Chapter 4

Computer Organization
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

**Data** might be fetched as a result of execution
Von Neumann Architecture

- CPU
  - ALU
  - Control Unit
- I/O Buses
- Memory Unit
- Devices
Von Neumann Machine Architecture

CPU = ALU + Cntrl Unit

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

Cntrl Unit
- fetch
- decode
- execute

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

Von Neumann 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 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];
}
```

Fetch

Decode(IR)
How does the system boot up?

- Bootstrap loader
- OS
- Application
A Bootstrap Loader

The power-up sequence

\[
\text{load } \text{PC}, \text{ FIXED\_LOC}
\]

Where FIXED\_LOC addresses the bootstrap loader (in ROM).

The bootstrap loader has the form:

\[
\begin{align*}
\text{load } R1, &= 0 \\
\text{load } R2, &= \text{LENGTH\_OF\_TARGET} \\
\text{loop: read } R1, \text{ FIXED\_DISK\_ADDRESS} \\
\text{store } R1, [\text{FIXED\_DEST}, R1] \\
\text{incr } R1 \\
\text{bleq } R1, R2, \text{ loop} \\
\text{br } \text{FIXED\_DEST}
\end{align*}
\]
Figure 4.2: The von Neumann Machine Architecture

Memory Unit

Central Processing Unit (CPU)
- Arithmetic-Logical Unit (ALU)
- Control Unit

Address Bus

Data Bus

Primary Memory Unit (Executable Memory)

Device Controller and Device
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

In OS

Device Driver

Device Controller

Device

Interfaces
Device Controller-Software Relationship

FIGURE 4.7
The Device-Controller-Software Relationship

Bus

Application Software

High-Level I/O Machine

Device Controller

Device

PCI

SCSI

Device driver

Standard Interface
Device Controller Interface

Driver places command if status "Done"

Driver interrogated these to check status of device

Interface to driver
Device Controller

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

Program  \[\text{write}\]  Controller

\[\text{Prints info}\]
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.
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 2\textsuperscript{nd} device) sends interrupt while the OS is handling prior interrupt?

If priority of 2nd interrupt higher than 1st then 1st interrupt suspended

\[ \text{Resumption of handling 1st interrupt} \]

\[ \text{2\textsuperscript{nd} interrupt handled} \]