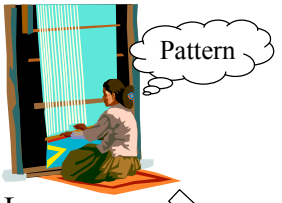
Slide 4-1

Computer Organization

Slide 4-2

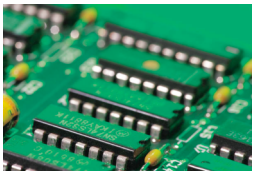
Stored Program Computers and Electronic Devices



Pattern

Jacquard Loom

Variable Program



Stored Program Device

Fixed Electronic Device

Program Specification

Slide 4-3

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

Slide 4-4

Assembly Language

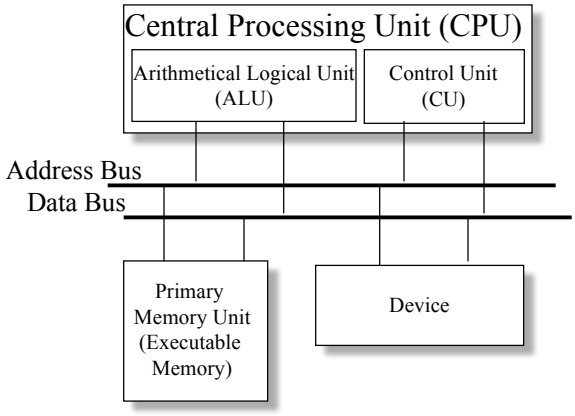
```
; Code for a = b + c
    load      R3,b
    load      R4,c
    add       R3,R4
    store     R3,a

; 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

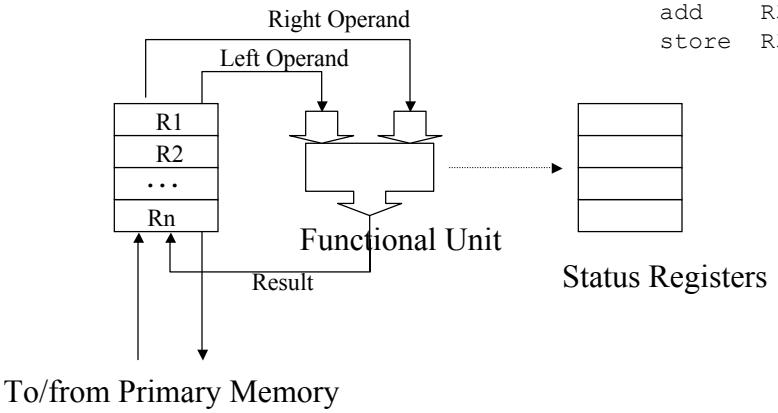
Slide 4-5



The ALU

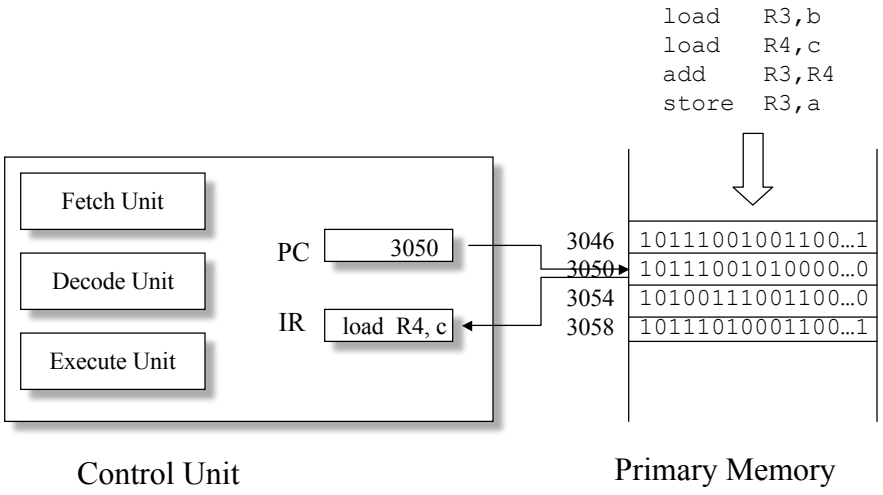
Slide 4-6

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



Control Unit

Slide 4-7



Control Unit Operation

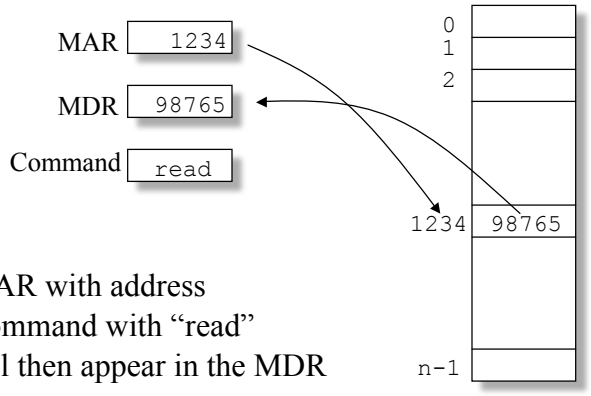
Slide 4-8

- *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

Slide 4-9

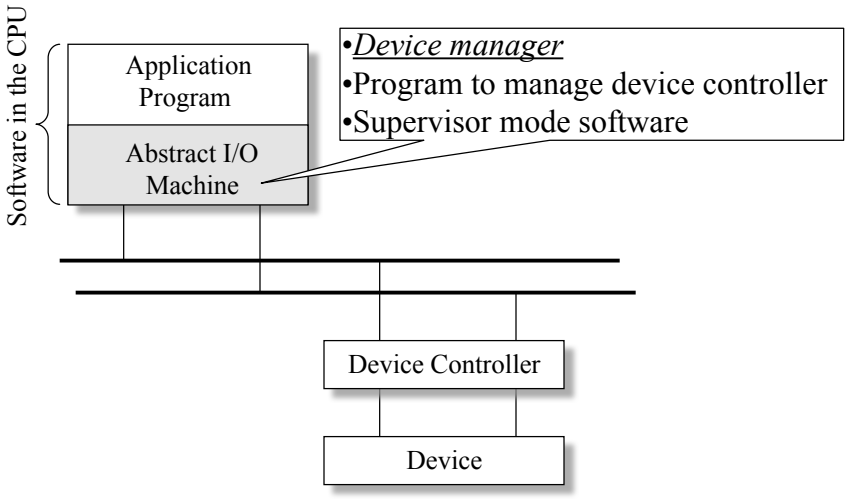


Read Op:

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

The Device-Controller-Software Relationship

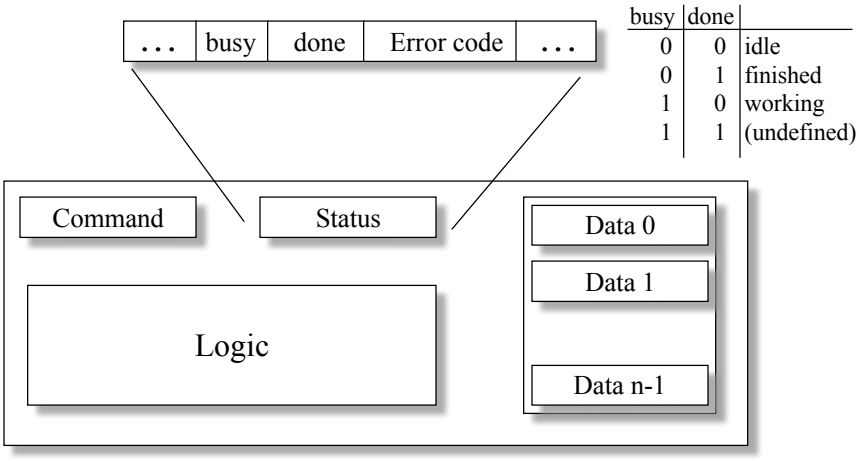
Slide 4-10



Device Controller Interface

Slide 4-11

Busy/done bits used to signal event occurrences to software and software to device



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

Performing a Write Operation

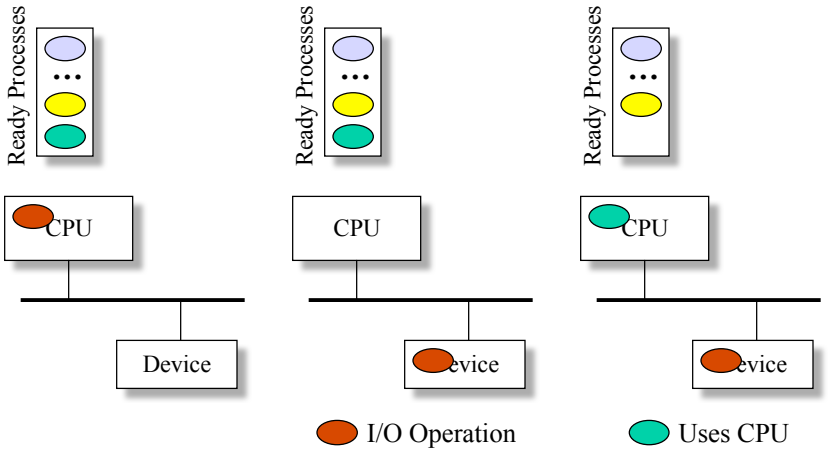
Slide 4-12

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

- 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

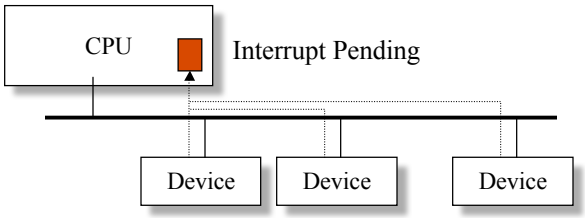
CPU-I/O Overlap

Slide 4-13



Slide 4-14

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

What happens if a second interrupt comes in the middle of the first?

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

Block other interrupt while processing first

```
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]
};
```

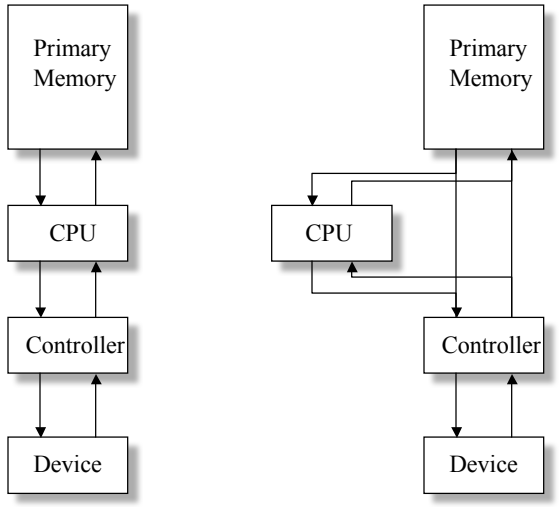

Revisiting the `trap` Instruction (Hardware)

```
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

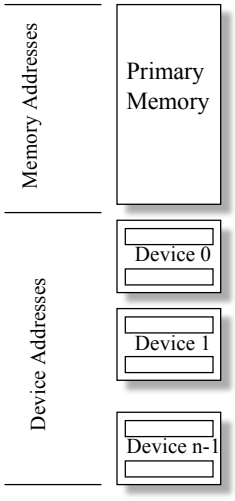
Slide 4-19



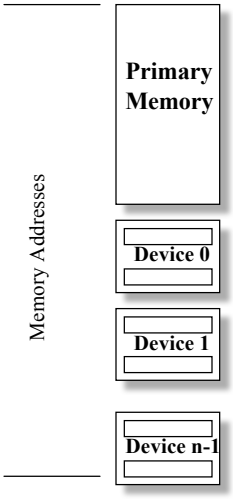
Addressing Devices

Slide 4-20

copy-in R3, 0x012, 4

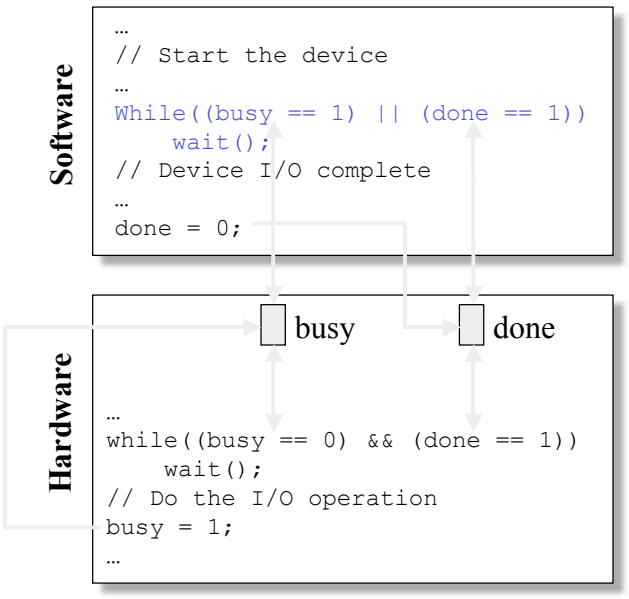


Load R3, 0xFFFF0124



Slide 4-21

Polling I/O



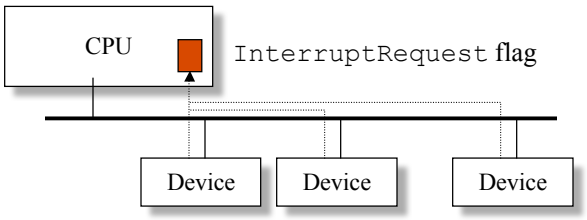
Slide 4-22

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

Slide 4-23



The Interrupt Handler

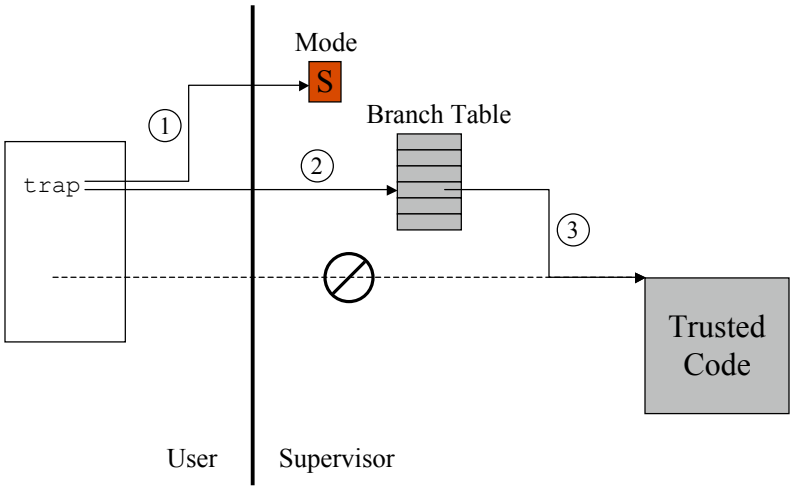
Slide 4-24

```
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

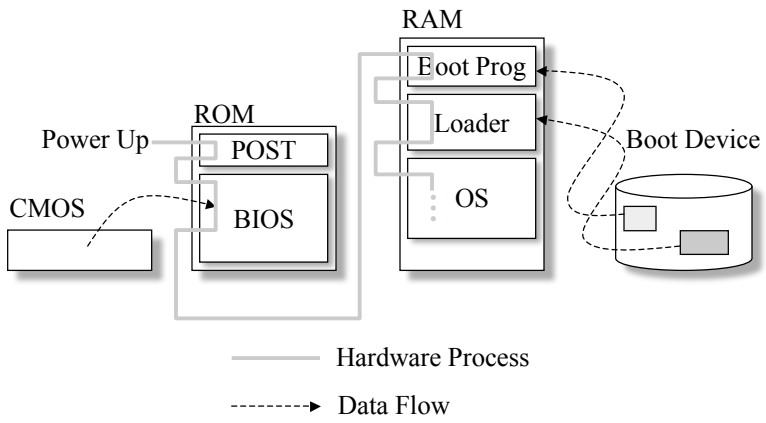

```
if(InterruptRequest && InterruptEnabled) {
/* Interrupt current process */
     disableInterrupts();
     memory[0] = PC;
     PC = memory[1];
}
```

The Trap Instruction Operation



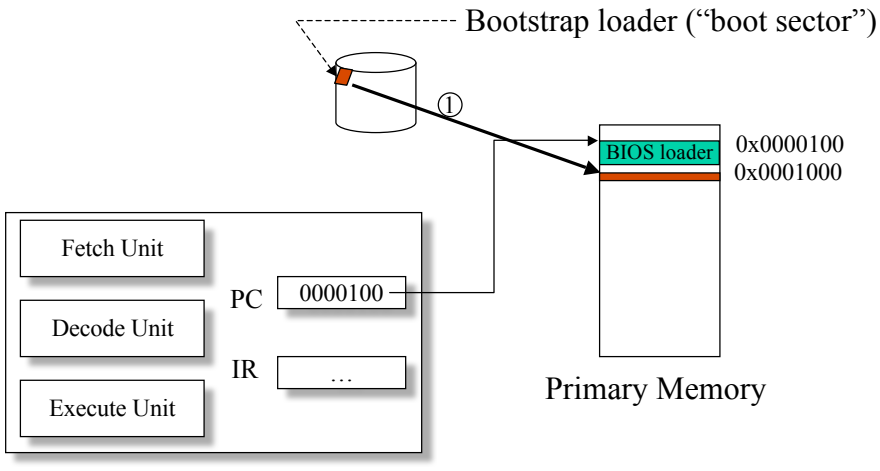
Intel System Initialization

Slide 4-27



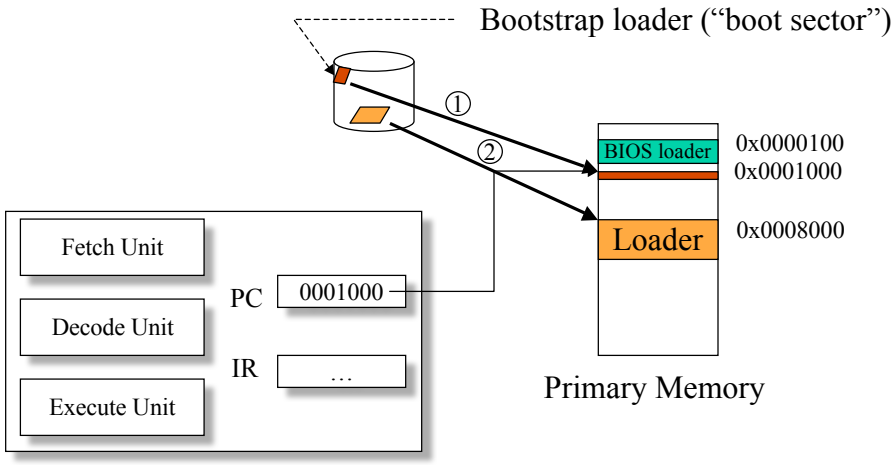
Bootstrapping

Slide 4-28



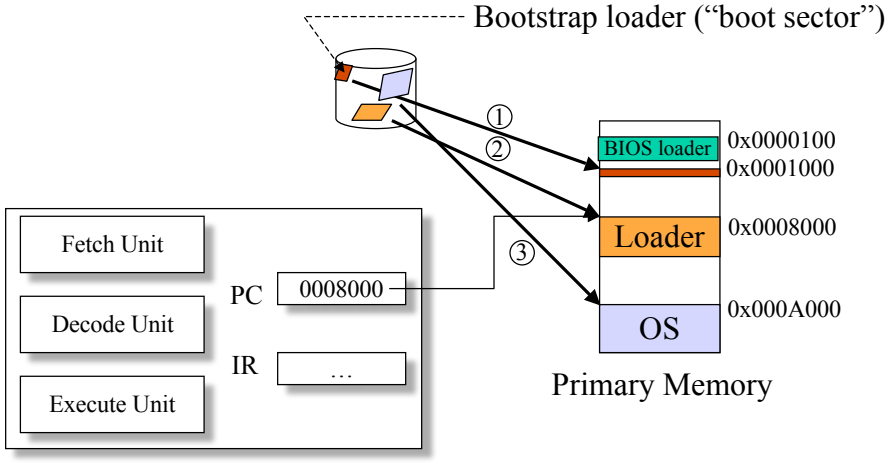
Slide 4-29

Bootstrapping



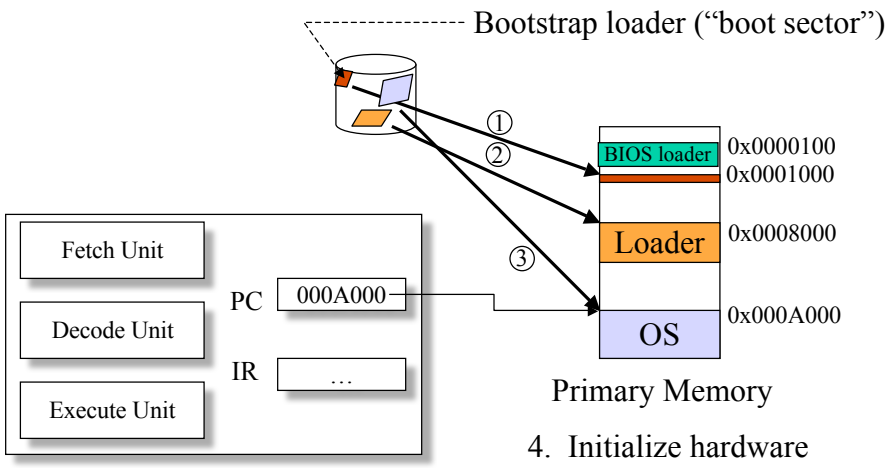
Slide 4-30

Bootstrapping



Bootstrapping

Slide 4-31



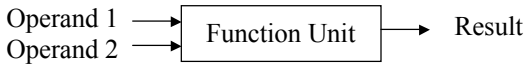
4. Initialize hardware
5. Create user environment
6. …

A Bootstrap Loader Program

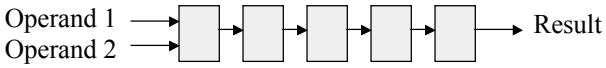
Slide 4-32

```
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

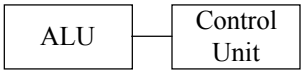


(a) Monolithic Unit

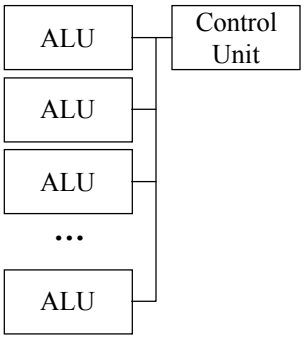


(b) Pipelined Unit

A SIMD Machine



(a) Conventional Architecture



(b) SIMD Architecture