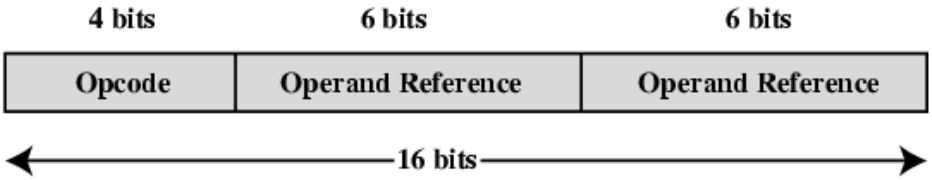
Lab 3 - Introduction to Assembly Language

3.1 x86 Assembly Instruction Format:

Almost every instruction in assembly/low language has two parts. One is **opcode** the other is address **references**. If we break down a 16-bit instruction, then we may get following components:



Opcode: This is a unique pattern which identify "what operation is to be done".

Operand References: This is the unique address of a memory location "where the operation is needed to be done". Operands can be immediate value, variable, memory address or register.

The basic idea with machine code is to use binary bytes to represent a computation. Different machines use different bytes, but Intel x86 machines use "0xc3" to represent the "ret" instruction, and "0xb8" to represent the "load a 32-bit constant into EAX" instruction.

```
0:     b8 05 00 00 00            mov    eax,0x5
5:     c3                        ret
```

0: and 5: are the address offsets of the instructions in bytes. "mov" is an instruction, encoded with the operation code or "opcode" 0xb8. Since mov takes an argument, the next 4 bytes are the constant to move into EAX.

The opcode 0xb9 moves a constant into ECX. 0xba moves a constant into EDX.

```
0:     b8 05 00 00 00            mov    eax,0x5
5:     b9 05 00 00 00            mov    ecx,0x5
a:     ba 05 00 00 00            mov    edx,0x5
```

x86 specifies register sizes using prefix bytes. For example, the same "0xb8" instruction that loads a 32-bit constant into EAX can be used with a "0x66" prefix to load a 16-bit constant, or a "0x48" REX prefix to load a 64-bit constant.

Here we're loading the same constant 0x12 into all the different sizes of EAX:

```
   0:  48 b8 12 00 00 00 00 00 00          mov    rax,0x12
   a:     b8 12 00 00 00                   mov    eax,0x12
   f:  66 b8 12 00                         mov    ax,0x12
  13:     b0 12                            mov    al,0x12
  15:     c3                               ret
```

3.2 x86 Instructions Disassembling

Disassembling your assembly or compiled code shows you both the instructions and the machine code that implements them. Not only are there hundreds of different x86 instructions, but there can also be dozens of different machine code encodings for a given instruction. Here are a few examples:

ASM	Machine Code	Description
add	0x03 *ModR/M*	Add one 32-bit register to another.
mov	0x8B *ModR/M*	Move one 32-bit register to another.
mov	0xB8 *DWORD*	Move a 32-bit constant into register eax.
ret	0xc3	Returns from current function.
xor	0x33 *ModR/M*	XOR one 32-bit register with another.
xor	0x34 *BYTE*	XOR register al with this 8-bit constant.

3.3 Encoding Memory and Register Operands

That last byte of most x86 instructions is called "ModR/M" in the Intel documentation, and it specifies what the source and destination are. It's just a bit field giving a selector called "mod" (which indicates whether r/m is treated as a plain register or a memory address), one register called "reg/opcode" (which is usually the destination register and determines the column in the ModR/M table), and a final register called "r/m" (usually the source register, which selects the row of the ModR/M table). These are stored in a single byte with this format:

mod	reg/opcode	r/m
2 bits, selects memory or register access mode: 0: memory at register r/m 1: memory at register r/m+byte offset 2: memory at register r/m + 32-bit offset 3: register r/m itself (not memory)	3 bits, usually a destination register number. For some instructions, this is actually extra opcode bits.	3 bits, usually a source register number. Treated as a pointer for mod!=3, treated as an ordinary register for mod==3. If r/m==4, indicates the real memory source is a SIB byte.

3.4 ModR/M Table

This is the "ModR/M" table for the meaning of ModR/M byte values:

r32(/r) reg=			EAX 000	ECX 001	EDX 010	EBX 011	ESP 100	EBP 101	ESI 110	EDI 111
effective address	*mod*	*R/M*	value of mod R/M byte (hex)							
[RAX]	*00*	*000*	00	08	10	18	20	28	30	38
[RCX]		*001*	01	09	11	19	21	29	31	39
[RDX]		*010*	02	0A	12	1A	22	2A	32	3A
[RBX]		*011*	03	0B	13	1B	23	2B	33	3B
[SIB]		*100*	04	0C	14	1C	24	2C	34	3C
[RIP + *DWORD*]		*101*	05	0D	15	1D	25	2D	35	3D
[RSI]		*110*	06	0E	16	1E	26	2E	36	3E
[RDI]		*111*	07	0F	17	1F	27	2F	37	3F
[RAX + *BYTE*]	*01*	*000*	40	48	50	58	60	68	70	78
[RCX + *BYTE*]		*001*	41	49	51	59	61	69	71	79
[RDX + *BYTE*]		*010*	42	4A	52	5A	62	6A	72	7A
[RBX + *BYTE*]		*011*	43	4B	53	5B	63	6B	73	7B
[SIB + *BYTE*]		*100*	44	4C	54	5C	64	6C	74	7C
[RBP + *BYTE*]		*101*	45	4D	55	5D	65	6D	75	7D
[RSI + *BYTE*]		*110*	46	4E	56	5E	66	6E	76	7E
[RDI + *BYTE*]		*111*	47	4F	57	5F	67	6F	77	7F
[RAX + *DWORD*]	*10*	*000*	80	88	90	98	A0	A8	B0	B8
[RCX + *DWORD*]		*001*	81	89	91	99	A1	A9	B1	B9
[RDX + *DWORD*]		*010*	82	8A	92	9A	A2	AA	B2	BA
[RBX + *DWORD*]		*011*	83	8B	93	9B	A3	AB	B3	BB
[SIB + *DWORD*]		*100*	84	8C	94	9C	A4	AC	B4	BC
[RBP + *DWORD*]		*101*	85	8D	95	9D	A5	AD	B5	BD
[RSI + *DWORD*]		*110*	86	8E	96	9E	A6	AE	B6	BE
[RDI + *DWORD*]		*111*	87	8F	97	9F	A7	AF	B7	BF
EAX	*11*	*000*	C0	C8	D0	D8	E0	E8	F0	F8
ECX		*001*	C1	C9	D1	D9	E1	E9	F1	F9
EDX		*010*	C2	CA	D2	DA	E2	EA	F2	FA
EBX		*011*	C3	CB	D3	DB	E3	EB	F3	FB
ESP		*100*	C4	CC	D4	DC	E4	EC	F4	FC
EBP		*101*	C5	CD	D5	DD	E5	ED	F5	FD
ESI		*110*	C6	CE	D6	DE	E6	EE	F6	FE
EDI		*111*	C7	CF	D7	DF	E7	EF	F7	FF

For complete reference visit this Link.

3.5 x86 Instructions Disassembling in Visual Studio

Screenshots

3.6 Assembly Instructions Examples

If **add a,b** is an instruction, then **add** is a mnemonic which represent unique opcode of add operation, which is telling the CPU to add something. **a,b** are the operands whose values will be added in the result of the operation.

Depending upon the architecture, we may have a third reference where the result of the operation will be stored or just one reference. In case of one reference, accumulator register “ax” will be used implicitly to hold the temporary data during arithmetic operations.

Add operation can be executed as:

One reference:

```
mov a        ; copy value of a into accumulator register "ax".
add b        ; add value of b into value of "ax" and keep in it.
sto c        ; copy the value of accumulator register in "c variable".
```

Two References:

```
add a,b            ;add value of b into a and keep the result in a.
sto c,a            ;copy the value of a into c.
```

Three References:

```
add c,a,b          ;add values of a and b, and store the result in c.
```

Some mnemonics are as follows:

Mnemonics	Syntax	Working
mov	mov op1,op2	Copy data from operand 2 to operand 1.
add	add op1,op2	Add operand 2 into operand 1 and keep it stored in operand 1.
sub	sub op1,op2	Subtract operand 1 from operand 2.
mul	mul op1	It multiplies operand to EAX and store result in edx:ex
jmp	jmp	Jump to a new location.
call	call proc1	Call a procedure.
imul	imul eax,ebx,5	EBX is multiplied by 5, and the product is stored in the EAX register.

Note: When two 32-bit integers are multiplied, output is 64-bit integer. In 32-bit architecture, 64-bit registers don't exist. Hence to accommodate 64-bit integer, upper 32-bits are moved to EDX and lower 32-bits are moved to EAX register.

An instruction contains four basic parts:

- Label (optional)
- Instruction mnemonic (required)
- Operand(s) (usually required)
- Comment (optional)

This is how the different parts are arranged:

```
[label:] mnemonic [operands] [;comment]
```

3.6.1 Label

A label is an identifier that acts as a place marker for instructions and data. A label placed just before an instruction point toward the instruction's address. Similarly, a label placed just before a variable point toward the variable's address. There are two types of labels:

- Data Label
- Code Label

A ***data label*** identifies the location of a variable, providing a convenient way to reference the variable in code. The following, for example, defines a variable named count:

```
count DWORD 100
```

The assembler assigns a numeric address to each label. It is possible to define multiple data items following a label. In the following example, array defines the location of the first number (1024). The other numbers following in memory immediately afterward:

```
Array  DWORD 1024, 2048
       DWORD 4096, 8192
```

A ***code label*** is used as targets of jumping and looping instructions. Code labels must end with a colon (:) character. For example, the following JMP (jump) instruction transfers control to the location marked by the label named target, creating a loop:

```
target:
       mov ax,bx
       ...
       jmp target
```

A code label can share the same line with an instruction, or it can be on a line by itself:

```
L1:    mov ax,bx
L2:
```

Label names follow the same rules we described for identifiers in Section 3.2.9 below.

3.7 First Assembly Language Program

let's look at a simple assembly language program that adds two numbers and saves the result in a register. We will call it the *AddTwo* program:

```
1: main PROC
2:     mov eax,5          ;move 5 to the eax register
3:     add eax,6          ;add 6 to the eax register
4:
5:     INVOKE ExitProcess,0      ;end the program
6: main ENDP
```

On the left side there are line numbers. You never actually type line numbers when you create assembly programs. Here is the explanation of each line:

- **Line 1** starts the main procedure, the entry point for the program.
- **Line 2** places the integer 5 in the EAX register.
- **Line 3** adds 6 to the value in EAX, giving it a new value of 11.
- **Line 5** calls a Windows service (also known as a function) named *ExitProcess* that halts the program and returns control to the operating system.
- **Line 6** is the ending marker of the main procedure.

This program does not display anything on the screen, but we could run it with Visual Studio's Debugger that would let us step through the program one line at a time and we can look at the register values.

3.7.1 Comments

Comments can be specified in two ways:

- Single-line comments, beginning with a semicolon character (;). All characters following the semicolon on the same line are ignored by the assembler.
- Block comments, beginning with the COMMENT directive and a user-specified symbol. All subsequent lines of text are ignored by the assembler until the same user-specified symbol appears. Here is an example:

```
COMMENT !
      This line is a comment.
      This line is also a comment.
!
```

We can also use any other symbol, as long as it does not appear within the comment lines:

```
COMMENT &
      This line is a comment.
      This line is also a comment.
&
```

3.7.2 Adding a Variable

Now modify the *AddTwo* program to save the result of our addition in a variable named sum. To do this, we will add a couple of markers, or declarations, that identify the code and data areas of the program:

```
1: .data     ;this is the data area
2: sum DWORD 0     ;create a variable named sum
3:
4: .code     ;this is the code area
5: main PROC
6:     mov eax,5    ;move 5 to the EAX register
7:     add eax,6    ;add 6 to the EAX register
8:     mov sum,eax  ;moving the value of EAX to sum variable
9:
10:    INVOKE ExitProcess,0      ;end the program
11: main ENDP
```

The **sum** variable is declared on **Line 2**, where we give it a size of 32-bits, using the **DWORD** keyword. There are a number of these size keywords, which work more or less like data types. But they are not as specific as types you might be familiar with, such as int, double, float, and so on. They only specify a size, but there's no checking into what actually gets put inside the variable. Remember, you are in total control.

The .code and .data directives, are called *segments*. So, you have the *code segment* and the *data segment*. Later on, we will see a third segment named *stack*.

3.7.3 Integer Literals (Constants)

An *integer literal* (also known as an *integer constant*) is made up of an optional leading sign, one or more digits, and an optional radix character that indicates the number's base:

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

We will use **Microsoft Syntax Notation** throughout the Manual whenever the syntax is written. Elements within square brackets [..] are optional and elements within braces {..} require a choice of one of the enclosed elements, separate by the | character. Elements in *italics* identify items that have known definitions or descriptions.

So, for example, 26 is a valid integer literal. It doesn't have a radix, so we assume it's in decimal format. If we wanted it to be 26 hexadecimals, we would have to write it as 26h. Similarly, the number 1101 would be considered a decimal value until we added a "b" at the end to make it 1101b (binary).

Here are the possible radix values:

- **d** decimal
- **h** hexadecimal
- **b** binary
- **q** or **o** octal
- **r** encoded real
- **t** decimal (alternate)
- **y** binary (alternate)

And here are some integer literals declared with various radixes. Each line contains a comment:

```
26          ;decimal
26d         ;decimal
11010011b   ;binary
42q         ;octal
42o         ;octal
1Ah         ;hexadecimal
0A3h        ;hexadecimal
```

Note: A hexadecimal literal beginning with a letter must have a leading zero to prevent the assembler from interpreting it as an identifier.

3.7.4 Constant Integer Expressions

A *constant integer expression* is a mathematical expression involving integer literals and arithmetic operators. Each expression must evaluate to an integer, which can be stored in 32-bits (0 through FFFFFFFFh). The arithmetic operators are listed in Table 3-1 according to their precedence order, from highest (1) to lowest (4). The important thing to realize about constant integer expressions is that they can only be evaluated at assembly time. From now on, we will just call them *integer expressions*.

Operator	Name	Precedence Level
()	Parentheses	1
+ , -	Unary plus, minus	2
* , /	Multiply, divide	3
MOD	Modulus	3
+, -	Add, subtract	4

Operator precedence refers to the implied order of operations when an expression contains two or more operators. The order of operations is shown for the following expressions:

```
4 + 5 * 2          Multiply -> add
12 -1 MOD 5        Modulus -> subtract
-5 + 2             Unary minus ->add
(4 + 2) * 6        Add -> multiply
```

3.7.5 Real Number Literals

Real number literals (also known as *floating-point literals*) are represented as either decimal reals or encoded (hexadecimal) reals. A *decimal real* contains an optional sign followed by an integer, a decimal point, an optional integer that expresses a fraction, and an optional exponent:

```
[sign]integer.[integer][exponent]
```

These are the formats for the sign and exponent:

```
sign        {+,-}
exponent    E[{+,-}]integer
```

Following are examples of valid decimal reals:

```
2.
+3.0
-44.2E+05
26.E5
```

3.7.6 Character Literals

A *character literal* is a single character enclosed in single or double quotes. The assembler stores the value in memory as the character's binary ASCII code. Examples are 'A', "d", etc.

The character literals are stored internally as integers, using the ASCII encoding sequence. So, when you write the character constant "A," it's stored in memory as the number 65 (41h in Hexadecimal).

3.7.7 String Literals

A *string literal* is a sequence of characters (including spaces) enclosed in single or double quotes: 'ABC', 'X', "Good night, Gracie", '4096', etc.

Embedded quotes are permitted when used in the manner shown by the following examples:

"This isn't a test" or 'Say "Good night," Gracie'.

Just as character constants are stored as integers, we can say that string literals are stored in memory as sequences of integer byte values. So, for example, the string literal "ABCD" contains the four bytes 41h, 42h, 43h, and 44h.

3.7.8 Reserved Words

Reserved words have special meaning and can only be used in their correct context. Reserved words, by default, are not case-sensitive. For example, MOV is the same as mov and Mov. Some of the reserved words are as follows:

- Instruction mnemonics, such as **MOV**, **ADD**, and **MUL**
- Register names
- Directives, which tell the 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
- Predefined symbols, such as @data, which return constant integer values at assembly time

3.7.9 Identifiers

An *identifier* is a programmer-chosen name. It might identify a variable, a constant, a procedure, or a code label. There are a few rules on how they can be formed:

- 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 $.
- An identifier cannot be the same as an assembler reserved word.

In general, it's a good idea to use descriptive names for identifiers, as you do in high-level

programming language code. But in assembly language try to name the identifier short.

Examples of well-formed names are:

```
lineCount    firstValue   index  line_count
myFile       xCoord       main   x_Coord
```

The following names are legal, but not as desirable:

```
_lineCount   $first       @myFile
```

Generally, you should avoid the @ symbol and underscore as leading characters, since they are used both by the assembler and by high-level language compilers.

3.7.10 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, but they let you define variables, macros, and procedures. They can assign names to memory segments and perform many other housekeeping tasks related to the assembler. Directives are not, by default, case sensitive. For example, **.data**, **.DATA**, and **.Data** are equivalent.

The following example helps to show the difference between directives and instructions. The **DWORD** directive tells the assembler to reserve space in the program for a doubleword variable. The **MOV** instruction, on the other hand, executes at runtime, copying the contents of myVar to the EAX register:

```
myVar  DWORD 26
mov    eax,myVar
```

3.7.11 Defining Segments

One important function of assembler directives is to define program sections, or segments. Segments are sections of a program that have different purposes. For example, one segment can be used to define variables, and is identified by the **.DATA** directive:

```
.data
```

The **.CODE** directive identifies the area of a program containing executable instructions:

```
.code
```

The **.STACK** directive identifies the area of a program holding the runtime stack, setting its size:

```
.stack 100h
```