*9/27 Lecture Notes*

Memory storage

Memory can only store integers.

46 🡪 stores the integer 46

23.567 🡪 stores the integer 23, and the integer 567

Hello! 🡪 stores the integer ASCII codes for each letter or symbol; ex: a=1, b=2, so on

A picture 🡪 stores the color of each individual pixel based on a color code (some amount of red, some amount of blue, some amount of green)

A video 🡪 Stores a lot of pictures in sequence, in the same way. Video can be *compressed* with clever programming – instead of saving every single image raw, it saves only the changes between frames

Audio 🡪 Measure voltage of sound waves over tight intervals, and save the voltages measured in sequence to be replayed by speakers

Theoretical computer and machine languages

*Memory* has addresses 0,1,2,3, etc. Each address contains any integer up to 5 digits (0 – 99999)

0: 01004

1: 21005

2: 00006

3: 99999

4: 00042

5: 00013

6: 37285

…

*CPU* has an accumulator and an instruction counter

Accumulator: 83209

Instruction Counter: 0

CPU does 3 things: **Fetch, Decode, Execute.** Repeats over and over and over.

Fetch step: CPU takes the number in the Instruction Counter and asks the memory unit for the number in that location (in this case 0). Memory gives back the number, in this case 01004.

Decode step: First 2 numbers are an operation code, tells the CPU what to do (add, subtract, etc) Say that opcode 01: load accumulator from memory location. Last 3 numbers give the address of the instructions; in this case 004 would mean memory address 4.

Execute step: CPU asks memory for the value in address 4 (00042) and changes the accumulator value to this value. The accumulator value is now 00042, the instruction counter goes up to 1 (because 1 cycle has completed) and the cycle resets to Fetch.

Fetch step 2: CPU asks memory unit for number in location 1. Memory gives the number, 21005.

Decode step 2: First 2 numbers are opcode; lets say opcode 21: add number at memory location to accumulator. Last 3 numbers are 005 🡪 5.

Execute step 2: CPU gets the number at location 5 (00013) and adds it to accumulator. Accumulator is now at 00055, instruction counter is now 2.

Fetch step 3: CPU asks memory for number in address 2, 00006.

Decode step 3: Lets say 00: store accumulator into memory location, 006.

Execute step 3: Address 6 is now 00055. Instruction counter is now 3.

Fetch step 4: CPU asks memory unit for location in unit 3, 99999.

Decode step 4: lets say 99: halt.

Execute step 4: Computer stops work.

So this program **added the values in addresses 4 and 5 and stored the value in address 6, then stopped work.**

Computers are not “smart”, they don’t know what they are doing when you give them commands. They simply run until they are told to stop, if the stop command never comes they will keep running forever.

Writing programs in this fashion technically works (this is how they did it in the 50s/60s with the earliest forms of computing), but people wanted an easier way to make it easier both to write and to spot bugs.

How bout words? Instead of a bunch of numbers we could write

LOAD PRICE

ADD FEES

STORE TOTALPRICE

HALT

PRICE DATA 42

FEES DATA 13

TOTALPRICE DATA

Etc.

One could translate these words into machine language and program much easier. Basically it is a way to talk to computers using our syntax and English rather than simply numbers, which is harder for our brains to think in.

These words are called *Assembly Language*. They are one step above machine language

It’s tedious, however, to constantly translate assembly language into machine language. So we use a program to do it for us. Because it is being done by a machine though we use a formal set of rules so the program can read it without ambiguity. This program is called an *assembler*.

The assembler is programmed to read letters from a file and recognize patterns (words), then translate the patterns it recognizes into numbers the machine can use.

We now have a 2 step procedure: human writes assembly language program, assembler translates it, machine reads the machine language and runs the program.

This is very “low level” programming because we are not very far removed from the machine language – only 1 step above talking directly to the machine. But there can now be 2 kinds of mistakes: a syntax or grammar error (ex: you misspell a word) or just a standard logic error.

Assembly language is very seldom used today but occasionally still used for very specific things.

Higher level languages

Higher level programming languages basically just use more translators to move up the ladder closer to English and make things easier for the programmer. FORTRAN in 1957 was the first higher level language and more have followed.

To do the same thing we did in machine language, but using a higher level language you would type:

integer price = 42  
integer fees = 13  
integer totalPrice = price + fees

Much easier to read and much easier to debug, also less tied to particular machines because can be translated to a variety of machine languages using a program called a *compiler*. Programmers no longer have to learn different languages for different hardware vendors.

Compilers are sticklers for syntax – very little leeway, they don’t try to “figure out what you meant”, they just refuse to translate, because it’s better to get an error than a fucked up program

If a compiler doesn’t know what you mean it’s called a *compiler error* or *syntax error*. Don’t guess!

*Runtime errors* or *logic errors* occur when the syntax is correct, but the program doesn’t do what you want it to. Ex: use incorrect numbers, reference wrong variables

There can be *compiler errors* where you type the right thing but it gets mistranslated. It sometimes happens but is exceedingly rare; don’t use it as an excuse for a bad program.