CHAPTER NINE

ASSEMBLY LANGUAGE PROGRAM

A program written in assembly language consists of a series of *instructions*--mnemonics that correspond to a stream of executable instructions, when translated by an assembler, that can be loaded into memory and executed.

For example, an x86/IA-32 processor can execute the following binary instruction as expressed in machine language (see x86 assembly language):

- Binary: 10110000 01100001 (Hexadecimal: B0 61)

The equivalent assembly language representation is easier to remember (example in Intel syntax, more *mnemonic*):

```
MOV AL, #61h
```

This instruction means:

- Move the value 61h (or 97 decimal; the h-suffix means hexadecimal; the pound sign means move the immediate value, not location) into the processor register named "AL".

The mnemonic "mov" represents the opcode **1011** which *moves* the value in the second operand into the register indicated by the first operand. The mnemonic was chosen by the instruction set designer to abbreviate "move", making it easier for the programmer to remember. A comma-separated list of arguments or parameters follows the opcode; this is a typical assembly language statement.

9.1 OVERVIEW OF ASSEMBLY LANGUAGE PROGRAMMING

In practice many programmers drop the word *mnemonic* and, technically incorrectly, call "mov" an *opcode*. When they do this they are referring to the underlying binary code which it represents. To put it another way, a mnemonic such as "mov" is not an opcode, but as it symbolizes an opcode, one might refer to "the opcode mov" for example when one intends to refer to the binary opcode it symbolizes rather than to the symbol--the mnemonic--itself. As few modern programmers have need to be mindful of actually what binary patterns are the opcodes for specific instructions, the distinction has in practice become a bit blurred among programmers but not among processor designers.

Transforming assembly into machine language is accomplished by an assembler, and the reverse by a disassembler. Unlike in high-level languages, there is usually a one-to-one correspondence between simple assembly statements and machine language instructions. However, in some cases, an assembler may provide *pseudoinstructions* which expand into several machine language instructions to provide commonly needed functionality. For example, for a machine that lacks a "branch if greater or equal" instruction, an assembler may provide a pseudoinstruction that expands to the machine's "set if less than" and "branch if zero (on the result of the set instruction)". Most full-featured assemblers also provide a rich macro language (discussed below) which is used by vendors and programmers to generate more complex code and data sequences.

Each computer architecture and processor architecture has its own machine language. On this level, each instruction is simple enough to be executed using a relatively small number of electronic circuits. Computers differ by the number and type of operations they support. For example, a new 64-bit machine would have different circuitry from a 32-bit machine. They may also have different sizes and numbers of registers, and different representations of data types in storage. While most general-purpose computers are able to carry out essentially the same functionality, the ways they do so differ; the corresponding assembly languages reflect these differences.

9.2 THE LINKER

The Linker is used for linking the output of an assembler to an executable file.

Basically a linker is used for large projects where there is more than one file to be assembled. Each of the assembly language files might contain references to code elements in the other files. A linker is the program that ties all the loose ends together and makes a single program out of the pieces.

Thus, technically, an assembler converts assembly language files to *object files* (basically machine code with some loose ends), and a linker connects all the object files together and makes a single executable program out of them.

However, there are some assemblers that do not require the use of a linker. E.g Flat Assembler FASM.

Actually, a linker is not just specific to assembly language programing. It is used fairly much regardless of the language in which you are programming.

An example of a linker is the ALINK

9.3 COMMON ASSEMBLERS

Some commonly used assemblers are listed below;

Lazy Assembler (LZASM)
GoASM
Flat Assembler (FASM)
MASM32
Pass32
Netwide Assembler (NASM)

In this course, the Microsoft Micro Assembler (MASM) will be used.

9.4 A SIMPLE HELLO WORLD PROGRAM USING FLAT ASSEMBLER(FASM)

```
Include ‘%fasminc%/win32ax.inc’

.code

start:

        invoke MessageBox, HWND_DESKTOP, ”Hello World! “, “Win32 Assembly”,MB_OK

        invoke ExitProcess, 0

.endstart
```

The first line *includes* a special macro file win32ax.inc

The next line tells the assembler we are going to include some code now.

The line start: is simply a label. It just gives a name to one of the lines of our program so we can refer to it elsewhere. We'll use labels often, for example when writing loops or using other jump instructions. The label will simply allow us to refer to that particular location within our code, by name; say for example if we wanted to jump to that location from somewhere else.

The program is ended with the .end directive. It takes one parameter, the name of a label corresponding to the entry point of the program (which doesn't need to be at the start of the program). When the program is loaded into memory by the operating system, execution of the program will begin at this point.

9.5 A SIMPLE HELLO WORLD PROGRAM USING NETWIDE ASSEMBLY LANGUAGE (NASM)

```
%include'system.inc'

section   .data
hello     db        'Hello, World!', 0Ah
hbytes    equ       hello

section   .text
global    _start
_start:
push      dword hbytes
```

```
11:      push     dword hello
12:      push     dword stdout
13:      sys.write
14:
15:      push     dword 0

16:    sys.exit
```

Here is what it does: Line 1 includes the defines, the macros, and the code from system.inc.

Lines 3-5 are the data: Line 3 starts the data section/segment. Line 4 contains the string "Hello, World!" followed by a new line (**0Ah**). Line 5 creates a constant that contains the length of the string from line 4 in bytes.

Lines 7-16 contain the code. Note that FreeBSD uses the *elf* file format for its executables, which requires every program to start at the point labeled **_start** (or, more precisely, the linker expects that). This label has to be global.

Lines 10-13 ask the system to write **hbytes** bytes of the **hello** string to **stdout**.

Lines 15-16 ask the system to end the program with the return value of **0**. The **SYS_exit** syscall never returns, so the code ends there.

CHAPTER TEN

JOB CONTROL LANGUAGE

10.1 INTRODUCTION

Job Control Language (**JCL**) is a scripting language used on IBM mainframe operating systems to instruct the system on how to run a batch job or start a subsystem. The term "Job Control Language" can also be used generically to refer to all languages which perform these functions.

JCL consists of control statements that do the following;

- introduce a computer job to the operating system
- request hardware devices
- direct the operating system on what is to be done in terms of running applications and scheduling resources

JCL is not used to write computer programs. Instead it is most concerned with input/output---telling the operating system everything it needs to know about the input/output requirements. It provides the means of communicating between an application program and the operating system and computer hardware.

JCL can be difficult because of the way it is used. A normal programming language, however difficult, soon becomes familiar through constant usage. This contrasts with JCL in which language features are used so infrequently that many never become familiar.

JCL can be difficult because of its design - JCL:

- consists of individual parameters, each of which has an effect that may take pages to describe
- has few defaults--must be told exactly what to do
- requires specific placement of commas and blanks
- is very unforgiving--one error may prevent execution

JCL is not necessarily difficult because most users only use a small set of similar JCL that never changes from job to job.

10.2 BASIC SYNTAX AND RULES OF JCL

```
//NAME    OPERATION   OPERAND,OPERAND,OPERAND   COMMENTS

|         |           |                         |
name      operation   operand field             comment
field     field                                 field
```

name field - identifies the statement so that other statements or the system can refer to it. The name field must begin immediately after the second slash. It can range from 1 to 8 characters in length, and can contain any alphanumeric (A to Z) or national (@ $ #) characters. operation field - specifies the type of statement: JOB, EXEC, DD, or an operand command. operand field - contains parameters separated by commas. Parameters are composites of prescribed words (keywords) and variables for which information must be substituted. comments field - optional. Comments can be extended through column 80, and can only be coded if there is an operand field.

General JCL Rules:

- Must begin with // (except for the /* statement) in columns 1 and 2
- Is case-sensitive (lower-case is just not permitted)
- NAME field is optional
- must begin in column 3 if used
- must code one or more blanks if omitted
- OPERATION field must begin on or before column 16
- OPERATION field stands alone
- OPERANDS must end before column 72
- OPERANDS are separated by commas

All fields, except for the operands, must be separated by one blank.

More on JCL

```
//LABEL        OPERATION        OPERAND,        OPERAND,
//                                      OPERAND,OPERAND,
//                                              OPERAND,
//  OPERAND
```

When the total length of the fields on a control statement exceeds 71 columns, continue the fields onto one or more following statements.

- Interrupt the field after a complete operand (including the comma that follows it) at or before column 71
- Code // in columns 1 and 2 of the following line
- Continue the interrupted statement beginning anywhere in columns 4 to 16.

Commenting JCL

//* THIS IS A COMMENT LINE

JCL should be commented as you would any programming language. The comments statement contains //* in columns 1 to 3, with the remaining columns containing any desired comments. They can be placed before or after any JCL statements following the JOB statement to help document the JCL. Comments can also be coded on any JCL statement by leaving a blank field after the operand field.

10.3 TYPES OF JCL STATEMENTS

JOB Identifies the beginning of a job

EXEC Indicates what work is to be done

DD Data Definition, i.e., Identifies what resources are needed and where to find them

10.4 THE JOB STATEMENT

The JOB statement informs the operating system of the start of a job, gives the necessary accounting information, and supplies run parameters. Each job must begin with a single JOB statement.//jobname JOB USER=userid

jobname - a descriptive name assigned to the job by the user which is the banner on your printout
- any name from 1 to 8 alphanumeric (A-Z,0-9) or national ($,@,#) characters
- first character must be alphabetic or national

JOB - indicates the beginning of a job

Userid - a 1 to 7 character user identification assigned to access the system

10.5 THE EXEC STATEMENT

Use the EXEC (execute) statement to identify the application program or cataloged or in-stream procedure that this job is to execute and to tell the system how to process the job.

//stepname EXEC procedure,REGION=####K

or

//stepname EXEC PGM=program,REGION=####K

stepname - an optional 1 to 8 character word used to identify the step
EXEC - indicates that you want to invoke a program or cataloged procedure
procedure - name of the cataloged procedure to be executed
program - name of the program to be executed
REGION=####K - amount of storage to allocate to the job

10.5 DATA DEFINITION (DD) STATEMENT

A DD (Data Definition) statement must be included after the EXEC statement for each data set used in the step. The DD statement gives the data set name, I/O unit, perhaps a specific volume to use, and the data set disposition. The system ensures that requested I/O devices can be allocated to the job before execution is allowed to begin.

The DD statement may also give the system various information about the data set: its organization, record length, blocking, and so on.

```
//ddname DD operand,operand,etc.
```

ddname - a 1 to 8 character name given to the DD statement

DD - DD statement identifier

operand - parameters used to define the input or output dataset

The DD Statement

- appears after an EXEC statement
- gives the system information on many things, including the dataset attributes, the disposition of the dataset when the job completes, and which input/output device(s) to use

Happy Lecture