**Chapter Three**

**Programming in Assembly**

1. **Programming in Assembly**

Two aspects must be mastered to do programming in assembly, which is a CPU specific low level programming language (and for us the CPU of focus is **Intel 8086)**.

1. Getting to know the CPU in focus
* Register sets
* Addressing modes
* Instruction sets
* Interrupts (Software interrupts)
* I/O handling (E.g. Disk I/O)
1. Mastering the assembler in which you do your programming on.

(To the most part, this is Elass for us)

The assembler translates assembly code to machine code so that a processor can execute it.

Generally, an assembly code may contain two things:

* 1. **Instructions**

Those parts of the assembly program translated into machine instructions by the assembler i.e. the life time of instructions is up to run time.

Generally, instructions have got the following formats:

**Label**  **: Mnemonic**  **Operand, Operand ; Comment**

* ⇩ can be ⇩

Optional Short words *No* Operand Optional

(Alpha numeric for the *One* Operand

not numeric alone instruction *Two* Operand

in Elass)

* 1. **Directives and Pseudo-opcodes**
		1. **Directives**

Parts of an assembly program which give “direction” to the assembler during the “assembly“ process, but are not translated into machine instructions i.e., the life time of directives is up to assembly time.

**Examples:**

**hex $** **:** this tells the assembler (for example - Elass) that every number that begin with $ is in hexadecimal number system.

**code segment/code ends :** this tells the assembler that all lines enclosed in code segement and code ends are codes (or more precisely instructions)

**functionName proc/functionName endp :** this tells the assembler that lines enclosed in functionName proc/functionName endp are function definition for function named function Name.

* + 1. **Pseudo-opcodes**

A pseudo-opcode is a message to the assembler, just like an assembler directive, however a pseudo opcode will emit object code bytes. Examples of pseudo-opcodes include DB, DW, DD, DQ , and DT . These instructions emit the bytes of data specified by their operands but they are not true machine instructions.

**Example:**

Reserving memory

 X DW ?

 ⇩ ⇩ ⇩

 Name Reserve Don’t Initialize

 ‘word’ size

 (16 bit)

***Note***: Because an assembler is nothing more than a translating program, the format and syntax of the instructions, directives and pseudo – opcodes depend on how the assembler is written, not on the microprocessor (CPU). (That is why there are plenty of assemblers for a specific CPU).

* 1. **Intel 8086 register sets**
* Refer chapter two. (Usage of each register will be demonstrated via the use of programs. However, you are expected to know the purpose of each register in general terms).
* All registers in 8086 are 16 bit wide, though, some are accessed with 8 & 16 bit (AX,BX,CX,DX,AH,AL,BH,BL,CH,CL,DH,DL) and only 9 positions were taken in the flag register.
* Intel 8086 has got a 20 bit/pin address bus, 16 bit/pin control bus, and 16 bit/pin data bus. So how much is the addressable memory size?
* I/O in Intel 8086 utilizes 8 & 16 bit.
* There is a 6 byte instruction queue embedded with Intel 8086. These will temporary store decoded instruction bit streams. When executed, instructions in machine code with Intel 8086 are of variable size. (Minimum 1 byte and Maximum 6 byte)
* All later generations of Intel processors are backward compatible with Intel 8086, though,
* With wider registers
* With wider buses
* More instruction sets
* More registers
* Factor clock speeds

Etc…

**Important Note: students should be able to attach the above points in this section to their Computer Architecture and Organization course.**

**Registers**

 AX – Accumulator registers, Used in general data operations including arithmetic and I/O.

 AH holds sub functions of software interrupts.

 BX – Base registers. Used for addressing and some computations.

 CX – Count registers. Used for counting loops, and shifts.

 DX – Data registers. Used for arithmetic operations and general data operations.

 SP, BP, SI, DI – used for offset addressing and other general purposes. (To be discussed later, with a program example).

 IP – Instruction pointer (like a program counter)

 At any point in your assembly programming, you can’t modify the value of IP (and also CS) **directly**. It is the assembler which manipulates it to control the program flow.

* 1. **Memory Segmentation**

**Questions**: - with 16 bit registers used for address information, how is it possible to manage a Physical memory that require a 20 bit physical address?

The solution is **memory segmentation.**

|  |
| --- |
|  |

 Beginning address of

Offset

 Or

Effective

Address

 Memory segment

 Memory segment (Max size 64Kbyte)

 = content of the (216)

 Corresponding

 Segment register

 (CS, DS, ES, SS)

The physical address in memory is (PA).

 PA = Beginning x 10h + Offset Address

 Segment or

 Address Effective Address

 Hex

 CS or DS or ES or SS Depend on the

 (For the corresponding addressing mode used

 Code, Data, Extra and

 Stack Segment).

 20 bit

* 1. **Software Interrupts**

Apart from hardware interrupts there are also things called ***software interrupts***, also known as ***BIOS calls***. The software interrupts work very much like ordinary functions: you set-up input, call the routine, and get some output back. The difference lies in how you reach the code; for normal functions you just, well, jump to the routine you want. Software interrupts use the INT instruction, which diverts the program flow to somewhere in **BIOS (ROM) or Operating System routines loaded in memory**, carries out the requested algorithm and then restores the normal flow of your program. This is similar to what hardware interrupts do; only now you raise the interrupt programmatically. Hence: software interrupts.

Interrupts can be seen as a number of functions. These functions make the programming much easier, instead of writing a code to print a character you can simply call the interrupt and it will do everything for you. There are also interrupt functions that work with disk drive and other hardware.

Interrupts are also triggered by different hardware, these are called **hardware interrupts**. Currently we are interested in **software interrupts** only.

To initiate a **software interrupt** the **INT** instruction is used, and it has very simple syntax:

**INT value**

Where **value** can be a number between 0 to 255 (or 0 to 0FFh),
generally we will use hexadecimal numbers.

**Interrupt Types**

|  |  |
| --- | --- |
| **Interrupt Types**  | **Description** |
| 0h-1Fh | BIOS Interrupts (These interrupts are generated by the ROM BIOS during the start up of the computer. These interrupts are used for general low-level services) |
| 20h-3Fh | DOS Interrupts (These interrupts are available when DOS is running and provide additional routines for enhanced access to devices and other resources) |
| 40h-7Fh | Reserved (These interrupts are available for use by other programs) |
| 80h-F0h | ROM BASIC (These interrupts are available when Basic is running) |
| F1h-FFh | Not used (reserved for user interrupts) |

You may think that there are only 256 functions, but that is not correct. Each interrupt may have sub-functions.
To specify a sub-function **AH** register should be set before calling interrupt.
Each interrupt may have up to 256 sub-functions (so we get 256 \* 256 = 65536 functions). In general **AH** register is used, but sometimes other registers maybe in use. Generally other registers are used to pass parameters and data to sub-function.

**The following are common BIOS/DOS interrupts: (h = hexadecimal)**

|  |
| --- |
| ***INT 10h Video Functions*** |
| **00h** | **Set Video Mode** |
| **01h** | **Set Cursor Size** |
| **02h** | **Set cursor position** |
| **03h** | **Read Cursor Position** |
| **05h** | **Select active page** |
| **06h** | **Scroll up screen** |
| **07h** | **Scroll Down screen** |
| **08h** | **Read attribute/character** |
| **09h** | **Display Attribute/character** |
| **0Ah** | **Display Character** |
| **0Bh** | **Set color palette** |
| **0Ch** | **Write pixel dot** |
| **0Dh** | **Read pixel dot** |
| **0Eh** | **Write teletype** |
| **OFh** | **Get Video Mode** |
| **10h** | **Access Pallette Registers** |
| **11h** | **Character Generator** |
| **INT 13h: Disk Functions** |
| **00h** | **Reset Disk** |
| **01h** | **Read Disk Status** |
| **02h** | **Read Disk Sectors** |
| **03h** | **Write sectors** |
| **04h** | **Verify Sectors** |
| **05h** | **Format Tracks** |
| **08h** | **Get disk drive parameters** |
| **09h** | **Initialize drive** |
| **INT 16h: Keyboard Operations** |
| **03h** | **Set typematic rate** |
| **05h** | **Keyboard write** |
| **10h** | **Read keyboard character** |
| **11h** | **Determine if character present** |
| **12h** | **Return keyboard shift status** |
| **INT 21h functions** |
| **01h** | **Keyboard input** |
| **02h** | **Display character** |
| **03h/04h** | **Communications input/output** |
| **05h** | **Printer output** |
| **06h** | **Direct keyboard input and display** |
| **07h** | **Direct keyboard input** |
| **08h** | **Keyboard input** |
| **09h** | **Display string**  |
| **0Ah** | **Buffered keyboard input** |
| **OBh** | **Check keyboard status** |
| **0Ch** | **Clear keyboard buffer** |
| **ODh** | **Reset disk drive** |
| **0Eh** | **Select default disk drive** |
| **19h** | **Get default disk drive** |
| **1Ah** | **Set disk transfer address** |
| **1Fh** | **Get default drive parameter block** |
| **25h** | **Set interrupt vector** |
| **29h** | **Parse file name** |
| **2Ah** | **Get system date** |
| **2Bh** | **Set system date** |
| **2Ch** | **Get system time** |
| **2Dh** | **Set system time** |
| **2Eh** | **Set/reset disk write verification**  |
| **32h** | **Get drive parameter block** |
| **35h** | **Get interrupt vector** |
| **36h** | **Get free disk space** |
| **3Bh** | **Change current directory** |
| **3Ch** | **Create file** |
| **3Dh** | **Open file** |
| **3Eh** | **Close file** |
| **41h** | **Delete file** |
| **42h** | **Move file pointer** |
| **43h** | **Get/set file attributes** |
| **4Ch** | **Terminate program** |
| **56h** | **Rename file/directory** |
| **5Ah** | **Create temporary file** |
| **5Bh** | **Create new file** |

**The Interrupts that we will use frequently are: Int 21h (Int $21) (mainly sub functions 01, 02, 08, 09 and Disk related sub functions)**

**Note:** (Interrupt Vector Table) - When power is applied to a computer, the POST procedure creates a table of interrupt vectors that is 1024 bytes and contains a maximum of 256 interrupts. This table lists pointers to interrupt service routines. The interrupt vector table starts at memory location 0000:0000h and ends at 0000:03FCh. An interrupt vector is a 4-byte value of the form offset; segment, which represents the address of a routine to be called when the CPU receives an interrupt. The interrupt vector table is first initialized by the start up ROM but changes are made to it's contents as the first ROM Extensions and later the operating system files are loaded. The ability to update the contents of the interrupt vector table provides a means to easily expand operating system services.

* 1. **Edit, Assemble, Link and Run an Assembly Program**
		1. **Assembly programming Syntax for ELASS**

The source code of an assembly language statement should have the following format:

 ***{identifier} keyword {{parameter}} {;comment}***

Each elements of a statement must appear in their appropriate order, but no significance is attached to the column in which an element begins. The assembler is not case sensitive.

A keyword in assembly language is eithera directive, an instruction statement or a data allocation statement.

An identifier is more like a label/variable or like a procedure name in higher-level languages. Identifiers are composed of the letters of the alphabet, the digits 0 through 9 and special characters \_, @, ? ! and $. The first character in an identifier may not be one of the digits 0 through 9 and may not be one of assembler’s keyword. There is no limit to the length of an identifier, but the length of each statement is limited to 240 characters. Since identifiers exist within statements, that limits their length.

A parameter is an operand for assembly keywords. Depending on the keyword type the program statement can have one or more parameters.

 A comment is a string of text that is used for program clarification. At the time of assembling the assembler ignores the comments.

Assembly language is called a low-level language because it allows programmers to operate at the level of the machine itself. What distinguishes assembly language from FORTRAN, BASIC, COBOL, Pascal, C and the other higher-level languages is the fact that each assembly language instruction translates into a single machine language instruction. By contrast, each instruction in a higher-level language might find itself being translated into tens or hundreds or even thousands of lines of machine language.

The real reason for writing programs in assembly language is to produce programs that run fast. An assembly language programmer writes only the code that is absolutely necessary to accomplish a desired task. The program will be much longer on paper than the equivalent program written in a higher-level language, but it will be much shorter in the machine. An assembly language program will typically occupy much less memory and run much faster than a program written in a higher-level language that does the same thing.

* + 1. **Assembling, Linking and Executing Programs**

The symbolic instructions that you code in assembly language (using a text editor such as notepad, as ELASS doesn’t have a built in Editor) are known as the *source program.* You use an assembler program to translate the source program into an intermediate (but non executable) code, known as *object program.* Finally, you use a linker program to complete the machine addressing for the object program, generating an *executable (machine) module.*

Once you have keyed in all statements for the program, examine the code for accuracy. As it stands, this source program is just a text file that cannot execute-you must first assemble and link it. The picture below provides a chart of the steps required to assemble, link and execute a program.



1. The *assembly* step involves translating the source code into object code and generating an intermediate .OBJ (object) file, or module. One of the assembler’s tasks is to calculate the offset for every data item in the data segment and for every instruction in the code segment. The assembler also creates a *header* immediately in front of the generated .OBJ module; part of the header contains information about incomplete addresses. The .OBJ module is not quite in executable mode.

2. The link step involves converting the .OBJ module to an .EXE (executable) machine code module. The linker’s tasks include completing any addresses left open by the assembler and *combining separately assembled programs* into one executable module.

3. The last step is to load the program for execution. Because the loader knows where the program is going to load in memory, it is now able to resolve any remaining addresses still left incomplete in the header. The loader drops the header and creates a program segment prefix (PSP) immediately before the segment loaded in memory.

* + 1. **Assembling a Source Program**

The assembler converts your source statements into machine code and displays any error messages on screen. Typical errors include a name that violates naming conventions, an operation that is spelled incorrectly, and an operand containing a name that is not defined. Because there are many possible errors (100 or more) and many different assemblers, you may refer to your assembler manual for a list. The assembler attempts to correct some errors but, in any event, reload your editor, correct the .ASM source program and reassemble it.

Possible output files from the assembly step are object (.OBJ), listing (.LST), and cross reference (.CRF or .SBR). You usually request an .OBJ file, which is required for linking a program into executable form. You will probably often request (if your assembler supports) an .LST file, especially when it contains error diagnostics or you want to examine the generated machine code. A .CRF file is useful for large programs where you want to see which instructions reference which data items.

* + 1. **Two-Pass Assemblers**

Assemblers typically make two or more passes through a source program in order to resolve forward references to addresses not encountered in the program. During pass 1, the assembler reads the entire source program and constructs a symbol table of names and labels used in the program, that is, names of data fields and program labels and their relative locations (Offsets) within the segment. Pass 1 also determines the amount of code to be generated for each instruction.

During pass 2, the assembler uses the symbol table that it constructed in pass 1. Now that it knows the length and relative position of each data field and instruction, it can complete the object code for each instruction. It then produces, on request, the various object (.OBJ), list (.LST) and cross-reference (.CRF) files

* + 1. **.COM vs .EXE**

For an .EXE program, the linker automatically generates a particular format and, when storing it on disk, proceeds with a special header block that is 512 bytes or more long. You can also write .COM programs for execution (if the assembler supports it). The advantages of .COM programs are that they are *smaller than comparable .EXE programs* and are more easily adapted to act as *memory resident* programs. The .COM format has its roots in earlier days of microcomputers, when program size was limited to 64K and accordingly somewhat primitive and limited.

*Differences between an .EXE and a .COM program:*

**Program Size:** A .COM program uses *one* segment for both instructions and data, basically restricted to a maximum of 64K, including the program segment prefix (PSP). The PSP is a 256byte (100h) block that the program loader inserts immediately preceding .COM and .EXE programs when it loads them from disk to memory. A .COM program is always smaller than its counterpart .EXE program; one reason is that a 512-byte header record that precedes an .EXE program on disk does not precede a .COM program.

**Segments:** The use of segments for .COM programs is significantly different (and easier) than for .EXE programs. A full .COM program combines the PSP, Stack, Data and Code segments into one code segment. For an .EXE program, you usually define a data segment and initialize DS with the address of that segment. Although you can define a stack segment for an .EXE program, the assembler automatically generates a stack for a .COM program.

**Initialization:** When the program loader loads a .COM program for execution, it automatically initializes CS, DS, SS, and ES with the address of the PSP. Because CS and DS now contain the correct initial segment address at execution time, a .COM program does not have to initialize them.

* + 1. **Editing a Source Program**

You can enter and edit your program using any text editor as long as it produces, which is called a text file. Practically all word processors and text editors can produce a pure text file.

* + 1. **Assembling the Source file**

To assemble the source program using ELASS, specify the following command line (after opening the command prompt):

 ***c:\path>*elass** filename

; Where filename is the name of an assembly language source code file.

The source code file must be stored with a filename extension of *.ASM*, but the extension may be omitted from the command line. ELASS will assemble the named source code file and generate an OBJECT file of the same name with a filename extension of .OBJ and store that file in the same directory in which it found the source code file.

* + 1. **Linking Object files**

To link a module or series of modules using ELINK, specify the following command line:

 ***c:\path>*elink** objfile {{+objfile}} {,runfile}

Where “objfile” is the name of an object file and runfile, if specified, is the name under which ELINK is to save the EXEcutable program file it produces. If runfile is omitted, ELINK will save the program file under the name of the first object file specified in the command line.

ELINK will recognize a command line parameter preceded by an **“@”** character as an automatic response file specification. An automatic response file is an ASCII text file consisting of one or more command line parameters whose contents will be incorporated by reference into the command line on which the automatic response file is specified.

For example, if a file name AUTORESF contained the following text:

 module01 + module02, program

then the command line:

 ***c:\path>*elink** @autoresf

will be logically equivalent to:

 ***c:\path>***elink module01.obj + module02.obj, program

* + 1. **Loading/Running**

You can Run/Load the program simply by using DOS command line and the file name. For example to load/run the program type the following on the DOS command line:

 ***c:\path>*filename** ***{parameter1}{*** ***parameter2}***

Where filename is the name of the executables file and parameter1 and parameter2 are the program parameters, which are optional for the pr