Administrative Stuff

• **Office Hours:** M 12-1, W 4-5

• When it is not my office hours, don’t call or stop by my office unless it is an emergency
  – *I will not answer any questions in person or on the phone unless it is during my office hours*

• I always answer email as soon as I can, regardless of whether or not it is my office hours
More Administrative Stuff

• Nachos questions: Ask David first
• David’s Office hours: T 7-9 pm, Th 10-12 am, F 12-2 (today: 12:30-2:00)
• Lectures and the book
  – Listening to lectures does not substitute for reading the book - it complements it
  – You cannot do well in this class without reading the book
Computer Organization
von Neumann Computer
The ALU

To/from Primary Memory

Status Registers
Memory Unit

MAR 1234
MDR 98765
Command write

1234 98765 n-1
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 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
10111010001100...1
10111001010000...0
10100111001100...0
10111010001100...1
10111001101100...1
10100110001100...0
10111001101100...1
10111001101100...1
Control Unit

Fetch Unit
Decode Unit
Execute Unit

PC
3050

IR
load R4,c

3046
3050
3054
3058

Primary Memory

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

10111001001100...1
10111001010000...0
10100111001100...0
10111010001100...1
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];  
};
```
Bootstrapping

Bootstrap loader ("boot sector")

Primary Memory
Bootstrapping

Bootstrap loader ("boot sector")

Primary Memory
Bootstrapping

Bootstrap loader ("boot sector")

Primary Memory

1

2

3

Loader

OS
Bootstrapping

Bootstrap loader ("boot sector")

1. Primary Memory
2. Loader
3. OS

4. Initialize hardware
5. Create user environment
6. …
Device Organization

Application Program

Abstract I/O Machine

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

Device Controller

Device
Device Controller Interface

<table>
<thead>
<tr>
<th>Command</th>
<th>Status</th>
<th>Logic</th>
<th>Data 0</th>
<th>Data 1</th>
<th>Data n-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>busy</td>
<td>done</td>
<td>Error code</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>busy</th>
<th>done</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>idle</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>finished</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>working</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>(undefined)</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;

• CPU waits while device operates
• Devices much slower than CPU
• Would like to multiplex CPU to a different process while I/O is taking place
Control Unit with Interrupt

PC = <machine start address>;
IR = memory[PC];
haltFlag = CLEAR;
while(haltFlag not SET) {
    execute(IR);
    PC = PC + sizeof(INSTRUCTION);
    IR = memory[PC];
    if(InterruptRequest) {
        memory[0] = PC;
        PC = memory[1];
        IR = memory[PC];
    }
}
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();
        returnToProcess();
    }
}
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) {
        disableInterupts();
        memory[0] = PC;
        PC = memory[1];
    }
}
Ensuring that **trap** is Safe

```java
eexecuteTrap(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 routine atomically
- A trap handler performs desired processing
- “A trap is a software interrupt”