**CS 271 – Week 1 Notes**

* Programming languages have levels
  + **Natural Languages** – English, Spanish, Chinese, etc.
  + **High Level Languages** – Java, C++, Python, etc.
  + **Low Level Languages** – Intel assembly, Mac assembly, etc.
  + **Machine Level Languages**  - Intel machine instructions, etc.
* The lower the level the language, the more specific it must be for the system being used (i.e. Windows, Mac, etc.)
* Assembly language is not machine language, it is a set of mnemonics that has a one to one correspondence with the machine instructions available on the particular architecture
* Idea is to write code and text for the assembler to then put into binary machine code for the target machine
* Level 2 is assembly language, level 1 is the program in machine code
* The operating system does partial translation and the hardware’s instruction set architecture provides a micro-program for each machine instruction or execution
* Level 0 is actual computer hardware, circuits, etc.
* Architecture refers to the design of the hardware whereas software refers to the instructions that control the hardware
  + Sometimes parts of an operating system are implemented into the hardware
* General architecture types from strongest to smallest
  + Super-Computer
  + Mainframe
  + Multiprocessor/Parallel
  + Server
  + Distributed Networks
  + Personal Computer
  + Micro-controller
* Sometimes assembly language will implement high-level language traits
* Everything inside a computer is represented as an “off and on switch”
  + Often 0 is “off” and 1 is “on”
  + A single switch is a binary digit or a “bit”
  + Combinations of these switches represent different info
  + A group of 8 bits is a “byte”
* Any program to be executed is must be loaded into a designated area of main memory
* Peripheral devices
  + Keyboard, mouse, microphones, etc.
* I/O Unit
  + Communicates between the CPU/Memory and peripheral devices. i.e computer screen, speakers, etc.
* Main Memory Unit
  + Contains “cells” with addresses, this is where program’s data is stored and programs being currently used by the CPU (volatile storage meaning data is lost if the computer loses power or is shut off, “temp memory”)
* CPU (Central Processing Unit)
  + Executes machine instructions, “brain” of the computer
* CPU Components
  + **Bus** – parallel “wires” used for transferring a set of electrical signals simultaneously
    - **Internal** – transfers signals among CPU components
    - **Control** – Carries signals for memory and I/O operations
    - **Address** – Links to a specific memory location
    - **Data** – Carries data between the CPU and memory
  + **Register** – fast local memory inside the CPU
  + **ALU** – Arithmetic/Logic Unit
  + **Microprogram** – Sequence of micro-instructions (implemented in the hardware) required to execute a machine instruction
  + **Micromemory** – Actual hardware circuits that implement the machine instructions and programs
* Busses are characterized by their capacity i.e. 16, 32, 64, etc.
* in a simple CISC model, programs must be stored in main memory in order to be executed, called the stored program concept
  + The broad class of computer architectures that use the stored program model is referred to VonNeumann architecture, which essentially means stored program
* **Instruction Execution Cycle**
  + 1. Fetch next instructions (at address in IP) into IR
  + 2. Increment IP to point to next instruction
  + 3. Decode instruction into IR
  + 4. If instruction requires memory access, determine memory address and fetch operand from memory into a CPU register, or send operand from a CPU register to memory
  + 5. Execute micro-program for instruction
  + 6. Got to step 1 (unless “halt” instruction has been executed)
  + Remember: default execution is sequential
* **EXAMPLE:** Execution of machine instruction by its corresponding microprogram, taking a value stored at the address “mem1” and adding it to general register 1 (R1)
  + **ADD R1, mem1 (this execution is written as text, but within the computer, this is a set of electrical signals)**
  + 1. Copy contents of R1 into ALU(Arithmetic and Logic Unit) at Operand\_ 1 register
  + 2. Move address named mem1 into MAR (memory address register)
  + 3. Signal memory fetch (gets contents of memory address currently in MAR and puts into MDR (memory data register))
  + 4. Contents of MDR are copied into ALU Operand\_2 register
  + 5. Signal ALU to perform addition operation
  + 6. Set Status register(flags for errors, issues, execution success, etc) depending on outcome of the calculation and copy contents of ALU Result to R1
* Intel IA 32 Architecture
  + Speed(distance/time) is measured in electronic units
    - K = 10^3 (1,000)
    - M = 10^6 (1,000,000)
    - G = 10^9 (1,000,000,000) etc.
    - Network speed of 8Mpbs is 8,000,000 bits per second
  + Size in bits, Bytes measured in binary units
    - Ki = 2^10, Mi = 2^20, Gi = 2^30
    - i.e. disk size of 200 GiB means 200 x 2^30 Bytes
* Changing Bytes to bits is done by multiplying by 8
* Changing bits to Bytes is dividing by 8
* The IA-32 has 2 modes of operation
  + Protected – certain memory and certain registers cannot be accessed by user programs
  + Real-address – allows user programs to do “just about anything”
* Has 2 processors in one
  + Integer unit – **HAS ONLY ONE SET OF REGISTERS**
  + Floating-point unit
* Has specific hardware implementations
  + Requires specific assembly languages, operating systems, etc.
* Whatever is stored in a memory cell or register is just a combination of electrical signals. One byte has 256 possible combinations of off/on switches in one byte. Each combination is called a code. OS and instruction decoder determines how the combos are made/executed
* Most common **REGISTERS** in programming are the 32-bit general purpose registers named EAX, EBX, ECX, EDX. (E means extended)
  + Multi-purpose registers will be used later on in the course
* Some “general-purpose” and “multi-purpose” registers are used for special purposes
  + **EAX and EDX** – automatically used by integer multiplication and division instructions
  + **ECX** – auto used as a counter for some looping instructions
  + **ESP** – used for referencing the system stack
* Some **REGISTERS** have 8 and 26-bit “sub registers”
  + **EXAMPLE** – Sub-registers of EAX
    - **AX** – refers to the least-significant 16-bits of EAX
    - **AL** – refers to the least-significant 8-bits of AX
    - **AH** – refers to the most-significant 8-bit of AX
    - **Think of this as dividing EAX into smaller and smaller parts, if one part is changed, the entire register is changed**
* **MASM Instruction Types**
  + **Move Data** – really means to copy data from a source to a destination (mov R2 mem2)
  + **Arithmetic/Compare Two Values** – performs an operation or comparison of two values
  + **Conditional/Unconditional branch, Call procedure, Loop control** – Interrupt the default sequence of instruction execution. Jump to a different part of a program, call a procedure, repeat a section of code, loop, etc.
  + **Input/Output** – Devices that communicate with users
* Directives tell the assembler how to interpret the code
  + Such as marking the beginning of program segments (i.e. .data, .code, etc)
  + Marks special labels (i.e. main proc, varName DWORD, etc.)
* MASM is not case sensitive, be concise and specific with code documentation, don’t forget a title (see lecture video 3, 2 min in)
  + Segments always start with a “ . “ such as “.code”, “.data”, etc.
  + “TITLE” directive is essentially the title of the program/code, identifies the file name, etc.
  + “INCLUDE” directive is similar to header files in C++
  + Global constants can then be defined
  + The “.data” segment specifies the block of memory where program data is stored. Specifies how the stored data will be used, variables are declared here also
    - “ds” register specifies the beginning address of the data segment
    - memory location names are declared and specify how stored data is interpreted
  + The “.code” segment specifies the block of memory where the program machine instructions are stored. Must contain a complete main procedure with the “main PROC” directive, “main ENDP” directive, and “END main” directive. Your actual executable code comes after you “main PROC” directive
  + Constants are usually ALL CAPS (i,e, UPPER\_LIMIT, LOWER\_LIMIT, etc.)
  + Comments can start anywhere in a line, no multi-line comment possible
  + Keep identifiers small and concise
  + Try to keep from using “reserved” words as identifiers
  + Variables when declared in “.data” segment start with name, type, initial value, then comment if necessary
  + “ ; “ denotes a comment in assembly language
  + Variable example: size DWORD 100 ;class size
  + “size” is the name, “DWORD” is how we will use that data, DWORD meaning a 4 byte integer, “100” is the initial value, and the comment is how we will use the variable
* **DATA TYPES AND THEIR USAGE**
  + **BYTE –** Character, string, 1-byte integer
  + **WORD –** 2-byte integer, address
  + **DWORD –** 4-byte unsigned integer, address
  + **FWORD –** 6-byte integer
  + **QWORD –** 8-byte integer
  + **There are more to be explored later…**
* **DATA SEGMENT EXAMPLE**
  + size DWORD 100 ;class size
  + celsius WORD -10 ;current Celsius temp
  + response BYTE ‘Y’ ;positive answer
  + myName BYTE “Casey Levy” ,0
  + Variable memory size is allocated in the order they are declared
  + REMEMBER: Names are just constants. The system substitutes the memory address for each occurrence of a name and the contents of a memory location may be variable
* **Literals** – actual numeric or character values, used for assigning specific values to registers, memory, constant names, etc.
  + Integer literals may be expressed in binary, octal, decimal, or hexadecimal by using a radix indicator. If the indicator isn’t present, the default is decimal.
  + Digits must be consistent with radix (e.g., 1011b, 235q, 2012d, 30h)
  + Hex values that start with a letter must have a leading 0 (e.g., 0A3h) or you can use the “0x” prefix instead of radix (e.g., 0xA3)
  + Character literals must be in quotes. They are just different interpretations of numbers, i.e. “A” is 65, “a” is 97, “\*” is 42
  + String literals are 2 characters or longer. Always add a “0” to the end of a string since the 0 detects the end of the string.
* Program Instruction Syntax
  + Executable code will be in the main procedure (main PROC)
  + Each insruction has a specific format
    - Label
    - Opcode (be indented, is a microprogram that tells what the program should do)
    - Operands
    - Comment
* Writing a MASM program (see lecture 5a in week 1)
* Know how to use the following operations
  + mov – move
  + add – add
  + sub – subtract
  + mul – multiply
  + div – divide
  + call – call operation
  + Examples:
    - mov op1, op2 - op2 is copied to op1, op2 then remains unchanged
    - add op1, op2 - op2 is added to op1
    - sub op1, op2 - op2 is subtracted from op1
    - inc op1 - add 1 to op1
    - dec op1 - subtract 1 from op1
  + The destination operand comes first
  + See week 1, lecture 5b, 4 min in for common errors
  + Memory cannot be moved to memory, it must be moved to a register first, then into memory
  + Some other common operations
    - Clrscr – clear the screen, screen is cleared and cursor is at upper left corner
    - Crlf – New line, cursor is at beginning of next new line
    - ReadInt – takes in a value from user, put into EAX first
    - ReadString – takes in string from user, address where you want to store the string in EDX must be known. The space must be declared and size needed, then you can put into EAX.
    - WriteInt/WriteDec – Writes an integer to the screen, value must be in EAX first
    - WriteString – Writes a null-terminated string to the screen, OFFSET memory location in EDX