Computer Organization

Stored Program Computers and Electronic Devices

Jacquard Loom

Fixed Electronic Device

Variable Program

Pattern

Stored Program Device
Program Specification

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
```

Machine 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
```

```machine_language
101110010011001
101110010100000
101001110011001
101110100011001
101110011011001
101110010011001
101110100011001
101110010100000
101001110011001
101110011011001
101110010011001
```
The von Neumann Architecture

Central Processing Unit (CPU)

Arithmetical Logical Unit (ALU)  Control Unit (CU)

Address Bus

Data Bus

Primary Memory Unit (Executable Memory)

Device

The ALU

load  R3, b
load  R4, c
add   R3, R4
store R3, a
Control Unit

- Fetch Unit
- Decode Unit
- Execute Unit

PC = 3050
IR = 3046

Primary Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>3050</td>
<td>load R3, b</td>
</tr>
<tr>
<td>3054</td>
<td>load R4, c</td>
</tr>
<tr>
<td>3058</td>
<td>add R3, R4</td>
</tr>
<tr>
<td>3062</td>
<td>store R3, a</td>
</tr>
</tbody>
</table>

Control Unit Operation

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

```c
PC = <machine start address>;
IR = memory[PC];
haltFlag = CLEAR;
while(haltFlag not SET) {
    execute(IR);
    PC = PC + sizeof(INSTRUCT);
    IR = memory[PC]; // fetch phase
};
```
Primary Memory Unit

Read Op:
1. Load MAR with address
2. Load Command with “read”
3. Data will then appear in the MDR

The Device-Controller-Software Relationship

• *Device manager*
• Program to manage device controller
• Supervisor mode software
Device Controller Interface

Busy/done bits used to signal event occurrences to software and software to device

<table>
<thead>
<tr>
<th>busy</th>
<th>done</th>
<th>Error code</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>idle</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>finished</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>working</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>(undefined)</td>
<td></td>
</tr>
</tbody>
</table>

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;
```

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

- CPU
- Device
- I/O Operation
- CPU
- Device
- Uses CPU

Determining When I/O is Complete

- CPU incorporates an “interrupt pending” flag
- When device.busy → FALSE, interrupt pending flag is set
- Hardware “tells” OS that the interrupt occurred
- *Interrupt handler* part of the OS makes process ready to run
Control Unit with Interrupt (Hardware)

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

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();  
  }
A Race Condition

What happens if a second interrupt comes in the middle of the first?

```c
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]
    }
}

Block other interrupt while processing first

Revisiting the trap Instruction (Hardware)

```c
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”
Direct Memory Access

Addressing Devices

copy-in R3, 0x012, 4

Load R3, 0xFFFF0124
Polling I/O

... // Start the device...
... While((busy == 1) || (done == 1))
    wait();
// Device I/O complete...
... done = 0;

... while((busy == 0) && (done == 1))
    wait();
// Do the I/O operation
    busy = 1;

Fetch-Execute Cycle with an Interrupt

while (haltFlag not set during execution) {
    IR = memory[PC];
    PC = PC + 1;
    execute(IR);
    if (InterruptRequest) {
        /* Interrupt the current process */
        /* Save the current PC in address 0 */
        memory[0] = PC;
        /* Branch indirect through address 1 */
        PC = memory[1];
    }
}
Detecting an Interrupt

The Interrupt Handler

```
Interrupt_Handler{
    saveProcessorState();
    for (i=0; i<Number_of_devices; i++)
        if (device[i].done == 1)
            goto device_handler(i);
/* Something wrong if we get here */
}
```
Disabling Interrupts

```c
if(InterruptRequest && InterruptEnabled) {
    /* Interrupt current process */
    disableInterrupts();
    memory[0] = PC;
    PC = memory[1];
}
```

The Trap Instruction Operation

```
 trap
  Mode
       \__\_
      ①  ②  ③
       /
     Branch Table

User
Supervisor
```

Trusted Code
Intel System Initialization

Bootstrapping

Hardware Process
Data Flow

Bootstrapping

Bootstrap loader ("boot sector")

Fetch Unit
Decode Unit
Execute Unit

Primary Memory

BIOS loader
0x0000100
0x0001000

BIOS loader

0000100

0000100
Bootstrapping

Bootstrap loader ("boot sector")

Primary Memory

Fetch Unit
Decode Unit
Execute Unit

0x0001000
0x0008000
0x000A000

0x0000100
0x0001000
0x0008000

BIOS loader
Loader
OS
Bootstrapping

Bootstrap loader ("boot sector")

1. BIOS loader
2. 0x0000100
3. 0x0001000
4. 0x0008000
5. 0x000A000

Primary Memory

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

A Bootstrap Loader Program

FIXED_LOC: // Bootstrap loader entry point
    load R1, =0
    load R2, =LENGTH_OF_TARGET
// The next instruction is really more like
// a procedure call than a machine instruction
// It copies a block from FIXED_DISK_ADDRESS
// to BUFFER_ADDRESS
    read BOOT_DISK, BUFFER_ADDRESS
loop:    load R3, [BUFFER_ADDRESS, R1]
         store R3, [FIXED_DEST, R1]
         incr R1
         bqueen R1, R2, loop
    br FIXED_DEST
A Pipelined Function Unit

Operand 1 → Function Unit → Result
Operand 2

(a) Monolithic Unit

Operand 1 → Function Unit → Result
Operand 2

(b) Pipelined Unit

A SIMD Machine

ALU → Control Unit

(a) Conventional Architecture

ALU
ALU
ALU
ALU
ALU

(b) SIMD Architecture