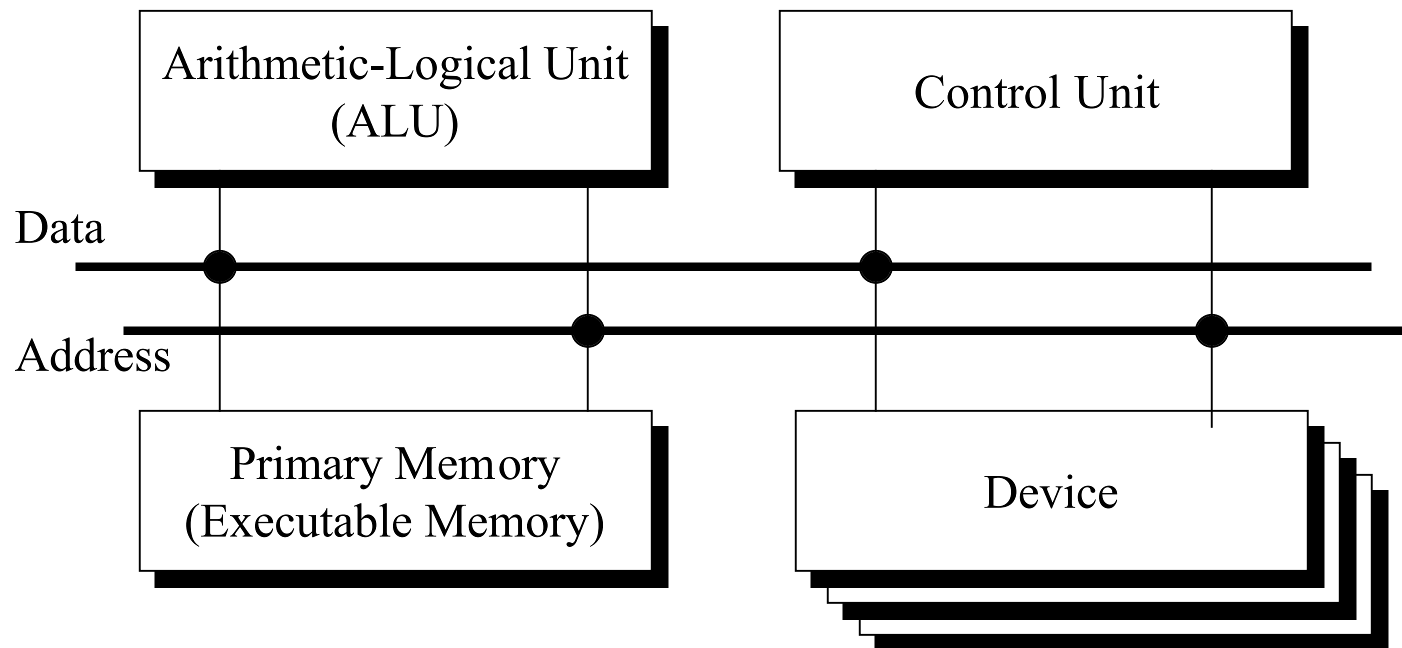
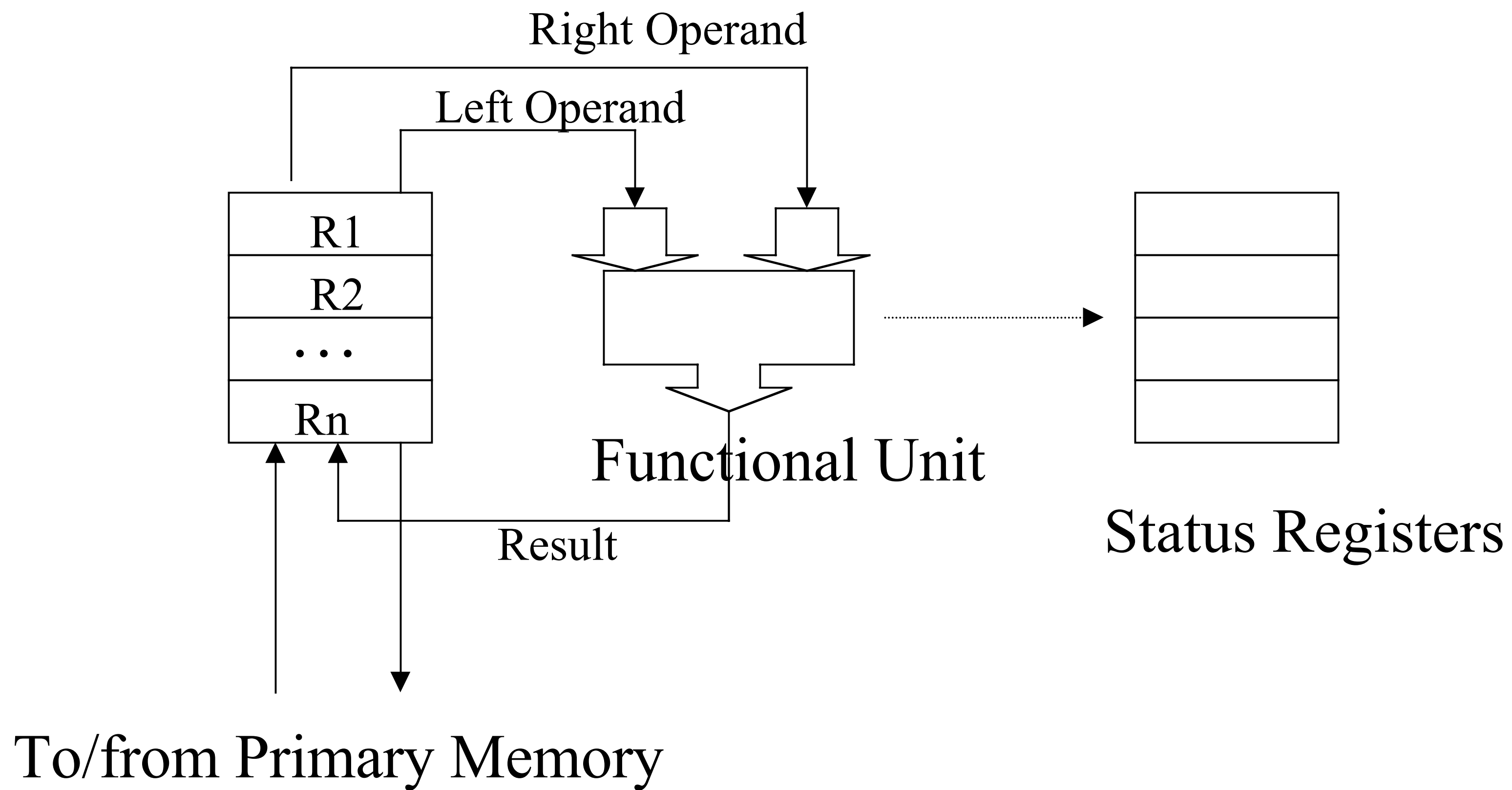
Computer Organization

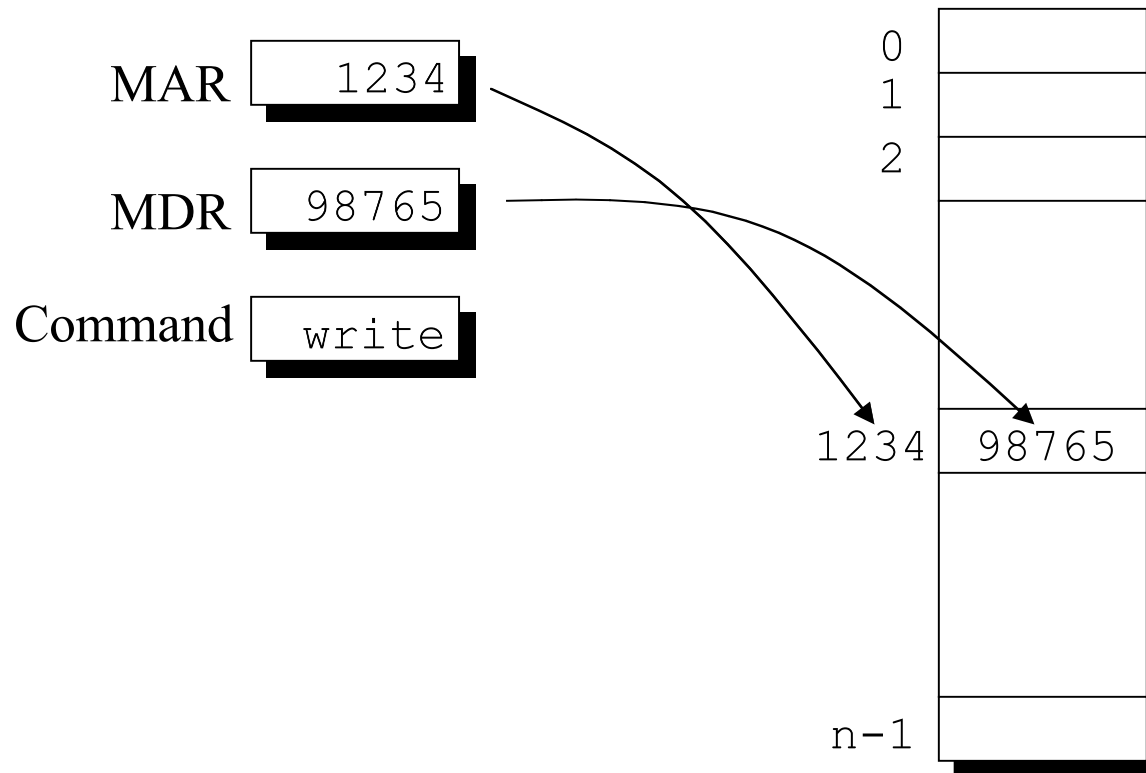
von Neumann Computer



The ALU



Memory Unit



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

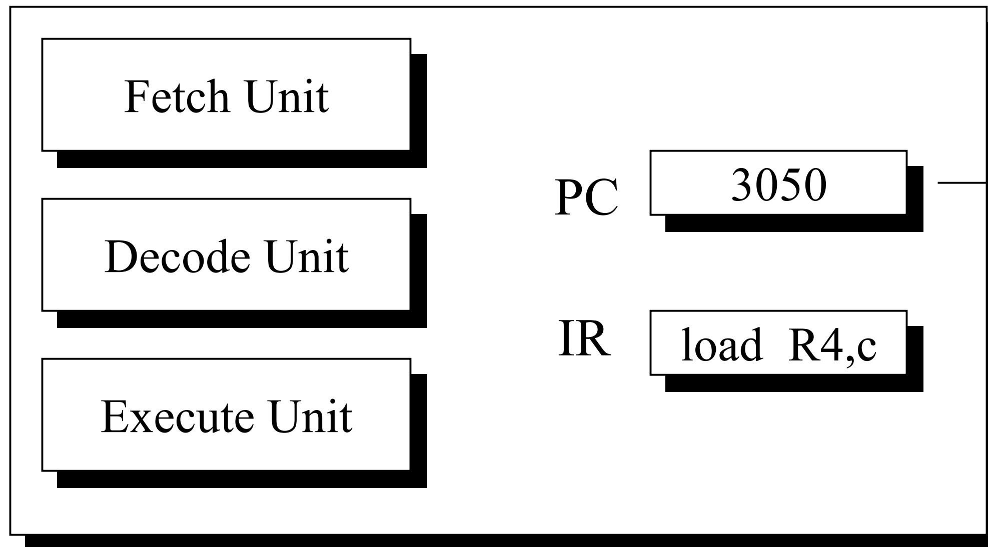
```
; Code for a = b + c
    load      R3,b
    load      R4,c
    add       R3,R4
    store     R3,a

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

Machine Language

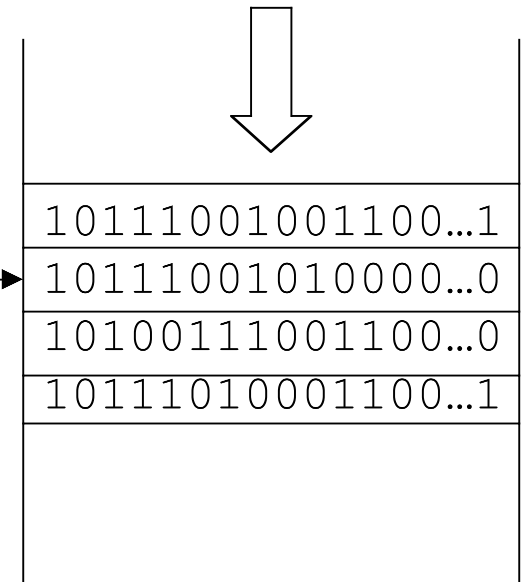
```
10111001001100…1
10111001010000…0
10100111001100…0
10111010001100…1
10111001010000…0
10100110001100…0
10111001101100…1
```

Control Unit

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



Control Unit



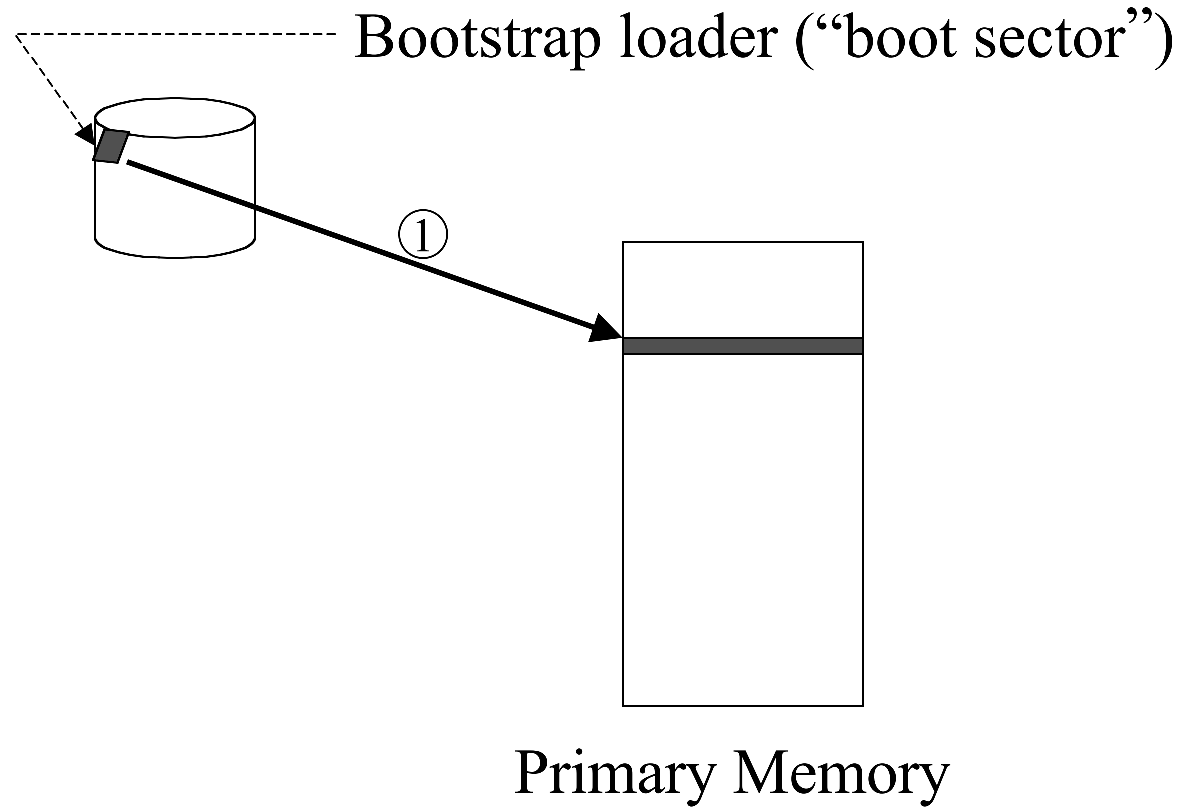
Primary Memory

Control Unit Operation

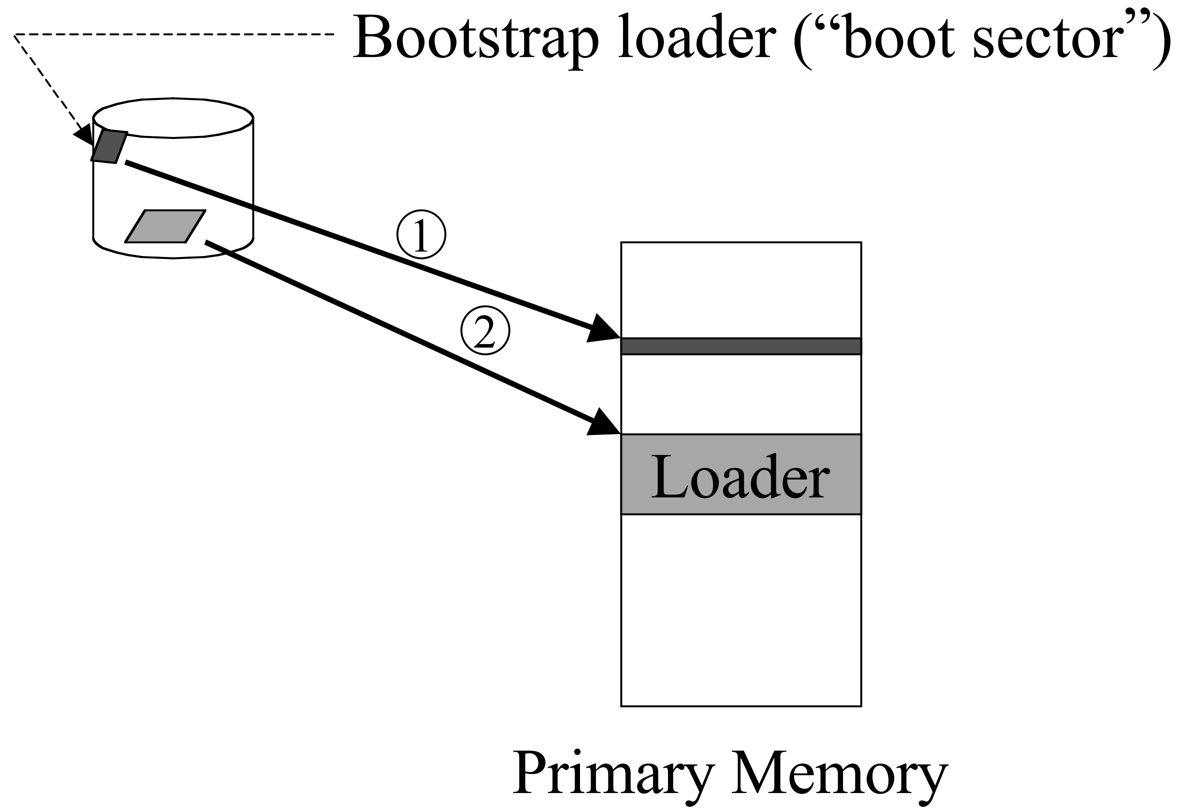
- *<u>Fetch phase</u>*: Instruction retrieved from memory
- *<u>Execute phase</u>*: ALU op, memory <u>data</u> reference, I/O, etc.

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

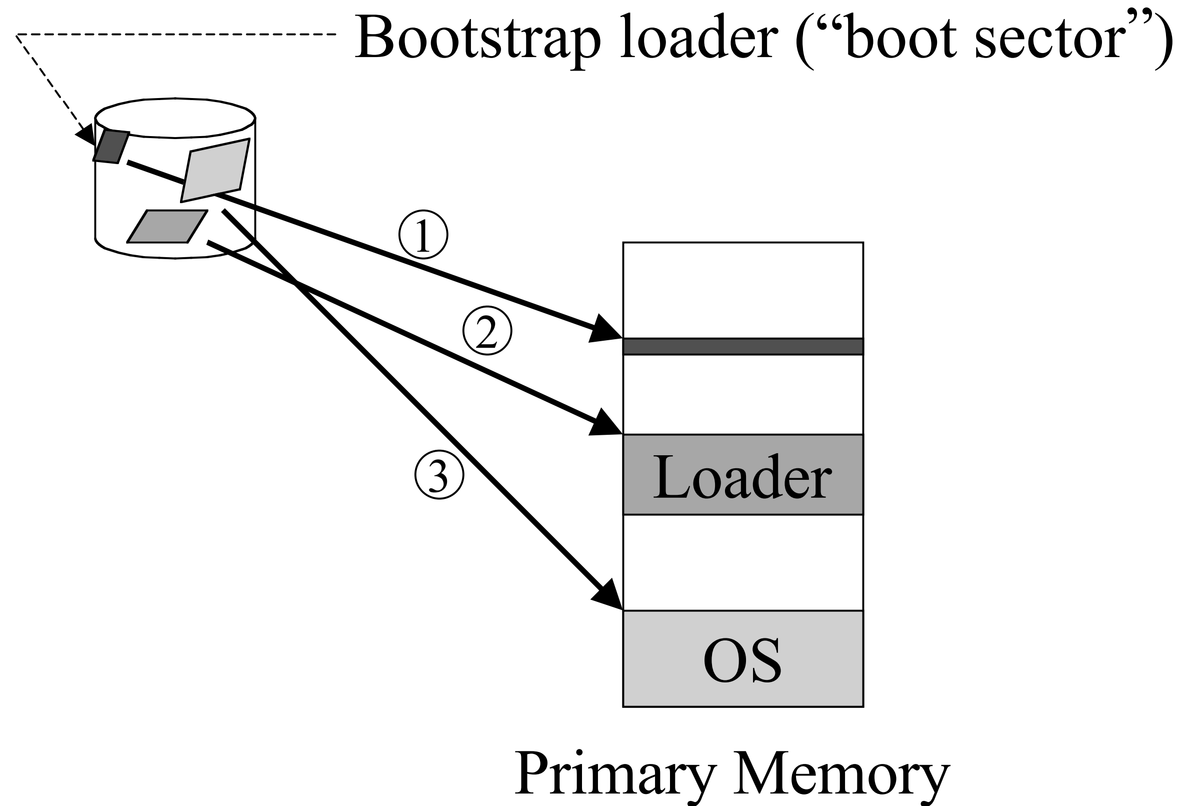
Bootstrapping



Bootstrapping

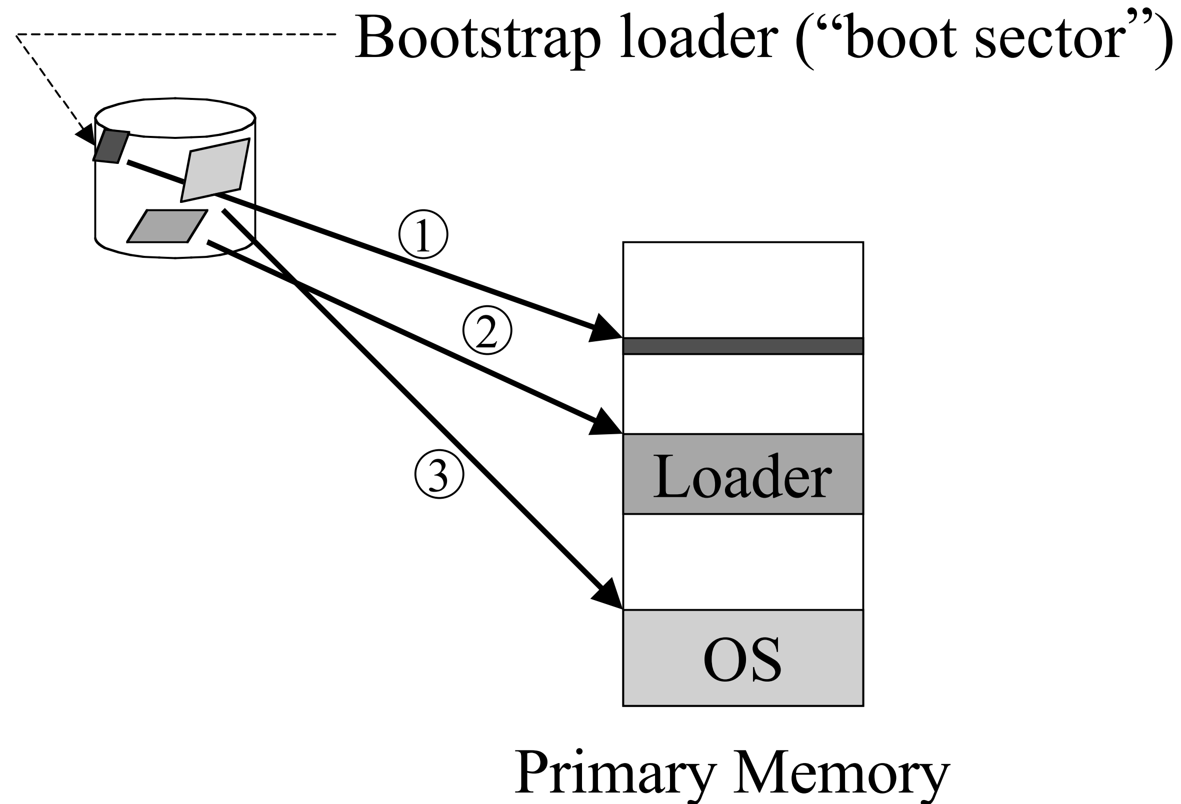


Bootstrapping



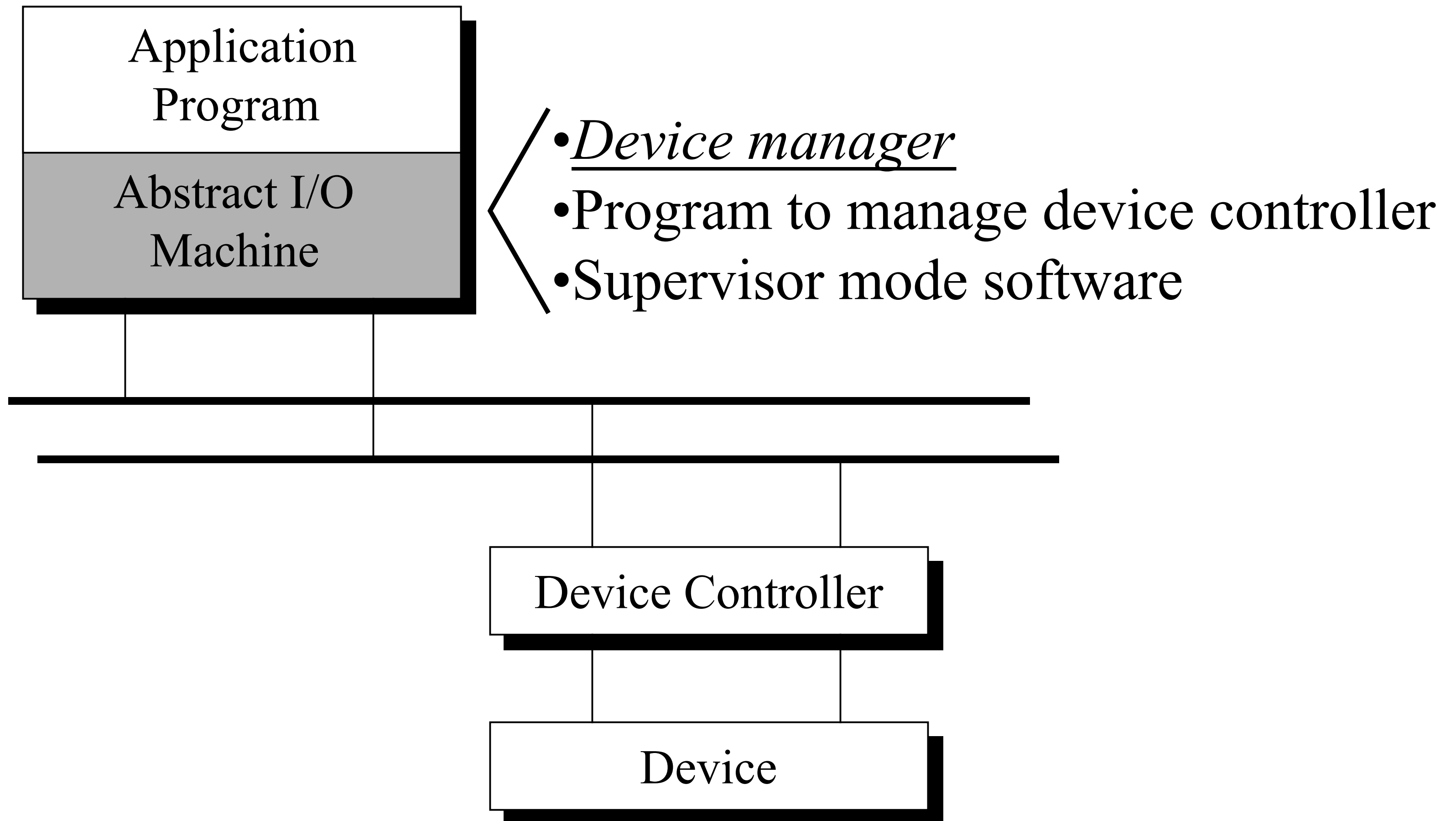
4. Initialize hardware
5. Create user environment
6. …

Bootstrapping

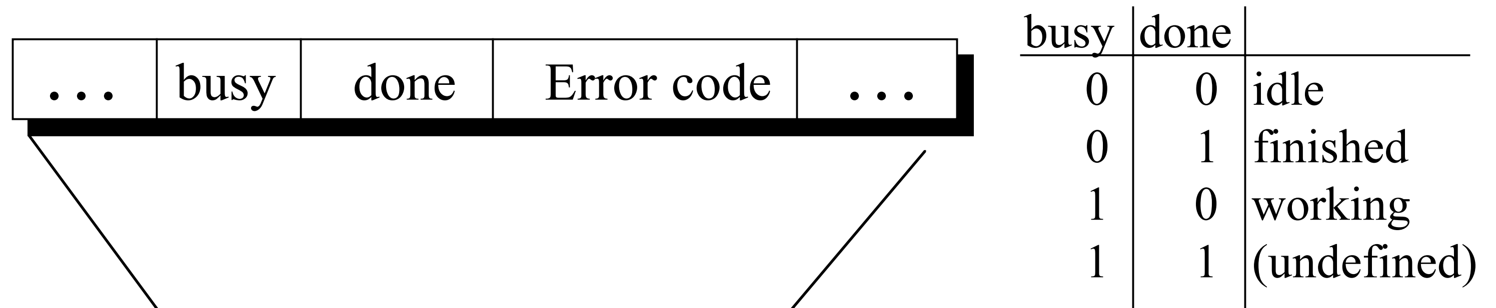


4. Initialize hardware
5. Create user environment
6. …

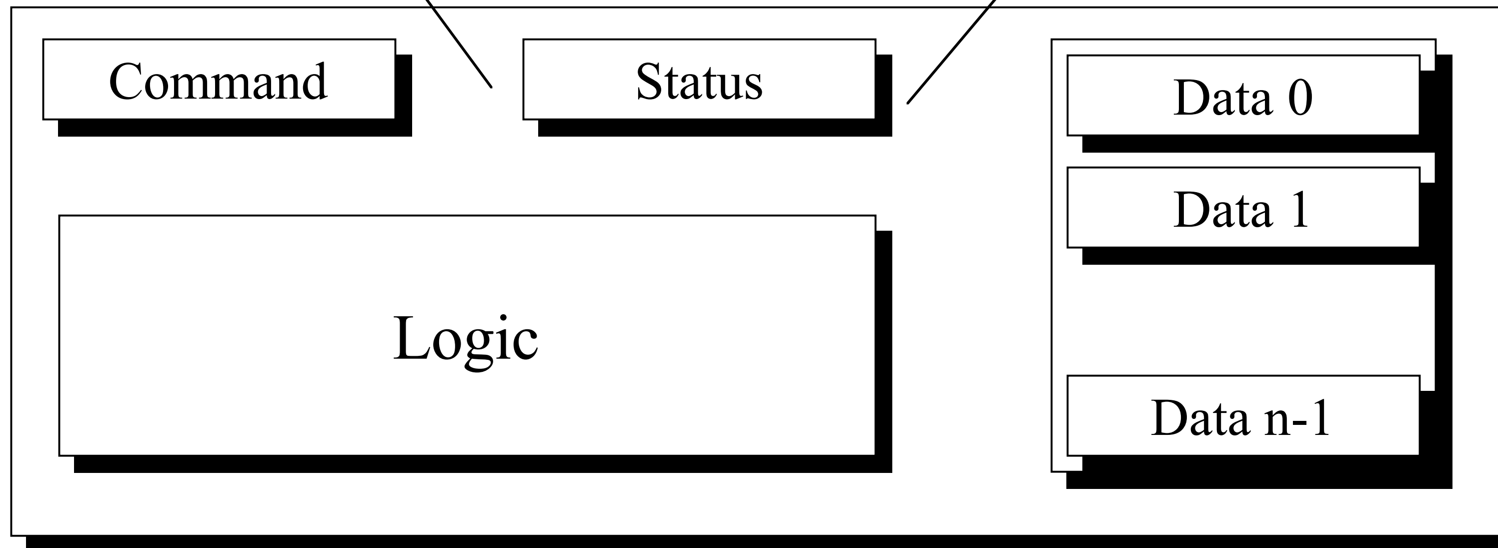
Device Organization



Device Controller Interface



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;
```

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

Control Unit with Interrupt

```
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]
};
```

Interrupt Handler

```
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();
    returnToProcess();
}
```

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]
            };
```

Ensuring that `trap` is Safe

```
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 routine atomically
- A trap handler performs desired processing
- "A trap is a software interrupt"