Key to reading the document:

Week number: purple, underlined and bold.

Chapter name: blue, underlined and bold.

Sub topic: blue, underlined.

Sub - sub - topic: black, bold, underlined

Sub - sub- sub topic :black underlined

Important words : coloured in blue

Everybody can comment by highlighting the phrases like this, so i can see what u want

FOR ALL EDITORS, leave a comment so i can see any changes u have made.

Week 1 + 2

Chapter 1: Fundamentals Concepts

Computer Architecture

- It describes the functions of the various components of the computer system.
- It also describes the function of the machine.
- The architecture tell us the *what* of the machine
- Includes:
    - System design
    - Computer design
- Deals with High level issues
- Involves logic components

Computer Organization

- It describes the how the functions are linked together
- It describes the operation of the components in relation to the hardware
- The organisation tells us the *how* of the machine
- Includes:
    - Computer design
    - Logic design\
    - Circuit design
    - Computer components
- Deals with low level issues
- Involves physical components

Introduction to Assembly Language

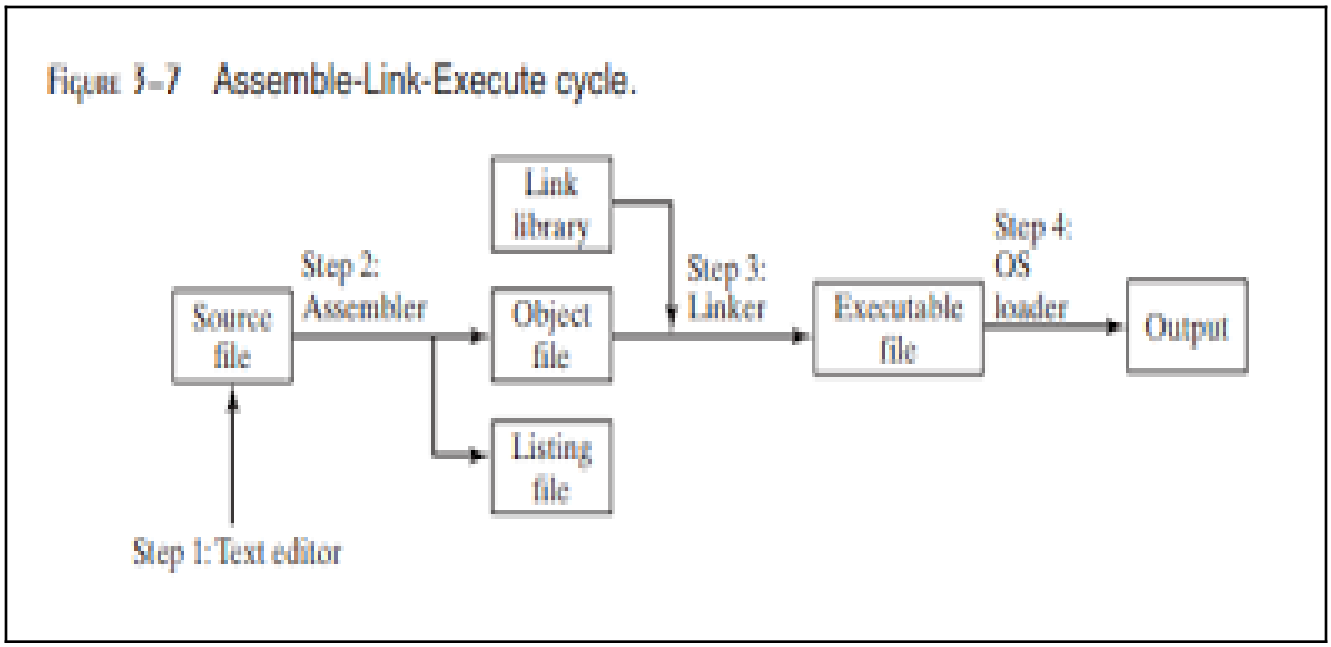
- A low level language designed for each specific processor
- Oldest programming language, which is closely related to machine language.
- It requires an assembler to execute.
- It has a one to one relationship with machine language, meaning that 1 instruction in assembly language is equivalent to 1 instruction in machine language
- It has a one to many relationship with high level languages, meaning that 1 instruction in a high level language is equivalent to many instructions in machine language.
- In modern times, assembly language is used to direct hardware manipulation and address critical performance issues
- It is also used from embedded and low level systems to device drivers.
- An embedded system is a microprocessor based system that controls components to perform a specific task.
- Assembly language is not portable, meaning it cannot be run by all machines as assembly language is made for a specific processor.

Assemble-Link Execute Cycle

- Assembly language cannot be directly executed, it must first be sent to the linker, which is a program which converts the code into an executable file.

- The process by which Assembly language is converted into machine language is called the assembly link execute cycle

- Step 1: The user writes the code
- Step 2: An object file is produced by the assembler, errors must also be corrected in this step
- Step 3: the linker reads the object file, it checks if there are any calls to the procedure for the in a link library, and copies them to the object file and produces an executable file.
- Step 4: the OS loader utility reads the executable file, and executes the program



Listing file

- It is a file that contains:
  - A copy of the source code, with line numbers
  - The numeric address of each instruction
  - The machine code byte of each instruction

    - A symbol table

Applications of Assembly Language and High level languages

Assembly language:

- Cryptographic algorithms that must always take strictly the same time to execute, preventing timing attacks.
- Situations where complete control over the environment is required, in extremely high-security situations where nothing can be taken for granted.
- Assembly language is useful in reverse engineering. Many programs are distributed only in machine code form which is straightforward to translate into assembly language by a disassembler, but more difficult to translate into a higher-level language through a decompiler. Tools such as the Interactive Disassembler make extensive use of disassembly for such a purpose. This technique is used by hackers to crack commercial software, and competitors to produce software with similar results from competing companies.
- Assembly language is used to enhance speed of execution, especially in early personal computers with limited processing power and RAM.
- Code that must interact directly with the hardware, for example in device drivers and interrupt handlers.

High level language:

- Used to develop many software applications

- Used in many places where code understandability is very important

Virtual Machine Concept (NEEDS TO BE REVISED)

- An effective method to explain the relationship between the hardware and the software of a machine is called a virtual machine.
- Each computer has its own native language for its own hardware.
- A machine language that is more human friendly is made above the machine's native language. This is then run on the machine.

Specific Machine level:

- There are 4 specific machine levels:
  - **Digital Logic:**
    - Consists of hardware such as memory, transistors, gates and system buses.
  - **Instruction Set Architecture(ISA):**
    - Known as conventional machine language
    - Controls programmable storage, data types and instruction sets.
    - Executed by level 1(Digital logic)
  - **Assembly language:**
    - One to one correspondence to machine language.
  - **High level language**

Week 3

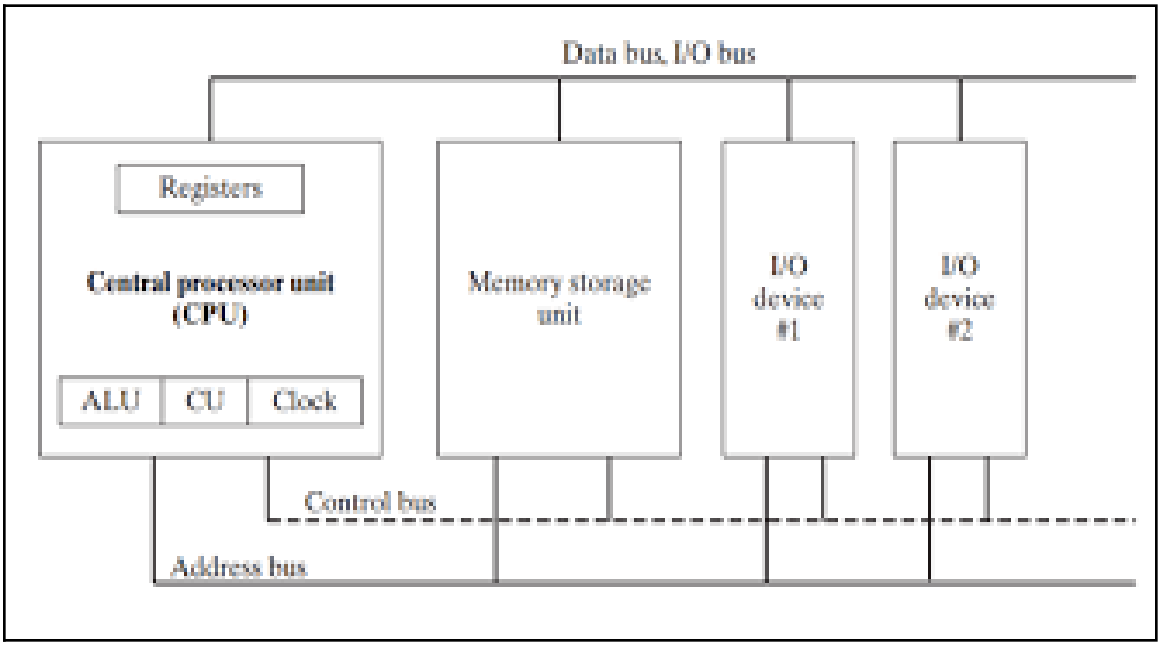
Chapter 2: X86 Architecture:

General concepts:

- A computer with a microprocessor is called a microcomputer
- A microprocessor is a digital component that consists of transistors on a semiconductor.
- One or more microprocessors usually serve as the central processing unit of a computer.

Basic microcomputer design:

- A microprocessor consists of a CPU, a memory and Input/output.



- The CPU, control and processing unit is the heart of the computer.
- It is responsible for calculations and logic operations

- It has 4 main parts:
    - Arithmetic and logic unit, ALU, which performs the mathematical and logical operations.
    - Registers which store the data for processing
    - The control unit is responsible for coordinating the sequence of execution steps
    - The clock is used to synchronise the operation of the CPU with the other components.
- The memory storage unit is where the instructions and data are stored, while a computer program is running.
- It also transfers data from RAM into the CPU and vice versa
- Programs that are not in the CPU cannot be executed or utilised, so they are copied into it.
- The Microprocessor also features connection or buses. There are 4 types of buses:
    - Data : transport data between processor and the memory
    - I/O(input/ output): Transports data between CPU and all the various input and output devices.
    - Control : sends controls signals to the devices and components of the computer

    - Address : holds and transports addresses between the CPU and the devices.

Clock:

- A machine instruction requires at least one clock cycle to execute.
- • The length of a clock cycle is the time required for one complete clock pulse, also known as a machine cycle.

Instruction Execution Cycle:

- The process by which an instruction is fetched from a memory location and executed
- The process:
    - Step 1: the instruction is fetched from the instruction queue, and the instruction pointer is incremented
    - Step 2:the instruction is decoded, and recognizes the binary pattern
    - Step 3: If there are operands involved then they are fetched
    - Step 4: the instructions are then executed, the Zero, carry and overflow flags are updated as well
    - Step 5: the result is stored if there was an output operand
- This process is simplified to fetch, decode and execute.
- Example z = x + y
    - An address is placed on the address bus

    - The code inside in the is made available, as per the instruction of the memory controller
    - The IP value determines the next instruction to be executed
    - It is analysed by the instruction decoder, the corresponding digital instruction are sent to the CU
    - Control bus carries signals that use the system clock to coordinate the transfer of data between different CPU components.

Reading from Memory

- Reading memory is done more slowly than reading an internal register.
- This is done in 4 steps:
    - Place the address on the address bus/
    - Change the value of the processors read pin
    - Wait 1 clock cycle, for the chip to respond
    - Copy data from data bus into destination operand.
- Each of these steps generally requires a single clock cycle.

Cache

- Reading from memory is slow, this causes a bottleneck, because all programs need to access to storage
- To solve this problem, designers used high speed memory known as a cache.
- This a memory which keeps all the most important, and most used programs, so that they don't need to be reloaded repeatedly

- If a processor finds the needed data in the cache then it is called a cache hit, otherwise it is called a cache miss.
- X86 processor has 2 types of cache:
    - Level 1/ primary: located on the cpu, it is the fastest
    - Level 2/ secondary: externally connect via high speed data bus. It is a little slower than level 1.
- Cache are in the class of RAM called static Ram which uses transistors, which reduces its storage capacity, but increases its access time. It is expensive aswell.
- On the other hand conventional RAM is classified as DRAM, which uses capacitors and a transistor. This means that it must be refreshed continuously, otherwise the data will be lost.

Loading and Executing a Program

- Step 1: The OS looks for the file name in the current disk, then in its available directories, if it is not found then an error message is displayed.
- Step 2: The OS retrieves the basic information about the file like the file directory, size, and physical location.
- Step 3: it is loaded into the next free slot in the memory,
    - A chunk of memory is allocated to the program, where the information of the file is entered into a table.
    - The OS adjusts the pointers so that they contain the addresses of the program.
- Step 4: the first machine instruction is executed, this is called a process

- Step 5: An identification number is allocated to the process, to keep track of it while it is running.
- Step 6: the OS responds to any query, interrupts and access to the system resources. according to the program
- Step 7: when the process ends, it is removed from the memory.

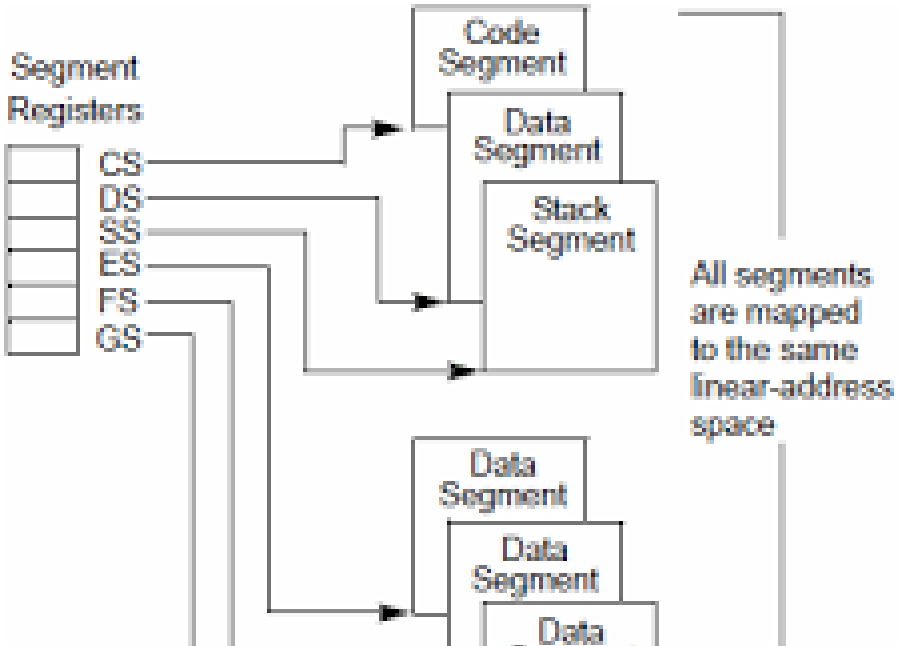
Mode of Operations

- There are 3 modes of operation:
    - Real mode:
        - It is the mode in which ANY part of the memory can be accessed, regardless of the use
        - This means that, without proper management and knowledge, a program using real addressing mode could easily overwrite the operating system and the system BIOS, trigger a physical hardware interrupt, or accidentally send a signal to a peripheral device. Not only could this cause a system to freeze or crash, but it also could cause data loss or physical damage to hardware.
        - Only 1 mb of memory can be addressed.
        - MS-Dos runs in this mode
    - Protected mode:
        - The native state of the processor
        - All 32 address lines are valid and can address up to 4G bytes of physical address space.

        - The OS allocates the memory, this prevents programs from accessing each others memory segments
        - Has Hardware support, and can quickly switch to protect the virtual memory.
        - Used by Windows and linux
        - Virtual 8086 mode:
            - A special case of protected mode.
            - 1 mb of address space is allocated for each program.
            - If anything happens inside the virtual memory, it won't affect the other programs.
            - A modern operating system can execute multiple separate virtual-8086 sessions
            - at the same time.
    - System management mode:
        - Provides a mechanism for implementation power management and system security
        - Handles chipset errors and safety functions.

Real mode addressing

- Can access to six segments



• Each segment is 64 kb
• Logical address of:
  ○ segment , 16 bits
  ○ Offset, 16 bits
• Linear address is 20 bits.	

- The code segment holds the base address for all executable instructions in the program
- The data segment holds the base address for variables.
- The extra segment is an extra data segment (often used for shared data)
- The stack segment holds the base address for the stack. The segment is also to

Linear address = segment * 10 + offset

- In protected mode, a task or program can address a linear space of 4 GB, while in real mode it is only 1 mbs.

[Basic Execution Environment]

- There are 3 registers in the basic execution environment:

- **Data /general Registers**:
  - Hold data for an operation
  - All these registers can be used for anything, but each also has its own special purpose.
  - There are 8 types of this register:
    - EAX, AX, AH, AL:
      - Preferred for arithmetic, logical and control operations
    - EBX, BX, BH, BL:
      - Used to serve as an address register
    - ECX, CX, CH, CL:
      - Serves as loop counter
    - EDX, DX, DH, DL
      - Used to do multiplication and division

- Address/ Segment registers:
  - Hold the address for an operation
  - There are 6 type of this register:
    - ESI/EDI:points to the source and destination string respectively in the string move instructions
    - EBP:(Extended Base Pointer) is used to reference function parameters and local variables on the stack
    - ESP:(extended stack pointer register) contains the offset for the top of the stack to addresses data on the stack

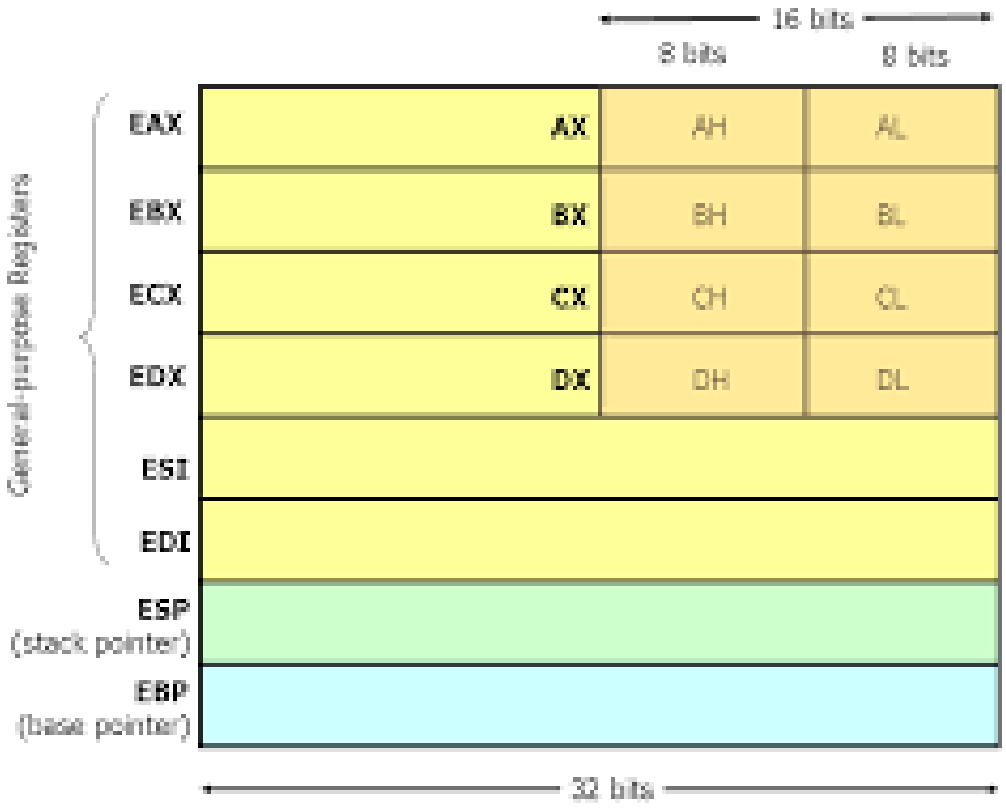
            - EIP:(extended instruction pointer) this stores the offset for the next instruction. It is updated every time an instruction is executed.
            - ECS:(extended …) contains the base address of the next instruction.
    - Status register: Keeps the current status of the processor
        - There are 6 status flags:
            - CF:the carry flag is set to 1 if a small number is subtracted from a larger number.
            - OF:The overflow flag is set to 1 when an operation results in a value larger than the destination storage capacity.
            - SF: The singed flag is set to 1, if an operations results in a negative number
            - ZF: The zero flag is set to 1, if an operation results in 0.
            - PF: the parity flag is set 1, if a number has an even number of 1s.
            - AC: the Auxiliary carry flag is set to 1, if in a binary coded decimal operation, the 3rd bit gives a carry to the 4th place.

29H = 0010 1001

+4CH = 0100 1100

75H = 0111 0101

^ here there is carry generated and forwarded to next nibble, so the auxiliary carry flag is set to one.



- Unlike high level languages, in assembly language u can't just move data, address or perform operations without the memory locations being the same size.
- Similarly the values in these registers must be within

certain limits.

Segments:

- A segment is a location in the register which stores some sort of designated data.
- There are 4 types of segments:
  - Code segments: hold base address for executable instruction
  - Data segment: hold base address of variables.
  - Extra segment: has many uses, often used for shared data
  - Stack segment: holds base address for the stack, stores interrupts and subroutines.

CHAPTER 3: ASSEMBLY LANGUAGE FUNDAMENTALS

BACKGROUND:

- Unlike traditional HLL, where variables can be declared anywhere and at any time. This cannot be done in assembly.
  - For declaring and initialising variables this needs to be done in the .DATA part of the code
  - Actual instructions and commands need to be written in the .CODE section of the program.
  - In the .CODE there is a main (like in c/ c++/ c#) we call this the main PROC (PROC is short for procedure).
  - To comment something use the ';' before a statement

Onward:Instructions:

- These are the statements which will be executed when the program will be run.
- Format:
  - Instruction<space> operands
- Where the instruction is the action we want the program

to perform, they are also called mnemonics, and cannot be used as variables.
- The operands can be the registers, values, or variables we want to manipulate
- Each instruction has a specific format of being used.
- Each instruction has a specific register it uses or can be used.
- We will have a look at each instruction in the next section

Data types initialization and declaration:

- To create a variable the format is :
  - <variable name> <directive initializer> <value(s)>
  - The name of the variable follow the same rules like those of c
  - The directive initializer defines the literal size of the variable. You can choose from:
    - Bytes: 8 bit unsigned
    - Sbytes:8 bit signed
    - Word: 16 bit unsigned
    - Sword:16 bit signed
    - DWORD: 32 bit unsigned
    - SDWORD:32 bit signed
  - Singed directive initializers can store negative values, while unsigned cannot.
  - Let’s look at some examples:

description	Assembly	C++
Declare a character	value1 BYTE 'A'	char value1 = a;
Declare a integer value, with 0	value2 BYTE 0	int value2 = 0;
Define a negative value -127	value4 SBYTE −128	int value4 = -128;
Define an uninitialized variable	value4 WORD (?)	int value4;
Define the smallest signed value	word2 SWORD -32768	
Define the largest unsigned word value	word1 WORD 65535	

ARRAY:

- To declare an array like in c++, instead you can write a number of values after the directive initialize.
- For example :

Description	Assembly	C++
Initialisé à 5 integer array	list BYTE 10, 20, 30, 40	List[] = {10, 20, 30, 40}

- Unlike C++, the arrays in assembly dont have type restriction. Meaning you can have array with letters, numbers and string :
-

Description	Assembly	C++
Initialisé à 3 integer array	list BYTE 10,‘a’,45h	This cannot be done

- Depending on the directive initialiser the offset will be different or will “jump”.
    - If you use a byte or sbyte directive then the offset of the array will jump by 1 after each element.
    - If you use a word or sword directive then the offset of the array will jump by 2.
    - If you use a dword or sdword directive then the offset of the array will jump by 4.
- There is a visual representation of this on the next page:

	data	address	
List_1 byte 10, 20, 30	10	0001	List 1
List_2 word 1000, 2000, 3000, 234H, 0ABh	20	0002	
	30	0003	
List_3 dword ABCh, DEFh, 12345h	1000	0000	List 2
	2000	0002	
	3000	0004	
	234	0006	

	AB	0008	
	ABC	0000	List 3
	DEF	0004	
	12345	0008	

STRINGS:

- To define strings we will do it like this

assembly	c++
String_1 byte “Hello world”,0	String string_1 = “hello world”

- What's with the 0? In c++ all strings end with a termination or \0 similarly we need to end a string manually with the 0, this is called a null byte
- When you use directive initialiser each letter in the string will occupy that memory.

- For example here we used byte thus each letter occupies 8 bits of data. If we had used word then each letter would take up 16 bits of memory space.

DUP OPERATOR:

- Sometimes in assembly we need to initialise array with the same value you could do it manually, but that would be impractical if you have a large number of elements
- To get around this u can use the DUP operator
- The specific format for this is:
    - <Variable_name> <directive initialiser> <number of time you want the value to repeat> DUP(<the value u want to repeat>)

- Here are some examples:

description	Assembly
Initialise an integer array of 3 bytes, where all values are 4	Arr_1 byte 3 DUP(4)
Initialise an character array of 5 word, where all values are ‘a’	Arr_2 WORD 5 DUP(‘a’)
Declare an uninitialized 50 Dword array	Arr_3 DWORD 50 DUP(?)
We can do similar thing with	Arr_4 word 5 DUP(“string”)

strings Note we don't need a null byte	
We also duplicate multiple variables like this	Arr_5 byte 10 DUP(2, ‘a’, “sarim”)

LITTLE ENDIAN ORDER:

- In the 8086 processor data is stored starting from the least significant byte first
- For example:

Var_1 dword 12345678h; declared in .data		
Starting from the least significant byte	offset	value
	0000	78
	0001	56
	0002	34
	0003	12

SYMBOLIC CONSTANTS:

- These are constants, which are mix between UDTs abd constants of c++, the only difference is that these constants can be either an expression, or a symbol or a piece of text, and can even be used as a data type.
- These constants cannot be modified during the execution of the program
- These constants are usually declared outside the .data and outside .code sections of the program.
- The format is:
    - <variable name> EQU <expression/ symbol/expression>

- For example:

Description	assembly
Declare a constant PI with its value to the 3rd decimal place	PI EQU <3.1416>
Declare a message	pressKey EQU <"Press any key to continue...",0>

- They can even be used to define variables, here are some example:

matrix2 EQU <10 * 10>
.data

Prompt byte pressKey; from the previous example
M1 word matrix2; here the value of M1 is 100

Current location counter:

- The $ sign returns the location of the current offset.
- This means we can get the size of any list, this is how:

```
.data
Selfptr DWORD $
List byte 10,20,30,40
List_size = $ - list;
```

- If it was a word array then we would divide the list size by 2 and by 4 if it was a DWORD array.

CHAPTER 4: DATA TRANSFER, ARITHMETIC AND ADDRESSING

Operands:

- An operands is the object, location, register we want to manipulate:
- There are 3 basic types of operands:
  - Integral literal/immediate eg(1, 300h ,)
  - Register
  - Memory(variabled)
- Instructions can have 1, 2, 3, or no operands, they look like this:
  - Mnemonic
  - mnemonic [destination]
  - mnemonic [destination],[source]
  - mnemonic [destination],[source-1],[source-2]

- In assembly when writing instructions we need to keep in mind the size of the register otherwise the assembler will throw an error
- Memory operands (variables)can be dereferenced to get their locations, like how it's done in c++ with an operator, this is called direct memory operands .

MOV instruction:

- This is 2 operand instructions, where the destination is replaced with the value of the source variable.
- It has 3 rules:
  - Operands must be the same size
  - Both of the operands cannot be memory operands
  - The IP, EIP or RIP cannot be the destination operand.
- It must be written in one of the following formats:
  - MOV reg,reg
  - MOV mem,reg
  - MOV reg,mem
  - MOV mem,imm
  - MOV reg,imm
- Note we cannot move the value of a variable(memory) to another variable directly we need to use a register.

MOVZX instruction:

- To move the value of a smaller register into a larger register you need to use the MOVZX instruction.
- It has 2 rules:
  - Both of the operands cannot be memory operands

    - The IP, EIP or RIP cannot be the destination operand.
- It has the same written formats as the MOV instruction

MOVSX instruction:

- To move a signed value into a larger register or variable use the MOVSX.
- It has the same rules and format as the MOVZX instruction.

XCHG instruction:

- To exchange 2 values use the XCHG instruction,
- It has the same rules as the MOV instruction.
- Formats:
    - XCHG reg,reg
    - XCHG reg,mem
    - XCHG mem,reg
- Note we cannot exchange the value of 2 variables or 2 integers.

ADD instruction:

- Adds the 2 operands together
- The format to write the instruction is:
    - ADD destination, source
- It has the same rules as the MOV instruction
- If the result produced is larger than the destination then the Overflow flag is set to one.

SUB instruction:

- Subtracts the 2 operands together.
- The format is:
    - SUB destination, source.

- It has the same rules as the MOV instruction
- If the result of the subtraction is equal to 0 then the zero flag is set to 1.
- If the result of the subtraction is a negative value then the signed flag is set to 1.
- If the destination is smaller than the source then the operation will result in the carry flag being set to 1.

NEG instruction:

- Reverses the sign of the given register
- If not done properly then the overflow flag will be set
- Format:
    - NEG address,
- Only registers can can be negated

OFFSET operator(NEEDS TO BE REVISED):

- Return the offset or location of the memory operand or data label.
- There are 2 types of offsets:
    - Direct offset operands: these access the values of variables with the help of an integer literal (usually done in arrays).
    - Formats of direct offset operands:
        - list_name + or - integer
        - [list_name + or - integer]
        - Eg : MOV BX, NUM+2
    - In direct offset operands: these types of operands help access the list with the help of pointers, which just means that a variable or a register has the offset of the list.

- Formats:
  - First get the offset of the list and then increment the register, usually the esi register.

```
.data
    val1 BYTE 10h,20h,30h
.code
    mov esi,OFFSET val1
    mov al,[esi]              ; dereference ESI (AL = 10h)

    inc esi
    mov al,[esi]              ; AL = 20h

    inc esi
    mov al,[esi]              ; AL = 30h
```

- Another way of indirect addressing is called indexed addressing where you give an integer value to access the address.
- There are 2 formats: [label + reg] or label[reg]
- When in use they look like:

PTR operator:

- It overrides the size of the declared variable.
- This can be used to store large data into smaller chunks
- It does this by turning the variable into an array, and using the little endian order stores the values
- Format:
  - Instruction reg, desired_type PTR variable

TYPE operator:

- The TYPE operator returns the size, in bytes, of a single element of a variable.
- This cannot return the size of an array
- Format:
    - Instruction reg/mem,TYPE mem

LENGTHOF operator:

- Counts the number of elements in the array.
- Format:
    - Instruction reg/mem,LENGTHOF list_name

SIZEOF operator:

- Return the size, in bytes of an array.
- Format:
    - Instruction reg/mem, SIZEOF list_name

LABEL directive:

LOOPS:

- There are 2 types of loops:
    - Conditional
    - Unconditional

Unconditional(JMP instruction):

- This just means that the loop will run no matter what happens, like an infinite while loop.
- Format:

```
SAMPLE:
    inc eax
    mov bx,ax
JMP SAMPLE
```

Conditional(LOOP instruction):

- This is will loop for a set number of instructions:
- This will be done with the help of ecx register.
- First we move the value of the number of iteration into ecx.
- Then we write the loop in the format:

```
mov eax,0
mov ecx,5
L1:
        inc ax
        call DumpRegs
loop L1
```

- We don't need to update the value of the ecx, it will decrement itself and check the value, if it has reached 0 it will not enter the loop.
- We can implement nested loops in assembly as well this can be done by saving the value of ecx and updating it at the end of the first loop, it will look like this:

```
INCLUDE Irvine32.inc
.code
main PROC
mov eax, 0
mov ebx, 0
mov ecx, 5
L1:
inc eax
mov edx, ecx; save the value of ecx
```

```
call dumpregs
mov ecx, 10; update the new value of ecx
        L2:
                inc ebx
                call dumpregs
        loop L2
mov ecx, edx; restore the value of ecx;
loop L1
exit
Main endp
```

Chapter 5: Procedures and stacks

Stacks:

Concept:

- Stacks are memory structures that are used to store data during runtime or after runtime.
- They follow the rule of last in first out. This means that anything that enters the stack first must leave it first.
- The best example to understand this is the stack of plates. To get the bottom plates you need to remove the top plates first otherwise the whole thing will topple over.

Operations:

PUSH/POP:

- To place a value into the stack we use the procedure push function

- When we push a value into the stack the value from whatever register or variable that is written is pushed into the stack.
- When anything is successfully pushed into the stack the stack pointer is decremented (by 4 in case of 32 bit operand, and 2 incase of 16 bit operand)
- The format is:
    - PUSH reg/mem16
    - PUSH reg/mem32
    - PUSH imm32
- To get a value from the stack we use the pop function
- When we pop a value into the stack we get the value of whatever the top of the stack holds and it is stored into the variable or register that was written with the pop instruction.
- When anything is successfully popped into the stack the stack pointer is incremented (by 4 in case of 32 bit operand, and 2 incase of 16 bit operand)
- The format is:
    - POP reg/mem16
    - POP reg/mem32
    - POP imm32

PUSHFD / POPFD:

- We can also push the values of the flags into the stack using the PUSHFD
- Form the format is PUSHFD or PUSHFD reg32
- To restore the values use POPFD reg32, to restore the values of the flags.

PUSHAD/POPAD:

- You can push all the current values of EAX, EBX, ECX, EDX into the stack with one command using PUSHAD
- And you can get all the values of the registers back using POPAD

PUSHA/POPA:

- PUSHA and POPA have the same function as PUSHAD/POPAD but it does this for all 16 bit registers.

Applications of stacks:

- To save and store registers
  - They can be used to store the value of ecx, for nested loops
- To save return address of a procedure
- To pass arguments
- To support local variables

PROC directive:

Concept:

- We can create functions like in c++ by creating a procedure.
- Format:
  - Sample PROC
  - .
  - .
  - .

    ret
  - Sample ENDP
- PROC defines the beginning of the function
- ENDP defines the last the ending of the function

- ret tells the program to return the main function.
- In Assembly all the variables are global functions meaning that any procedures can access any variable
- To call a procedure use CALL function_name
- Just like c++ procedures in assembly can all other procedures.
- It is good practice to push the value of the registers into the stack.

Conditional Processing:

Comparison instruction:

AND instruction:

- Performs bitwise multiplication
- Format:
  - AND reg, reg
  - AND reg, mem
  - AND reg, imm
  - AND mem, reg
  - AND mem, imm
- Both operands must be the same size.
- If you AND 1110 1110, 0, the remaining places of the second operand will be filled with 0s.
- It is used for masking.
- It clears the overflow and the carry flags
- It modifies the sign, zero, and parity flags according to the destination operand.

AND al, 0; can be used to set the zero flag to 1
AND al, 7Fh; can be used to clear the sign flag.

OR instruction:

- Perform bitwise addition
- Format:
  - OR reg,reg
  - OR reg,mem
  - OR reg,imm
  - OR mem,reg
  - OR mem,imm
- The operands must be the same size.
- It will clear the carry and the overflow flags
- It will set the sign, zero,and parity flag according to the value

OR al, 1; can be used to set the zero flag to 0
OR al, 80h; can be used to set the sign flag to 1
OR eax, 0; can set the overflow to 0;

XOR instruction:

- Performs XOR instruction
- Same format an rules as OR and AND instruction
- It clears the overflow and carry flags
- It modifies sign, parity, and destination according to the output.
- Can be used to check the parity flag

NOT instruction:

- Inverts all the operands
- Format:
  - NOT reg

    - NOT mem
- Does not affect any flags.

TEST instruction:

- Does the AND operation without changing the destination.
- Can be used to test if a bit is switched on
- It modifies sign, parity, and zero according to the output.
- Same format as AND instruction.

CMP instruction:

- It performs subtraction of destination and source, but does not modify any of the values.
- Also known as applied subtraction.
- Same format and rules as the AND instruction.

The table below shows which flags will be set if the following conditions are met for unsigned values

cmp dst, src	ZF	CF
dst = src	1	0
dst < src	0	1
dst > src	0	0

Setting and Clearing Individual CPU Flags:

- You can also set the carry flag to 1 directly using: stc
- You can also clear the carry flag directly using: clc
- You can also set the overflow flag by:

```
Mov al, 7Fh;
Inc al; overflow set to 1;
```

Conditional Jumps:

• The conditional jump instructions can be divided into four groups:

1. Jumps based on specific flag values
2. Jumps based on equality between operands or the value of (E)CX
3. Jumps based on comparisons of unsigned operands
4. Jumps based on comparisons of signed operands.

Jump on the basis of flags:

Mnemonic	Description	Flags
JZ	Jump if zero	ZF = 1
JNZ	Jump if not zero	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if not carry	CF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if signed	SF = 1
JNS	Jump if not signed	SF = 0
JP	Jump if parity (even)	PF = 1
JNP	Jump if not parity (odd)	PF = 0

Jump on the basis of CMP for signed:

Mnemonic	Description
JG	Jump if greater (if *leftOp* > *rightOp*)
JNLE	Jump if not less than or equal (same as JG)
JGE	Jump if greater than or equal (if *leftOp* ≥ *rightOp*)
JNL	Jump if not less (same as JGE)
JL	Jump if less (if *leftOp* < *rightOp*)
JNGE	Jump if not greater than or equal (same as JL)
JLE	Jump if less than or equal (if *leftOp* ≤ *rightOp*)
JNG	Jump if not greater (same as JLE)

Jumps based on comparisons of unsigned operands

Mnemonic	Description
JA	Jump if above (if *leftOp* > *rightOp*)
JNBE	Jump if not below or equal (same as JA)
JAE	Jump if above or equal (if *leftOp* >= *rightOp*)
JNB	Jump if not below (same as JAE)
JB	Jump if below (if *leftOp* < *rightOp*)
JNAE	Jump if not above or equal (same as JB)
JBE	Jump if below or equal (if *leftOp* <= *rightOp*)
JNA	Jump if not above (same as JBE)

- We cannot use jump instructions for unsigned operands because when the comparison is being done the sign will not be maintained thus resulting in a logical error.

Jumping based on conditional loops:

LOOPZ/ LOOPE:

- Works the same as the loop instruction but instead of only checking the value of ecx it also checks the zero flag if it is 0

LOOPNZ/ LOOPNE:

- Works the opposite of LOOPZ/ LOOPE meaning it checks if the value of ecx and it checks if the zero flag is set to 1

Translating c/c++ code into assembly:

Simple **if**

c++/c	**assembly**

if(a > b)
  Max = a;
	Mov eax, a;
	Cmp eax, b;
	Jg label_1
	label_1:
	Mov max, a;

Simple **if…else**

if(a > b)
  Max = a;
Else
  Max = b;
	Mov eax, a;
	Cmp eax, b;
	Jg label_1:
	Mov max, b;
	Label_1
	Mov max, a;

Simple **if** with **AND**

if(a>b && a>c)
  Max = a
	Mov eax, a;
	Cmp eax, b;
	Jg label_1:
	Jmp else_label

	label_1:
	Cmp eax, c;
	Jle label_2
	Mov max, a;
	
	label_2:

Simple **if** with **OR**

if(a>b || a > c)
Max = a
	Mov eax, a
	Cmp eax, b
	Jg label_max;
	Cmp eax,c
	Jle else_label
	Label_max:
	Mov max, a

Simple **while loop**

while(a < b)
a++
	Mov eax, a
	Top:
	Cmp eax, b
	Jg exit_loop
	Inc eax;
	Jmp Top
	exit_loop

Chapter 7: Integer Arithmetic:

General concepts:

- This chapter is about the manipulation of the binary values in the registers and how can they be altered using different techniques

Shifting:

Logical shift:

- There are 2 types shift left and shift right
- These operations simply shift the bits from the left to the right or vice versa, by pushing 0s into the MSB incase of shift left or LSB in case of Shift right.
- This also does not maintain the sign of a negative number when the number is shifted.
- The LSB is moved into the carry flag in case or left shift.
- The MSB is moved into the carry flag in case of right shift
- Format:
    - **SHL destination, count.**
    - SHL reg, imm
    - SHL mem, imm
    - SHL reg, CL
    - SHL mem, CL
- Mnemonic for shift right is **SHR**
- Shift towards left ← can be seen as bitwise multiplication(in powers of 2) and shift right → is seen as bitwise division(in powers of 2).
- To divide a number by an integer that is not the power of 2:
    1. Break the divisor into powers of 2
    2. divide the number individually by these components
    3. And then add then together

4. Here is an example:

{**INSERT PICTURE!!!!!!!!!!!! :)**}

Arithmetic shift:

- There are 2 types arithmetic shift: arithmetic left shift and arithmetic right shift
- These operations shift the bits from the left to the right or vice versa, they also maintain the sign of a negative operand by inserting a 1 into the MSB
- The LSB is moved into the carry flag in case or left arithmetic shift.
- The MSB is moved into the carry flag in case of right arithmetic shift
- Format:
    - **SAL destination, count.**
    - SAL reg, imm
    - SAL mem, imm
    - SAL reg, CL
    - SAL mem, CL
- Mnemonic for shift arithmetic right is **SAR**
- Shift towards left ← can be seen as signed bitwise multiplication(in powers of 2) and shift right → is seen as signed bitwise division(in powers of 2) .

Shift double:

- Shift double shifts the corresponding bit from the source to the opposite corresponding destination.
- This means that incase of shift left double the MSB of the source will move to the LSB of the destination, and in

case of shift right double the LSB of the source will become the MSB of the destination.

- The sources do not change.
- The bits that are shifted out are moved into the carry flag.
- Format:
    - **SHLD destination, source, CL/imm**
    - SHLD reg16, reg16, imm
    - SHLD mem32, reg16, imm
    - SHLD reg32, reg32, imm
    - SHLD mem32, reg32, imm
- Mnemonic for shift right double is **SHRD**

Rotating:

Logical or simple rotate:

- The rotate function moves the MSB into the LSB, in case of the left rotation or vice versa in case or right rotation.
- In the case of Left Rotation the carry flag will reflect the value of the MSB, and in the case of right rotation the carry flag will reflect the value of LSB.
- Format:
    - **ROL destination, count**
    - ROL reg, imm
    - ROL mem, imm
    - ROL reg, CL
    - ROL mem, CL
- Mnemonic for right rotate is ROR

Rotation with carry:

- The rotation with carry function moves the value of the carry flag to the LSB incase of left rotation and will move

the value of the carry flag to the MSB incase of right rotation.

- Format:
  - **RCL destination, count**
  - RCL reg, imm
  - RCL mem, imm
  - RCL reg, CL
  - RCL mem, CL
- Mnemonic for right rotate with carry is RCR

MUL/IMUL:

- MUL multiplies the operand with the value of eax, and stores it in eax.
- The overflow flag will never be updated incase of multiplication, as the destination will always be twice the size of the multiplicand, here is the table:

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

- MUL sets the carry and overflow to flags to 1,if the upper half of the product in not equal to 0;
- Whenever the multiplication operation is performed half of the product is stored in 2 parts, meaning if u were to multiply ax with bx then the product will be able to accommodate up to 32 bits.
- For example multiplying:

1. al = 10 And bl = 10 will result in ax = 0064h

2. ax = 1234 and bx = 1234 will result in ax storing the lower half which is 34CCh and the upper half is stored in dx which is 17

- IMUL is used the same as MUL instruction but it preserves the sign of the result as well.
- This is done with the help of sign extensions CBW, CWD, CDQ.
- IMUL supports multiplication up to 1, 2, and 3 operands for multiplication

CBW, CWD, CDQ:

- To maintain or extend the sign of a value in the register, we will use one of these instruction:
    1. CBW, is used for extending the sign of AL into AH.
    2. CWD, is used for extending the sign of AX into EAX.
    3. CDQ, is used for extending the sign of EAX into EDX.
- These instructions do not have any operands.

```
.data
byteVal SBYTE -101              ; 9Bh
.code
mov  al,byteVal                 ; AL = 9Bh
cbw                             ; AX = FF9Bh
```

DIV/IDIV:

- The div instruction is used to divide the value of eax with either another register or a variable in assembly.
- It is a single operand instruction.

- Format:
  - DIV reg/mem8
  - DIV reg/mem16
  - DIV reg/mem32
- The quotient and remainder stored in different places when eax, is divided:

Dividend	Divisor	Quotient	Remainder
AX	reg/mem8	AL	AH
DX:AX	reg/mem16	AX	DX
EDX:EAX	reg/mem32	EAX	EDX

- When we are performing singed division, we need to extend the sign of the value that is being divided(aka dividend) otherwise the program will produce an incorrect result.
- For example we want to divide -48(decimal) by 5 so we need to do this:

```
.data
byteVal SBYTE -48               ; D0 hexadecimal
.code
mov  al,byteVal                 ; lower half of dividend
cbw                             ; extend AL into AH
mov  bl,+5                      ; divisor
idiv bl                         ; AL = -9, AH = -3
```

Otherwise you will get this:

```
.data
byteVal SBYTE -48              ; D0 hexadecimal
.code
mov  ah,0                      ; upper half of dividend
mov  al,byteVal                ; lower half of dividend
mov  bl,+5                     ; divisor
idiv bl                        ; AL = 41, AH = 3
```

ADC, SBB:

ADC:

- It means add with carry.
- It is the same as add, but it also adds the contents of the carry flag allowing us to adding big numbers
- Same restrictions as the ADD instructions:
    - ADC reg,reg
    - ADC mem,reg
    - ADC reg,mem
    - ADC mem,imm
    - ADC reg,imm
- Also the operands must be the same size.

SBB:

- It means to subtract by borrowing.
- It is the same as the sub instruction but it also subtracts the carry flag from the destination operand.
- It has the same restriction as the sub instruction.

Chapter 8: Advanced procedures

Introduction:

- We learned in simple procedures how make and call some procedures and pass some values into the procedure using registers
- But because the number of registers are limited we can only pass a limited number of parameters.
- We use a new way to pass parameters and different ways to manipulate the stack and function calls.
- We will also create local or temporary variable in the procedure
- One thing we cannot do is that we cannot directly return values from a procedure.

Ways to send values into a procedure:

- We learned we can pass parameters using register like this:

```
.code
main PROC
    mov eax, 12h
    mov ecx, 3;
call foo;
exit
main ENDP

foo proc
Loop1:
    inc eax;
    call writehex;
    call crlf
    loop Loop1
ret
foo endp
```

- No we will use the stack, we will first push the values on the stack
- Then we will save the value of the return address, which is the value stored in EBP, by pushing it
- And then save the values that are in the stack with the help of esp. BUT we do not pop the value out.
- At the end of the function before the ret instruction we will pop the value of EBP so we can return back to the main.
- It will look like:

```
main PROC
    mov eax, 10
    mov ecx, 3;
    push eax;
    push ecx;
call foo;
exit
main ENDP

foo proc
    push ebp;
    mov ebp, esp;
    mov eax, [ebp + 12];
    mov ecx, [ebp + 8]
Loop1:
    inc eax;
    call writehex;
    call crlf
    loop Loop1
    pop ebp
ret
foo endp
```

- Sometimes passing parameters in such a way can cause the return address to be lost, to combat this we can do either of these 2 things:
    1. We can add the number of parameters times 4 to esp. In our example we have 2 parameters so 2*4=8 ans we will add 8 to esp, this is called the C Calling convention
    2. Otherwise when we are about to return we can add the number of parameters times 4 to the ret in our example we will write ret 8. This is called the STD calling conventions

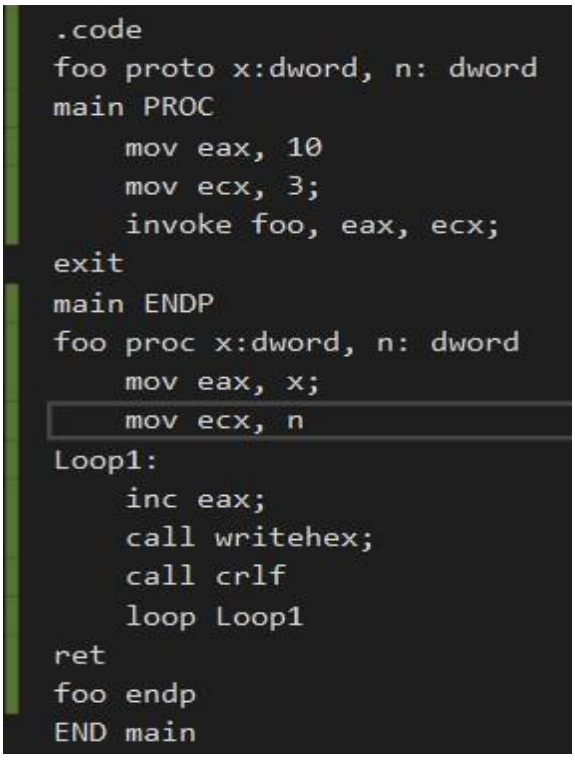
The LEA instruction:

- This instruction is used to get the offset of an array if it has been passed as a parameter through the stack.
- It will move the offset from the stack into esi or edi.
- It will be used something like this:

```
makeArray PROC
    push  ebp
    mov   ebp,esp
    sub   esp,32              ; myString is at EBP-30
    lea   esi,[ebp-30]        ; load address of myString
    mov   ecx,30              ; loop counter
L1: mov   BYTE PTR [esi],'*'  ; fill one position
    inc   esi                 ; move to next
    loop  L1                  ; continue until ECX = 0
    add   esp,32              ; remove the array (restore ESP)
    pop   ebp
    ret
makeArray ENDP
```

Passing parameters:

- There is a 3rd way of passing parameters, this includes the use of INVOKE, and PROTO.
- First we need to specify the parameters with their types such as, write this beside the procedure name, use commas to separate them:
  - N : word ; u will take a parameter of type word
  - Array : ptr byte ; u will take an address of a byte array
  - X : byte, Y: word, Z: dword
  - To pass the offset of an array u need to use ADDR operator and name of the array.
- Next you need to write the name of the procedure followed by PROTO the parameter list, in the .code section before the main proc.
- And then you can use INVOKE, the format is INVOKE function_name, parameter1, parameter2, ….
- It will look like:



```
.code
foo proto x:dword, n: dword
main PROC
    mov eax, 10
    mov ecx, 3;
    invoke foo, eax, ecx;
exit
main ENDP
foo proc x:dword, n: dword
    mov eax, x;
    mov ecx, n
Loop1:
    inc eax;
    call writehex;
    call crlf
    loop Loop1
ret
foo endp
END main
```

Creating local variables:

- What if we want to create local variables to save temporary values?
- There are 3 way of doing it

The manual way:

- We can do it in the same way by pushing EBP,
- Then we can subtract the value of ESP by the number of variables * 4. For example we want 2 local variables so we will subtract 8 from the esp.
- Remember to remove these variables from the stack otherwise the stack will be corrupted
- Finally remember to mov the value of ebp to esp so it can pop the top of the stack with the return address
- It will look like this:

```
.code
main PROC
    call foo
exit
main ENDP
foo proc
    push ebp
    mov ebp, esp
    sub esp, 8
    mov dword ptr [ebp - 4], 10;
    mov dword ptr [ebp - 8], 43;
    mov esp, ebp
    pop ebp
ret 8
foo endp
END main
```

Using the Enter and leave:

- ENTER performs the same function as the manual one but it replaces 3 lines of code:

```
.code
main PROC
    call foo
exit
main ENDP
foo proc
    push ebp
    mov ebp, esp
    sub esp, 8
    mov dword ptr [ebp - 4], 10;
    mov dword ptr [ebp - 8], 43;
    mov esp, ebp
    pop ebp
ret 8
foo endp
END main
```

To

```
.data
.code
main PROC
    call foo
exit
main ENDP
foo proc
    enter 2, 0;
    mov dword ptr [ebp - 4], 10;
    mov dword ptr [ebp - 8], 43;
    mov esp, ebp
    pop ebp
ret 8
foo endp
END main
```

- Leave replaces the last 2 lines of the procedure, it restores the value of ebp to esp as well as pop the return address.
- So the code becomes:

```
.data
.code
main PROC
    call foo
exit
main ENDP
foo proc
    enter 2, 0;
    mov dword ptr [ebp - 4], 10;
    mov dword ptr [ebp - 8], 43;
    mov esp, ebp
    pop ebp
ret 8
foo endp
END main
```

to

```
.code
main PROC
    call foo
exit
main ENDP
foo proc
    enter 2, 0;
    mov dword ptr [ebp - 4], 10;
    mov dword ptr [ebp - 8], 43;
    leave
ret 8
foo endp
END main
```

Using the Local directive:

- The easiest way is to use the local directive so we can avoid the whole problem entirely
- Now you just need to name the local variable and its type in the format: temp:dword, i:byte, ….
- The program will look like:

```
.code
main PROC
    call foo
exit
main ENDP
foo proc
    local temp1:dword, temp2: dword
    mov dword ptr [ebp - 4], 10;
    mov dword ptr [ebp - 8], 43;
ret 8
foo endp
END main
```

Recursion:

- It means to repeatedly call a function from inside itself, this can be done in assembly
- It can also be done using any way that has been taught in this chapter.
- U can use invoke or call
- U can send parameters or non at all
- U can also use the stack or use the register method they work in recursion.

Chapter 9:Strings shit

Introduction:

- This is a small chapter with only 5 instructions and 2 more concepts
- These involve strings and deal with 2 pointers: the source pointer ESI and the destination pointer EDI.
- Any thing ending in B is used for byte strings and arrays
- Anything ending in W is used for word strings and arrays
- Anything ending in D is used for Dword strings and arrays

The directional flag:

- The most important part of this chapter.
- The flag indicates whether the value of edi and esi will be incremented or decremented.
- It will be incremented or decremented by the corresponding value 1 for byte(B), 2 for word(W) or 4 for dword (D).
- To set the flag to be incremented use the CLD or clearing the directional flag
- To set the flag to be decremented use the STD or setting the directional flag.

MOVSB, MOVSW, MOVSD:

- This instruction is used to move the value pointed by esi into the location pointed by edi.

REP, REPZ, REPE, REPNZ, REPNE:

- This instruction repeats a single instruction a specified number of times
- Image a single line loop

- REPZ and REPE repeats while ecx > 0 and the 0 flag is clear
- REPNZ and REPNE repeats while ecx > 0 and the flag is set.
- When we put these together:

```
.data
source DWORD 20 DUP(0FFFFFFFFh)
target DWORD 20 DUP(?)
.code
cld                                ; direction = forward
mov   ecx,LENGTHOF source          ; set REP counter
mov   esi,OFFSET source            ; ESI points to source
mov   edi,OFFSET target            ; EDI points to target
rep   movsd                        ; copy doublewords
```

CMPSB, CMPSD, CMPSD:

- It is the same thing as CMP, but it will compare the source location and the destination location.
- We can use the same jump instruction for unsigned.
- We can use it like this:

```
.data
      source DWORD count DUP(?)
      target DWORD count DUP(?)
.code
      mov ecx,LENGTHOF source
      mov esi,OFFSET source
      mov edi,OFFSET target
      cld                ; direction = forward
      repe cmpsd         ; repeat while equal
```

- U can also extend the jumping to different labels

SCASB, SCASW, SCASD:

- This instruction will compare the value of al, ax, or eax with the location pointed by esi.
- The comparison will be done on the basis of B for al, W for word or D for Dword.
- For example we can look for a letter in an array

```
.data
alpha BYTE "ABCDEFGH",0
.code
mov   edi,OFFSET alpha          ; EDI points to the string
mov   al,'F'                    ; search for the letter F
mov   ecx,LENGTHOF alpha        ; set the search count
cld                             ; direction = forward
repne scasb                     ; repeat while not equal
jnz   quit                      ; quit if letter not found
dec   edi                       ; found: back up EDI
```

STOSB, STOSW, STOSD:

- This instruction will store the value of al, ax, or eax into the location pointed by edi.
- The storing will be done on the basis of B for al, W for word or D for Dword.

LODSB, LODSW, LODSD:

- Used to load the value pointed by esi into al, ax, or eax.
- The loading will be done on the basis of B for al, W for word or D for Dword.

2 dimensional arrays:

- We cannot create 2 dimensional arrays in assembly, we can stimulate it

- We imagine we have columns after some constant number

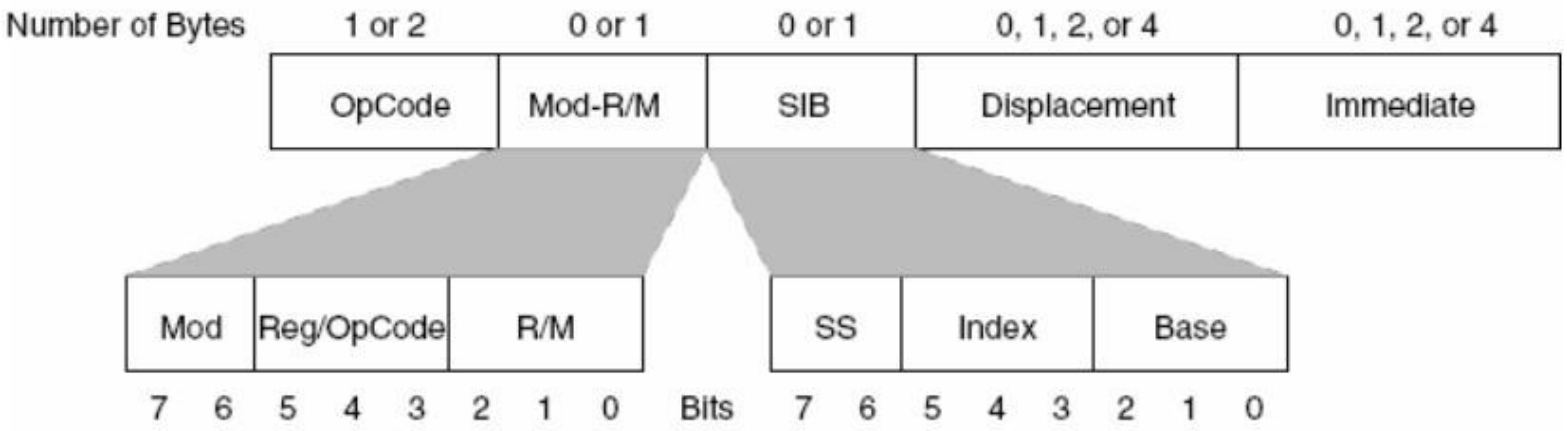
Chapter 10:Encoding and decoding

Introduction:

- Encoding the process of converting the instructions into hexadecimal so that the instructions maybe given to the processor, this is called as lexical analysis
- We are not concerned how the analysis is performed
- We will be focusing on how to convert the instructions.

The format:

- This is how the entire Instruction will be divided:



The opcode:

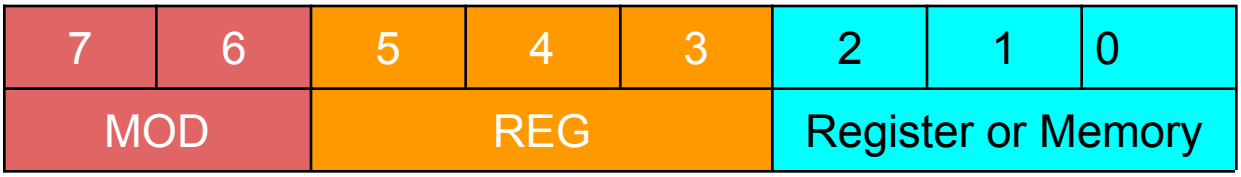
- Each instruction has a unique hexadecimal code
- The opcode is further divided into segments like:

- We are not concerned with the first 6 bits
- The bit marked D, show if the instruction deals in register to memory or vice versa
    - if **D is set to 0,** that mean that the destination is memory location and the source is a register, eg: mov var1, al
    - if **D is set to 1,** that means that the destination is a register and the source is a memory location, eg: mov al, var1.
- The bit marked S shows if the instruction is dealing with an 8 bit or an 16 bit operand, 32 bit operands are considered as 16 bit operands.
    - If **S is set to 0,** this means that the registers and memory location are 8 bit in size, eg mov bl, var2.
    - If **S is set to 1,** this means that the registers and memory location are 16 bit in size, eg mov cx, [esi].
- Here is a table to make it easier:

D	S	Destination	Source
0	0	8 bit memory	8 bit register
0	1	16/32 bit memory	16/32 bit register
1	0	8 bit register	8 bit memory
0	1	16/32 bit register	16/32 bit memory

MOD-R/M:

- This byte has 3 parts these are the most important
- Here is a visualizer



MOD:

- The MOD tell us weather this instruction has 1 , 2 or no operands
- Here is the table u must consult:

MOD value	Explanation
00	Memory with no displacement
01	Memory with 1 byte or 2 hex digits of displacement
10	Memory with 2 byte or 4 hex digits of displacement
11	Only registers are being used

For example:

Mov [esi + 01 23 4fh], bx	MOD = 10
MOV [esi + 34h], bx	MOD = 01
MOV value, bx	MOD = 00
Mov AX, BX	MOV = 11

REG and R/M:

- The reg part tells us what register is being utilised
- The R/M tells us what is the other operand.
- To get the value of R/M, u first need to see the value of MOD, there are 4 tables from which u need to pick one

<table>
<tr><th>Value of MOD</th><th>The table to pick from:</th></tr>
<tr><td>00</td><td>
<table>
<tr><th>R/M</th><th>MOD = 00</th></tr>
<tr><td>000</td><td>(BX) + (SI)</td></tr>
<tr><td>001</td><td>(BX) + (DI)</td></tr>
<tr><td>010</td><td>(BP) + (SI)</td></tr>
<tr><td>011</td><td>(BP) + (DI)</td></tr>
<tr><td>100</td><td>(SI)</td></tr>
<tr><td>101</td><td>(DI)</td></tr>
<tr><td>110</td><td>DIRECT ADDRESS</td></tr>
<tr><td>111</td><td>(BX)</td></tr>
</table>
</td></tr>
<tr><td>01</td><td>
<table>
<tr><th>R/M</th><th>MOD = 00</th><th>MOD = 01</th></tr>
<tr><td>000</td><td>(BX) + (SI)</td><td>(BX) + (SI) + D8</td></tr>
<tr><td>001</td><td>(BX) + (DI)</td><td>(BX) + (DI) + D8</td></tr>
<tr><td>010</td><td>(BP) + (SI)</td><td>(BP) + (SI) + D8</td></tr>
<tr><td>011</td><td>(BP) + (DI)</td><td>(BP) + (DI) + D8</td></tr>
<tr><td>100</td><td>(SI)</td><td>(SI) + D8</td></tr>
<tr><td>101</td><td>(DI)</td><td>(DI) + D8</td></tr>
<tr><td>110</td><td>DIRECT ADDRESS</td><td>(BP) + D8</td></tr>
<tr><td>111</td><td>(BX)</td><td>(BX) + D8</td></tr>
</table>
</td></tr>
</table>

10

R/M	MOD = 00	MOD = 01	MOD = 10
000	(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16
001	(BX) + (DI)	(BX) + (DI) + D8	(BX) + (DI) + D16
010	(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16
011	(BP) + (DI)	(BP) + (DI) + D8	(BP) + (DI) + D16
100	(SI)	(SI) + D8	(SI) + D16
101	(DI)	(DI) + D8	(DI) + D16
110	DIRECT ADDRESS	(BP) + D8	(BP) + D16
111	(BX)	(BX) + D8	(BX) + D16

11

R/M	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

- Another general way to understanding this is:

	7	6	5	4	3	2	1	0
The op code	The MOD		The code for the source operand			The code for the destination operand		

- This means the table for the values of MOD is **only for the destination operand.**
- For the source operand or REG always refer to the row of the MOD value of 11.

Displacement:

- The displacement for esi, edi or any displacement is placed after the 2 byte
- It is also placed in the little endian order

Immediate: these are placed at the end of the after any displacement

Some examples:

1. MOV [esi + 32], dx.

1	0	0	0	1	0	0	1

- D = 0 as we as the destination is memory and the source is a register
- W = 1 as we are dealing with 16 bit operands

0	1	0	1	0	1	0	0
MOD		REG			Register or Memory		

- MOD = 01 as it is a memory location with displacement of 1 byte
- The Source is DX, which in the row of MOD = 11 is equal to 010
- For the destination we look at the value of MOD (01) and look at the corresponding row we can see that the value of SI with D8(displacement of 8 bits) is equal to 100.
- The final answer come out to:

1000 1001 0101 0100 its hex is 89 54, but esi also has a displacement of 32 so we will attach that as well to get the final result: 89 54 32 that is our answer.

Unusual cases:

No operands:

- Sometimes there are instances where the instruction does not have any operands
- In those cases we do not use the previously mentioned format, instead we just refer to the Encoding manual.

1 operand:

Immediate operand:

- When the instruction is only an immediate, then we will write the op code and we will ignore the MOD - R/M and displacement, but we will write the immediate byte.
- For example the PUSH 17097ch, it will become 50 7c0917, where 50 is the opcode for PUSH and the immediate is in little endian format.

Single operand: register

- First we look up the hex code for the instruction then add the value of the register to the hex code from the table below

R/M	W = 0	W = 1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

- For example PUSH BX, the opcode for push is 50 and the value of bx is 3 together they make up : 53 which is the answer.

2 operand:

Register with an immediate (8 bit or 16 bit):

- If we see an instruction with a register and an immediate operand we will first look up the hex code of the instruction
- We will add the value of the register and then concat the immediate
- For example: MOV DL, 17.the hex code is B0 value of DL is 02 the answer will become: 8A 17 or B2 17
- For example: MOV BX, 1, the hex code for MOV is B8 add 03, code of BX, and concat the immediate value: Answer: BB 0100.(bcoz 0001 in little endian format is: 0100).