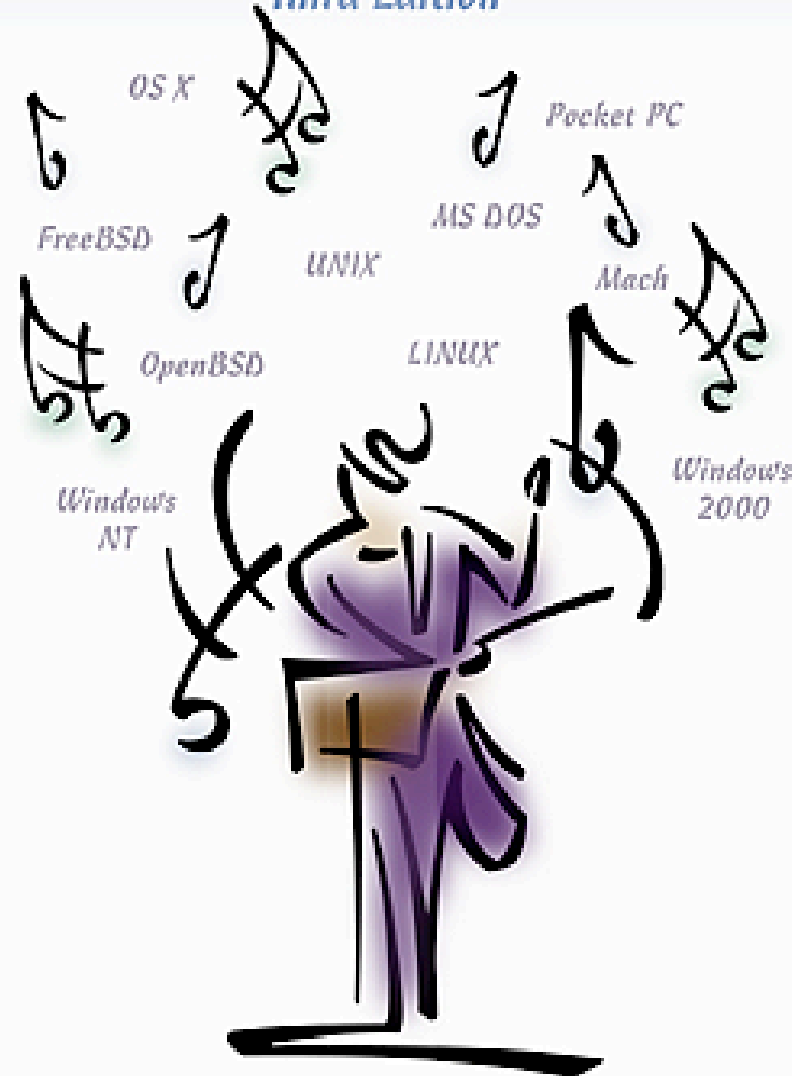


OPERATING SYSTEMS

Third Edition



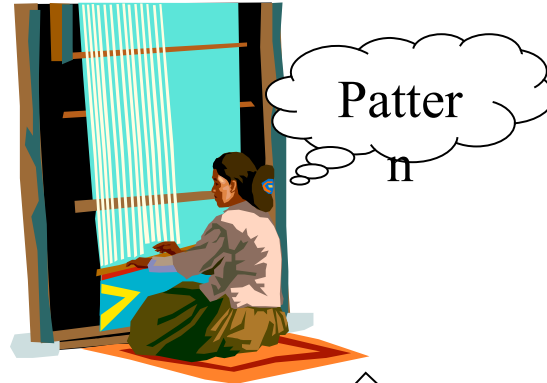
GARY NUTT



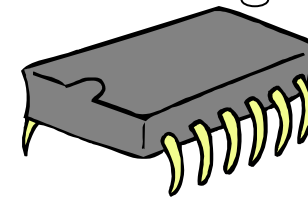
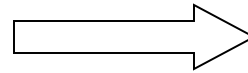
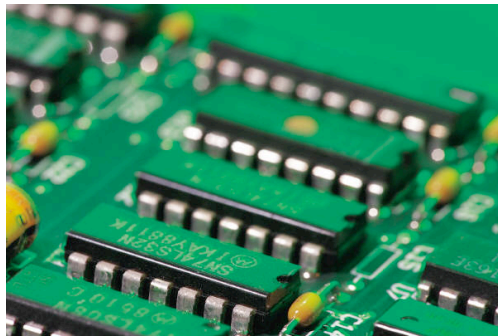
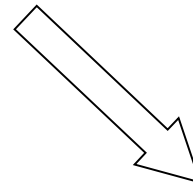
Computer Organization



Stored Program Computers and Electronic Devices



Jacquard Loom



Stored Program Device

Fixed Electronic Device

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

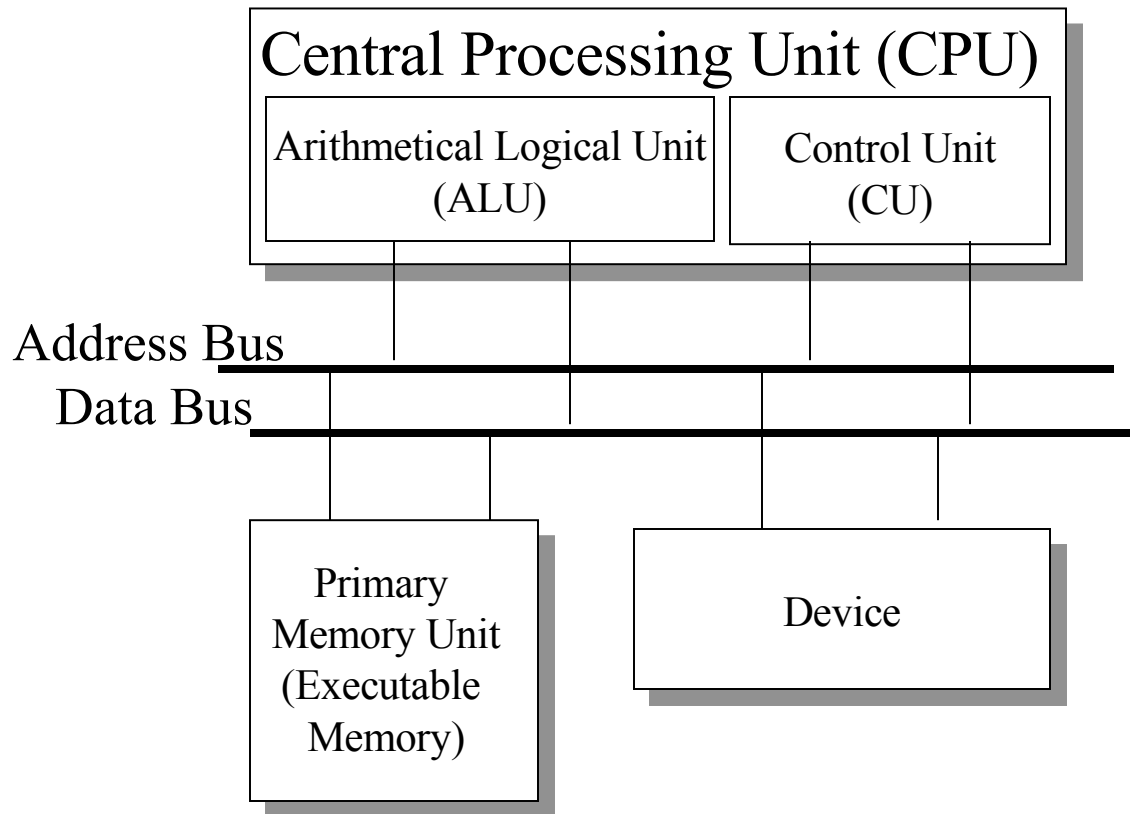
10111010001100...1

10111001010000...0

10100110001100...0

10111001101100...1

The von Neumann Architecture

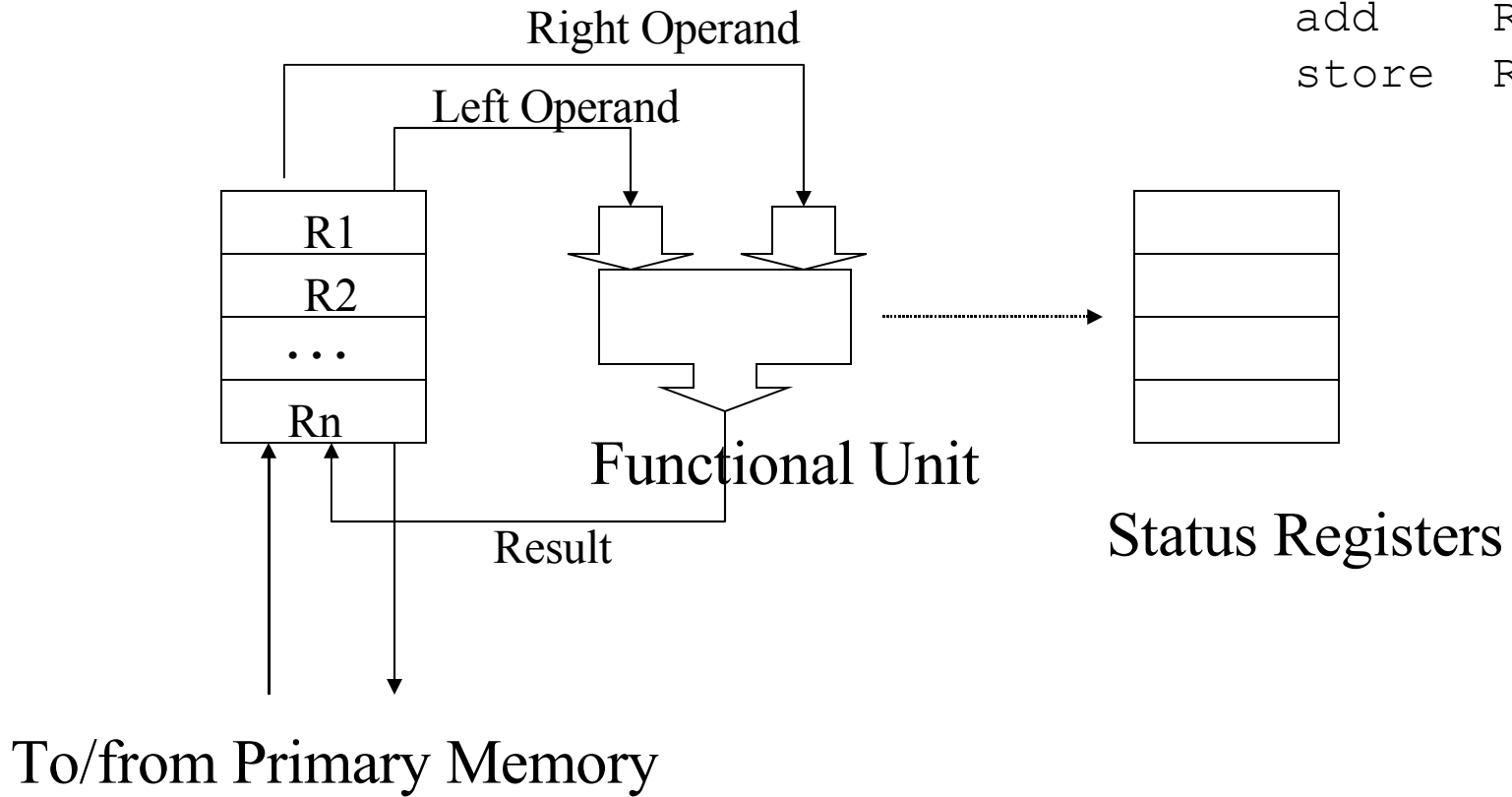


The ALU



Slide 4-7

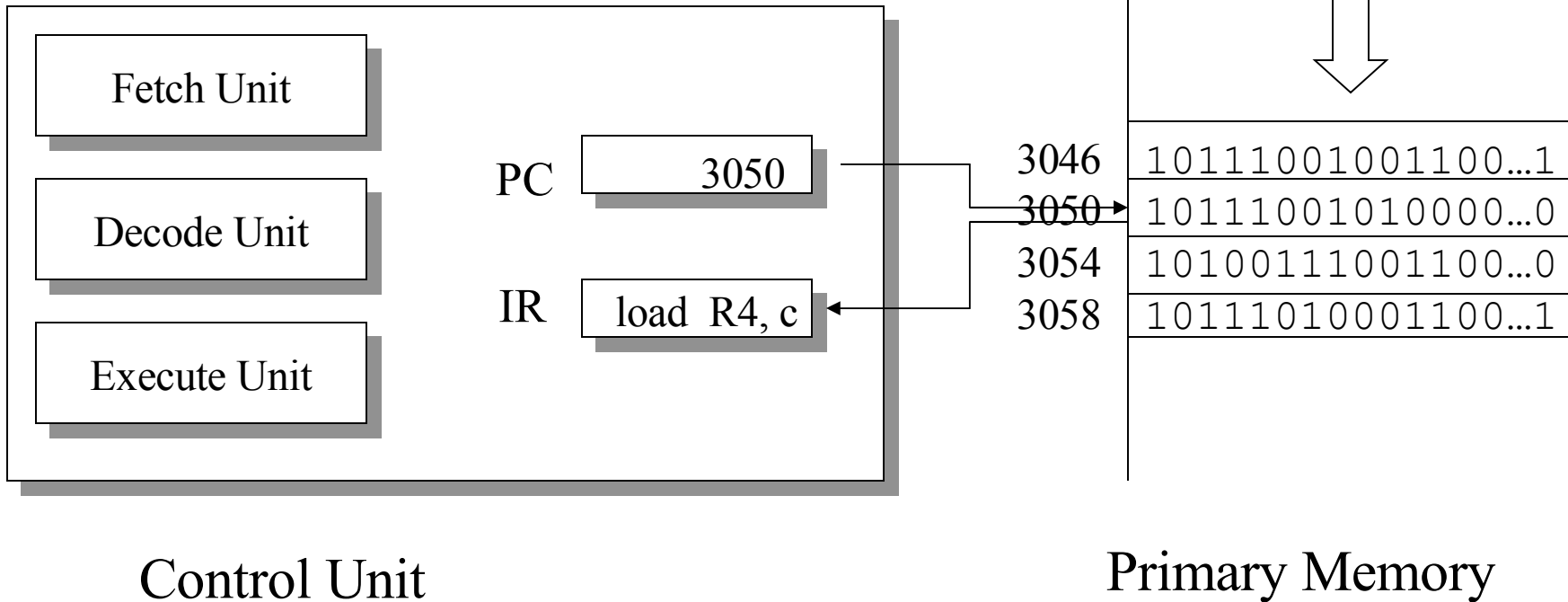
```
load  R3, b
load  R4, c
add   R3, R4
store R3, a
```



Control Unit



```
load R3, b
load R4, c
add R3, R4
store R3, a
```



Control Unit Operation

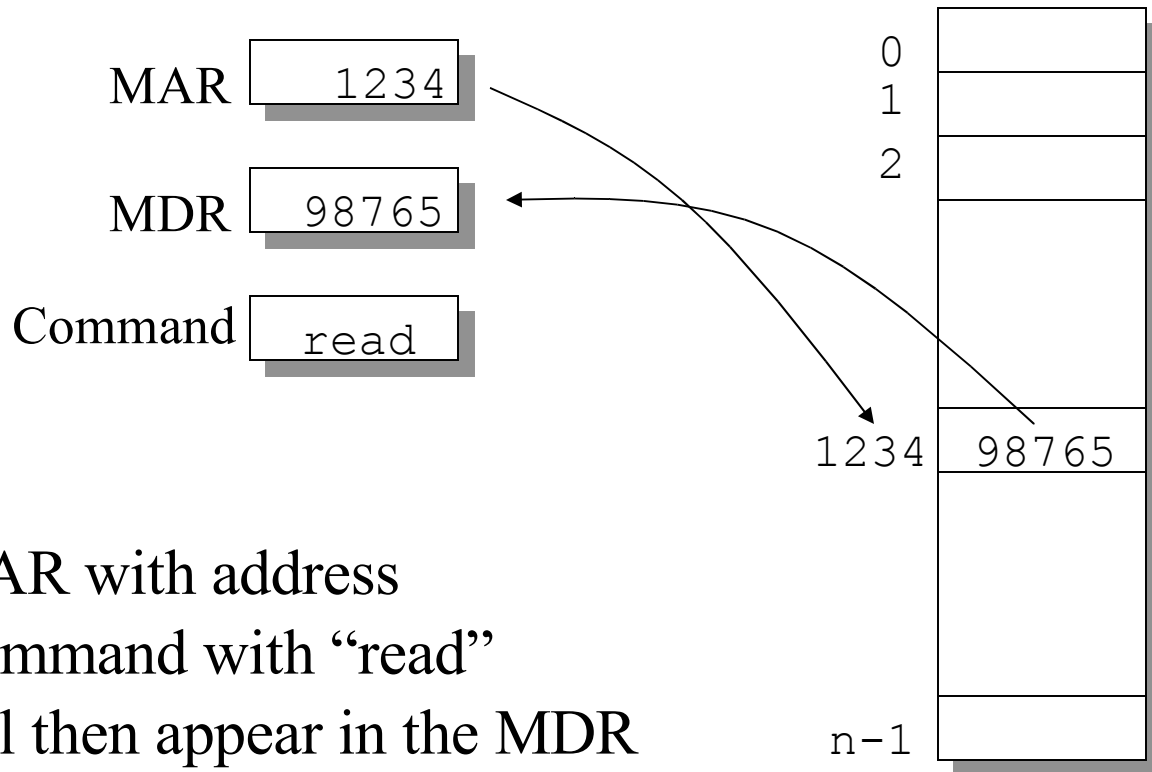


- Fetch phase: Instruction retrieved from memory
- Execute phase: ALU op, memory data reference, I/O, etc.

```
PC = <machine start address>;
IR = memory[PC];
haltFlag = CLEAR;
while(haltFlag not SET) {
    execute(IR);
    PC = PC + sizeof(INSTRUCT);
    IR = memory[PC]; // fetch phase
};
```



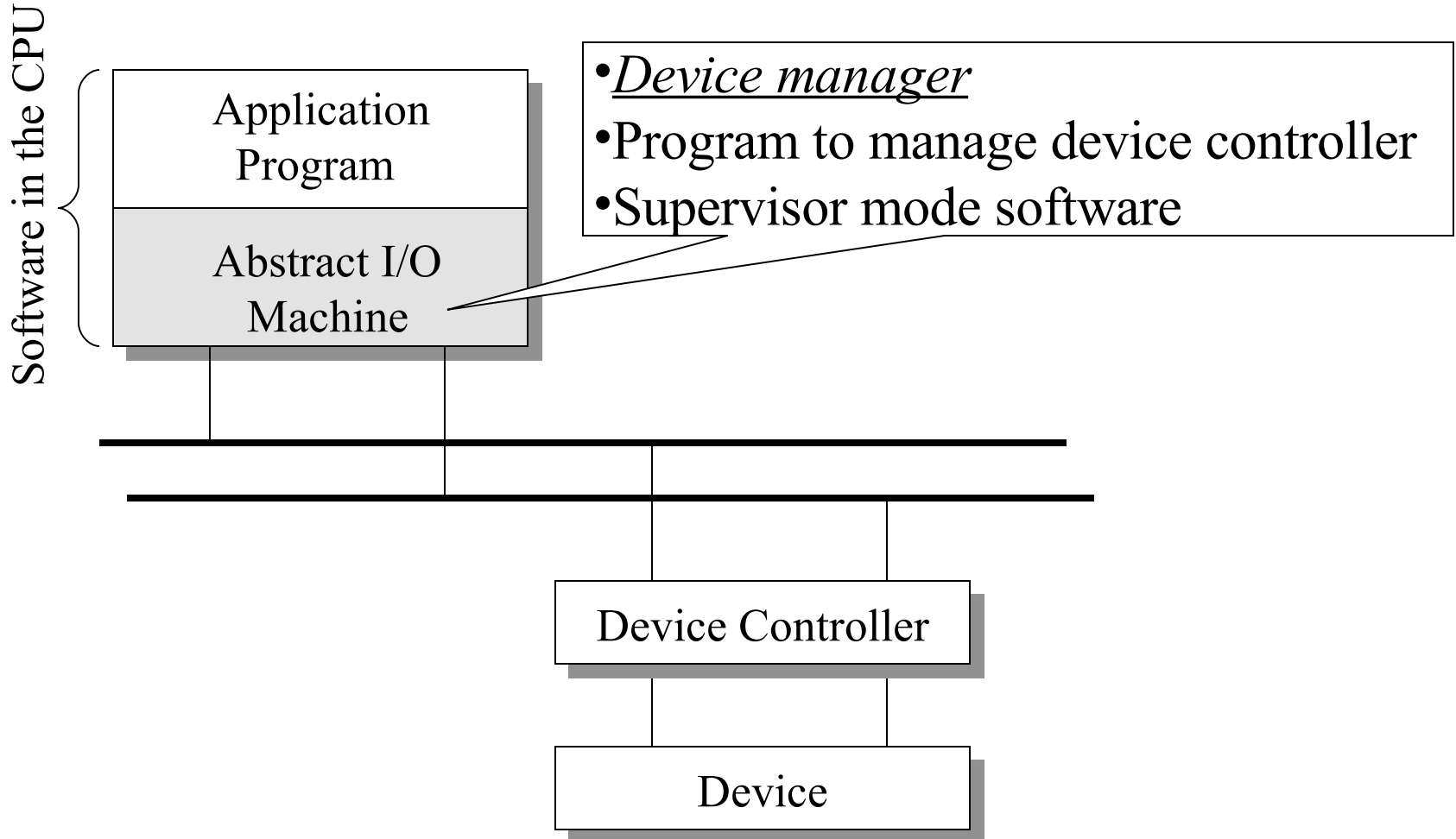
Primary Memory Unit



Read Op:

1. Load MAR with address
2. Load Command with “read”
3. Data will then appear in the MDR

The Device-Controller-Software Relationship

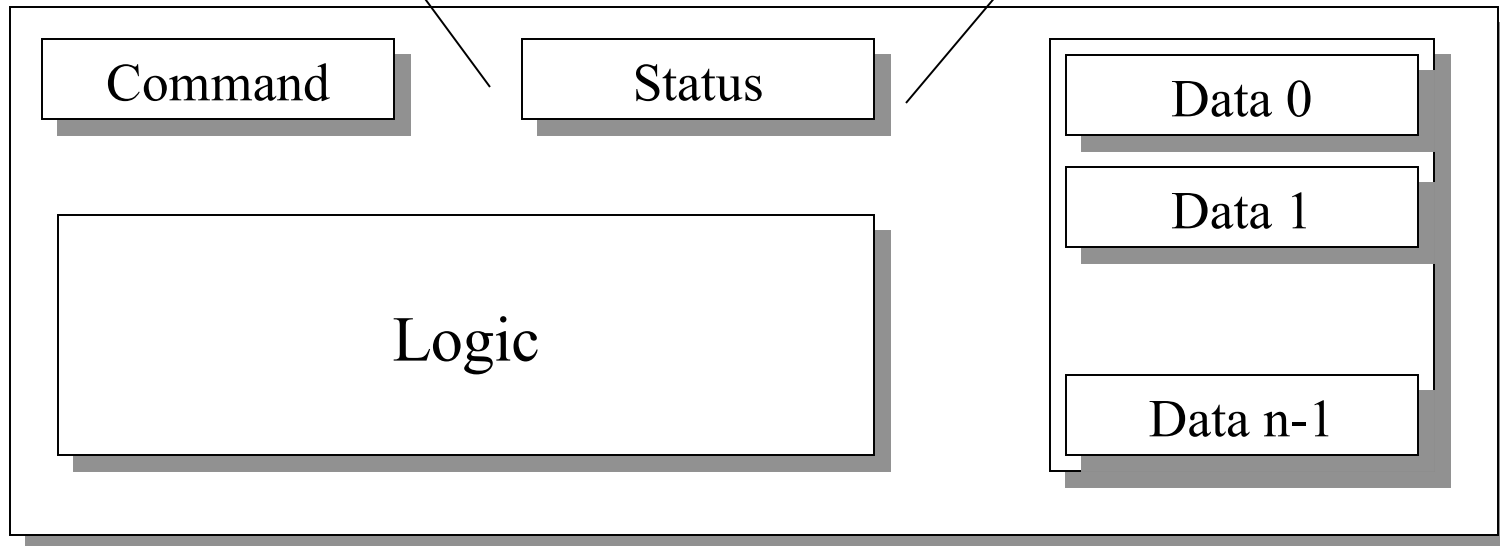


Device Controller Interface



...	busy	done	Error code	...
	0	0		
	0	1		
	1	0		
	1	1		

busy	done	
0	0	idle
0	1	finished
1	0	working
1	1	(undefined)



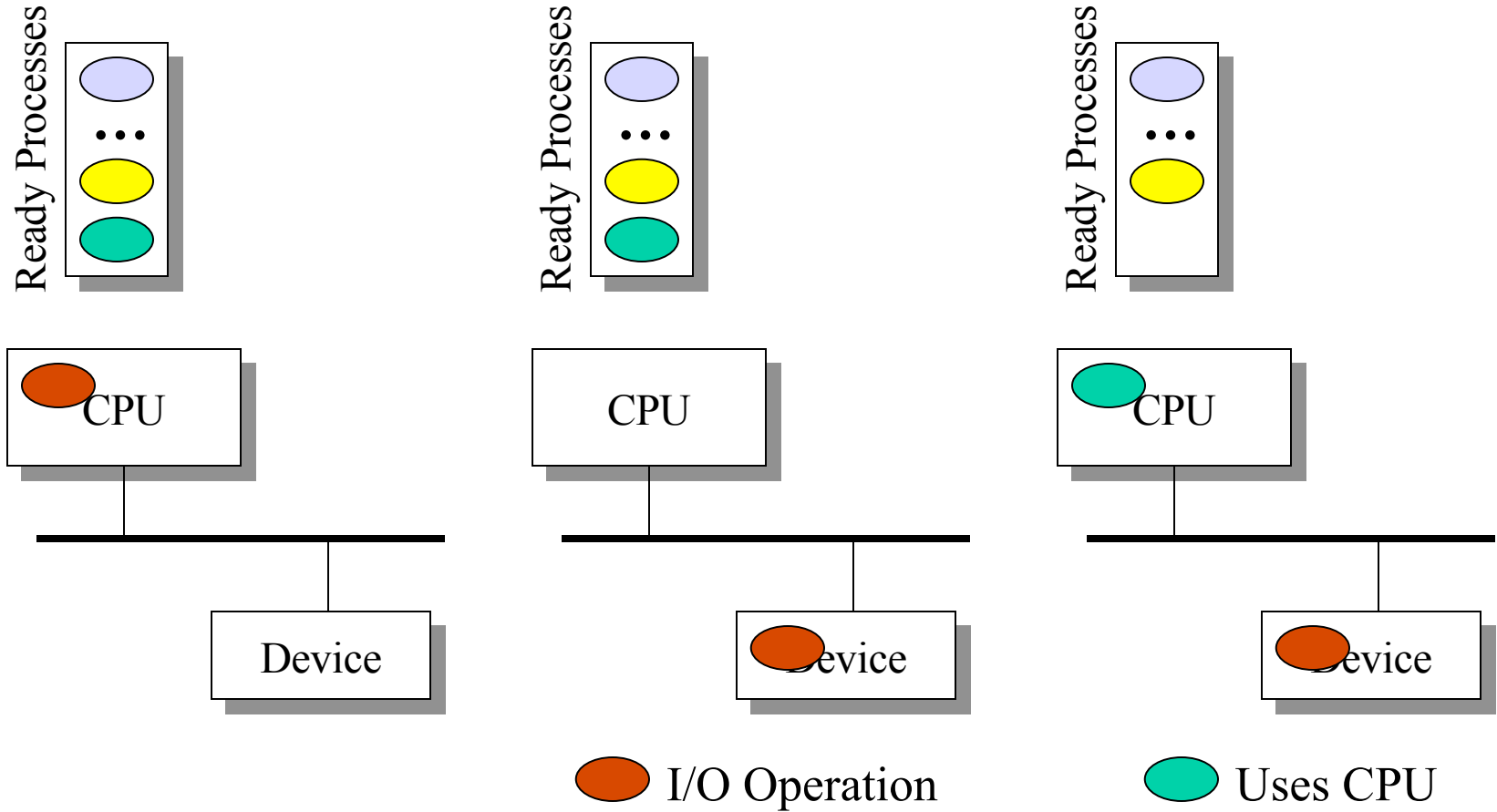


Performing a Write Operation

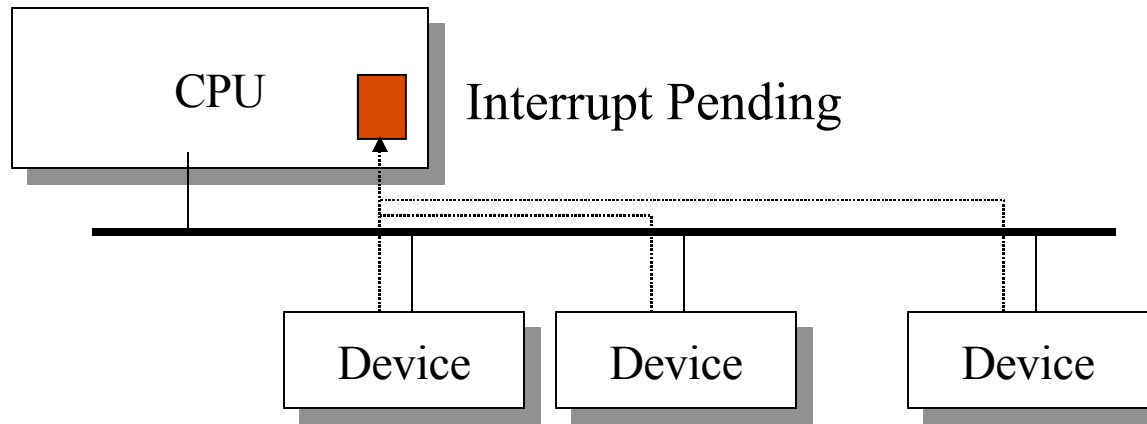
```
while(deviceNo.busy || deviceNo.done) <waiting>;  
deviceNo.data[0] = <value to write>  
deviceNo.command = WRITE;  
while(deviceNo.busy) <waiting>;  
deviceNo.done = TRUE; actually should be  
FALSE!
```

- Devices *much* slower than CPU
- CPU waits while device operates
- Would like to multiplex CPU to a different process while I/O is in process

CPU-I/O Overlap



Determining When I/O is Complete



- CPU incorporates an “interrupt pending” flag
- When device.busy FALSE, interrupt pending flag is set
- Hardware “tells” OS that the ***interrupt*** occurred
- ***Interrupt handler*** part of the OS makes process ready to run



Control Unit with Interrupt (Hardware)

```
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]
    }
};
```

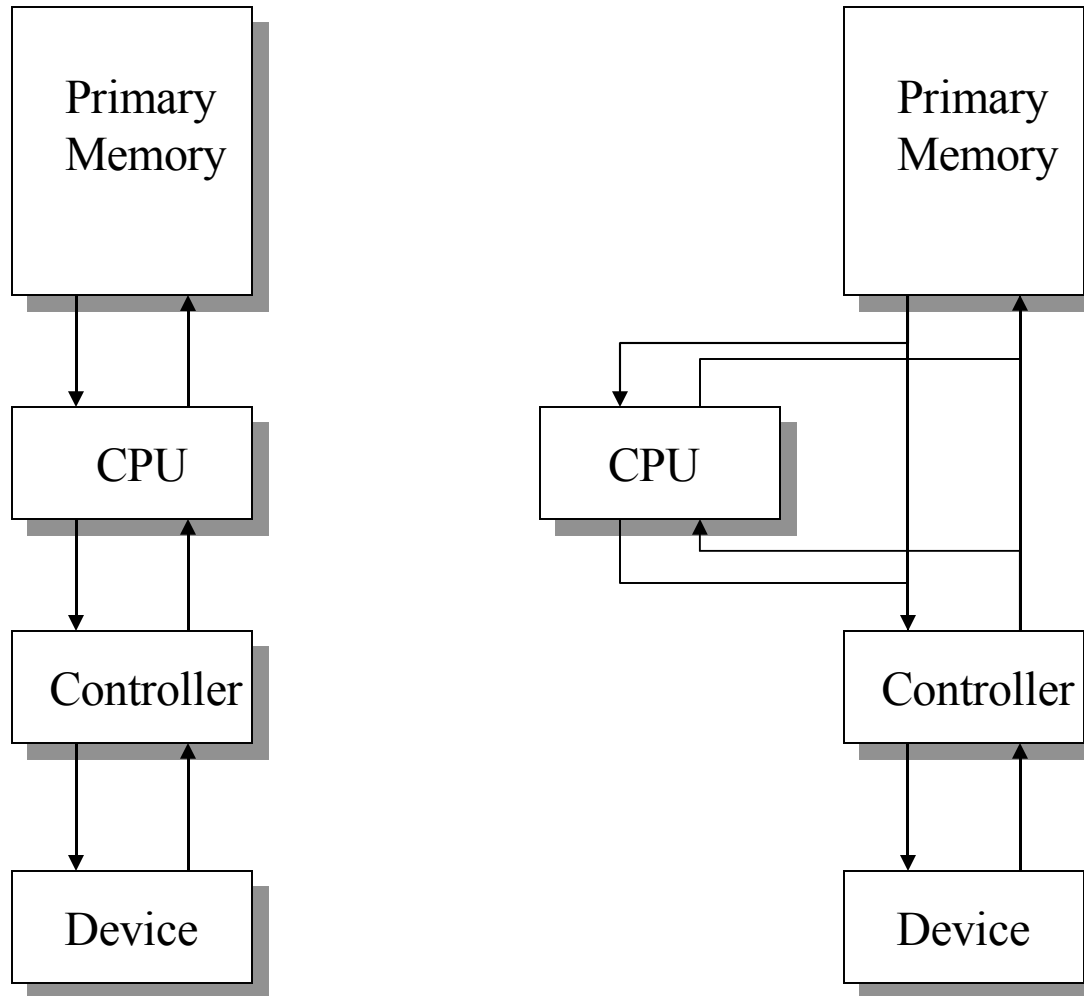


Revisiting the `trap` Instruction (Hardware)

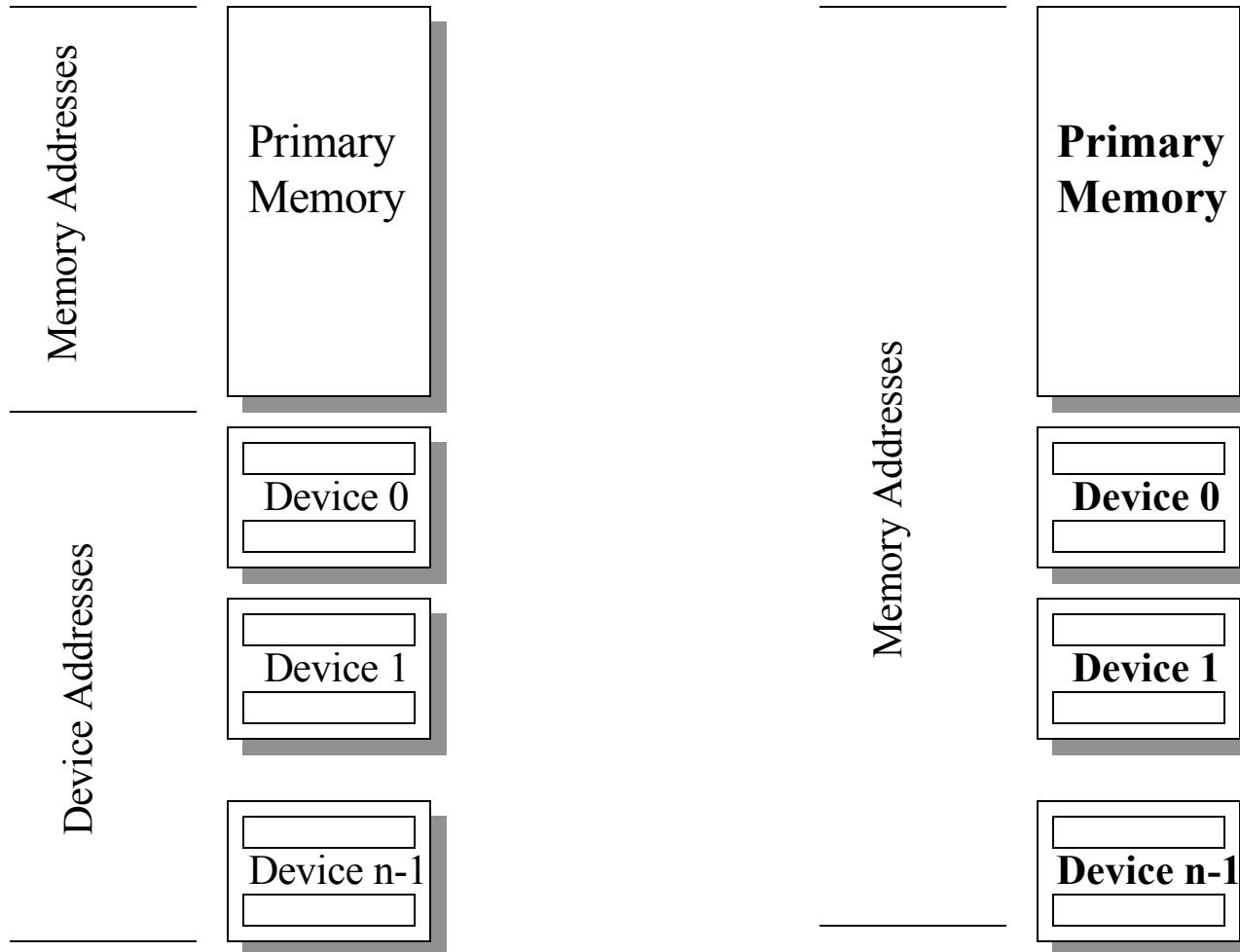
```
executeTrap(argument) {  
    setMode(supervisor);  
    switch(argument) {  
        case 1: PC = memory[1001]; // Trap handler 1  
        case 2: PC = memory[1002]; // Trap handler 2  
        . . .  
        case n: PC = memory[1000+n]; // Trap handler n  
    };  
};
```

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

Direct Memory Access



Addressing Devices





Polling I/O

Software

```
...  
// Start the device  
...  
While((busy == 1) || (done == 1))  
    wait();  
// Device I/O complete  
...  
done = 0;
```

Hardware

```
...  
while((busy == 0) && (done == 1))  
    wait();  
// Do the I/O operation  
busy = 1;  
...  
done
```



busy



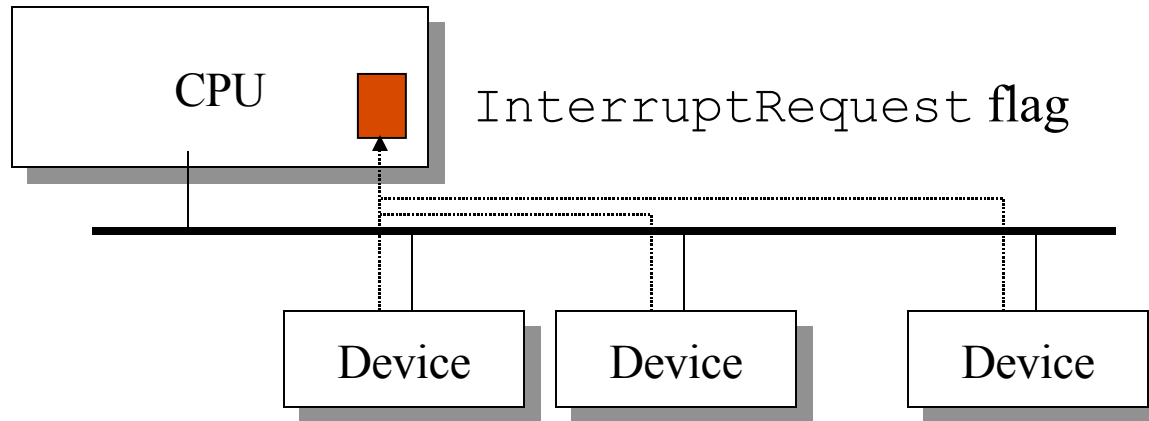
done



Fetch-Execute Cycle with an Interrupt

```
while (haltFlag not set during execution) {
    IR = memory[PC];
    PC = PC + 1;
    execute(IR);
    if (InterruptRequest) {
        /* Interrupt the current process */
        /* Save the current PC in address 0 */
        memory[0] = PC;
        /* Branch indirect through address 1 */
        PC = memory[1];
    }
}
```

Detecting an Interrupt



The Interrupt Handler



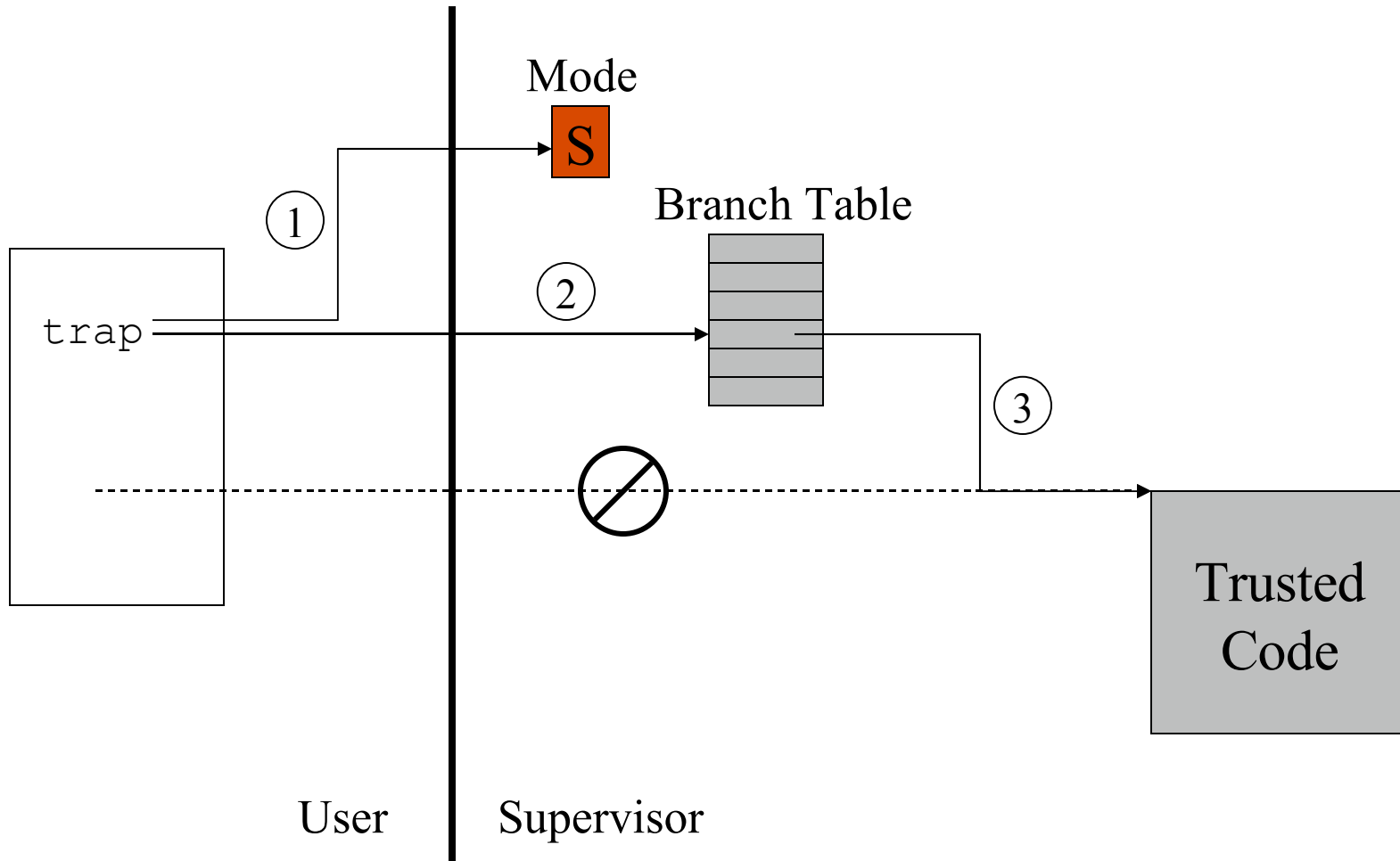
```
Interrupt_Handler{
    saveProcessorState();
    for (i=0; i<Number_of_devices; i++)
        if (device[i].done == 1)
            goto device_handler(i);
    /* Something wrong if we get here */
}
```

Disabling Interrupts

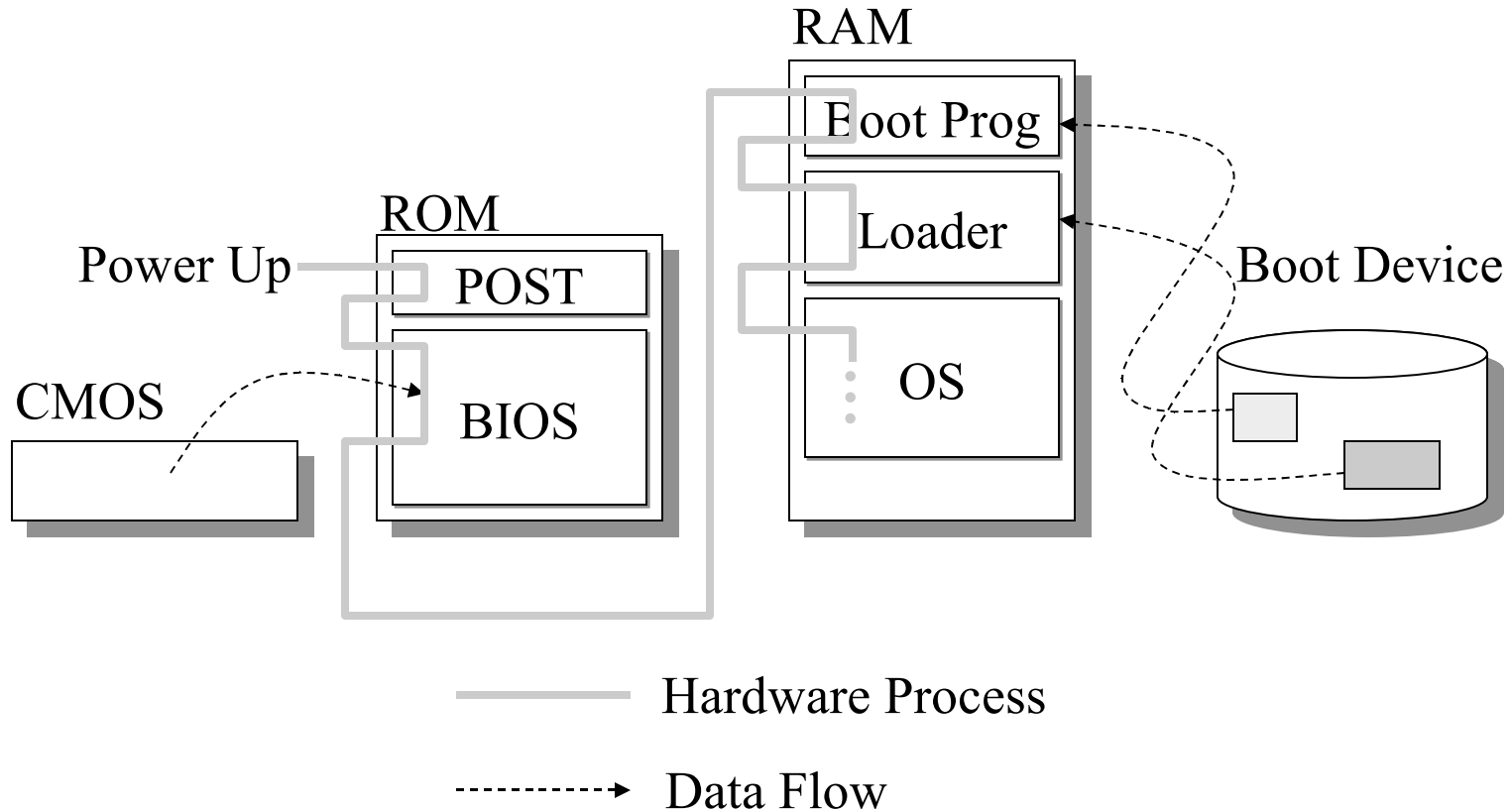


```
if (InterruptRequest && InterruptEnabled) {  
    /* Interrupt current process */  
    disableInterrupts();  
    memory[0] = PC;  
    PC = memory[1];  
}
```

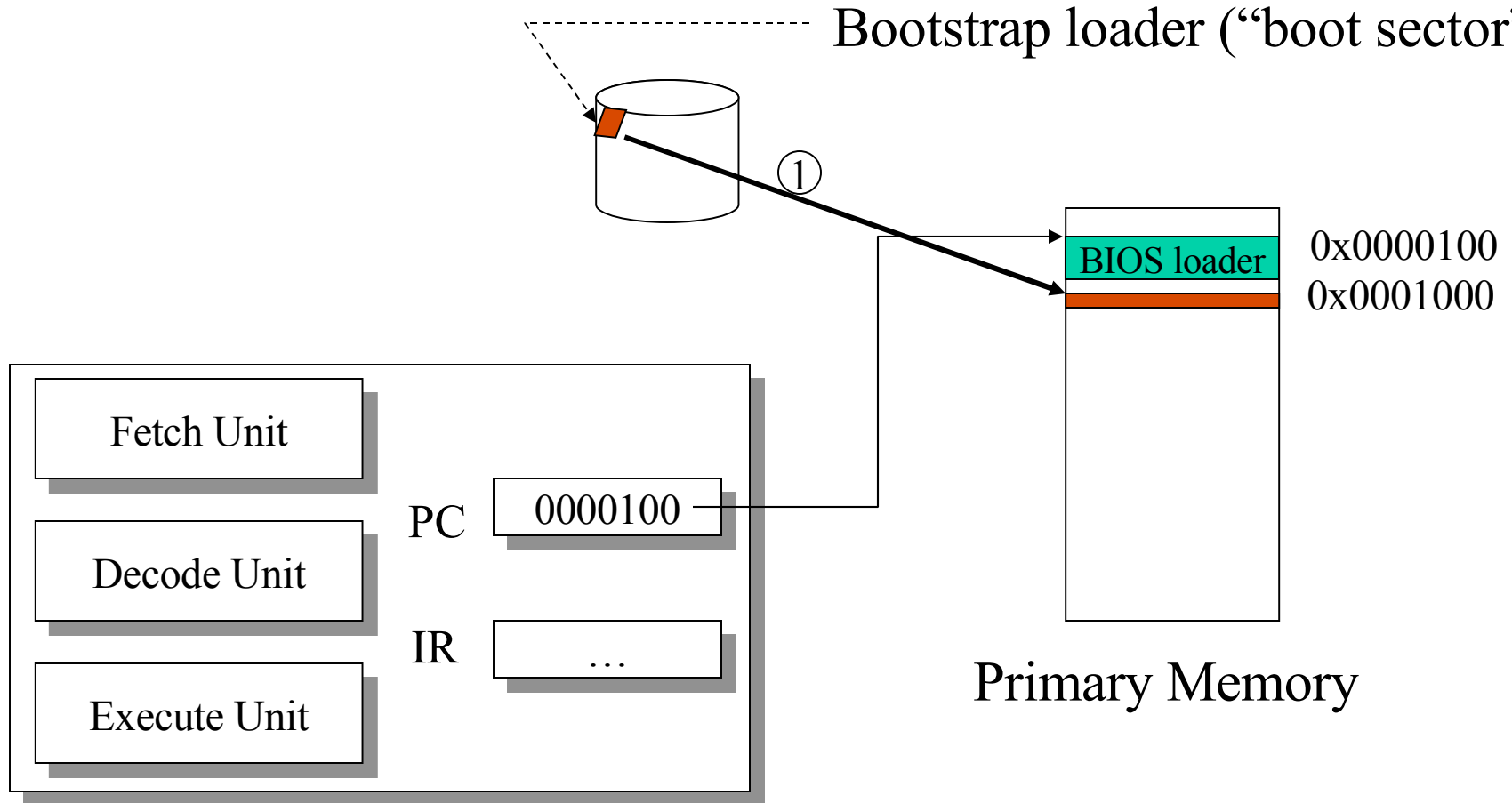
The Trap Instruction Operation



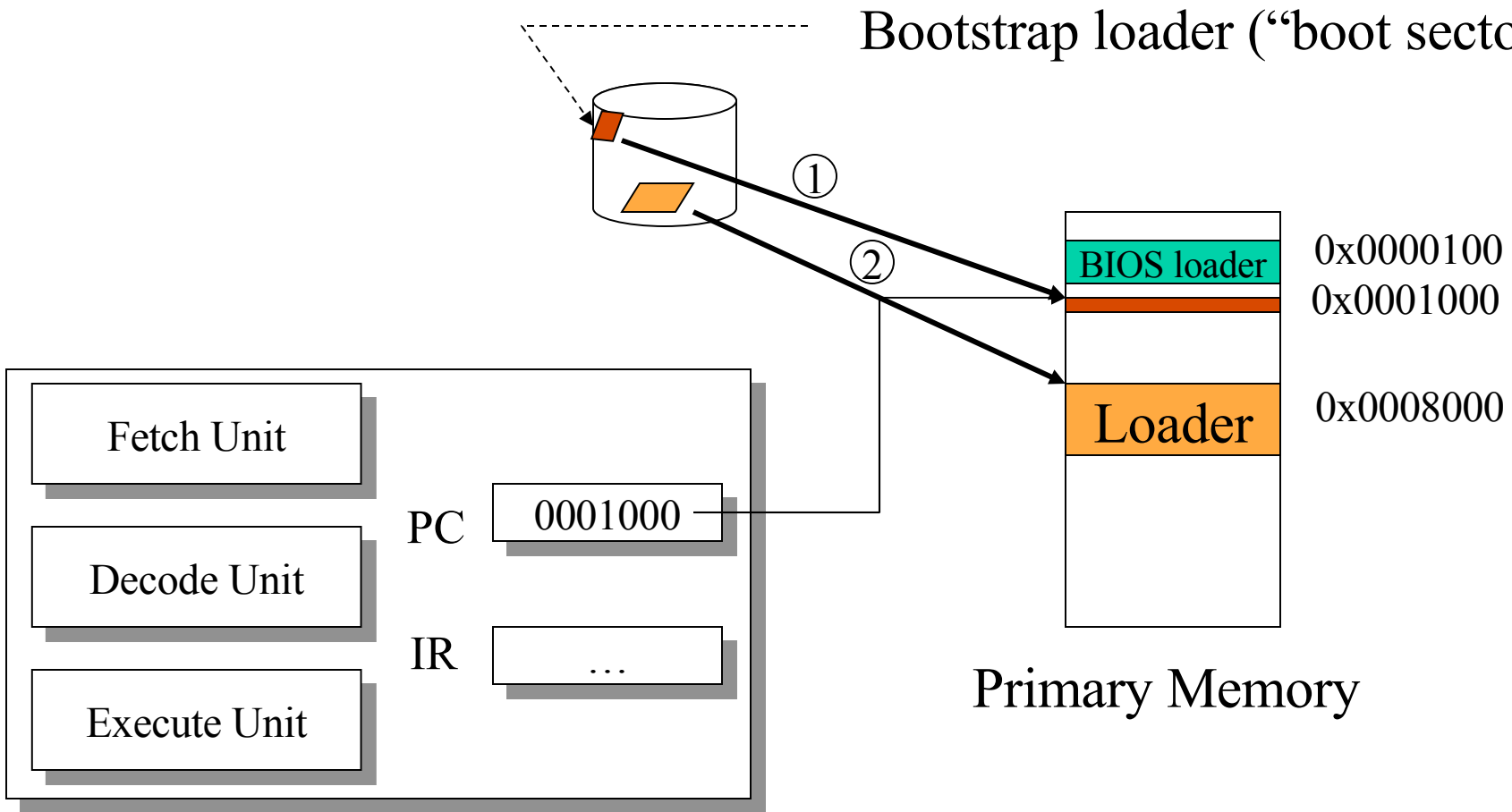
Intel System Initialization



Bootstrapping



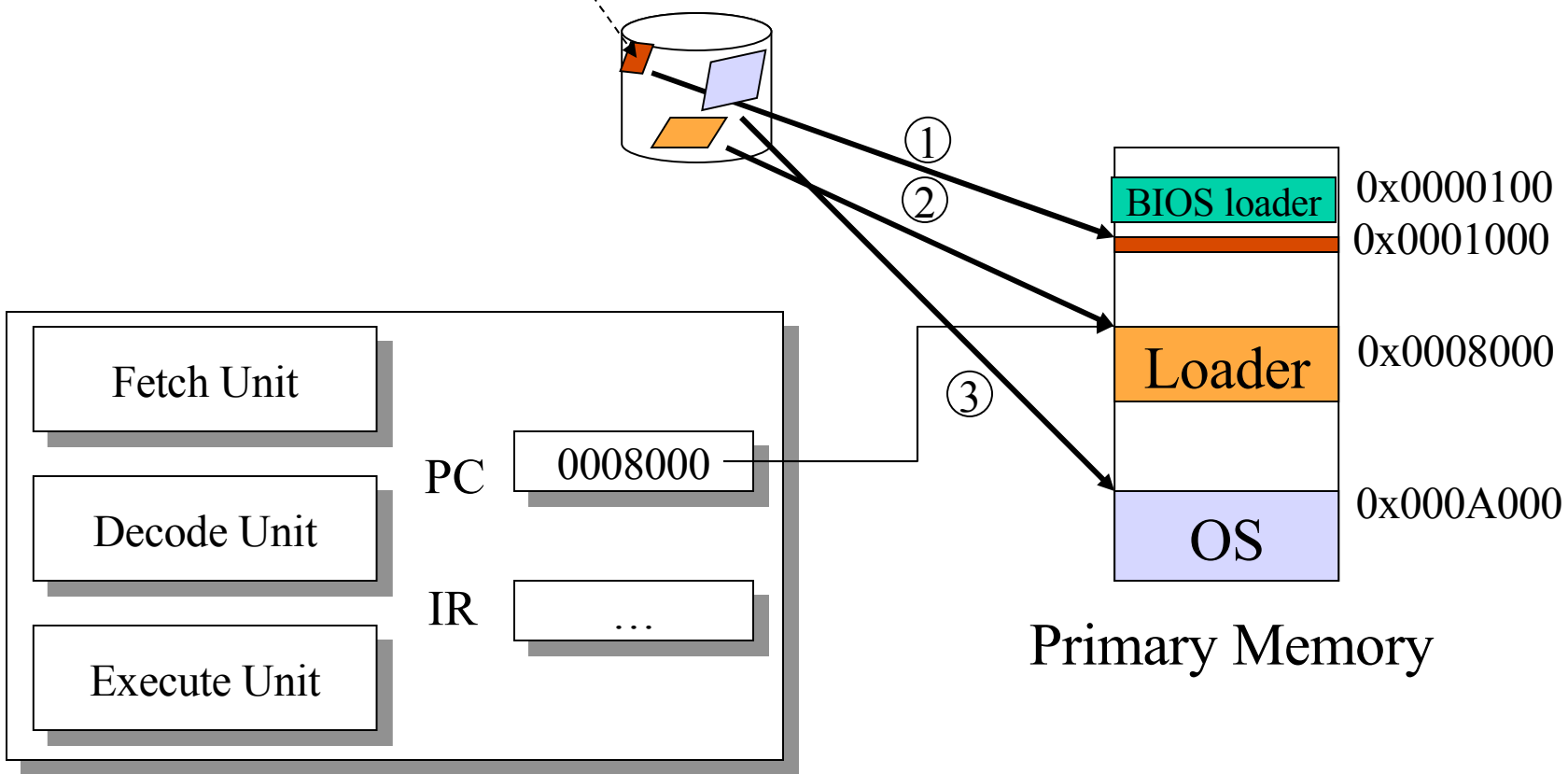
Bootstrapping



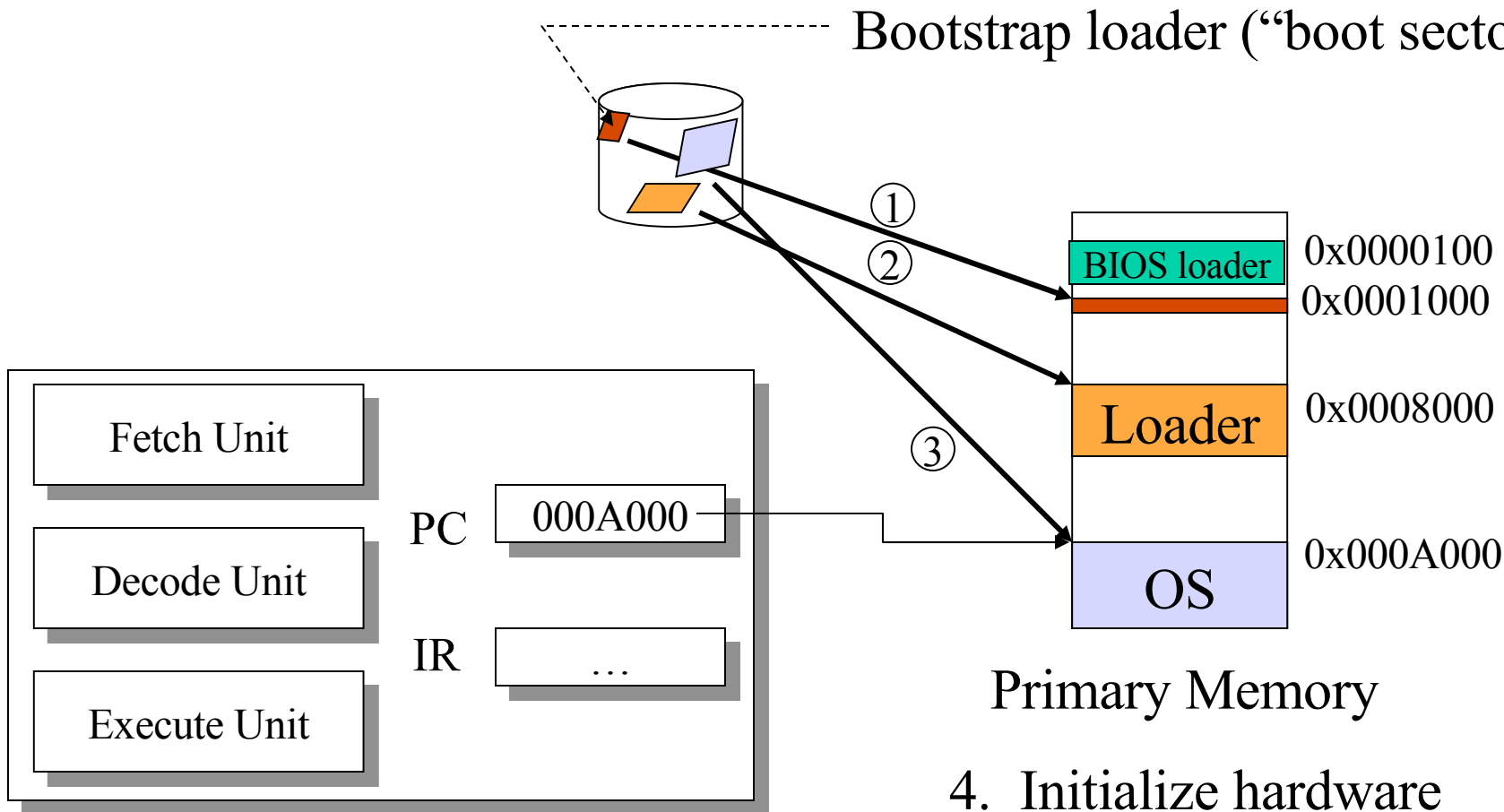


Bootstrapping

Bootstrap loader (“boot sector”)



Bootstrapping



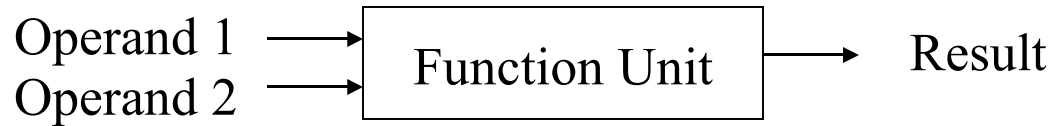
4. Initialize hardware
5. Create user environment
6. ...

A Bootstrap Loader Program

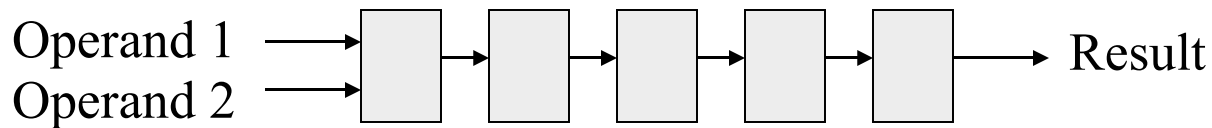


```
FIXED_LOC:                // Bootstrap loader entry point
    load  R1, =0
    load  R2, =LENGTH_OF_TARGET
// The next instruction is really more like
// a procedure call than a machine instruction
// It copies a block from FIXED_DISK_ADDRESS
// to BUFFER_ADDRESS
    read  BOOT_DISK, BUFFER_ADDRESS
loop: load  R3, [BUFFER_ADDRESS, R1]
    store R3, [FIXED_DEST, R1]
    incr  R1
    bleq  R1, R2, loop
    br    FIXED_DEST
```

A Pipelined Function Unit



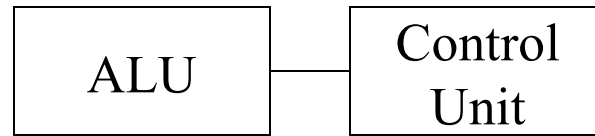
(a) Monolithic Unit



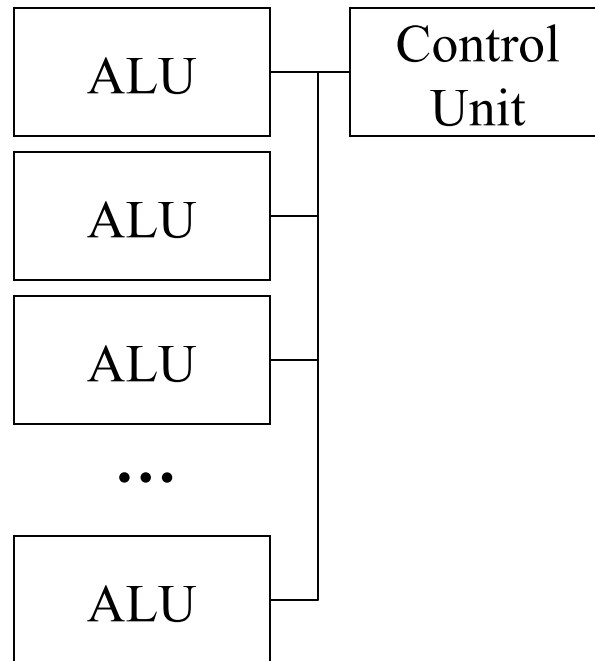
(b) Pipelined Unit



A SIMD Machine



(a) Conventional Architecture



(b) SIMD Architecture