Stored Program Computers and Electronic Devices

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

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

The von Neumann Architecture

The ALU

Control Unit

load R3, a
load R4, c
add R3, R4
store R3, a

Primary Memory

Control Unit

Fetch Unit
PC
IR
Load Unit
Store Unit
Control Unit Operation

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

```
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

```
MAR | 0
---|---
MDR | 0
Command | read
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

The Device-Controller-Software Relationship

- Device manager
- Program to manage device controller
- Supervisor mode software

Device Controller Interface

Performing a Write Operation

```
while(deviceNo.busy || deviceNo.done) <waiting>
deviceNo.data[0] = <value to 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

```

```

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)

```cpp
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)

```cpp
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

```cpp
saveProcessorState() {  
  for(i=0; i<NumberOfRegisters; i++)  
    memory[K+i] = R[i];  
  for(i=0; i<NumberOfStatusRegisters; i++)  
    memory[K+NumberOfRegisters+i] = StatusRegister[i];  
};
```

Revisiting the `trap` Instruction (Hardware)

```cpp
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

```plaintext
Primary Memory  
CPU  
Controller  
Device  
```

```plaintext
Primary Memory  
CPU  
Controller  
Device  
```
Addressing Devices

Polling I/O

// Start the device
while((busy == 1) || (done == 1))
    wait();

// Device I/O complete
done = 0;

while((busy == 0) && (done == 1))
    wait();

busy = 1;

Software

Hardware

while(busy == 1) 
    wait();

// Do the I/O operation
busy = 1;

while(busy == 0) 
    wait();

// Device I/O complete
done = 0;

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

if(InterruptRequest && InterruptEnabled) {
    /* Interrupt current process */
    disableInterrupts();
    memory[0] = PC;
    PC = memory[1];
}
The Trap Instruction Operation

Intel System Initialization

Bootstrapping

1. The Trap Instruction Operation
2. Mode
3. Trusted Code

1. Intel System Initialization
2. ROM
3. CMOS
4. RAM
5. Boot Device
6. POST
7. BIOS
8. Boot Prog
9. Boot Loader
10. OS
11. …
12. Hardware Process
13. Data Flow
14. Power Up
15. BIOS loader
16. 0x0000100
17. 0x0001000
18. 0x0008000
19. 0x000A000

1. Bootstrapping
2. Bootstrap loader (“boot sector”)
3. Primary Memory
4. Fetch Unit
5. Decode Unit
6. Execute Unit
7. PC
8. IR
9. OS
10. …
11. 0x0000000
12. 0x0008000
13. 0x000A000

1. Bootstrapping
2. Bootstrap loader (“boot sector”)
3. Fetch Unit
4. Decode Unit
5. Execute Unit
6. PC
7. IR
8. OS
9. …
10. 0x0000000
11. 0x0008000
12. 0x000A000

1. Bootstrapping
2. Bootstrap loader (“boot sector”)
3. Fetch Unit
4. Decode Unit
5. Execute Unit
6. PC
7. IR
8. OS
9. …
10. 0x0000000
11. 0x0008000
12. 0x000A000

1. Bootstrapping
2. Bootstrap loader (“boot sector”)
3. Fetch Unit
4. Decode Unit
5. Execute Unit
6. PC
7. IR
8. OS
9. …
10. 0x0000000
11. 0x0008000
12. 0x000A000

1. Bootstrapping
2. Bootstrap loader (“boot sector”)
3. Fetch Unit
4. Decode Unit
5. Execute Unit
6. PC
7. IR
8. OS
9. …
10. 0x0000000
11. 0x0008000
12. 0x000A000

4. Initialize hardware
5. Create user environment
6. …
A Bootstrap Loader Program

```plaintext
FIXED_LOC: // Bootstrap loader entry point
    load R1, =8
    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
    bleq R1, R2, loop
    br FIXED_DEST
```

A Pipelined Function Unit

```
Operand 1 --> Function Unit --> Result
Operand 2

(a) Monolithic Unit
```

(b) Pipelined Unit

A SIMD Machine

```
ALU
Control
Unit

(a) Conventional Architecture
```
```
ALU
ALU
ALU
...
ALU

(b) SIMD Architecture
```

Fun New Stuff!

- You are sitting on the cusp of a grand new frontier!
  - UNI Processors
  - SMP design
  - Hyperthreading
  - Multicore

UNI Processor

- UNI processor == one processor
- Standard Von Neuman architecture holds

SMP Systems

- SMP stands for “Symmetric Multi Processor”
- Symmetric means that all processors have the same number of registers, L1/L2 cache, and so on.
- Think for a second...how would you schedule jobs on an SMP system – Or at least think: How would scheduling on an SMP system be different than on a UNI-processor system?
Hyperthreading

- Hyperthreading is “Hype”
- Hyperthreading is a marketing gimmick
- On high performance systems and today's top clusters, no one in their right minds enables hyperthreading. It's just stupid.
- Hyperthreading gives a virtual dual-pipeline channel that can double-up the pipeline on a single CPU. In the theoretical sense, it can double the computational capacity of a single UNI-processor design.

Hyperthreading

- In reality... what you have is double the contention. What hyperthreading doesn't give you is two sets of registers, two sets of cache, and two pipes into memory. So ???
- L1 and L2 cache is expensive, and for good reason. When you stay inside of L1/L2 cache, your performance is outstanding!
- When you contend for cache lines, you have a lot of cache misses ... what happens when you have misses?

Multicore

- Multicore is the Engineer's answer to the Marketing hype of hyperthreading.
- One chip of silicon the size of one processor, but multiple ALU's that
  - function independently
  - have a private set of distinct registers*
  - have their own cache lines
  - have their own lines to memory.

Why should you care?

- Programming a system with multiple CPUs, either virtual or physical, is just different from programming a uniprocessor system.
- The early Xbox had 3 cores.
- 16-core+ laptops are right around the corner.