Week 1 Notes:

Von Neumann

- Von Neumann: ENAIC design in 1943
- 1944: Neumann joins Eckert and Mauchly
- Von Neumann Architecture Model, consisting of a CPU (central processing unit for operations and sequence of operations), memory unit (code and data), and Input/Output mechanisms
- CPU ←→ I/O, CPU ←→Memory
- RAM (Random Access Memory)
- Basic unit of memory: byte (1 byte = 8 bits, such as "0110 1011")
    - Bytes tend to carry numerical information/data/content
        - (1) 0000 0011 = 3
        - (2) 0000 0001 = 1
        - (3) 0000 0010 = 4
        - (4) 0000 0001 = 1
        - (5) 0001 1001 = 25
        - (6) 0000 1001 = 9
        - (7) 0000 0010 = 2
        - (8) 1010 0111 = 167
        - (9) 1111 1011 = -5
    - From right to left: Address, Content, Human
    - Each byte has a unique address
        - Byte addressable memory
        - Each have the same number of bits i.e., 16-bit address

Conceptual View of Memory (8-bit addresses example)

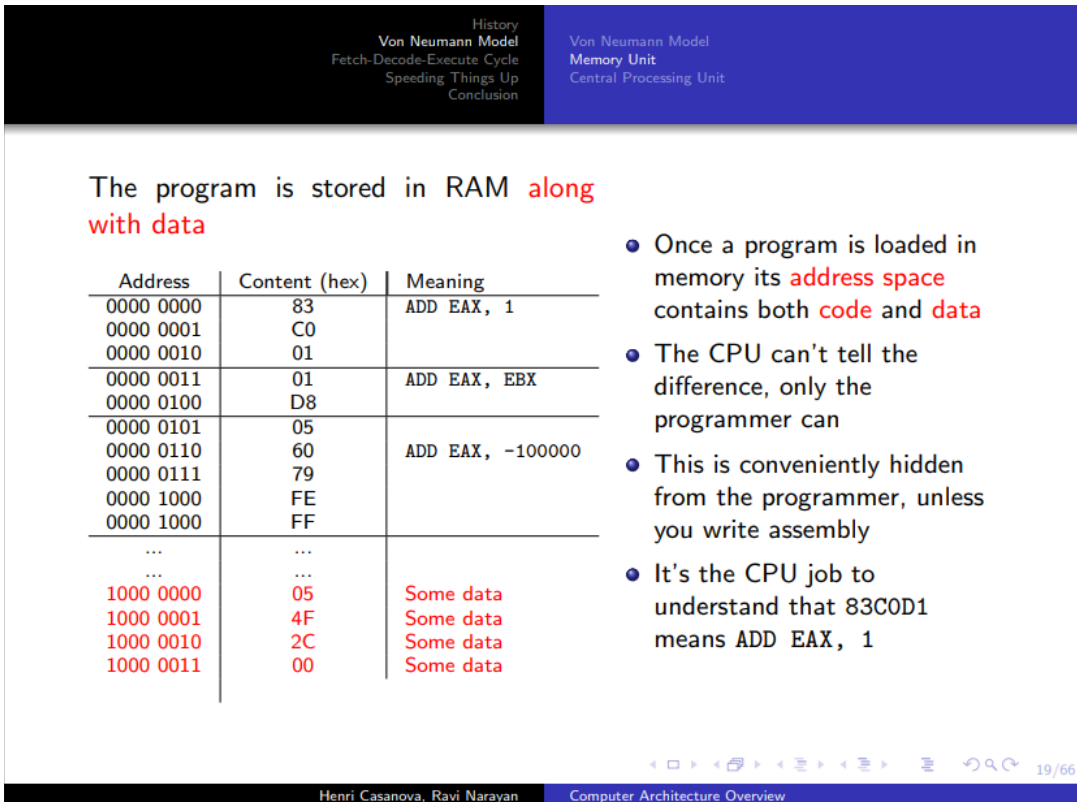
Let's consider a memory 8-bit addresses with this initial state.
We can write a program that does "At address 1000 0000, store the address of the first '9' (0000 1001) in memory"

Address	Content
0000 0000	0000 0011
0000 0001	0000 0001
0000 0010	0000 0100
0000 0011	0000 0001
0000 0100	0000 0101
0000 0101	0000 1001
0000 0110	0000 0010
0000 0111	0000 0110
0000 1000	0000 0101
...	...
1000 0000	0110 0101
1000 0001	1001 0111

$\Longrightarrow$

Address	Content
0000 0000	0000 0011
0000 0001	0000 0001
0000 0010	0000 0100
0000 0011	0000 0001
0000 0100	0000 0101
0000 0101	0000 1001
0000 0110	0000 0010
0000 0111	0000 0110
0000 1000	0000 0101
...	...
1000 0000	0000 0101
1000 0001	1001 0111

- Address is just information
    - Content at memory location is address of another memory location (pointer/reference)
    - In the example, we have the value 9 although it could be another address
- Good for C, though any high-level program is capable of helping
    - Unsigned long x= 42;
    - Int *ptr = (int *)x;
    - Easy to make program crash versus other languages
- Let's consider the following pseudo-code:
    - Step 1) Set the content of variable A to the content at address 1000 0000
    - Step 2) Set the content of variable B to the content at address 1000 0001
    - Step 3) Add A and B together and store the result in A
    - Step 4) Set the content at address 1000 0001 to the contents of A
    - Step 5) Go back to Step 1
        - or in assembly (pseudo-)instructions:
        - // MIPS-like (ICS 331)
        - S1: LOAD A, (1000 0000)
        - S2: LOAD B, (1000 0001)
        - S3: ADD A, B
        - S4: STORE A, (1000 0010)
        - S5: JMP S1
        - // x86-like (ICS 312)
        - S1: MOV AL, [1000 0000]
        - S2: MOV BL, [1000 0001]
        - S3: ADD AL, BL
        - S4: MOV [1000 0010], AL
        - S5: JMP S1
- Instructions tend to be encoded in binary
    - More instruction = larger .exes, though some x86 instructions can be shorter
    - Assembler: transforms assembly code into binary



- Memory is an indexed array of bytes
  - Integers, character codes, floating point numbers, RGB values, coordinates in space-time, images, etc.
  - Addresses
  - Instructions
- What is a CPU?
  - Registers (a few bytes of memory in which things can be written/read extremely fast
    - Data is stored here
    - Special purpose registers for specific purposes
      - Some are managed by the OS itself
    - General purpose registers for storage of program data
      - i.e., 2 + 3 = 5, 5 is stored in registered
    - Not many registers. Too many would make the CPU expensive
    - Compilers are great for utilizing the registers to the best of their abilities
  - ALU (arithmetic logic unit) – computations
    - Mathematical operations
    - Integer and floating point arithmetic
    - Tends to take 1 cycle
    - Operands and results must all be in registers
  - Control Unit: the hardware that makes everything work, aka the boss
    - In charge of controlling the program execution
      - **Program Counter**: contains address of next instruction that must be executed next

        - **Current Instruction**: the binary code of the instruction that is currently being executed
    - Knows how to decode instructions and makes it happen
- Fetch-Decode-Execution cycle
    - Control Unit **Fetches** the next program instruction from memory via the program counter, increments the program counter
    - Instruction is then **decoded**. Signals are then sent to the hardware components such as the memory controller, ALU, I/O controller, etc.
    - Instruction is then **executed**
        - Read from memory into registers when needed
        - Computed via the ALU, results stored in registers
        - Register values written back to memory when needed
    - Repeat
- Let's consider a simplistic hypothetical Von Neumann architecture
    - Memory contains 256 × 1 byte
    - CPU has 2 "data" registers (A and B), 2 "control" registers
    - (Program Counter and Current Instruction)
    - CPU instructions encoded on 1 byte (8 bits): 3-bit "opcode"
    - (operation code) and 5-bit operands:
        - Opcode 000: Load to register A from memory
        - Opcode 001: Load to register B from memory
        - Opcode 010: Add B to A; store the result in A
        - Opcode 011: Store the value of A to memory
        - Opcode 100: Jump
        - Opcode 111: Halt (program terminates)
- We will assume that initially A = 5 and B = 151
- From the previous slide, our instructions are as follows:
    - Opcode 000: Load to register A from memory
    - Opcode 001: Load to register B from memory
    - Opcode 010: Add B to A; store the result in A
    - Opcode 011: Store the value of A to memory
    - Opcode 100: Jump
    - Opcode 111: Halt (program terminates)
- So, for instance, here are meanings of example instructions:
    - 00010111: Load the byte in RAM at address 00010111 into register A ("LOAD A, (10111)" in MIPS-like assembly)
    - 010?????: A = A + B (we don't care what the 5 trailing bits are because this instruction takes no operand)
    - 10000011: Jump to the instruction at address 00000011 and execute it
- See slides 47 through 55 for visual representation