Fundamental of Assembly Language

Instructions

An *instruction* is a statement that becomes executable when a program is assembled. Instructions are translated by the assembler into machine language bytes, which are loaded and executed by the CPU at runtime. An instruction contains four basic parts:

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

This is the basic syntax:

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

Label

A *label* is an identifier that acts as a place marker for instructions and data. A label placed just before an instruction implies the instruction's address. Similarly, a label placed just before a variable implies the variable's address.

Data 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; address is 4096 and number  store on the address is 8192
```

Code Label: A label in the code area of a program (where instructions are located) must end with a colon (:) character. Code labels are used as targets of jumping and looping instructions. 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
```

Label names are created using the rules for identifiers one can use the same code label more than once in a program as long as each label is unique within its enclosing procedure. (A procedure is like a function.)

Instruction Mnemonic

An *instruction mnemonic* is a short word that identifies an instruction. In English, a *mnemonic* is a device that assists memory. Similarly, assembly language instruction mnemonics such as mov, add, and sub provide hints about the type of operation they perform. Following are examples of instruction mnemonics:

mov:	Move (assign) one value to another
add:	Add two values

sub:	Subtract one value from another
mul:	Multiply two values
jmp:	Jump to a new location
call:	Call a procedure

Operands

Assembly language instructions can have between zero and three operands, each of which can be a register, memory operand, constant expression, or input-output port. A *memory operand* is specified by the name of a variable or by one or more registers containing the address of a variable. A variable name implies the address of the variable and instructs the computer to reference the contents of memory at the given address. Following are examples of assembly language instructions having varying numbers of operands.

The STC and NOP instruction, for example, has no operands:

```
stc ; set Carry flag and
NOP; no operation
```

The INC instruction has one operand:

```
inc eax ; add 1 to EAX
```

The MOV instruction has two operands:

```
mov count,ebx ; move EBX to count
```

In a two-operand instruction, the first operand is called the *destination*. The second operand is the *source*. In general, the contents of the destination operand are modified by the instruction. In a MOV instruction, for example, data is copied from the source to the destination.

The IMUL instruction has 3 operands, in which the first operand is the destination, and the following 2 operands are source operands:

```
imul eax,ebx,5
```

In this case, EBX is multiplied by 5, and the product is stored in the EAX register.

Comments

Comments are an important way for the writer of a program to communicate information about the program's design to a person reading the source code. The following information is typically included at the top of a program listing:

- Description of the program's purpose
- Names of persons who created and/or revised the program
- Program creation and revision dates
- Technical notes about the program's implementation

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.

```
Mov eax, 5; I am a comment.
```

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. For example,

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

```
!
```
other symbol can be used:
```
COMMENT &
This line is a comment.
This line is also a comment.
&
```
The NOP (No Operation) Instruction

The safest (and the most useless) instruction you can write is called NOP (no operation). It takes up 1 byte of program storage and doesn't do any work. It is sometimes used by compilers and assemblers to align code to even-address boundaries. In the following example, the first MOV instruction generates three machine code bytes. The NOP instruction aligns the address of the third instruction to a doubleword boundary x86 processors are designed to load code and data more quickly from even doubleword addresses.
```
00000000 66 8B C3 mov ax,bx
00000003 90 nop ; align next instruction
00000004 8B D1 mov edx,ecx
```

Example: Adding and Subtracting Integers

This section introduces a short assembly language program that adds and subtracts integers. Registers are used to hold the intermediate data, and we call a library subroutine to display the contents of the registers on the screen. Here is the program source code:
```
TITLE Add and Subtract (AddSub.asm)
; This program adds and subtracts 32-bit integers.
INCLUDE Irvine32.inc
code
main PROC
mov eax,10000h ; EAX = 10000h
add eax,40000h ; EAX = 50000h
sub eax,20000h ; EAX = 30000h
call DumpRegs ; display registers
exit
main ENDP
```

Let's go through the program line by line. Each line of program code will appear before its explanation.

TITLE Add and Subtract (AddSub.asm)

The TITLE directive marks the entire line as a comment. anything you want can be put on this line.

; This program adds and subtracts 32-bit integers.

All text to the right of a semicolon is ignored by the assembler, so we use it for comments.

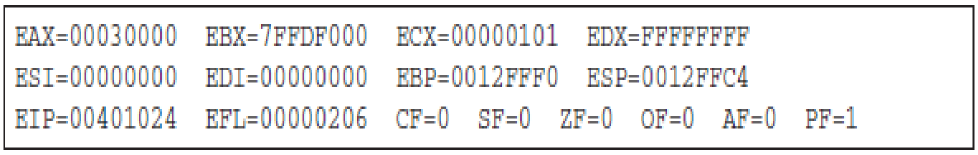
INCLUDE Irvine32.inc

The INCLUDE directive copies necessary definitions and setup information from a text file named *Irvine32.inc*, located in the assembler's INCLUDE directory.

.code

The **.code** directive marks the beginning of the *code segment*, where all executable statements in a program are located.
main PROC
The PROC directive identifies the beginning of a procedure. The name chosen for the only procedure in our program is **main**.
mov eax,10000h ; EAX = 10000h
The MOV instruction moves (copies) the integer 10000h to the EAX register. The first operand (EAX) is called the *destination operand*, and the second operand is called the *source operand.*
The comment on the right side shows the expected new value in the EAX register.
add eax,40000h ; EAX = 50000h
The ADD instruction adds 40000h to the EAX register. The comment shows the expected new value in EAX.
sub eax,20000h ; EAX = 30000h
The SUB instruction subtracts 20000h from the EAX register.
call DumpRegs ; display registers

The CALL statement calls a procedure that displays the current values of the CPU registers. This can be a useful way to verify that a program is working correctly.
exit
main ENDP
The **exit** statement (indirectly) calls a predefined MS-Windows function that halts the program.
The ENDP directive marks the end of the **main** procedure. Note that **exit** is not a MASM keyword; instead, it's a macro command defined in the *Irvine32.inc* include file that provides a simple way to end a program.
END main
The END directive marks the last line of the program to be assembled. It identifies the name of the program's *startup* procedure (the procedure that starts the program execution).
Program Output The following is a snapshot of the program's output, generated by the call to DumpRegs:


```
EAX=00030000  EBX=7FFDF000  ECX=00000101  EDX=FFFFFFFF
ESI=00000000  EDI=00000000  EBP=0012FFF0  ESP=0012FFC4
EIP=00401024  EFL=00000206  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=1
```

Program Template

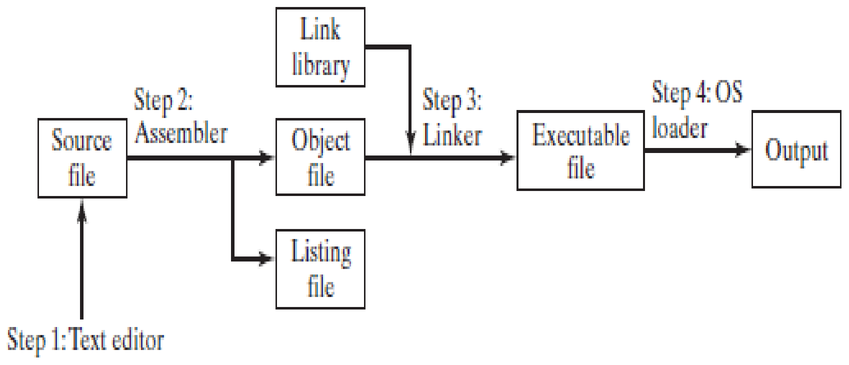
Assembly language programs have a simple structure, with small variations. To begin a new program, start with an empty shell program with all basic elements in place. One can avoid redundant typing by filling in the missing parts and saving the file under a new name.
The following protected-mode program (*Template.asm*) can easily be customized. Note that comments have been inserted, marking the points where your own code should be added:

```
TITLE Program Template (Template.asm)
; Program Description:
```

```
; Author:
; Creation Date:
; Revisions:
; Date:
INCLUDE Irvine32.inc
.data
; (insert variables here)
.code
main PROC
; (insert executable instructions here)
exit
main ENDP
; (insert additional procedures here)
END main
```

Assembling, Linking, and Running Programs

A source program written in assembly language cannot be executed directly on its target computer. It must be translated, or *assembled* into executable code. In fact, an assembler is very similar to a *compiler*, the type of program you would use to translate a C++ or Java program into executable code. The assembler produces a file containing machine language called an *object file*. This file isn't quite ready to execute. It must be passed to another program called a *linker*, which in turn an *executable file*. This file is ready to execute from the MS-DOS/Windows command prompt.



The Assemble-Link-Execute Cycle

The process of editing, assembling, linking, and executing assembly language programs is summarized in above. Following is a detailed description of each step.

Step 1: A programmer uses a **text editor** to create an ASCII text file named the *source file*.

Step 2: The **assembler** reads the source file and produces an *object file*, a machine-language translation of the program. Optionally, it produces a *listing file*. If any errors occur, the programmer must return to Step 1 and fix the program.

Step 3: The **linker** reads the object file and checks to see if the program contains any calls to procedures in a link library. The **linker** copies any required procedures from the link library, combines them with the object file, and produces the *executable file*.

Step 4: The operating system **loader** utility reads the executable file into memory and branches the CPU to the program's starting address, and the program begins to execute.