Lab # 01

Exploring Instruction Set Architecture x86 Machines

1. Instruction Set Architecture (ISA)

The ISA of a machine is the set of its attributes a system programmer needs to know in order to develop system software or an assembler requires for translation of an Assembly Language (HLL) code into machine language. Examples of such attributes are (but not limited to):

- *Instruction Set*
- *Programmer Accessible Registers:* these are the general purpose registers (GPR) within a processor in contrast to some special purpose registers only accessible to the system hardware and Operating System (OS)
- *Memory-Processor Interaction*
- Addressing Modes: means of specifying operands in an instruction (e.g. immediate mode, direct mode, indirect mode, etc.)
- *Instruction Formats:* breakup of an instruction into various fields (e.g. opcode, specification of source and destination operands, etc.) ISA is also known as the programmer‟s view or software model of the machine.

ISA is also known as the programmer‟s view or software model of the machine.

2. ISA of x86 Machines

This represents a family of machines beginning with 16-bit 8086/8088 microprocessors. (An n-bit microprocessor is capable of performing n-bit operations). As an evolutionary process, Intel continued to add capabilities and features to this basic ISA. The 80386 was the first 32-bit processor of the family. The ISA of 32-bit processor is regarded as IA-32 (IA for Intel Architecture) or x86-32 by Intel. IA-64 was introduced in Pentium-4F and later processors. Operating Systems are now also categorized on the basis of the architecture they can run on. A 64-bit OS can execute both 64-bit and 32-bit applications. We will limit scope of our discussion to IA-32. The ISA of 8086/88 processor shown in Figure 1-1

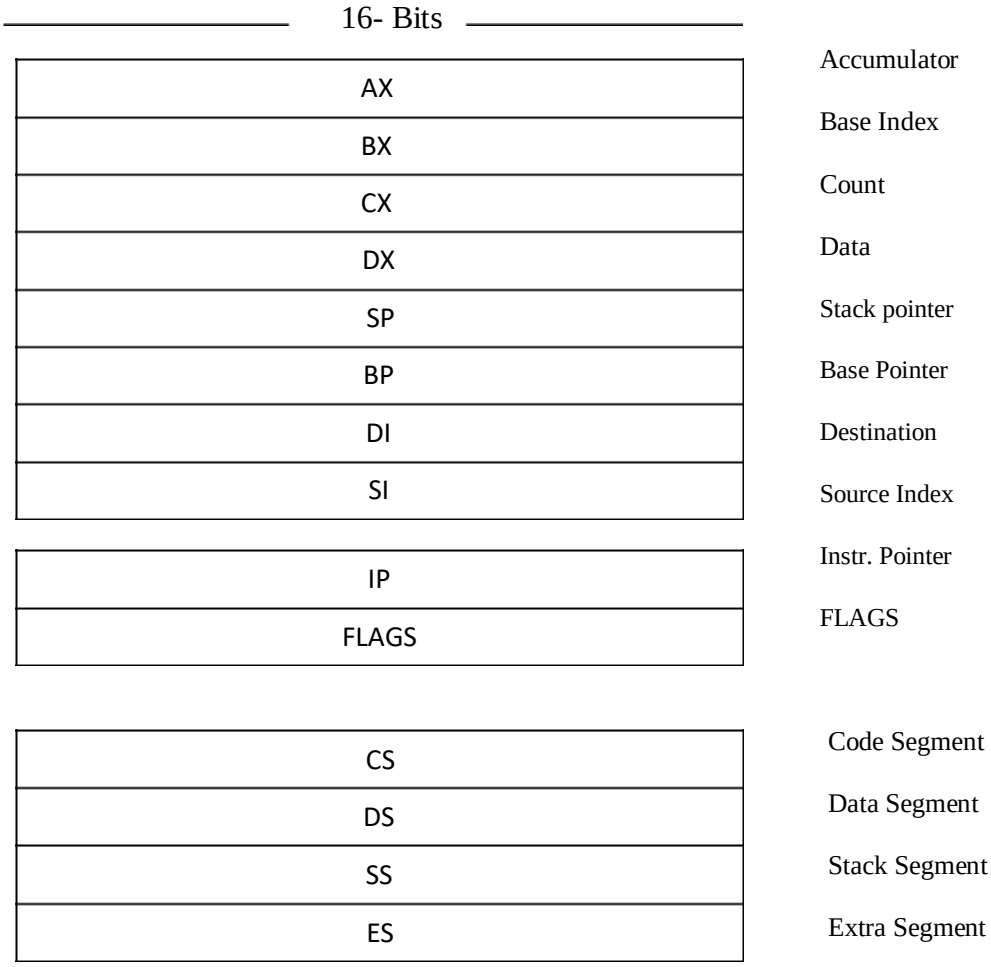


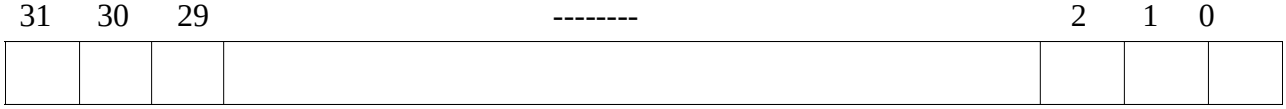
Figure 1-1 Internal Architecture of 8088/8086 microprocessor

2.1 Registers

Registers are temporary storage locations inside the processor. A register can be accessed more quickly than a memory location. Different registers serve different purposes. Some of them are described below:

2.1.1 General-Purpose Registers

EAX, EBX, ECX and EDX are called data or general purpose registers. (E is for extended as they are 32-bit extensions of their 16-bit counter parts AX, BX, CX and DX in 16-bit ISA). Other General purpose registers includes BP, DI and SI registers . Bits in a register are conventionally numbered from right to left, beginning with 0 as shown below.



Apart from accessing the register as a whole, EAX, EBX, ECX, EDX registers can be accessed in pieces as illustrated in Figure 1-2

.

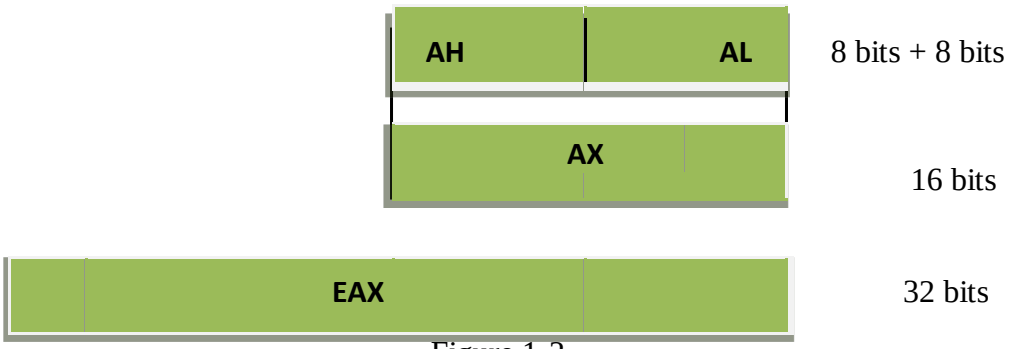


Figure 1-2

Register	Purpose
AX	For 16/32 bits operations, AX registers stores operands for arithmetic operations.
BX	Used to hold starting location of a memory region within data segment.
CX	It defined as a counter, primarily used in loop instructions.
DX	It is used to hold the part of result from multiplication or a part of dividend before division.

2.1.2 Index Registers

ESI(Extended Source Index) and EDI(Extended Destination Index) registers are respectively used as source and destination addresses in string operations. They can also be used to implement array indices.

2.1.3 Pointer Registers

The EIP (Extended Instruction Pointer) register contains the offset in the current code segment for the next instruction to be executed. (Segments will be explained shortly). ESP(Extended Stack Pointer) and EBP(Extended Base Pointer) are used to manipulate stack - a memory area reserved for holding parameters and return address for procedure calls. ESP holds address of top of stack, location where the last data item was pushed. EBP is used in procedure calls to hold address of a reference point in the stack.

2.1.4 Flags Register

EFLAGS register is never accessed as a whole. Rather, individual bits of this register either control the CPU operation (control flags) or reflect the outcome of a CPU operation (status flag). Table below gives some of the commonly used control and status flags.

BIT	Name	Type	Description
0	Carry Flag (CF)	Status	Contains carry from a high-order (leftmost) bit following an arithmetic operation; also, contains the contents of the last bit of a shift or rotate operation.
2	Parity Flag (PF)	Status	It indicates the parity of result i.e. if lower order 8 bits of the result contains eve number of 1"s the parity flag is set to 1.0 otherwise.
4	Auxiliary Carry (AF)	Status	If an operation performed in ALU generates a carry/borrow from lower nibble (D_0-D_3) to upper nibble (D_4-D_7), the AF is set to1, 0 otherwise.
6	Zero Flag (ZF)	Status	Indicates the result of an arithmetic or comparison operation (0 = nonzero and 1 = zero result)
7	Sign Flag (SF)	Status	Contains the resulting sign of an arithmetic operation (0 = positive and 1 = negative).
8	Trap Flag (TF)	Control	Permits operation of the processor in single-step mode.
9	Interrupt Flag (IF)	Control	Indicates that all external interrupts, such as keyboard entry, are to be processed or ignored.
10	Direction Flag (DF)	Control	Determines left or right direction for moving or comparing string (character) data.
11	Overflow Flag (OF)	Status	Indicates overflow resulting from some arithmetic operation

Table below gives the clear understanding that how these registers can be view and use in programming**.**

SYMBOL	*FLAGS*	CLEAR (0)	SET (1)
O	Overflow Flag	NV	OV
D	Direction Flag	UP	DN
I	Interrupt Flag	DI	EI
S	Sign Flag	PL	NG
Z	Zero Flag	NZ	ZR
A	Auxiliary Flag	NA	AC
P	Parity Flag	PO	PE
C	Carry Flag	NC	CY

2.2 Memory Addressing

A 32-bit processor uses 32-bit addresses and thus can access 232B = 4GB physical memory. Depending on the machine, a processor can access one or more bytes from memory at a time. The number of bytes accessed simultaneously from main memory is called word length of machine. Generally, all machines are byte-addressable i.e.; every byte stored in memory has a unique address. However, word length of a machine is typically some integral multiple of a byte. Therefore, the address of a word must be the address of one of its constituting bytes. In this regard, one of the following methods of addressing (also known as byte ordering) may be used. Big Endian – the higher byte is stored at lower memory address (i.e. Big Byte first). MIPS, Apple, Sun SPARC are some of the machines in this class. Little Endian - the lower byte is stored at lower memory address (i.e. Little Byte first). Intel‟s machines use little endian.

Consider for example, storing 0xA2B1C3D4 in main memory. The two byte orderings are illustrated below.

Address	Contents
2030	A2
2031	B1
2032	C3
2033	D4

BIG

Contents	Address
D4	2030
C3	2031
B1	2032
A2	2033

LITTLE

2.3 Memory Models

IA-32 can use one of the three basic memory models:

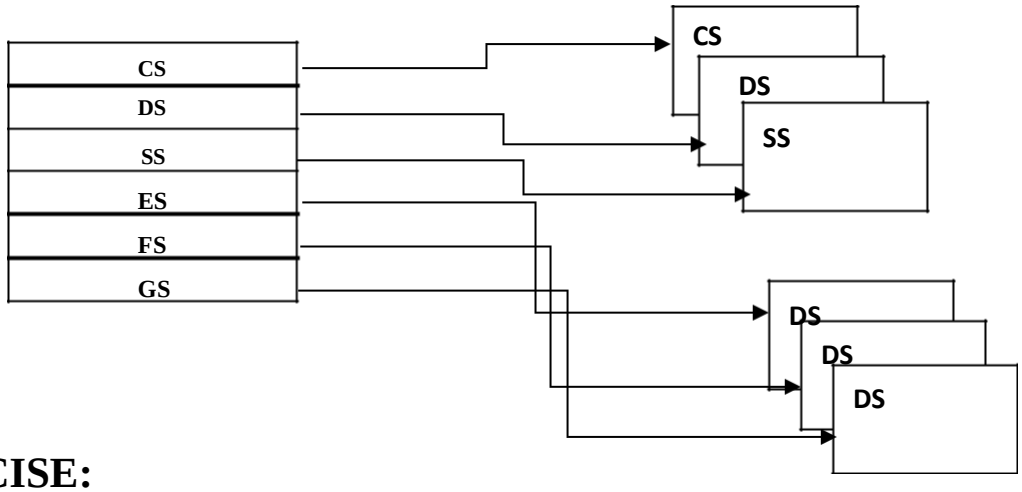
Flat Memory Model – memory appears to a program as a single, contiguous address space of 4GB. Code, data, and stack are all contained in this address space, also called the linear address space

Segmented Memory Model – memory appears to a program as a group of independent memory segments, where code, data, and stack are contained in separate memory segments. To address memory in this model, the processor must use segment registers and an offset to derive the linear address. The primary reason for having segmented memory is to increase the system's reliability by means of protecting one segment from other.

2.4 Segment Registers

The segment registers hold the segment selectors which are special pointers that point to start of individual segments in memory. The use of segment registers is dependent on the memory management model in use.

The segment registers (CS, DS, SS, ES, FS, and GS) hold 16-bit segment selectors. Each of the segment registers is associated with one of three types of storage: code, data, or stack. For example, the CS register contains the segment selector for the code segment, where the instructions being executed are stored. The processor fetches instructions from the code segment, using a logical address that consists of the segment selector in the CS register and the contents of the EIP register. The EIP register contains the offset within the code segment of the next instruction to be fetched.



EXERCISE:

1. Fill in the following tables to show storage of 0xABDADDBA at address 1996 in the memory of a machine using (i) little endian (ii) big endian byte ordering.

Contents

	Address
	1996
	1997
	1998
	1999

BIG Endian

Contents

LITTLE Endian

2. What is the significance of learning ISA of a processor?

3. Show the ECX register and the size and position of the CH, CL, and CX within it.

4. For each add instruction in this exercise, assume that EAX contains the given contents before the instruction is executed. Give the contents of EAX as well as the values of the CF, OF, SF, PF, AF and ZF after the instruction is executed. All numbers are in hex. (Hint: add eax, 45 adds 45 to the contents of register eax and stores the result back in eax)

Contents of EAX (Before)	Instruction	Contents of EAX (After)	CF	OF	SF	PF	AF	ZF
00000045	Add eax,45							
FFFFFF45	Add eax,45							
000000045	sub eax , 46							
FFFFFF45	Sub eax, 9A							
FFFFFFFF	Add eax, 1							

Lab # 02
Introduction to Assembly Language Tools

Introduction to Assembly Language Tools

Software tools are used for editing, assembling, linking, and debugging assembly language programming. You will need an assembler, a linker, a debugger, and an editor. These tools are briefly explained below.

Assembler
An **assembler** is a program that converts **source-code** programs written in **assembly language** into **object files** in machine language. Popular assemblers have emerged over the years for the Intel family of processors. These include MASM (Macro Assembler from Microsoft), TASM (Turbo Assembler from Borland), NASM (Netwide Assembler for both Windows and Linux), and GNU assembler distributed by the free software foundation.

Linker
A **linker** is a program that combines your program's **object file** created by the assembler with other object files and **link libraries**, and produces a single **executable program**.

Debugger
A **debugger** is a program that allows you to trace the execution of a program and examine the content of registers and memory.

Editor
You need a text editor to create assembly language source files. You can use NotePad , or any other editor that produces plain ASCII text files.

Emu8086

Emu8086 combines an advanced source editor, assembler, disassemble and software emulator (Virtual PC) with debugger. It compiles the source code and executes it on emulator step by step.
Visual interface is very easy to work with. You can watch registers, flags and memory while your program executes.
Arithmetic & Logical Unit (ALU) shows the internal work of the central processor unit (CPU).
Emulator runs programs on a Virtual PC, this completely blocks your program from accessing real hardware, such as hard-drives and memory, since your assembly code runs on a virtual machine, this makes debugging much easier. 8086 machine code is fully compatible with all next generations of Intel's microprocessors, including Pentium II and Pentium 4. This makes 8086 code very portable, since it runs both on ancient and on the modern computer systems. Another advantage of 8086 instruction set is that it is much smaller, and thus easier to learn.

EMU8086 Source Editor
The source editor of EMU86 is a special purpose editor which identifies the 8086 mnemonics, hexadecimal numbers and labels by different colors as seen in Figure 1.

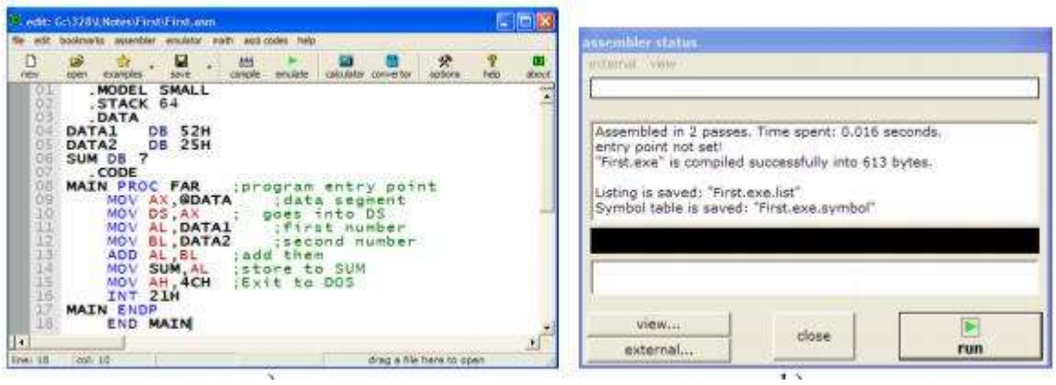


Figure:1

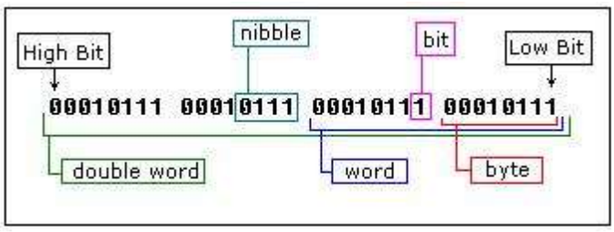
The compile button on the taskbar starts assembling and linking of the source file. A report window is opened after the assembling process is completed. Figure 2 shows the emulator of 8086 which gets opened by clicking on emulate button.

Figure 2

Data Representation in EMU8086

Binary System

Computers are not as smart as humans are (or not yet), it's easy to make an electronic machine with two states: **on** and **off**, or **1** and **0**. Computers use binary system, binary system uses 2 digits: **0, 1**And thus the **base** is **2**. Each digit in a binary number is called a **BIT**, 4 bits form a **NIBBLE**, 8 bits form a **BYTE**, two bytes form **WORD**, two words form a **DOUBLE WORD** (rarely used)



In emu8086, There is a convention to add "b" in the end of a binary number, this way we can determine that 101b is a binary number with decimal value of 5.

Decimal System

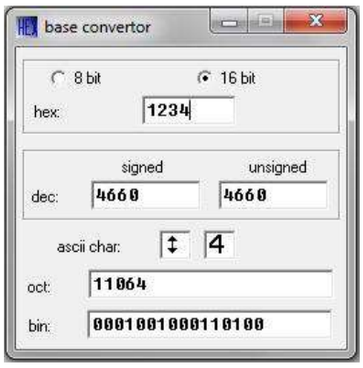
Most people today use decimal representation to count. In the decimal system there are 10 digits: **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**. These digits can represent any value, for example: **754**.

Hexadecimal System

Hexadecimal System uses 16 digits:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F and thus the base is 16. Hexadecimal numbers are compact and easy to read. There is a convention to add "h" in the end of a hexadecimal number.

Base Converter

Emulator 8086 provides base converter facility which can convert any number (signed/unsigned) into octal, hexadecimal, decimal and binary number systems.



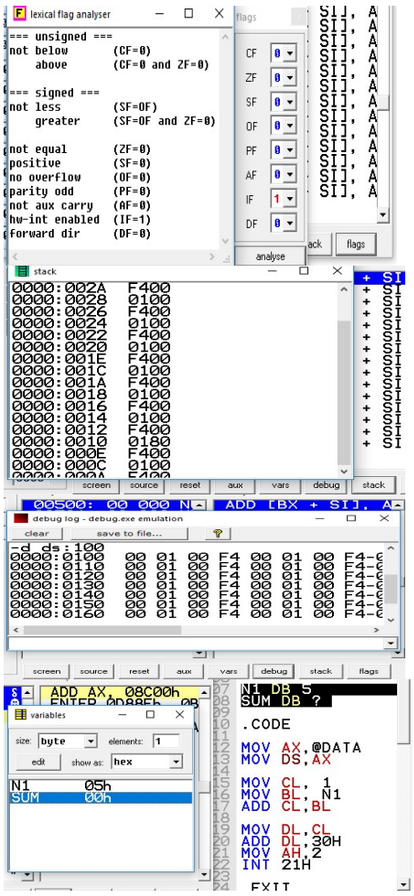
Example: Use Emu8086 to make the following calculations:

10100101b = ? d
1234h = ? d
39d = ? h

1. Start *Emu8086* by selecting its icon from the start menu, or by running Emu8086.exe
3. Choose “Math” and specify “Base Convertor” in emu8086.
4. Enter one of the numbers like in the Figure .

Emulator 8086 Environment

- **Flags** : By pressing “flags” in the bottom-right corner of the window(figure 3)you can observe flag values at the execution of instruction written in a program ,

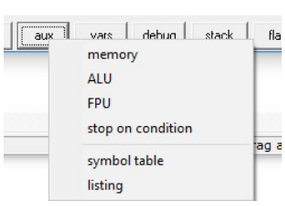


- **Stack:** stack segment window is popup by pressing “stack”. Stack is directly affected when call and return instructions used in the program.

- **Debug**: emulator86 also provide debug environment where you can view and also edit the contents of memory segments , register values through debug commands.

- **Variables**: Variable/constants declare in the program can also be viewed through “vars” .

- **Aux:** you can view various options(listing ,memory, ALU and FPU) under this tab as mention in the figure below.



User: ISHAAN,glaitm
Key:27R3VDEFYFX4NoVC3FRTQZX

EXERCISE:

1. Name four software tools used for assembly language programming.

_____

_____

_____

2. Convert the following numbers using Base Converter:
   a) 10110011b (binary to hexadecimal)
   b) 455d (decimal to binary)
   c) 2AFh (hexadecimal to binary)
   d) 0A1h (hexadecimal to binary)

   Do all the calculations manually and compare with the results obtained using base converter.

_____

_____

_____

_____

_____

_____

3 . Write each of the following 8-bit Signed Binary Integers in Decimal:

a) 11011100 = b) 10010001 =

c) 10001111 = d) 10000000 =

Lab # 03

Assembly Language Program Structure

An **assembly language** (or **assembler language**) is a low-level programming language for a computer, or other programmable device, in which there is a very strong correspondence between the language and the architecture's machine code instructions. Assembly language is converted into executable machine code by a utility program referred to as an assembler; the conversion process is referred to as assembly, or assembling the code.

How Does Assembly Language Relate to Machine Language?

Machine language is a numeric language specifically understood by a computer's processor (the CPU). All x86 processors understand a common machine language.
Assembly language consists of statements written with short mnemonics such as ADD, MOV, SUB, and CALL. Assembly language has a *one-to-one* relationship with machine language: Each assembly language instruction corresponds to a single machine-language instruction.

How Does C++ and Java Relate to Assembly Language?

High-level languages such as C++ and Java have a one-to-many relationship with assembly language and machine language. A single statement in C++ expands into multiple assembly language or machine instructions. We can show how C++ statements expand into machine code. Most people cannot read raw machine code, so we will use its closest relative, assembly language.

Is Assembly Language Portable?

A language whose source programs can be compiled and run on a wide variety of computer systems is said to be *portable*. A C++ program, for example, should compile and run on just about any computer, unless it makes specific references to library functions that exist under a single operating system. A major feature of the Java language is that compiled programs run on nearly any computer system.
Assembly language is not portable because it is designed for a specific processor family. There are a number of different assembly languages widely used today, each based on a processor family. Some well-known processor families are Motorola 68x00, x86, SUN Sparc, Vax, and IBM-370. The instructions in assembly language may directly match the computer‟s architecture or they may be translated during execution by a program inside the processor known as a *microcode interpreter.*

Assembly Language Syntax:

Name: operation operand (s) ;comment

- **Name field**

Assembler translate name into memory addresses. It can be 31 characters long. The NAME field allows the program to ref of code by name.

Examples of legal names

- COUNTER1
- @character
- SUM_OF_DIGITS
- .TEST

Examples of illegal names

- TWO WORDS
- 2abc
- A45.28

- **Operation field**

It contains symbolic operation code (opcode) called "**mnemonics**"(MOV, ADD, e.t.c.). The mnemonic (instruction) and operands together accomplish the tasks for which program was written. The assembler translates mnemonics into machine language opcode.

- **Operand field**

It specifies the data that are to be acted on by the operation. An instruction may have a zero, one or two operands.

Examples

- NOP
- INC AX
- ADD AX, 2

- **Comment field**

A semicolon marks the beginning of a comment. Good programming practice dictates comment on every line

Examples

- MOV CX, 0 ; move 0 to CX
- MOV CX, 0 ; CX counts terms, initially 0

✓ Program Structure:

The machine language programs consist of code, data and stack. Each part occupies a memory segment. The same organization is reflected in an assembly language program. This time, the code, data and stack structured as program segments. Each program segment is translated into a memory segment by the assembler.

✓ Memory Models:

The size of code and data in a program can have determined by specifying a memory model using the **.MODEL** directive. The syntax is **.MODEL** memory_model The most frequently used memory models are SMALL, MEDIUM, COMPACT and LARGE. They are described in table below. Unless there is a lot of code or data, the appropriate model is SMALL. The .MODEL directive should come before any segment definition.

Model	Description
SMALL	Code in one segment Data in one segment
MEDIUM	Code in more than one segment Data in one segment
COMPACT	Code in one segment Data in more than one segment
LARGE	Code in more than one segment Data in more than one segment No array larger than 64KB
HUGE	Code in more than one segment Data in more than one segment Arrays may be larger than 64KB

- **Stack Segment:**

The purpose of the stack segment declaration is to set aside a block of memory (the stack area) to store the stack. The declaration syntax is *.STACK size*For example, .STACK 100H sets aside 100h bytes for the stack area (a reasonable size for most applications). If size is omitted, 1KB is set aside for the stack-area.

- **Data Segment:**

A program‟s data segment contains all the variable definitions. Constant definitions are often made here as well, but they may be placed elsewhere in the program since no memory allocation is involved. To declare a data segment, we use the directive .DATA, followed by variable and constant declaration. For example,

```
.DATA
Word1 Dw 2
 Msg Db “This Is A Message”
```

- **Code Segment:**

The code segment contains a program‟s instructions. The declaration syntax is *.CODE*

Here name is the optional name of the segment (there is no need for a name in a SMALL program, because the assembler will generate an error). Inside a code segment, instructions are organized as procedures. The simplest procedure definition is:

Name PROC ;body of the procedure
Name ENDP ;where name is the name of the procedure;

PROC and ENDP are pseudo-ops that delineate the procedure. Here is an example of a code segment definition:

```
.CODE
MAIN            PROC
 ; main procedure instructions
MAIN ENDP
; other procedures go here
End Main
```

```
.MODEL SMALL
.STACK 100H
 .DATA
; data definitions go here
.CODE
MAIN PROC
; instructions go here
MAIN ENDP
 ;other procedures go here
END MAIN
```

;The last line in the program should be the END directive, followed by name of the main procedure

ASCII Character Chart ASCII, American Standard Code for Information Interchange, is a scheme used for assigning numeric values to punctuation marks, spaces, numbers and other characters. ASCII uses 7 bits to represent characters. The values 000 0000 through 111 1111 or 00 through 7F are used giving ASCII the ability to represent 128 different characters. An extended version of ASCII assigns characters from 80 through FF.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	⌂

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	ø
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	▒	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	|	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	▒	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	⌂

Disk operating system (DOS) routines

INT 21H is used to invoke a large number of DOS function. The type of called function is specified by pulling a number in AH register. For example

AH=1 input with echo

AH=2 single-character output

AH=9 character string output

AH=8 single-key input without echo

AH=0Ah character string input

Single-Key Input with Echo: Output: AL= ASCII code if character key is pressed, otherwise 0.

```
MOV AH,1
INT 21      ; read character will store in AL register
```

Single –Key Input without Echo:

```
MOV AH,8
INT 21 ; read character will store in AL register
```

Single-Character Output: DL= ASCII code of character to be displayed.

```
MOV AH,2
MOV DL,"?"
INT 21h     ; displaying character ?
```

Input a String:

```
MOV AH, 0A
INT 21
```

Display a String: DX= offset address of a string. String must end with a „$" character.

```
LEA DX, n1
MOV AH,9
INT 21
```

EXERCISE:

Q1 Write down the CODE of following scenario

- Input a one-digit number from keyboard
- Save that number to DL register
- Display the number you have entered in new line

Q2 Write down the CODE to input First Letter of your name and NOT echo.

Q3 Write down the CODE to Display your name through ASCII Code (One character @ a time) each new character should display in new line.

Q4 Design first letter of your name through suitable ASCII codes.

Lab # 04

Assembler Directives, Constants & Identifiers

Directives:

A directive is a command embedded in the source code that is recognized and acted upon by the assembler. Directives do not execute at runtime. Directives can define variables, macros, and procedures. They can assign names to memory segments and perform many other housekeeping tasks related to the assemble or assembler directive is a message to the assembler that tells the assembler something it needs to know in order to carry out the assembly process; for example, an assemble directive tells the assembler where a program is to be located in memory.

Data Declarations:

The default number system used by the assembler is decimal. Using other number systems entail appropriate suffixes as shown below:

Number System	**Suffix**
Binary	B
Hexadecimal	H
Octal	O or Q
Decimal	None

Table 1

A hexadecimal value must start with a digit. For example, code 0a8h rather than a8h to get a constant with value A816. The assembler will interpret a8h as a name.

Some Assembler Directives:

Pseudo-op	**Stands for**
DB	define byte
DW	define word
DD	define double word (two consecutive words)
DQ	define quad word (four consecutive words)
DT	define ten bytes (five consecutive words.

Byte Variables

Assembler directive format assigning a byte variable

Name DB initial value

A question mark ("?") place in initial value leaves variable uninitialized

```
I       DB  4           ;define variable I with initial value 4
J       DB  ?           ;Define variable J with uninitialized value
Name    DB  "Course"    ;allocate 6 bytes for name
K       DB  5, 3,-1     ;allocate 3 bytes
```

Other data type variables have the same format for defining the variables like:

```
Name        DW          initial value
```

Named Constants

- EQU pseudo-op used to assign a name to constant.
- Makes assembly language easier to understand.
- No memory allocated for EQU names.

```
LF      EQU     0AH
```

- MOV DL, 0AH
- MOV DL, LF

```
PROMPT  EQU     "Type your name"
```

- MSG DB "Type your name"
- MDC DB PROMPT

Integer Constants

An *integer constant* (or integer literal) is made up of an optional leading sign, one or more digits, and an optional suffix character (called a *radix*) indicating the number‟s base:

[{+ | -}] *digits* [*radix*]

Radix may be one of the following (uppercase or lowercase), If no radix is given, the integer constant is assumed to be decimal. (Refer table 1 for radix examples)

Character Constants

A *character constant* is a single character enclosed in single or double quotes.
Examples are 'A', "d".

String Constants

A *string constant* is a sequence of characters (including spaces) enclosed in single or double quotes: “ABC' , "Good night, Gracie” etc.

Reserved Words

Reserved words have special meaning in Assembler and can only be used in their correct context. There are different types of reserved words:

• Instruction mnemonics, such as MOV, ADD, and MUL
• Register names
• Directives, which tell Assembler how to assemble programs
• Attributes, which provide size and usage information for variables and operands. Examples are BYTE and WORD
• Operators, used in constant expressions

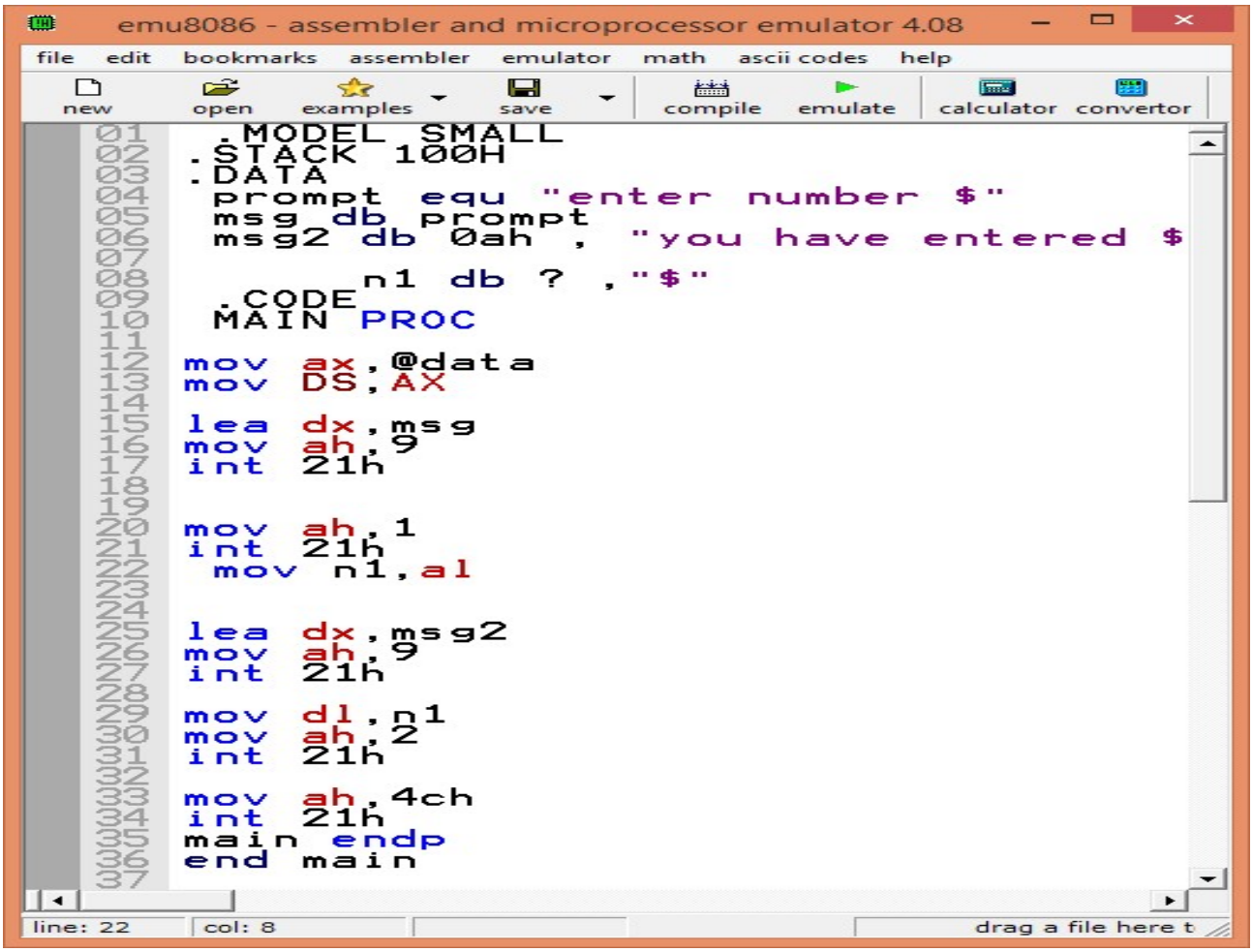
Identifiers

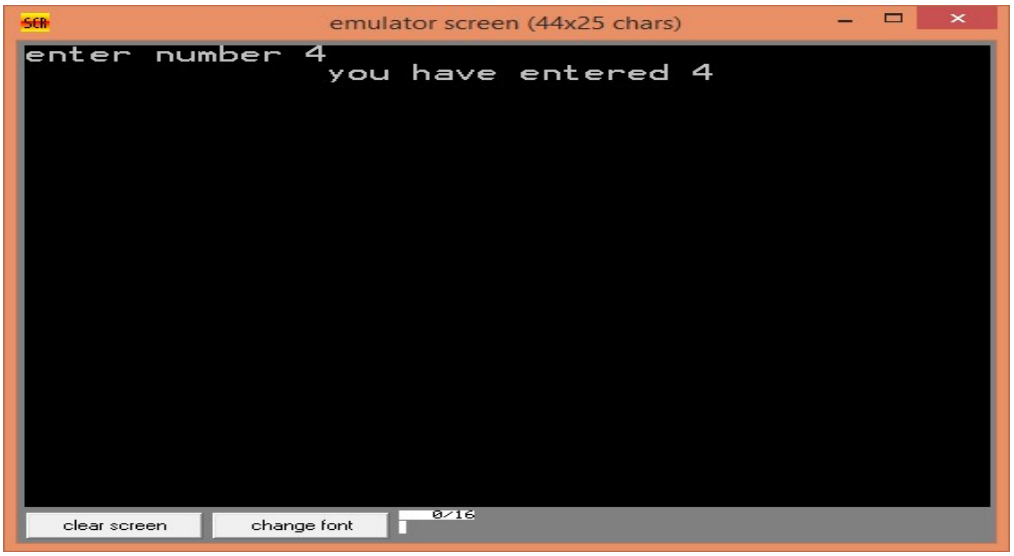
An *identifier* is a programmer-chosen name. It might identify a variable, a constant, a procedure, or a code label. Keep the following in mind when creating identifiers:

• They may contain between 1 and 247 characters.
• They are not case sensitive.
• The first character must be a letter (A..Z, a..z), underscore (_), @ , ?, or $.
Subsequent characters may also be digits.
• An identifier cannot be the same as an assembler reserved word.

```
.data
   prompt equ "enter  number $"
   msg db prompt
   msg2 db  0ah,"you have enterded $"
   n1 db ?,"$"

.code
main proc
 mov ax,@data  ; load data to DS
 mov DS, AX
lea dx,msg
mov ah,9
int 21h
mov ah,1
int 21h
mov n1,al  ; save al value to n1

lea dx,msg2
mov ah,9
int 21h
mov dl,n1
mov ah,2
int 21h
mov ah,4ch
int 21h

main endp
end main
```

```
C:\>link l41.obj

Microsoft (R) Overlay Linker  Version 3.60
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.

Run File [L41.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:

C:\>l41
enter  number 4
you have enterded 4
C:\>
```





EXERCISE:

Q1. Refer to p-1, assemble the program and examine the listing file. Determine the opcode and size of each instruction inside the main procedure.

_____

_____

_____

_____

_____

Q2. Refer to p-1, examine the listing file and determine how many bytes reserved by each constant/ variable also determine the address range occupied by each variable.

_____

_____

_____

Q3. Find the values that the assembler will generate for each of the directives below. Write your answer using two hex digits for each byte generated.

i.	Byte1 DB 10110111b	Value: _____
ii.	Byte2 DB 0B8h	Value: _____
iii.	Byte 3 DB „D“	Value: _____
iv.	Byte 3 DB „d“	Value: _____
v.	Byte4 DW 10000000b	Value: _____
vi.	Byte5 DW 43,41,4F	Value: _____

Q4. Write an assembly language program to swap numbers stored at value1 and value2. Assume value1 is last-digit-of-roll# and value2 is second-last-digit-of-your-roll#

Q5 Write down the code to generate following output using DB.

C
A
O

Lab # 05

Arithmetic Instructions

Integer Addition and Subtraction Instructions

Each addition/Subtraction instruction has the following syntax:

Add/sub destination, source Operation : Operand1 ⟵ Operand1+Operand2

Possible Operands:

Operand1	**Operand2**
Register	Register
Memory	Register
Register	constant data
Memory	constant data

The integer in source is added to the integer at destination and the sum replaces the old value at destination. The SF, ZF, OF, CF, PF and AF flags are set according to the value of the result of the operation. A subtraction instruction also has the same format. The integer at source is subtracted from the integer at destination and the difference replaces the old value at destination. One reason that 2's complement notation is used to represent signed numbers is that it does not require special hardware for addition or subtraction – the same circuits can be used to add/subtract unsigned and 2's complement numbers.

Two very useful instructions are inc and dec instructions which have the following formats:

Inc/Dec Operand Operation: Operand ⟵ Operand ± 1

These instructions treat the value of the destination as an unsigned integer. They are especially useful for incrementing and decrementing counters. They sometimes take fewer bytes of code than corresponding addition and subtraction instructions.

Another useful instruction is neg instruction having the following format:

Neg Operand Operation: Operand ⟵ 0 - (OPERAND)

It negates (takes the 2's complement of) destination replacing the value in the destination by the new value. Hence, a positive value gives a negative result and negative value will become positive.

Multiplication Instructions

There are two versions of multiplication instructions in the 80x86 assembly language. The mul instruction is for unsigned multiplication. Operands are treated as unsigned numbers. The imul instruction is for signed multiplication. Operands are treated as signed numbers and result is

positive or negative depending on the signs of the operands. The formats of these instructions are discussed below.

MUL Operand	Operation: AX	AL * (8 bit value)
	DX AX	AX * (16 bit value)

Single operand may be byte, word, double word or quad word in register or memory (not immediate) and specifies one factor – that is the other number to be multiplied is always the accumulator. Location of this factor is implied. For example, AL for byte-size source, AX for word source and EAX for double word source. When a byte source is multiplied by the value in AL, the product is put in AX. When a word source is multiplied by the value in AX, the product is put in DX:AX (this strange placement is to keep backward compatibility) with the high-order 16 bits in DX and the low-order 16 bits in AX. When a double word source is multiplied by the value in EAX, the product is put in EDX:EAX with the high order 32 bits in DX and the low-order 32 bits in AX. In each case the source operand is unchanged unless it is half of the destination. The format of imul instruction as presented below:

IMUL Operand

Division Instructions

The division operation generates two elements - a quotient and a remainder. The dividend is an accumulator. This instruction can work with 8-bit, 16-bit or 32-bit operands. There are two types of division instructions in x86 assembly. The instruction idiv source is for signed operands, while div source is for unsigned operands. The source identifies the divisor which may be in byte, word, double word or quad word. It may be in memory or register, but not an immediate operand. The dividend is always double the size of divisor.

DIV Operand (S8/S16) Operation: AL ← Q [(AX)/S8]

AH ← R[(AX)/S8]

AX ← Q[(DXAX)/S16]

DX ← R[(DXAX)/S16]

All division operations must satisfy the relation: *dividend = quotient*divisor + remainder* .For signed division, the remainder will have same sign as dividend and the sign of quotient will be positive if signs of divisor and dividend agree, negative otherwise.

EXERCISE:

Q1 Assumes num1 and num2 3 & 2 respectively, Your assembly code should generate the output as mention below:

Num1+num2 = a ; where a,b,c,d are the results of specific operation.
Num1- num2 = b
Num1* num2= c
Num1 / num2= d

Q2 i. write an assembly language program , assume 8,3,2 values for variables x, y and z ,the output should display an appropriate label and value of the expression x –2 y + 2z

Q2 ii. Modify above code by taking input for x, y and z ,and compute the value of x –2 y + 2z

Q3 Examine the list file of Q2i and write down the bytes reserved in memory by x,y,z and also note the operation codes of each instruction

Lab # 06

Program Control Instructions (Implementing Branching in x86 Assembly Language)

Conditional execution in assembly language is accomplished by several looping and branching instructions. These instructions can change the flow of control in a program. Program control execution is observed in two scenarios.

- Un Conditional Jump
- Conditional jump

Unconditional Branches (Jumps)

An unconditional branch (jump) instruction transfers control to a specified label in the program without testing any condition. This is similar to goto in a HLL. Transfer of control may be forward, to execute a new set of instructions or backward, to re-execute the same steps. The 80x86 jump instruction has the following format:

Syntax: Jmp *label*

Here a program is presented that uses jump instruction to loop forever through the program and calculates the 1+2+3+……+n sum .

```
.code
Main proc
        Mov  ax,0
        Mov  bx,0
Forever: Inc bx
        Add ax,bx
        Jmp forever
Main endp
```

Conditional Branches

Unlike an unconditional branch, a conditional branch tests a condition before transfer of control.This is performed by a set of jump instructions *j<condition>* depending upon the condition. The conditional instructions transfer the control by breaking the sequential flow and they do it by changing the offset value in IP.

This conditional branching is used to implement if structures, other selection structures, and loop structures in 80x86 assembly language. Most conditions considered by the conditional jump instructions are settings of flags in the FLAGS register. For example, jz lable1 means to transfer control to the instruction to label1, if the zero flag ZF is set to 1. Conditional branch instructions

don not modify flags; they just react to previously set flag values. Most common way to set flags for conditional branches is to use compare instruction that has the following format:

Syntax: **cmp** operand1, operand2

Flags are set the same as for the subtraction operation operand1–operand2. Operands, however, are not changed. Following are the conditional jump instructions used on unsigned data used for logical operations:

Instruction	Description	Flags tested
JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump not Equal or Jump Not Zero	ZF
JA/JNBE	Jump Above or Jump Not Below/Equal	CF, ZF
JAE/JNB	Jump Above/Equal or Jump Not Below	CF
JB/JNAE	Jump Below or Jump Not Above/Equal	CF
JBE/JNA	Jump Below/Equal or Jump Not Above	AF, CF

The following conditional jump instructions have special uses and check the value of flags

Instruction	Description	Flags tested
JXCZ	Jump if CX is Zero	None
JC	Jump If Carry	CF
JNC	Jump If No Carry	CF
JO	Jump If Overflow	OF
JNO	Jump If No Overflow	OF

JP/JPE	Jump Parity or Jump Parity Even	PF
JNP/JPO	Jump No Parity or Jump Parity Odd	PF
JS	Jump Sign (negative value)	SF
JNS	Jump No Sign (positive value)	SF

Example-1

Un Conditional	**Conditional**	**Both Un-conditional & Conditional**

Mov dl,1
Add dl,30h
Start: Mov ah,2
Int 21h
Inc dl
Jmp **start**
	Mov dl,1
	Add dl,30h
	Start:
	Mov ah,2
	Int 21h
	Inc dl
	Cmp dl,35h
	Jne **Start**
		Mov dl,1
		Add dl,30h **Endpara:**
		Start:
		Mov ah,2 Mov ah,4ch
		Int 21h Int 21h
		Inc dl
		Cmp dl,5
		Jne **start**
		Jmp endpara

Implementation of if Structure

Let's implement the following pseudo-code in x86 assembly language.

```
if (total ≥100)
   then add value to total;
end if;
```

Assuming total and value are in memory and count in CX, the assembly code is shown below:

```
cmp total, 100
jae addValue
addValue: mov bx, value
          add total, bx
```

EXERCISE:

Q1 write an assembly language program that repeatedly (forever) calculates 1*2*3*…*n.

Q2 Assume for each part of this exercise that the AX contains 00 4F and the doubleword referenced by value contains FF38. Determine whether each of the following conditional branch instructions causes a transfer of control to label dest.

i. cmp ax,value
   jb dest

Your answer: YES ☐ NO ☐

Reason

ii.cmp ax,value
   ja dest

Your answer: YES ☐ NO ☐

Reason

iii. cmp ax,04fh
je dest

Your answer: YES ☐ NO ☐

Reason

iv. add ax,200
jne dest

Your answer: YES ☐ NO ☐

Reason

Q3 Write down the code to find number input by user is EVEN or ODD ✓ Save "Enter number " @ Location loc1
✓ Save "Number is EVEN"@ Location loc2
✓ Save "Number is ODD" @ Location loc3

Hint:
Save the input to another register(Say bl) Mov 2 to Al
Divide bl to al

Q4 Write down the code to find Largest number using @least one conditional & unconditional jump instructions. Input two numbers from user; Output should look like as:

- Enter number 1:
- Enter number 2:
  Largest number is :
  (save above strings @ locations msg1,msg2 and msg3of DS)

Q5 Repeat the Q4 to find largest number by using @least one conditional & unconditional jump instructions. Input two numbers from user ; Output should look like as:

- Enter number 1:
- Enter number 2:
  Largest number is :

- ✓ (save above strings @ locations msg1,msg2 and msg3of DS)
  Save numbers (input by user) @ locations num1 and num2 of DS

Lab 07

Implementation of Loop Structures in x86 Assembly Language

Looping (repeated execution of a program fragment) is the fundamental capability that gives programming languages and computers the real power. a loop is a sequence of instruction s that is continually repeated until a certain condition is reached. Typically, a certain process is done, such as getting an item of data and changing it, and then some condition is checked.

Syntax: loop destination_label

The CX register is used as a counter for loop instructions i.e. count (number of times a loop runs) is set to CX register, the loop instruction causes CX to be decremented, and if cx != 0, jump to destination_label.
The destination label must precede the loop instruction by no more than 126 bytes.

Destination_Label:

It is the address (or Label) of the code that we want to execute again and again

Loop Body:

Program portion executed repeatedly. This is the actual work to be accomplished. The rest is loop overhead. Goal to minimize that overhead

```
;initialize cx to loop_count
destination_label:

        ;body of the loop
loop destination_label
```

Example:

```
        mov cx,5
        mov dl,0                ; dl stored code of ASCII character @ 0
        Add dl,30h              ; add 30 to dl ,now dl =30+0=30 ;code of 0
        mov ah,2                ; display number
again:  int 21h
        Add dl,2                ; Add 2 to dl ,dl=30+2=32

        loop again              ; go to again
```

Commonly used loop structures include while and for loops. This lab describes, in detail, implementation of both of these structures in 80x86 assembly language.

Implementation of while Loop

A while loop is a pre-test loop – the continuation condition, a Boolean expression, is checked before the loop body is executed. Whenever it is true the loop body is executed and then the continuation condition is checked again. When it is false execution continues with the statement following the loop. It may take several 80x86 instructions to evaluate and check a continuation condition.

A while loop can be indicated by the following pseudocode design:

```
while condition

... { body of loop }

end while;
```

An 80x86 implementation of a while loop follows a pattern much like this one:

```
while1:  .              ; code to check Boolean expression

         …. body:   ; loop body

  jmp while1 ; go check condition again

endWhile1
```

As an example, consider the following pseudocode:

```
while (sum < 1000) loop
add count to sum;
add 1 to count;
 end while;
```

Assuming sum in memory and count in CX, the corresponding assembly code follows:

```
 whileSum:

      cmp   sum, 1000
      jnl   endWhileSum
      add   sum, cx
       inc   cx
       jmp   whileSum
endWhileSum:
```

Consider another example. Suppose that the integer base 2 logarithm of a positive integer number needs to be determined. The integer base 2 logarithm of a positive integer is the largest integer x such that < number. The following pseudocode will do the job.

```
 x := 0;
 twoTox :=1;
while twoTox < number loop
multiply twoTox by 2;
add 1 to x;
 end while;
subtract 1 from x;
```

Assuming that the number references a word in memory, the following 80x86 code implements the design, using the AX register for two To x and the ECX register for x. ;

```
.DATA
   number DB      05
.CODE
 main        PROC
 mov   cx, 0      ; cx := 0
 mov   ax, 1      ; ax := 1
whileLE:
      cmp    ax, number ; ax <= number?
      jnle  endWhileLE ; exit if not body:
      add   ax, ax      ;
      inc   cx         ; add 1 to x
     jmp   whileLE     ; go check condition again
endWhileLE:
     dec   cx         ; subtract 1 from cx
     mov   ax, 4ch
     int 21h; exit with return code 0
main        ENDP
END main
```

Implementation of For Loop

The for loop is a counter-controlled loop that executes once for each value of a loop index (also known as loop counter) in a given range. Often the number of times the body of a loop must be executed is known in advance, either as a constant that can be coded when a program is written, or as the value of a variable that is assigned before the loop is executed. The for loop is ideal for coding such a loop. A for loop can be described by the following pseudocode:

```
 for index := initialValue to finalValue loop
 ... { body of loop }
end for;
```

Consider the following pseudocode:

```
 sum := 0
for count := 20 down to 1 loop
add count to sum;
end for;
```

Assuming sum in AX and count in CX, the corresponding assembly code follows:

```
 mov   ax, 0
 mov   cx, 20
 forCount:
         add   ax, cx
  loop forCount
```

Implementation of until Loop

An until loop is a post-test loop – the condition is checked after the body of loop is executed. In general, an until loop can be represented as follows:

repeat

... { body of loop }

until termination condition;

EXERCISE:

Q1. Write down the code to Print following output:

123456789
1357
2468

Q2.i Consider the following pseudocode; and write its equivalent assembly program.

```
Repeat:
   add count to sum;
  add 1 to count;
 until (sum > 1000);
```

ii. for index := 1 to 50 loop

```
   ... { loop body using index }
end for;
```

iii. repeat

```
    add 2*count to sum;
   add 1 to count;
  until (sum > 1000);
```

Q3 Write down the assembly code of a program that can take input from user 3 times and calculate their sum.

Lab # 08

Procedures in x86 Assembly Language

The 80x86 architecture enables implementation of procedures that are similar to those in high-level language. These procedures can be called from high-level language program or can call high-level language procedures. There are three main concepts involved: (1) transfer of control from calling program (procedure) to the called procedure and back, (2) passing parameters from calling program (procedure) to the called procedure and results back to the calling program, and
(3) development of procedure code that is independent of the calling program. A procedure is a subprogram that is essentially a self-contained unit. Main program or another subprogram calls a procedure. A procedure may simply do a task or it may return a value. Value-returning procedure is sometimes called a function.

Procedures are valuable in assembly language for the same reasons as in a HLL. However, sometimes assembly language can be used to write more efficient code than is produced by a HLL compiler and this code can be put in a procedure called by a HLL program that does tasks that don't need to be as efficient.

Procedure Definition In a code segment following *.CODE* directive, the procedure body is bracketed by PROC and ENDP directives giving procedure name as the label.

```
.CODE
procName PROC
procedure body
 ...
procName ENDP
```

- **Call Instruction**

Transferring Control to a Procedure In the "main" program or calling procedure, control is transferred to the called procedure using CALL instruction. Call causes the procedure named in the operand to be executed. When the called procedure completes, execution flow resumes at the instruction following the call instruction . return address is (IP) is pushed to stack.

Syntax :

Call procedure_name

The next instruction executed will be the first one in the procedure.

➢ **Return from Procedure (ret)**

The ret instruction transfers control to the return address located on the stack. This address is usually placed on the stack by a call instruction. Issue the ret instruction within the called procedure to resume execution flow at the instruction following the call. Return address (IP) is removed from the stack.

Syntax:

RET

Or *RET imm*

Example 1:

```
.MODEL small
.Stack 100h
.CODE                  ; start of main program code
      main proc
       mov    cl, 3    ; move data into registers
       mov    bl, 2
     call add2nums;  push address of next instruction onto the stack, and pass control
       mov ah,4ch ; exit code
         int 21h
  main ENDp          ; end of source code
add2nums proc
       add    cl, bl    ; Add two numbers
       mov   dl,cl
       add    dl,30h
       mov   ah,2h
       int 21h
      ret ; Pop the address from stack
      add2nums endp
end main
```

Output

```
5
```

Calling a Procedure within a Procedure

```
.data
Msg db ? ,"$"
.code
Main proc

Mov ax,@data
Mov ds,ax

; Calling Three Procedures
Call input          ;PUSH address of next instruction onto the stack, and pass control
Call save

Mov ah,4ch
Int 21h
```

```
Main endp

Input proc
Mov ah,1
Int 21h

ret                    ; POP the address from the stack
Input endp
Save proc
Mov msg,al
Call Display           ;Call "Display" with in "Save"
ret
Save endp

Display proc
Mov dl,0ah
Mov ah,2
Int 21h
Lea dx,msg
Mov ah,9
Int 21h
Ret
Display Endp
End main
```

Output:

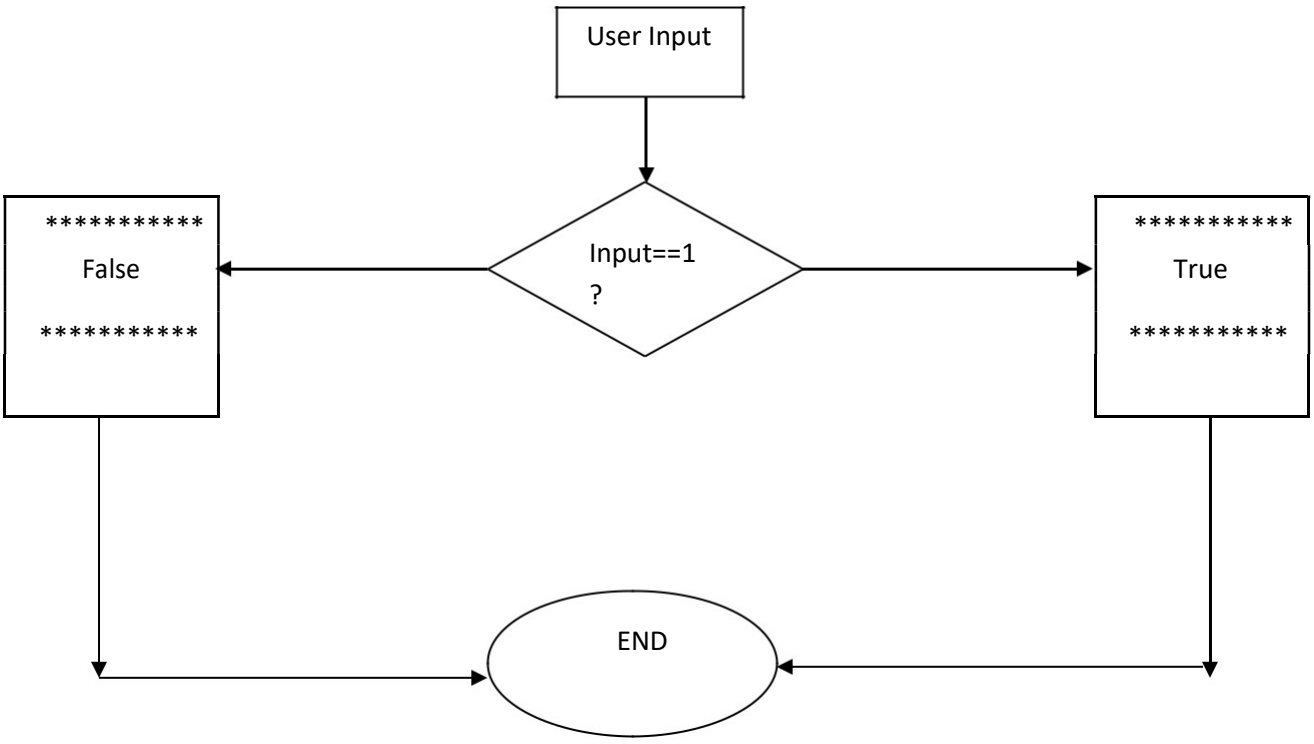
```
5
5
```

EXERCISE:

Q1. Write assembly code of a procedure that computes the value of *b*b-4*a*c* ,where a,b,c are user provided inputs

Q2. Write code of a program that perform 4 basic arithmetic operations on user provided input . code should have at least four procedures.

Q 3 Write down the code of following scenario. Program should have 2 procedures .



Lab # 9

Development of Macros in x86 Assembly Language

A procedure definition often includes parameters (also called formal parameters). These are associated with arguments (also called actual parameters) when the procedure is called. For a procedure's in (pass-by-value) parameters, values of the arguments are copied to the parameters when the procedure is called. These values are referenced in the procedure using their local names (the identifiers used to define the parameters).

Parameter values are normally passed on the stack. They are pushed in the reverse order from the argument list. As an example, consider the pseudocode: sum := add2(value1, value2)

The 80x86 implementation follows:

```
push   cx       ; assuming value2 in CX

push   value1    ; assuming value1 in memory

call   add2      ; call procedure to find sum

add    sp, 8   ; remove parameters from stack

mov    sum, ax  ; sum in memory
```

If the stack is not cleaned, and a program repeatedly calls a procedure, eventually the stack will fill up causing a runtime error with modern operating systems.

Macro Definition and Expansion :A macro expands to the statements it represents. Expansion is then assembled. It resembles a procedure call, but is different in the way that each time a macro appears in a code, it is expanded. In contrast, there is only one copy of procedure code. A macro is defined as follows:

name MACRO *list of parameters*

…..assembly language statements

ENDM

The Macro directive indicate the beginning of the macro definition and the ENDM directives signals the end. Code place between these two directives.

- ✓ Name must be unique and must follow assembly language conventions
- ✓ Arguments are names of parameters or even registers that are mentioned in the body.

After the Macro has been written it can be invoked or called by its name , and appropriate values are substituted for parameters

Example:

```
String Macro Data1
        MOV AH,9
        MOV DX, OFFSET DATA1
        INT 21H
    ENDM
```

The above code is the Macro Definition. Argument Data1 is mentioned in the body of Macro is already been defined in data segment as shown below :

Message1 DB "What is your name? $"

In the code segment, the Macro can be invoked by its name with the user"s actual data

```
String    Message1      ; Invoke the Macro
```

Parameters in the MACRO directive are ordinary symbols, separated by commas. The assembly language statements may use the parameters as well as registers, immediate operands, or symbols defined outside the macro.

Comments in Macro:

There are basically two types of comments in the Macro: List able and non-list able. If comments are precedes by a single semicolon (;) , they will show up in the .lst file , but if comment precedes by double semicolon(;;) ,they will not appear in the .lst file when the programmed is assembled.

EXERCISE:

Q1) Write code that calculate and print value of Length * Breadth where length and breadth are 2 & 4 respectively.

Q2) Repeat Q1-lab # 8, Write Macro definition of a procedure having 3 arguments a,b,c and called it from main procedure.

Q3) Write down the assembly code of following output using Only One Macro definition

My name is xxxxx
My rollnumber is yyyyy
What is Your name

Q4) Using Macros write down the code to solve following expression: x –2 y + 2z
Where x,y,z are values provided by the user.Code should have at least two Macros

Lab # 10

Array Processing in x86 Assembly Language

In programming, a series of objects all of which are the same size and type called "**Array**".
Each object in an array is called an **array element**. The important characteristics of an array are:

- ✓ Each element has the same data type (although they may have different values).
- The entire array is stored contiguously in memory (that is, there are no gaps between elements).

Arrays can have more than one dimension. A one-dimensional array is called a *vector* ; a two-dimensional array is called a **matrix**. Programs often use arrays to store collections of data values. Loops are commonly used to manipulate the data in arrays. Storage for an array can be reserved using the *DUP* directive in the data segment of a program. For Example:

array1 DD 25, 47, 15, 50, 32 ; creates an array of 5 doublewords with initial values as specified.

Declare an array

array2 DD 1000 DUP (?) ; creates an array of 1000 logically uninitialized doublewords

Access to array elements

This part is easy similar to other language, you need the right index and then you access your element. That"s means if you want to access the first element of the array A, you just write: A[0].

Create a 2 dimensional array

Let"s assume we have 2 arrays A and B of dimension DIM. We will compute all possible products among values of first array and values of second array, we will put results in a matrix of DIMxDIM dimension
Here is the algorithm I will use:

for each element1 in array A, starting from 0:
for each element2 in array B, starting from 0:
M[i][j] = element1*element2

In 8086, there is no real structure for 2 dimensional array in memory.
So we will map our matrix in a linear memory. That"s mean we declare an array of DiM*DIM elements and every element of the matrix will be in that array.

The mapping is simple to access to the J^{th} row and the K^{th} column we just do: M[J,K] = M[J * DIM + K]

The second trick is about the size of every element of the matrix. Since every element is a product of two element of 1 byte , we will use 2 bytes for the size of every element of the matrix. So the mapping formula is now: M[J,K] = M[2 * (J * DIM + K)] because every element take one more byte on memory.
The result:

```
1
2           .MODEL small
3           .STACK
4           .DATA
5             DIM  EQU 9
              A   DB  1, 2, 3, 4, 5, 6, 7, 8, 10
6             B   DB  11, 12, 13,14, 15, 16, 17, 18, 19
7             M   DW (DIM * DIM) DUP (?)
8             CDB?
9             TEMP DW ?
10           TEMP2 DW ?
11          .CODE
12            Main Proc
13            MOV AX, 0
14            MOV CX, DIM
15            MOV DI, 0
16        X:  MOV C, DIM1
17            MOV BX, 0
18        Y:  MOV AX, 0
19            MOV AL, A[DI]
20            MUL B[BX]
21    ;M[J,K] = M[2 * (J * NUM_COLS + K)]
22            MOV TEMP,AX
23            XOR AX,AX
24            MOV AX,DI
25            MOV TEMP2,BX
26            MOV BX,DIM1
27            MUL BX
28            MOV BX,TEMP2
```

```
29          ADD AX,BX
30          MOV TEMP2,BX
31          MOV BX,2
32          MUL BX
33          MOV BX,TEMP2
34          MOV TEMP2,DI
35          MOV SI,AX
36          MOV AX,TEMP
37          MOV M[SI],AX
38          MOV AX, 0
39          INC BX
40          DEC C
41          CMP C,0
42          JNZ Y
43          MOV AX, 0
44          INC DI
45          DEC CX
46          CMP CX,0
47          JNZ X
48           ;EXT
49          ;END
```

EXERCISE:

Q1) Calculate and display the sum of three numbers entered by user, save input in an array.

Q2) Enter 10 elements in an array using input function and display in new line.

Q3)write a code to Save your name in an array and Display in reverse order .