Chapter 4

Computer Organization
Program Specification

Source

\[
\text{int } a, b, c, d; \\
\ldots \\
a = b + c; \\
d = a - 100; \\
\]

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 \( a = b + c \)
```assembly
load   R3, b
load   R4, c
add    R3, R4
store  R3, a
```

; Code for \( d = a - 100 \)
```assembly
load   R4, =100
subtract R3, R4
store  R3, d
```

**Machine Language**

<table>
<thead>
<tr>
<th>Code in Assembly Language</th>
<th>Machine Language</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>load R3, b</code></td>
<td>10111001001100...1</td>
</tr>
<tr>
<td><code>load R4, c</code></td>
<td>10111001010000...0</td>
</tr>
<tr>
<td><code>add R3, R4</code></td>
<td>10100111001100...0</td>
</tr>
<tr>
<td><code>store R3, a</code></td>
<td>10111010001100...1</td>
</tr>
<tr>
<td><code>load R4, =100</code></td>
<td>10111001010000...0</td>
</tr>
<tr>
<td><code>subtract R3, R4</code></td>
<td>10100111001100...0</td>
</tr>
<tr>
<td><code>store R3, d</code></td>
<td>10111001101100...1</td>
</tr>
</tbody>
</table>
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 \( \rightarrow \) View bits as instruction
  - Execute

**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

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

Address Bus
Data Bus

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
Program Specification (revisited)

Source
```c
int a, b, c, d;
...
a = b + c;
d = a - 100;
```

Assembly Language
```assembly
; 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
```
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]
}
OS boot-up...

- How does the system boot up?
  - Bootstrap loader
  - OS
  - Application
A Bootstrap Loader

The power-up sequence

load PC, FIXED_LOC

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]
icr R1
bleq R1, R2, loop
br FIXED_DEST

Address of BS Loader

Fetch
Reads OS in
Decode
Execute
Branches to OS

CS 3204 - Arthur 12
Memory Unit

FIGURE 4.2
The von Neumann Machine Architecture

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

Central Processing Unit (CPU)

Arithmetic-Logical Unit (ALU)

Control Unit

Address Bus

Data Bus

Primary Memory Unit (Executable Memory)

Device Controller and Device

Device & Device Controller

In OS

Device Driver

Device Controller

Interfaces
Device Controller-Software Relationship

![Diagram of Device-Controller-Software Relationship]

- Bus
- Application Software
- High-Level I/O Machine
- Device driver
- PCI
- Standard Interface
- SCSI
- Device Controller
- Device

FIGURE 4.7
The Device-Controller-Software Relationship
Device Controller Interface

Driver places command if status “Done”

Driver interrogated these to check status of device

Interface to driver

---

CS 3204 - Arthur
Device Controller

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

```
Program
  ↓
write
  ↓

Controller
  ↓
  Prints info
```
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

---

CS 3204 - Arthur
How do interrupts factor in?

- **Scenario (1)**
  - Program:
    ```c
    while device_flag busy {}
    ```
  - => Busy wait - consumes CPU cycles

- **Scenario (2)**
  - Program:
    ```c
    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 2\textsuperscript{nd} interrupt higher than 1\textsuperscript{st} then 1\textsuperscript{st} interrupt suspended

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

\[ \rightarrow \] Resumption of handling 1\textsuperscript{st} interrupt
Control Unit with Interrupt (H/W)

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
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();
}
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;
        memory[1] = PC;
        PC = memory[1];
    }
};
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”