Today’s topics

**Machine Architecture**
- The basic machine
- Basic programming
- Assembler programming

**Upcoming**
- Language Translation

**Reading**
*Great Ideas, Chapters 9*
Computer Architecture

- **Definition of computer architecture**
  - The programmer’s view of the computer hardware

- **Hardware – Software Boundary**
  - Not well defined
  - Much hardware is programmed
  - Some hardware instantiates programming steps
  - An imbedded program that cannot be changed could be called hardware

- **Firmware**
  - Sometimes used to describe programming that is seldom changed
  - Typically stored in *read-only* memory (cannot change)
Basic Computer

- **Extremely Primitive**
  - Cannot understand any Java or English-like commands
  - There is no command to carry out the `while` statement
  - Make up in speed what it gives up in complexity

- Use a *translator* to transform program to machine’s native language
  - Called *compiler*
  - High-level language like Java called the *source* language
  - Target language is called *machine* language
  - Machine language is what the hardware responds to
Machine Language

- **Machine language is the most primitive**
  - Everything represented by numbers
  - At hardware level, numbers are in binary
  - Numbers represent *instructions* (code)
  - **AND** Numbers represent *data*
  - *Context* of use decides whether number is data or instruction

- **In practice, seldom program in machine language**

- **Use a language, very close to machine language called** *Assembler Language*
  - *Symbolic* in nature (as opposed to numeric)
  - Each instruction number has a mnemonic
  - E.g., 12 is **ADD**
  - Locations also given names (sometimes *variable* name)
Architectural Features

- **Memory**

- **Central Processing Unit (CPU) seen as set of Registers**

  - IP: Instruction pointer
  - IR: Instruction Register
  - AX: Arithmetic Register/Accumulator
  - CF: Condition Flag
Simple Program

- Show in assembler rather than machine language
  
  ```assembly
  copy   ax, x
  add    ax, y
  copy   z, ax
  ```

- Implements
  
  ```
  z = x + y;
  ```

- **Remember, really ALL NUMBERS**

  - Could be:
    
    ```
    20 101
    12 102
    21 103
    ```

  - If `copy-into = 20`, `add = 12`, and `copy-out = 21` and `x` is stored in `101`, `y` in `102`, and `z` in `103`
Fetch - Execute Cycle

- Clock systematically leads machine cycle thru steps

- FETCH
  - Get instruction from memory
    - IP register (also called program counter or PC) says *where from*
  - Increment IP (to point to *next* instruction)

- EXECUTE
  - Decode instruction
    - Figure out what is wanted (add?, copy? …)
    - Extract memory address from instruction
    - If needed, get info from memory
  - Carry out instruction
    - I.e., add info to Accumulator (AX)
More Instructions

- **copy and add**
  - Implicit right to left movement
  - Most instructions involve accumulator (AX)

- **in and out**
  - Like getInt and setInt in Java
  - in goes from keyboard to AX
  - out goes from AX to screen

- **Go through another example -- program to perform:**

```java
{  
x = a.getInt();
y = b.getInt();
z = (x + y);
c.setInt(z);
}
```
sum.as

0    in     ax
1    copy   x, ax
2    in     ax
3    copy   y, ax
4    copy   ax, x
5    add    ax, y
6    copy   z, ax
7    copy   ax, z
8    out    ax

20  x  0
21  y  0
23  z  0

Sample I/O:
<23
<16
>39
More Instructions

- Need to handle Java if and while instructions
- Use `cmp` instruction
  - Compares values in AX and memory location
  - Sets carry flag (CF) to
    - B below (AX less than memory) or
    - NB not below (AX greater or equal to memory)
- Use `jump` instructions to take advantage of this new info
  - `jnb` instruction jumps to new location if CF set to NB
  - `jb` instruction jump to new location if CF set to B
  - `jmp` always jumps, regardless of CF state
- Can now implement code involving if
Program to write out the larger of two numbers read in:

```
in ax
    copy r, ax
in ax
    copy s, ax
    copy ax, s
    cmp ax, r
    jnb there
    copy ax, r
    out ax
    jmp quit
there
    copy ax, s
    out ax
quit
r 0
s 0
Sample I/O: <33 <44 >44
```
Tracing

- **Tracing is often the only way to figure out assembler programs**
  - Number your statements (for reference)
    - Can also use actual memory addresses if known
  - Set up column heading for variables (memory) expected to change
  - Step through the program
    - You play to role of computer
    - Use notes and/or extra columns to keep track of
      - Input and output
      - State of the Condition Flags (CF)
  - Trace with test data
    - Until done *or*
    - Until program is understood
Programming Loops

- Now use new instructions to do the equivalent of `while`
- We noted that syntax for `if` and `while` were same
  - Assembler code surprisingly similar for these two
  - Major addition is the *update*
  - Also need to *jump* back to *beginning* of loop

- Demonstrate with code equivalent to:

```java
{  
    limit = 0;  
    sum = 0;  
    x = a.getInt();  
    while (limit < x)  
    {  
        sum = (sum + x);  
        x = a.getInt();  
    }  
    b.setInt(sum);  
}
```
```
summer.as

  0  copy  ax, #C0
  1  copy  limit, ax
  2  copy  ax, #C0
  3  copy  sum, ax
  4  in    ax
  5  copy  x, ax
  6  #L0  copy  ax, limit
  7  cmp   ax, x
  8  jnb   #L1
  9  copy  ax, sum
 10  add   ax, x
 11  copy  sum, ax
 12  in    ax
 13  copy  x, ax
 14  jmp   #L0
 15  #L1  copy  ax, sum
 16  out   ax

  40  limit   0
  41  #C0     0
  42  sum     0
  43  x       0

Notes:
  #L0=6
  #L1=15
```
Another looping example

- Calculate N! (N factorial) but do it with a loop this time
- Code is equivalent to the following Java:

```java
{ 
    n = a.getInt();
    i = 1;
    fact = 1;
    while (i < n+1) 
    { 
        fact = (fact * i);
        i = (i + 1);
    } 
    b.setInt(fact);
}
```
fact.as

1  in   ax
2  copy n, ax
3  copy ax, #C1
4  copy i, ax
5  copy fact, ax
6  #L0 copy ax, n
7  add ax, #C1
8  copy E0, ax
9  copy ax, i
10 cmp ax, E0
11 jnb #L1
12 copy ax, fact
13 mult ax, i
14 copy fact, ax
15 copy ax, i
16 add ax, #C1
17 copy i, ax
18 jmp #L0
19 #L1 copy ax, fact
20 out ax
21 halt

Notes:
#L0=6
#L1=19

CompSci 001
Assembler Programming Notes

- **Note that previous program added the `mul` instruction**
  - Most hardware has standard arithmetic support
  - Historically not the case

- **The best way to follow such a program is by tracing**
  - See trace for `fact.as` program on web page

- **Writing assembler programs from scratch**
  - Not that hard
  - Can get quite used to working at this level
  - Was done for efficiency reasons
    - Could do better than automatic translation (e.g., compiler)
  - However, remember **15 lines of code a day**
    - This figure is language independent!
    - Compilers have gotten better than the average programmer
Handling List or Arrays

- Need extra hardware to do this well
  - Have registers that point to the list/array
  - Increment these registers to step through list/array

- Can be done with our limited hardware
  - Involves having the program modify itself
  - Not hard to write
  - Errors in such self-modifying code very hard to find!

- Additional Features Desired (minimal upgrade)
  - Need for more registers
  - Handling function/method calls
    - Need to “remember” where you came from
    - Jump to statement after that when done
Modern Hardware

❖ Memory Size
  - PC’s often have gigabyte of memory now
  - What does this do to the size of the instruction?

❖ Lots of Registers
  - It is not unusual to have 32 accumulators
  - What does this do to the size of the instruction?

❖ Memory Hierarchy
  1. Registers
  2. Cache Memory
  3. Main Memory
  4. Disk (virtual memory)
  5. Offline storage (tapes, CDROMs, DVDs, etc.)