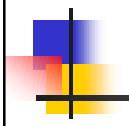
Chapter 4

Computer Organization



Von Neuman Concept

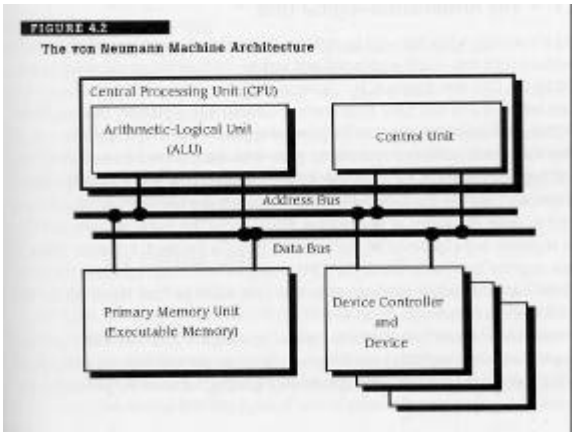
- Stored program concept
- General purpose computational device driven by internally stored program
- Data and instructions look same i.e. binary
- Operation being executed determined by HOW we look at the sequence of bits
  - Fetch
  - Decode
  - Execute

  } View bits as instruction

Data might be fetched as a result of execution

Von Neuman Architecture

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

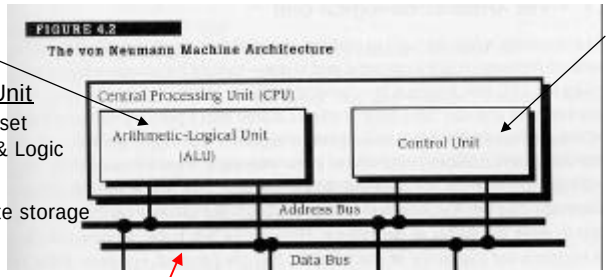


Von Neuman Machine Architecture

CPU = ALU + Cntrl Unit

ALU

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



Cntrl Unit

- fetch
- decode
- execute

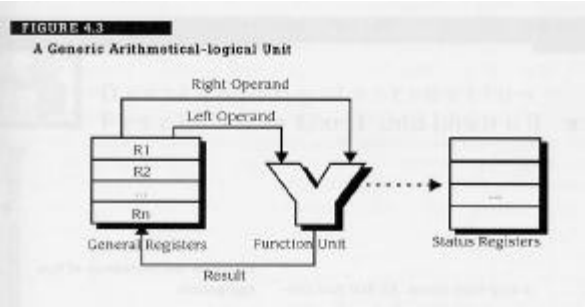
ALU

Buses

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

Von Nuemann Bottleneck

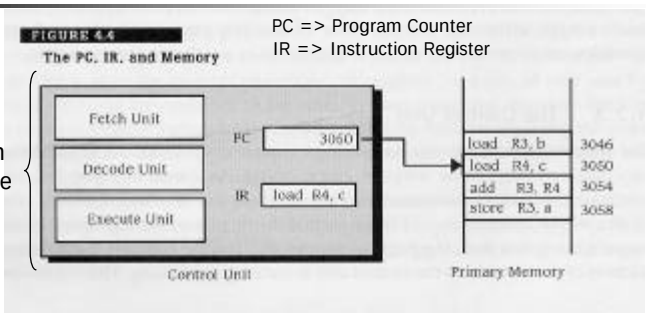
CPU: **ALU** Component



- 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

CPU: **Control Unit** Component

Von Nuemann 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

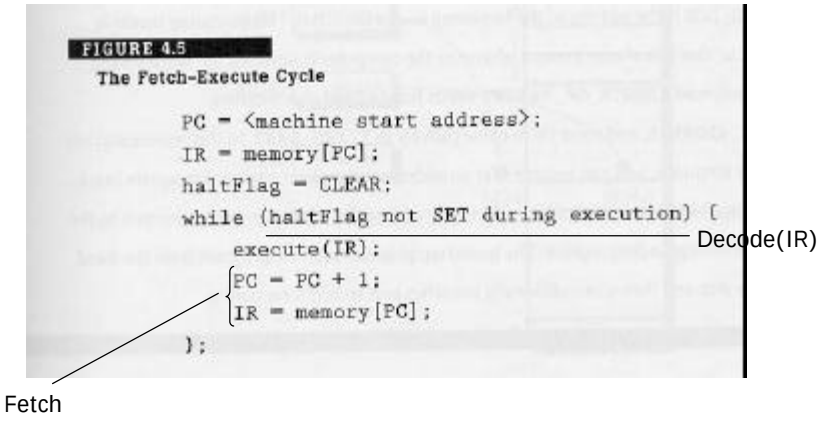


FIGURE 4.5
The Fetch-Execute Cycle

```
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

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

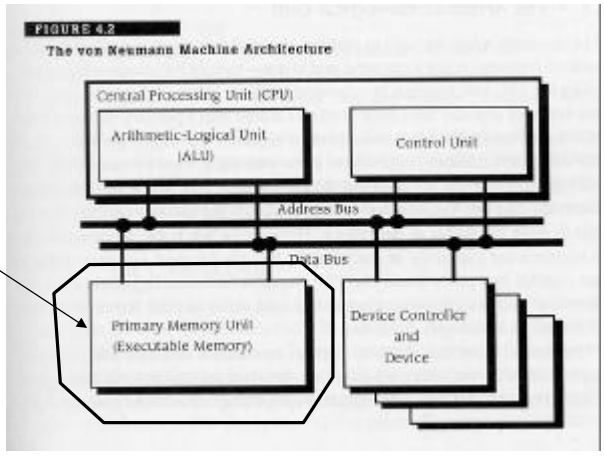
Decode

Execute

Branches to OS

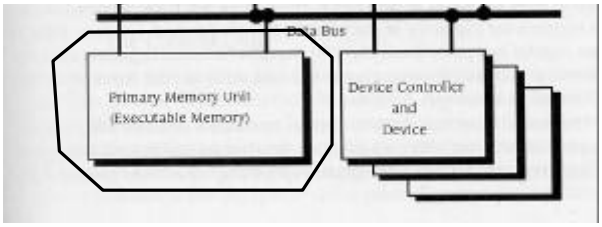
Memory Unit

Memory Unit



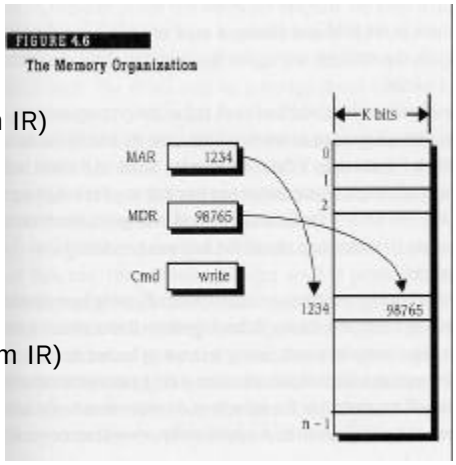
Memory Unit

- 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

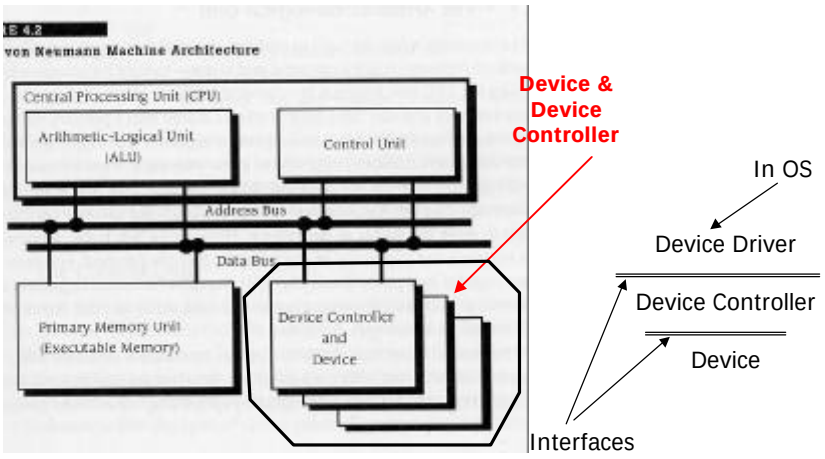


Memory Access

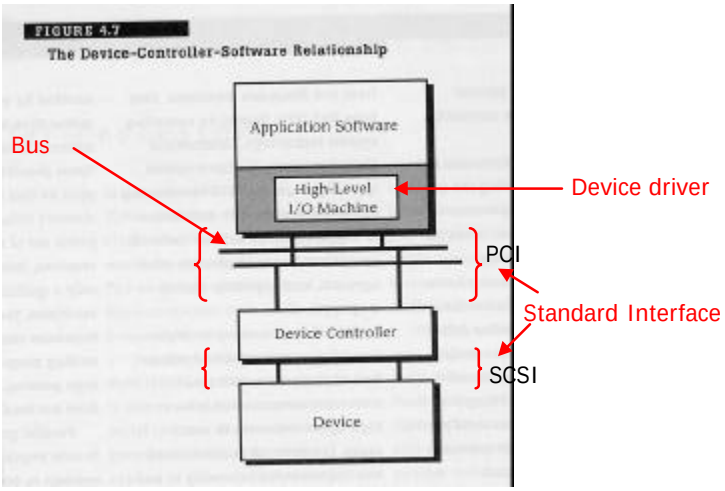
- Read from Memory
  - `MAR ← MemAddr`
  - `CMD ← 'Read OP'` (from IR)
  - Execute
    `MDR ← Mem[ MAR ]`
- Write to Memory
  - `MAR ← MemAddr`
  - `CMD ← 'Write OP'` (from IR)
  - Execute
    `Mem[ MAR ] ← MDR`



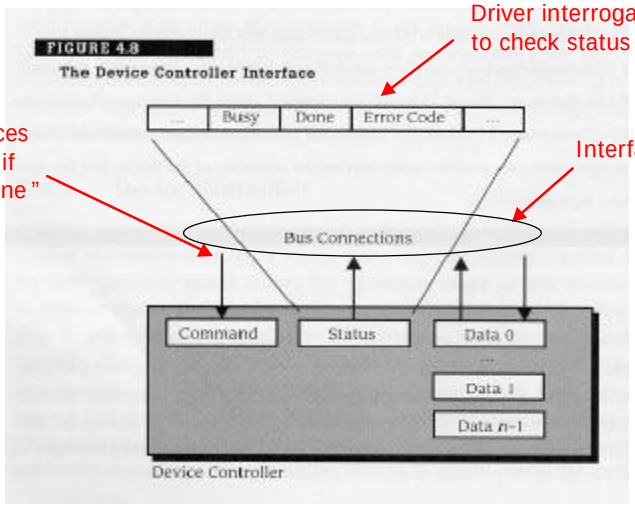
Device & Device Controller



Device Controller-Software Relationship

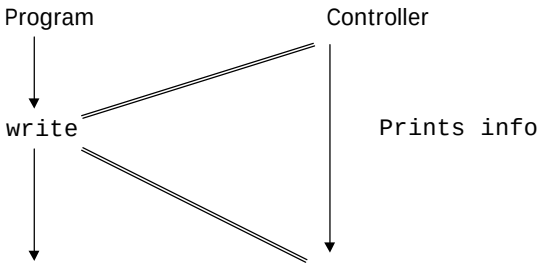


Device Controller Interface



Device Controller

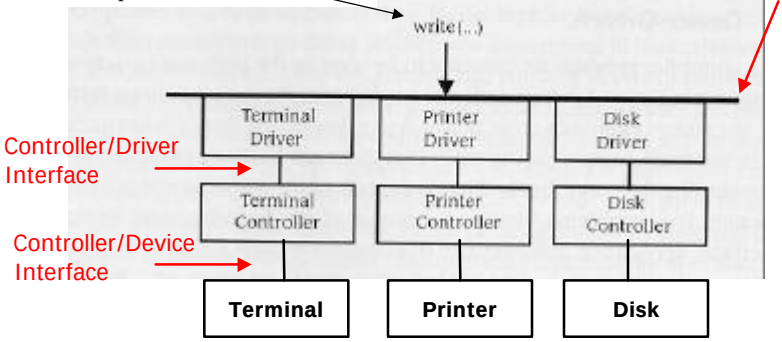
- Device controller is a processor and allows 2 parts of the process to proceed concurrently



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