Buses**

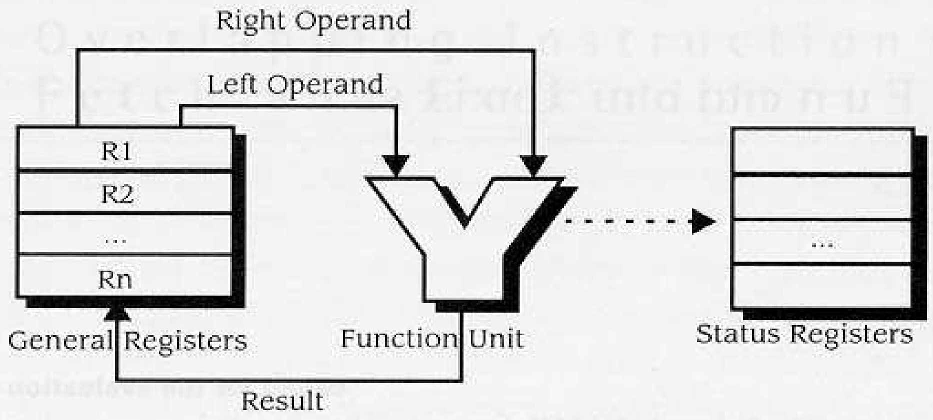
Address Bus / Data Bus wires over which Instr / data is transferred from memory to ALU

**Von Neumann Bottleneck**

# CPU: **ALU** Component

**FIGURE 4.3**

**A Generic Arithmetical-logical Unit**



- Assumes instruction format: OP code, LHO, RHO
  - Instruction / data fetched & placed in register
  - Instruction / data retrieved by functional unit & executed
  - Results placed back in registers

- Control Unit sequences the operations

# Program Specification (revisited)

Source

```
int a, b, c, d;
. . .
a = b + c;
d = a - 100;
```

Assembly Language

```
; Code for a = b + c
    load      R3,b
    load      R4,c
    add       R3,R4
    store     R3,a

; Code for d = a - 100
    load      R4,=100
    subtract  R3,R4
    store     R3,d
```

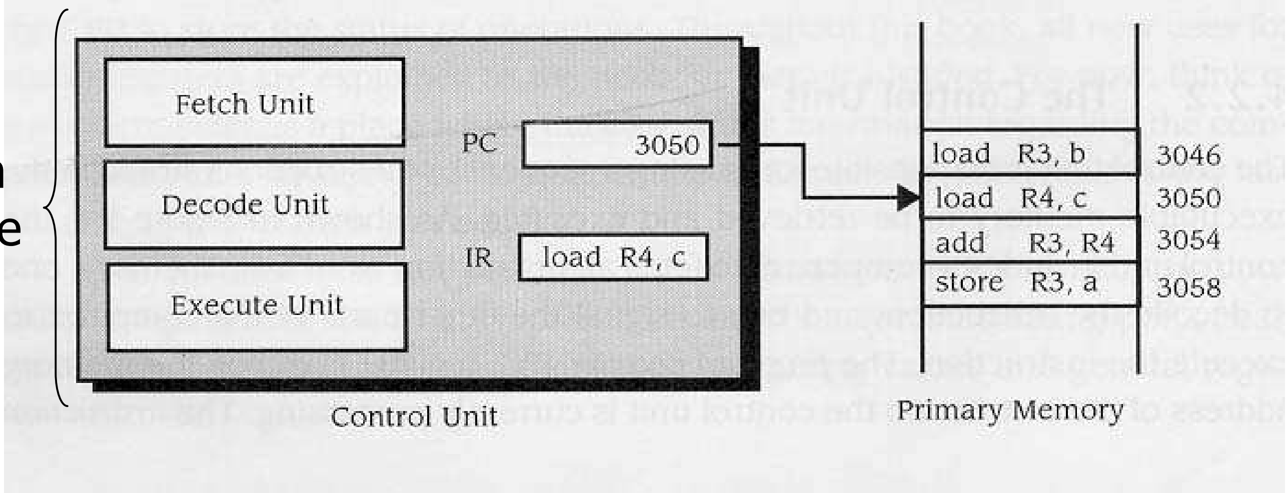
# CPU: **Control Unit** Component

PC => Program Counter
IR => Instruction Register

**FIGURE 4.4**

**The PC, IR, and Memory**

Von Neumann Execution Cycle



- Fetch Unit
  - Get instruction at location pointed to by PC and place in IR
- Decode Unit
  - Determine which instruction & signal hardware that implements it
- Execute Unit
  - Hardware for instruction execution (could cause more data fetches)

# Fetch – Execute cycle

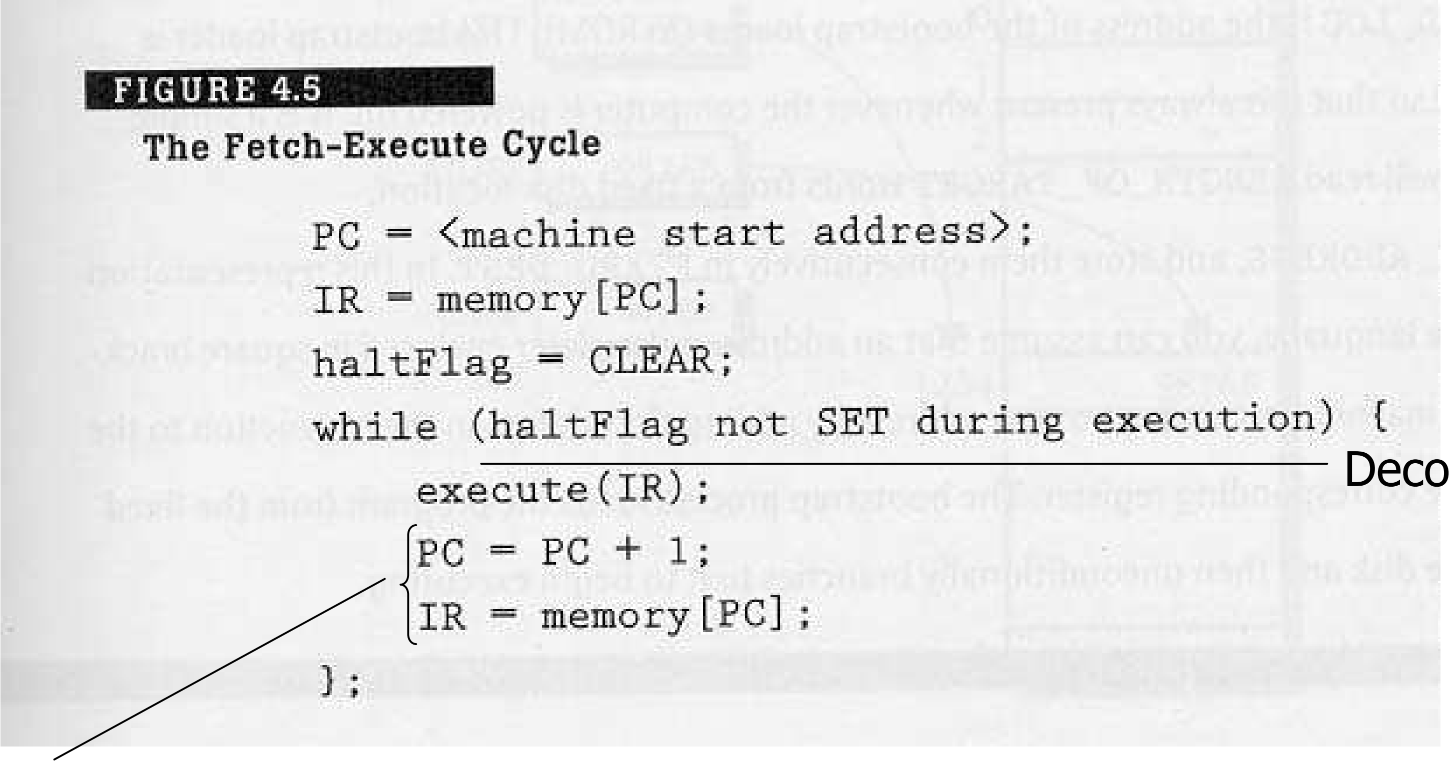


**FIGURE 4.5**

**The Fetch-Execute Cycle**

```
PC = <machine start address>;
IR = memory[PC];
haltFlag = CLEAR;
while (haltFlag not SET during execution) {
    execute(IR);
    PC = PC + 1;
    IR = memory[PC];
};
```

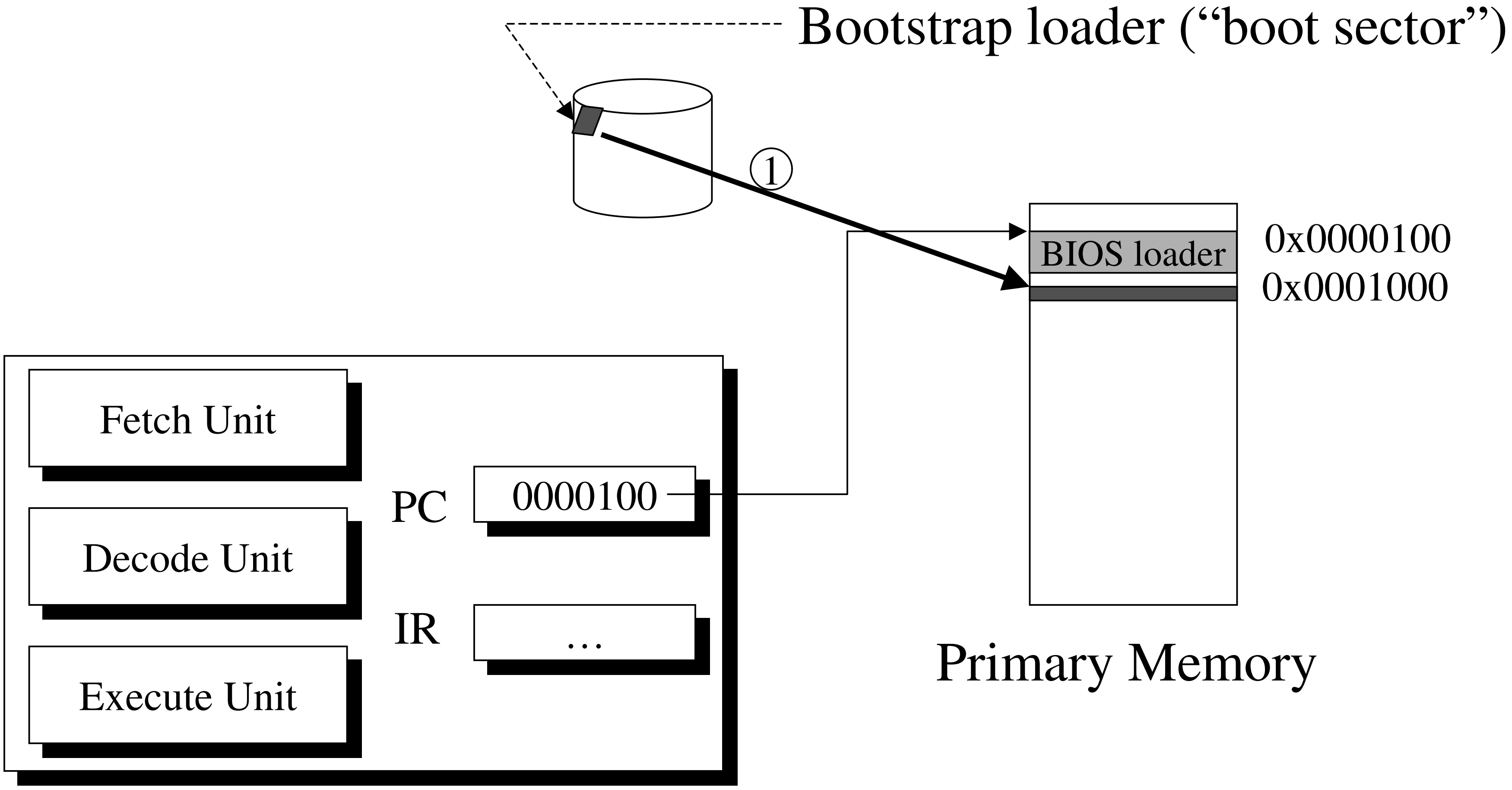
Decode(IR)

Fetch

# OS boot-up…

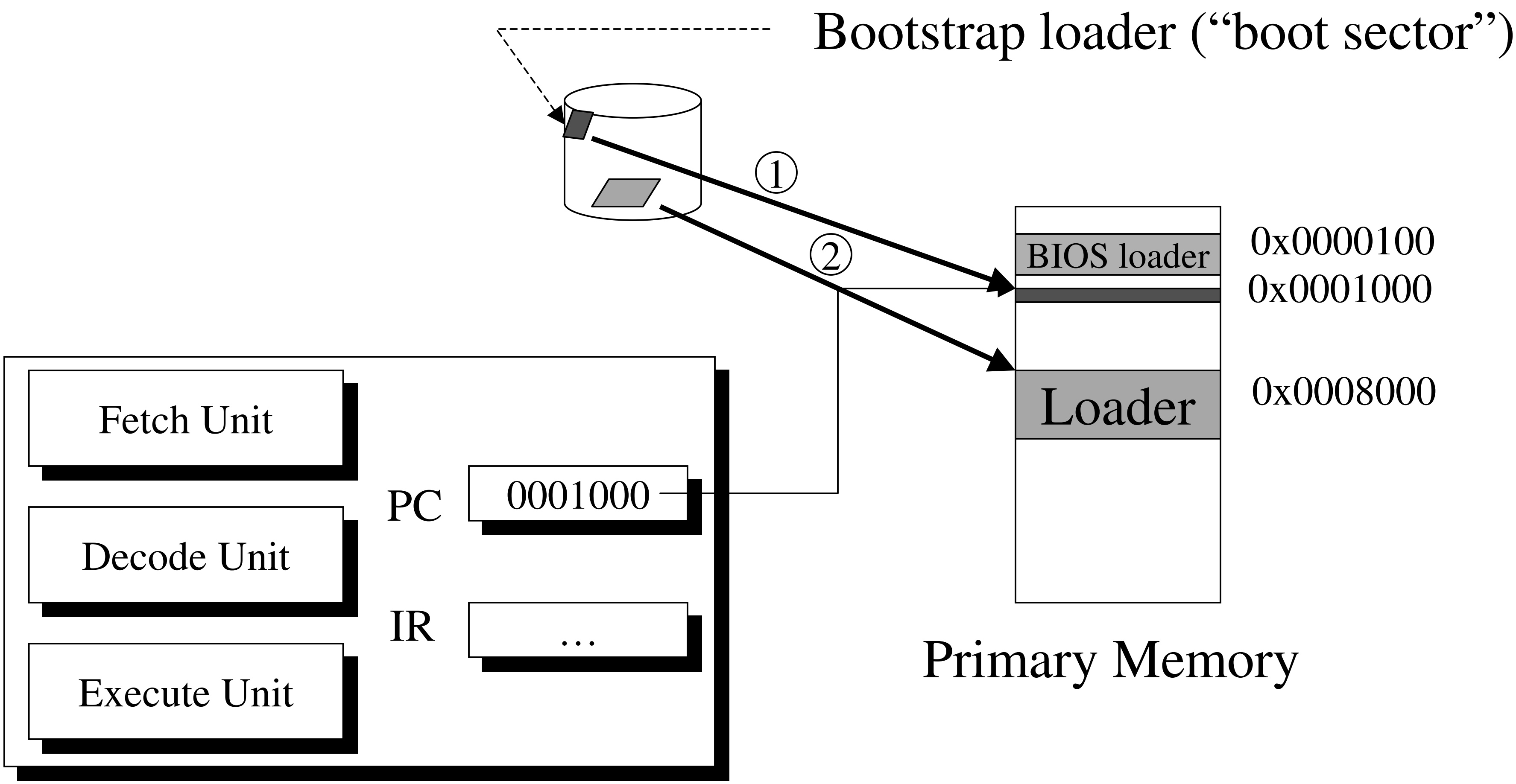
- How does the system boot up ?
  - Bootstrap loader
  - OS
  - Application

# Bootstrapping

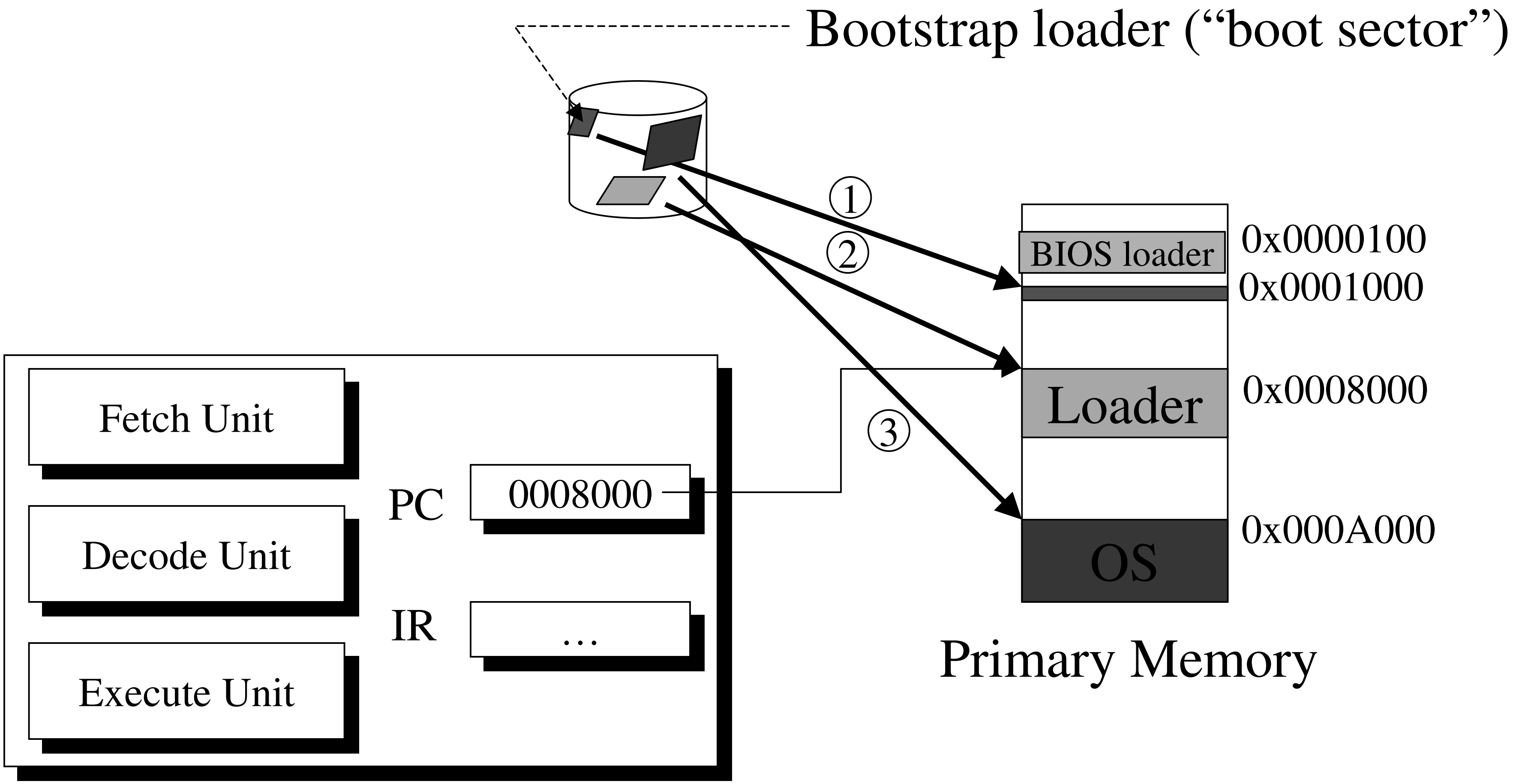


Primary Memory

# Bootstrapping



# Bootstrapping



# A Bootstrap Loader

The power-up sequence

```
        load PC, FIXED_LOC
```

Address of BS Loader

Where FIXED_LOC addresses the bootstrap loader (in ROM).

The bootstrap loader has the form:

```
        load R1, =0
        load R2, = LENGTH_OF_TARGET
loop:   read R1, FIXED_DISK_ADDRESS
        store R1, [FIXED_DEST, R1]
        incr R1
        bleq R1, R2, loop
        br FIXED_DEST
```

Reads OS in

Fetch

Decode

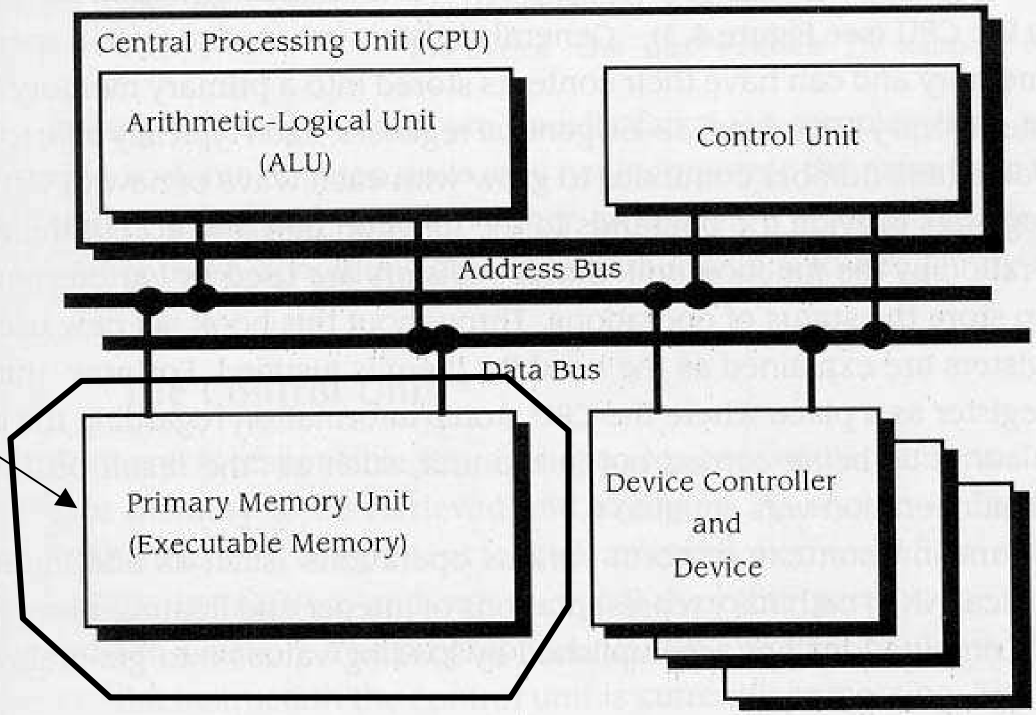
Execute

Branches to OS

# Memory Unit

**FIGURE 4.2**

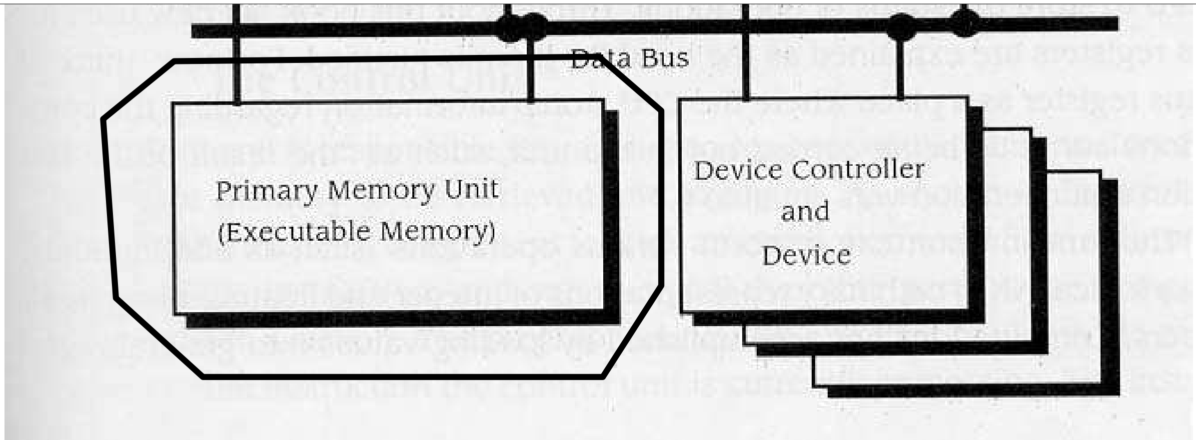
**The von Neumann Machine Architecture**



Memory Unit

# Memory Unit

- Memory Unit contains
  - Memory
    - Instructions & Data
  - MAR (Memory Address Register)
  - MDR (Memory Data register)
  - CMD (Command Register)
  - Get instruction at location pointed to by PC and place in IR



# Memory Access

- Read from Memory
  - `MAR ← MemAddr`
  - `CMD ← 'Read OP'` (from IR)
  - Execute

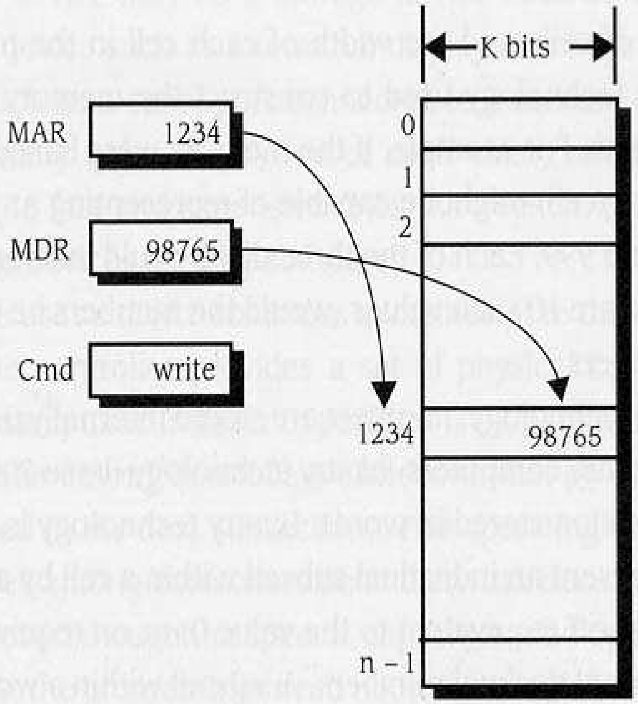
    `MDR ← Mem[ MAR ]`

- Write to Memory
  - `MAR ← MemAddr`
  - `CMD ← 'Write OP'` (from IR)
  - Execute

    `Mem[ MAR ] ← MDR`

**FIGURE 4.6**

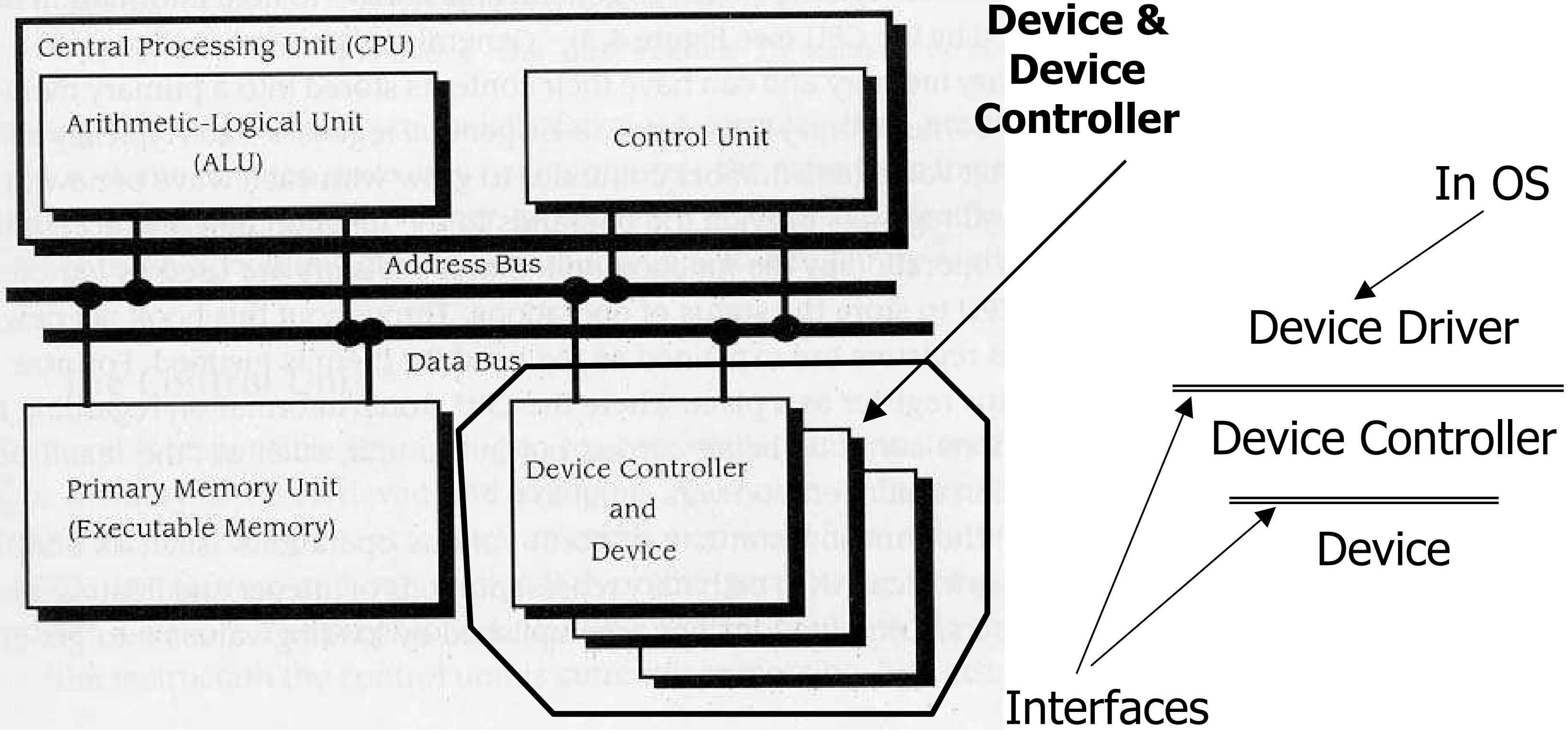
**The Memory Organization**



# Device & Device Controller

RE 4.2

**von Neumann Machine Architecture**



# Device Controller-Software Relationship

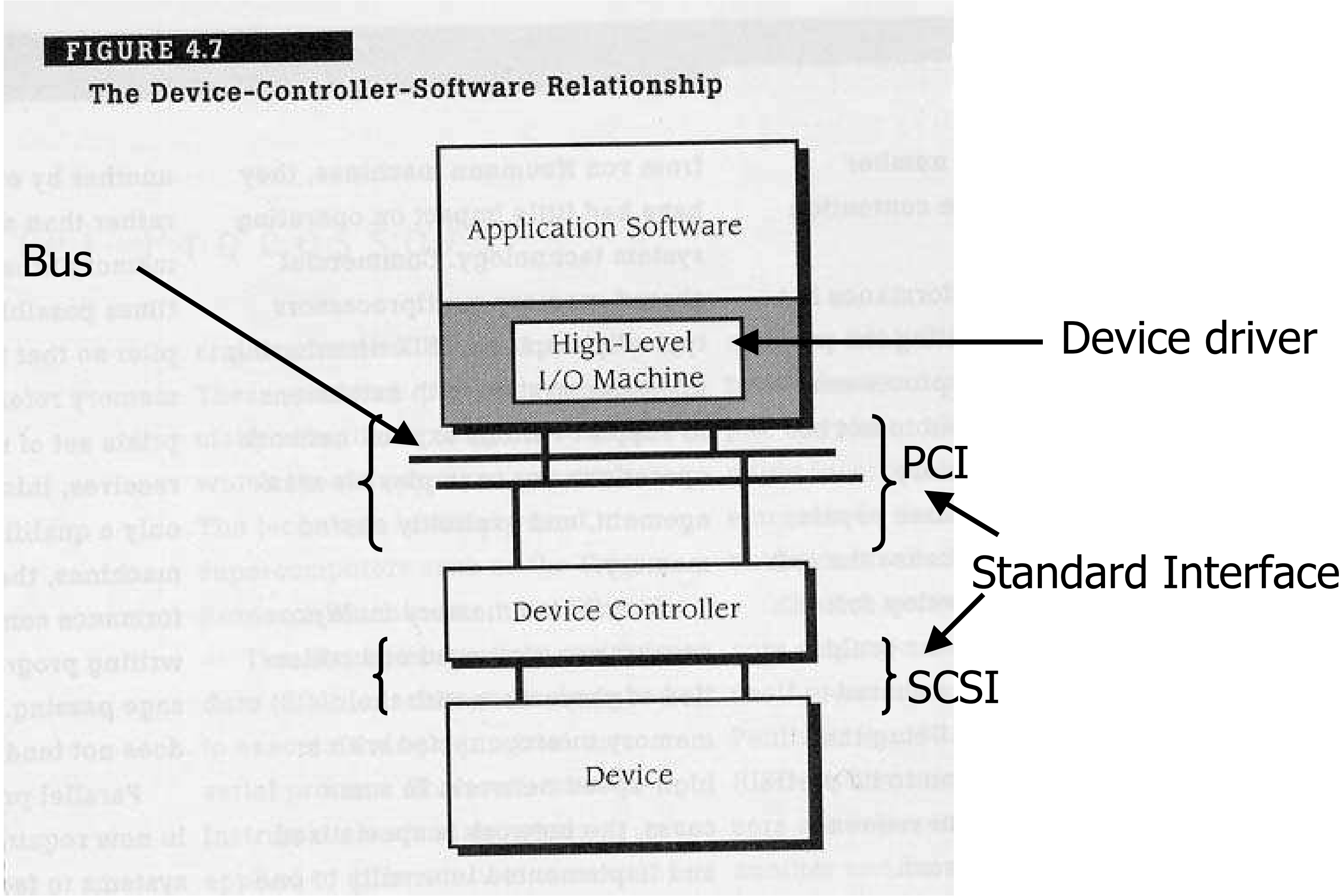


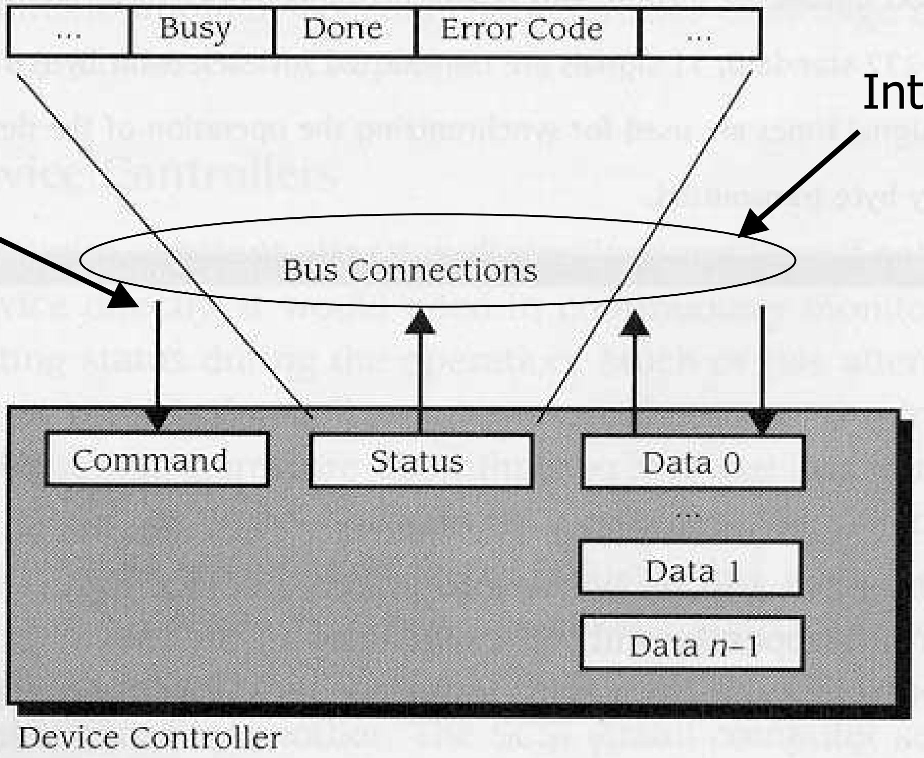
**FIGURE 4.7**

**The Device-Controller-Software Relationship**

# Device Controller Interface

**FIGURE 4.8**

**The Device Controller Interface**



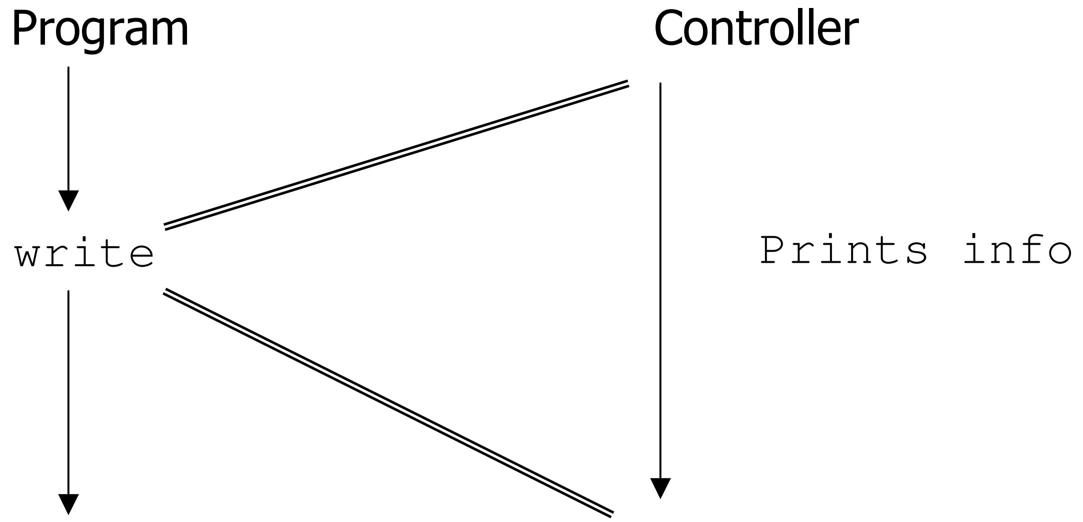
Driver interrogated these to check status of device

Driver places command if status "Done"

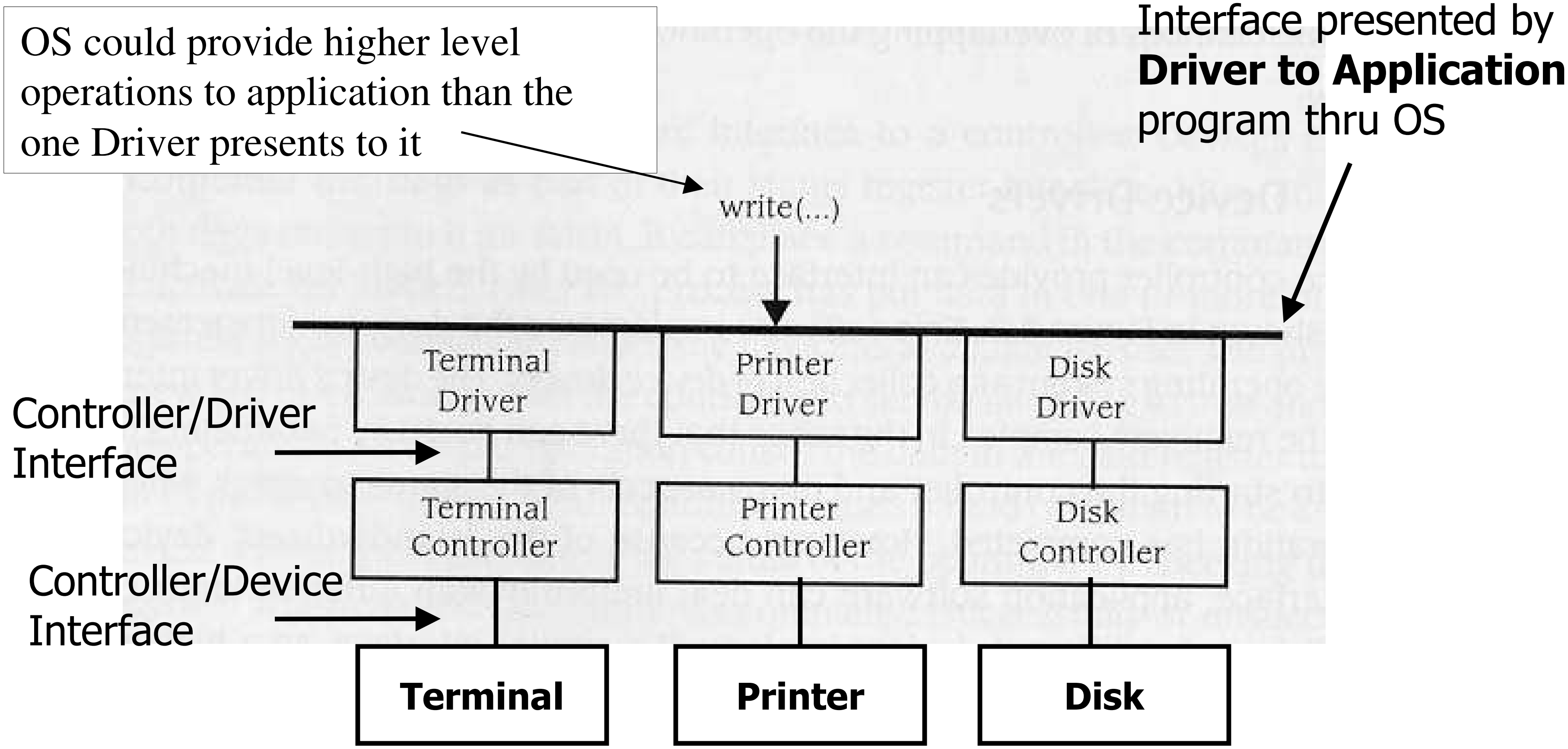
Interface to driver

# Device Controller

- Device controller is a processor and allows 2 parts of the process to proceed concurrently



# Device Driver Interface



OS could provide higher level operations to application than the one Driver presents to it

Interface presented by **Driver to Application** program thru OS

Controller/Driver Interface

Controller/Device Interface

# How do interrupts factor in ?

- Scenario (1)
  - Program:

```
while device_flag busy {}
```

**=>** Busy wait - consumes CPU cycles

- Scenario (2)
  - Program:

```
while (Flag != write) {
  sleep( X )
}
```

**=>**If write available while program sleeping - inefficient

# How do interrupts factor in ? …

- Scenario (3)
  - Program:

    `issues “write”`

Driver:

- Suspend program until write is completed,

  then program is unsuspended

**This is Interrupt-driven**

# Interrupts Driven Service Request

- Process is suspended only if driver/controller/device cannot service request
- If a process is suspended, then, when the suspended process' service request can be honored
  - Device interrupts CPU
  - OS takes over
  - OS examines interrupts
  - OS un-suspends the process
- Interrupts
  - Eliminate busy wait
  - Minimizes idle time

# Interrupts …

Interrupt Handler in OS:

```
disables interrupts
           :
           :   Interrupt processed
           :
           :
  enables interrupts
```

**What if multiple devices (or 2nd device) sends interrupt while the OS is handling prior interrupt ?**

If **priority** of 2nd interrupt higher than 1st then 1st interrupt suspended ⇒ 2nd interrupt handled ⇒ Resumption of handling 1st interrupt

# Control Unit with Interrupt (H/W)

```
PC = <machine start address>;
IR = memory[PC];
haltFlag = CLEAR;
while(haltFlag not SET) {
    execute(IR);
    PC = PC + sizeof(INSTRUCT);
    IR = memory[PC];
    if(InterruptRequest) {
        memory[0] = PC;
        PC = memory[1]
};
```

memory[1] contains the address of the interrupt handler

# Interrupt Handler (Software)

```
interruptHandler() {
➡   saveProcessorState();
    for(i=0; i<NumberOfDevices; i++)
        if(device[i].done) goto deviceHandler(i);
    /* something wrong if we get to here … */


deviceHandler(int i) {
    finishOperation();
    returnToScheduler();
}
```

# A Race Condition

```
saveProcessorState() {
    for(i=0; i<NumberOfRegisters; i++)
        memory[K+i] = R[i];
    for(i=0; i<NumberOfStatusRegisters; i++)
        memory[K+NumberOfRegisters+i] = StatusRegister[i];
}
```

---

```
PC = <machine start address>;
IR = memory[PC];
haltFlag = CLEAR;
while(haltFlag not SET) {
    execute(IR);
    PC = PC + sizeof(INSTRUCT);
    IR = memory[PC];
    if(InterruptRequest && InterruptEnabled) {
        disableInterupts();
        memory[0] = PC;
        PC = memory[1]
};
```

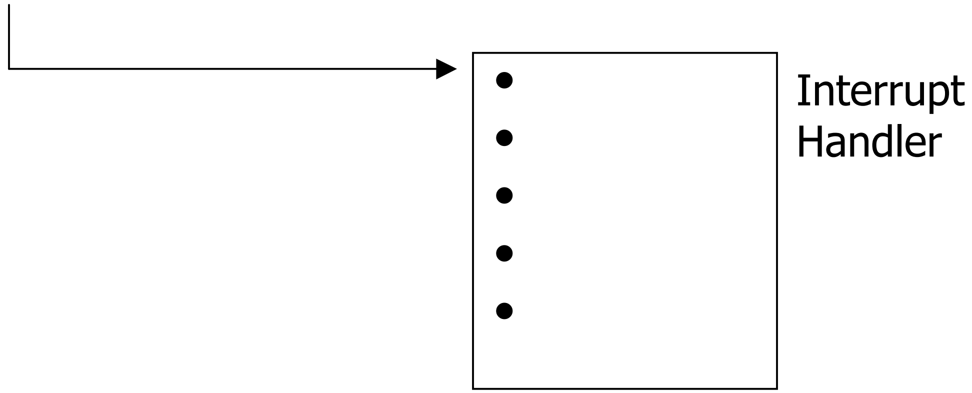
# Revisiting the `trap` Instruction (H/W)

```
executeTrap(argument) {
    setMode(supervisor);
    switch(argument) {
    case 1: PC = memory[1001];  // Trap handler 1
    case 2: PC = memory[1002];  // Trap handler 2
    . . .
    case n: PC = memory[1000+n];// Trap handler n
};
```

- The trap instruction dispatches a trap handler routine atomically
- Trap handler performs desired processing
- “A trap is a software interrupt”

# Requesting Service from OS

- Kernel functions are invoked by “trap”



- System call
  - Process traps to OS Interrupt Handler
  - Supervisor mode set
  - Desired function executed
  - User mode set
  - Returns to application

# Requesting Svc:  System Call

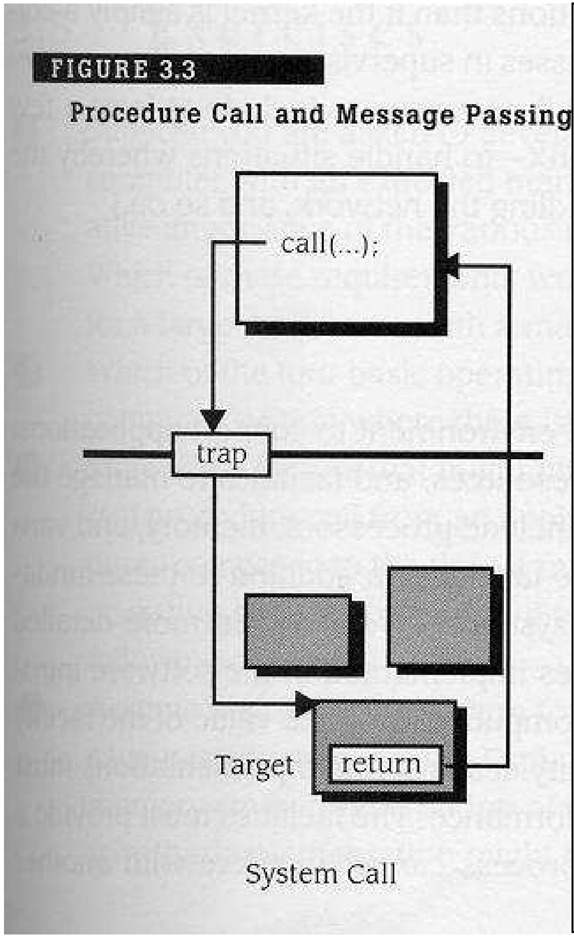


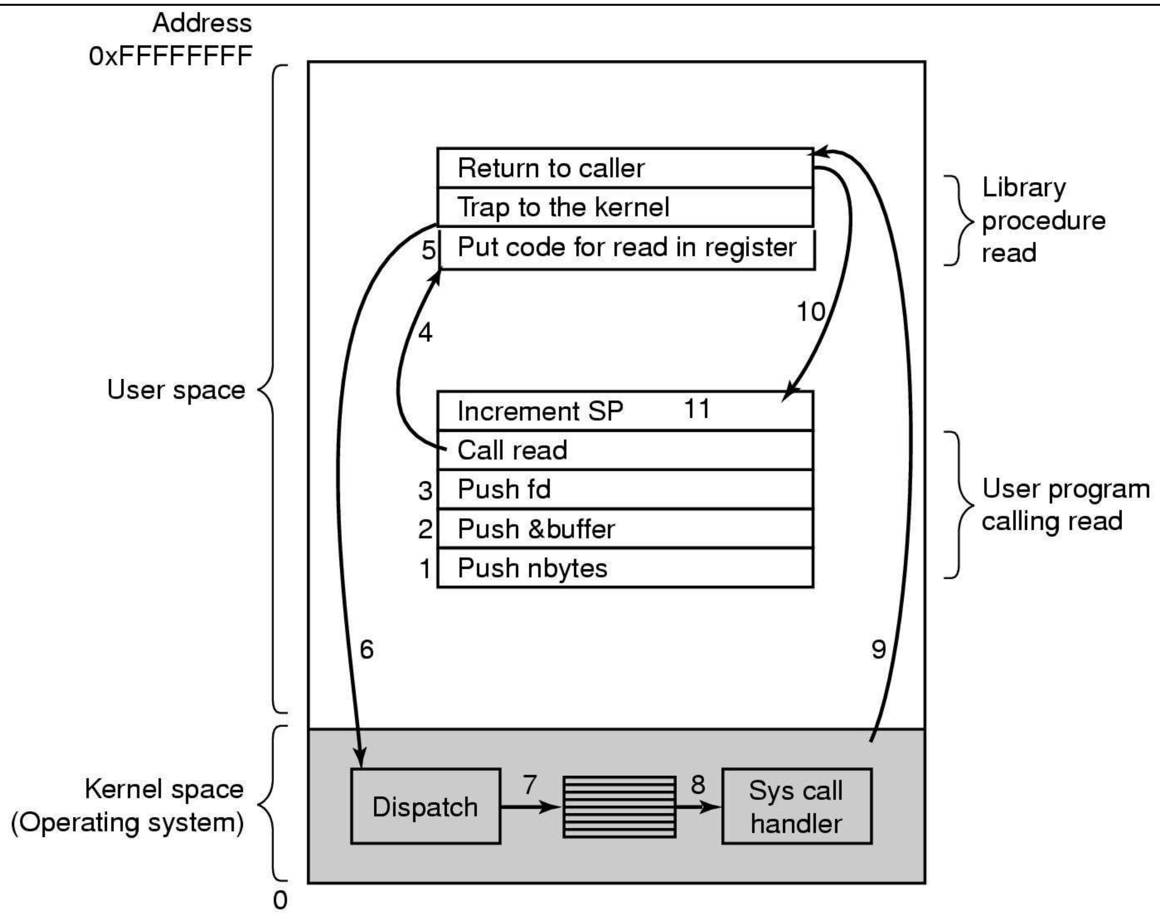
FIGURE 3.3

**Procedure Call and Message Passing**

System Call

# Steps in making a system call

Taken from Modern Operating Systems, 2nd Ed, Tanenbaum, 2001



There are 11 steps in making the system call read (fd, buffer, nbytes)