

Chapter 4

Computer Organization

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
```

CS 3204 - Arthur

2

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
```

CS 3204 - Arthur

3

Von Neumann Concept

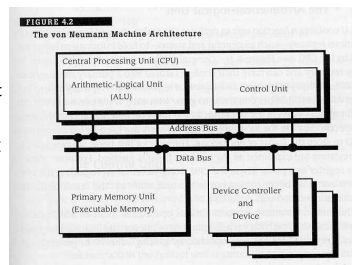
- Stored program concept
 - General purpose computational device driven by internally stored program
 - Data and instructions look same i.e. binary
 - Operation being executed determined by HOW we look at the sequence of bits
 - Fetch
 - Decode
 - Execute
- View bits as instruction
- Data might be fetched as a result of execution

CS 3204 - Arthur

4

Von Neumann Architecture

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

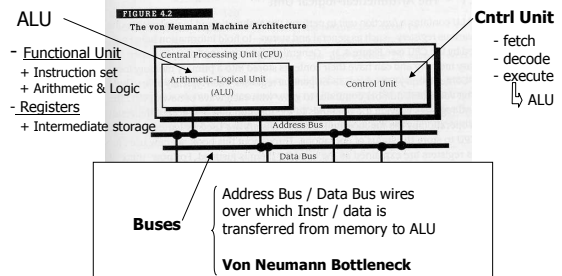


CS 3204 - Arthur

5

Von Neumann Machine Architecture

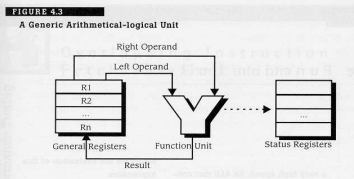
CPU = ALU + Cntrl Unit



CS 3204 - Arthur

6

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

CS 3204 - Arthur

7

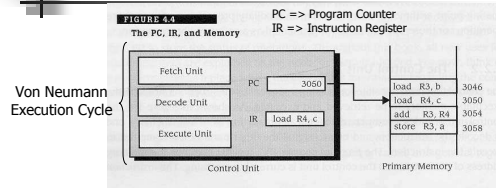
Program Specification (revisited)

Source	Assembly Language
<code>int a, b, c, d;</code>	
<code>...</code>	
<code>a = b + c;</code>	<code>; Code for a = b + c</code>
	<code>load R3, b</code>
	<code>load R4, c</code>
	<code>add R3, R4</code>
	<code>store R3, a</code>
<code>d = a - 100;</code>	<code>; Code for d = a - 100</code>
	<code>load R4, =100</code>
	<code>subtract R3, R4</code>
	<code>store R3, d</code>

CS 3204 - Arthur

8

CPU: Control Unit Component



- 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)

CS 3204 - Arthur

9

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)

CS 3204 - Arthur

10

OS boot-up...

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

CS 3204 - Arthur

11

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

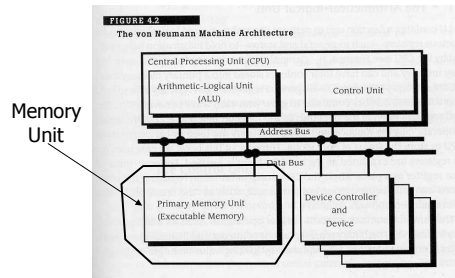
Fetch
Decode
Execute

Branches to OS

CS 3204 - Arthur

12

Memory Unit



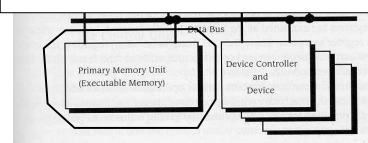
CS 3204 - Arthur

13

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



CS 3204 - Arthur

14

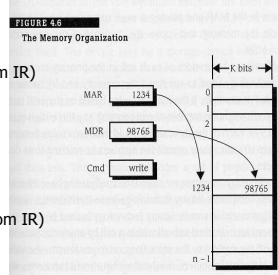
Memory Access

Read from Memory

- MAR \leftarrow MemAddr
- CMD \leftarrow 'Read OP' (from IR)
- Execute
 - MDR \leftarrow Mem[MAR]

Write to Memory

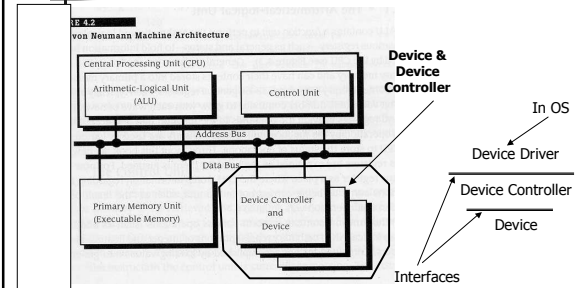
- MAR \leftarrow MemAddr
- CMD \leftarrow 'Write OP' (from IR)
- Execute
 - Mem[MAR] \leftarrow MDR



CS 3204 - Arthur

15

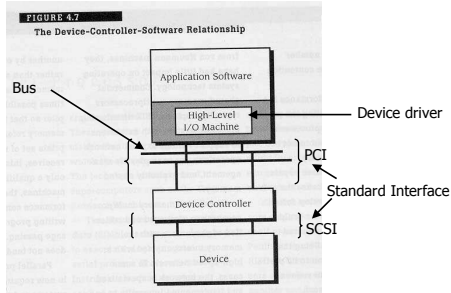
Device & Device Controller



CS 3204 - Arthur

16

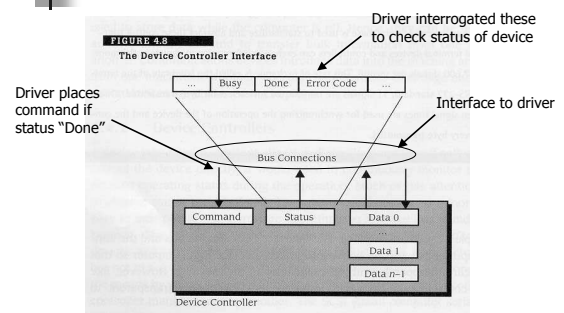
Device Controller-Software Relationship



CS 3204 - Arthur

17

Device Controller Interface

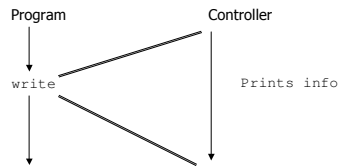


CS 3204 - Arthur

18

Device Controller

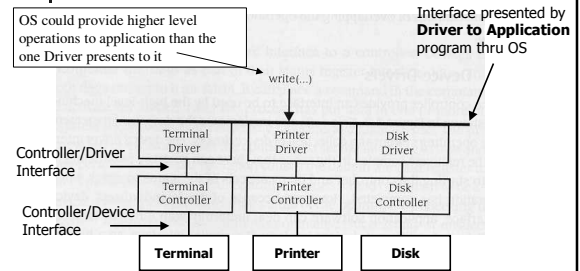
- Device controller is a processor and allows 2 parts of the process to proceed concurrently



CS 3204 - Arthur

19

Device Driver Interface



CS 3204 - Arthur

20

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

CS 3204 - Arthur

21

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

CS 3204 - Arthur

22

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

CS 3204 - Arthur

23

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

CS 3204 - Arthur

24

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

CS 3204 - Arthur

25

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();
    }
}
```

CS 3204 - Arthur

26

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) {
        disableInterrupts();
        memory[0] = PC;
        PC = memory[1]
    }
};
```

CS 3204 - Arthur

27

Revisiting the trap Instruction (H/W)

```
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"

CS 3204 - Arthur

28