Objectives

- To introduce various addressing modes
- To provide an introduction to 80x86 assembly language programming
- To provide an introduction to the tools for generating, loading and executing assembly language programs
- To provide an introduction to the different types of instructions

Addressing Modes

- In order to understand how does a computer operates in the lowest level and to write an assembly program, addressing modes must be thoroughly understood.
- There are many addressing modes, but the most common are easy to master and represent most instructions in most programs.
- Register addressing, immediate addressing, direct addressing, and simple forms of indirect addressing are the most common addressing modes.

Data Addressing Modes

- Data addressing modes are presents with the MOV instruction because it is by far the most common instruction in a program.
- MOV really moves nothing. MOV copies the source into the destination. It probably should be named copy, but it is not.
Register

- Both operands are registers
  - MOV EAX,EBX
  - MOV CX,DX
  - MOV AH,AL
  - MOV AX,DS
  - MOV ES,CX
  - ADD AL,CL (most instructions use it)
  - OR AX,DX

Immediate

- Move an “immediate” value (contained in the instruction) to the destination
  - MOV EAX,234H
  - MOV CX,2
  - MOV AL,34H
  - ADD AL,3
  - SUB CL,4
  - AND EAX,1

Direct

- The destination or source is a memory location specified by a logical name or the specific Seg:Offset value(s)
  - MOV BOB,EAX ; BOB is a variable name
  - MOV FRED,CX ; FRED is another name
  - MOV BILLY,AL ; so is BILLY
  - MOV EDI,RALPH ; and RALPH, STEVE &
  - MOV AX,STEVE ; BARNEY. However, the
  - MOV DS,BARNEY ; data type can be different
  - MOV AL,DS,[1000H] ; location at DS:[1000H]
Register Indirect

- The register content is an offset value (consider Pointer in C)
- MOV AL,[BX] ; Move Word from DS:[BX]
- MOV AX,[EBX] ; Move 32 bits from DS:[EBX]
- MOV [EDI],EAX ; Move to DS:[EDI]
- MOV [EAX],EDX ; Move to [EAX]
- MOV BYTE PTR [EAX],6 ; Move byte to DS:[EAX]
- MOV WORD PTR [ECX],12 ; Move Word to DS:[ECX]
- MOV DWORD PTR [ESI],2345H ; Move DWord to DS:[ESI]

Base Plus Index

- An extension of Indirect address mode. A combination of Base Register and one of the Index registers
- MOV AL,[BX+SI] ; DS:[BX+SI] -> AL
- MOV [BX+DI],AX ; DS:[BX+DI] -> AX
- MOV [BP+SI],EAX ; SS:[BP+SI] -> EAX
- MOV AL,[BP+DI] ; SS:[BP+DI] -> AL
- MOV WORD PTR [BX+SI],5 ; 5 (16bit) -> DS:[BX+SI]
- ADD AL,[BX+DI] ; AL = AL + DS:[BX+DI]

Register Relative

- = Indirect Register Mode plus offset
- MOV AL,[BX+3] ; DS:[BX+3] -> AL
- MOV AX,[DI+20H] ; DS:[DI+20H] -> AX
- MOV [EDI+200H],EAX ; EAX -> DS:[EDI+200H]
- MOV [BX-33],ECX ; ECX -> DS:[BC+2]
- ADD BYTE PTR [BX+2],5 ; Add 5 to Bye location at DS:[BX+S]
Relative Plus Index
- Register indirect plus offset.
- MOV AL,[BX+SI+22]
- MOV [BX+DI-22],AX
- MOV EAX,[EBX+EDI+100H]

Scaled Index
- Not available in real mode
- MOV EAX,[EBX+4*ECX]
- MOV AX,[EDI+2*EBX]
- MOV [EAX+2*EBX],DX
- MOV [4*ECX],EBX
- ADD AL,[ECX+EBX]

Program Flow
- Program flow instructions are unconditional (JMP) or conditional (JNZ).
- Flow instructions are short (+127, -128), near (±32K), or far (anywhere in the memory).
- Conditionals do not contain the far type.
- Labels are followed by a colon if they are jumped to in a program.

Stack Addressing
- The SS (stack segment) and the SP/ESP are added to form an address in the stack.
- The stack is an area of memory that functions as a last-in, first-out (LIFO) memory. That is if a 1 followed by a 2 are placed on the stack the 2 comes out of the stack first, followed by the 1.
- PUSH and POP are used to store data on the stack and to remove data from the stack as words or doubleword.
Detailed Objectives:

• Explain the meaning of the following concepts
  – relationship between machine level and mnemonic (Assembly) programming
  – Assembler, Compiler, Linker and Loader
  – Assembly Language and Mnemonics
  – MS-DOS debugger DEBUG
  – Turbo/Microsoft Assemblers (TASM / MASM)
  – instruction formats
  – data transfer instructions
  – arithmetic instructions

• Know how to perform the following operations:
  – use the debugger to view the raw memory contents of either data or code
  – display values with an appropriate format in the watch window within the debugger
  – write pure assembly language programs
  – single step through machine level instructions within the debugger
  – display the contents of the CPU registers while single stepping
  – use the DEBUG facility to enter machine code and data, single step and view register contents

Executing Computer Instructions using DEBUG

• Debug is a simple, easy-to-use debugger program supplied with MS-DOS. It is used for testing and debugging executable programs.
• Debug displays all program code and data in hex format
• Any data that you enter into memory must also be in hex format

• It also provides a single-step mode which allows the execution of a program one instruction at a time, which is very useful for viewing the effect of each instruction on memory locations and registers
• Debug does not distinguish between lower and upper case letters, so commands may be entered either way
• Debug can be run from inside Windows by selecting the MS-DOS prompt from the Windows Start menu:
• Select Start|Programs|MS-DOS Prompt
• When MS-DOS prompt appears, type **Debug** and press Enter.
• You will see Debug’s command prompt, which is a hyphen ("-"). Commands may be typed in upper or lowercase letters. A command may be followed by one or more parameters.
• To see some of the functions provided by Debug enter a ‘?’ at Debug’s prompt. Debug will respond with a list of the available commands.

**Some commonly used Debug commands:**

• **A [address]**  **Start assembling** a program, placing each instruction in memory. Optionally an integer parameter can be supplied which specifies the hex location where the first instruction is to be placed
• **D [range]**  **Dump** (display) the contents of memory
• **E address [list]**  **Enter** data (bytes) into memory beginning at specified location

**Examples**

```plaintext
C: \WINDOWS>debug
  -> 0100 b8 07 00 00 00 10 00 00 c3 05 00 03 c0 05 00 00 16 01 b4 4c cd 21
  -> d ; To display memory locations
1486:0100 B8 07 00 00 00 10 00-00 c3 05 00 03 c0 05 00 00 16 01 b4 4c cd 21
1486:0110 b4 4c cd 21 c3 05 00 03 c0 05 00 00 16 01 b4 4c cd 21
1486:0120 e8 80 3e 61 e2 00 74 34 00 75 14 be 61 e2 e8 98 02 2e a1 18
1486:0130 00 c3 00 00 9e 02 00 00 3d-41 00 74 07 eb 74 06 e8 48 02 80 3e-3c
1486:0140 00 c3 00 00 9e 02 00 00 3d-41 00 74 07 eb 74 06 e8 48 02 80 3e-3c
1486:0150 00 c3 00 00 9e 02 00 00 3d-41 00 74 07 eb 74 06 e8 48 02 80 3e-3c
1486:0160 ba 81 e9 09 df 0f 0d-01 ba 5f 0e 2e 1486:0170 ba 81 e9 09 df 0f 0d-01 ba 5f 0e 2e

-r ; to view the contents of all registers
AX=0000 BX=0000 CX=0000 DX=0000
SP=FFE5 BP=0000 SI=0000 DI=0000
DS=1486 ES=1486 SS=1486 CS=1486
IP=0100 NV UP EI PL NZ NA PO NC
1486:0100 B80700 MOV AX,0007
```
- `r cs` ; to view and change the contents of one reg.
  CS 1486
  :
  ; hit enter if no change
- `r cs`
  CS 1486
  :2540
  ; change the contents of CS to 2540H

- ; enter code/data at specified address (below)
  -e 2540:0200 b8 07 00 8b d8 b8 10 00 03 c3 05 15 00 a3 16 01
    b4 4c cd 21
  -d 2540:0200 ; display contents starting at
    2540:0200 b8 07 00 8b d8 b8 10 00 03 c3 05 15 00 A3 16 01
    2540:0210 B4 4C CD 21 00 00 00 00 00 00 00 00 00 00 00 00
    2540:0220 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    2540:0230 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    2540:0240 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    2540:0250 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    2540:0260 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    2540:0270 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

- ; r cs
  CS 2540
  :; change the contents of IP to offset 0000H
  -r ip
  IP 0000
  ; show me the contents of IP
  ip

• You can change the contents of most of the registers when in DEBUG
• It may not be possible to change the contents of some memory locations. Why?
• To trace or execute a machine language program with Debug, type the name of the program as a command line parameter, eg,

  Debug test.exe

- ; enter code/data at specified address (below)
  -e 2540:0200 b8 07 00 8b d8 b8 10 00 03 c3 05 15 00 a3 16 01
    b4 4c cd 21
  -d 2540:0200 ; display contents starting at
    2540:0200 b8 07 00 8b d8 b8 10 00 03 c3 05 15 00 A3 16 01
    2540:0210 B4 4C CD 21 00 00 00 00 00 00 00 00 00 00 00 00
    2540:0220 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    2540:0230 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    2540:0240 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    2540:0250 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    2540:0260 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    2540:0270 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

- ; enter code/data at specified address (below)
  -e 2540:0200 b8 07 00 8b d8 b8 10 00 03 c3 05 15 00 a3 16 01
    b4 4c cd 21
  -d 2540:0200 ; display contents starting at
    2540:0200 b8 07 00 8b d8 b8 10 00 03 c3 05 15 00 A3 16 01
    2540:0210 B4 4C CD 21 00 00 00 00 00 00 00 00 00 00 00 00
    2540:0220 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    2540:0230 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    2540:0240 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    2540:0250 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    2540:0260 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    2540:0270 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

• Eg, you can write a program in C or assembly language, compile/assemble that program to create an executable file, and then use DEBUG to step through it.
• Debug accepts commands in either upper or lower case. (i.e, like assembler, it is case insensitive.)

When Debug is first loaded, the following defaults are in effect:

1. All segment registers are set to the bottom of free memory, just above the debug.exe program
2. IP is set to 0100h.
3. Debug reserves 256 bytes of stack space at the end of the current segment
4. All of available memory is allocated (reserved)
5. BX: CX are set to the length of the current program or file
6. The flags are set to the following values: NV (overflow flag clear), UP (direction flag = up), EI (interrupts enabled), PL (sign flag = positive), NZ (zero flag clear), NA (auxiliary Carry flag clear), PO (odd parity), NC (carry flag clear).
Introduction to Assembly Language

- Numeric constants
  - Numeric constants are made of numerical digits with, possibly, a sign and a suffix. Ex:
    - \(-23\) (a negative integer, base 10 is default)
    - \(1011\)b (a binary number)
    - \(1011\) (a decimal number)
    - \(0A7Ch\) (an hexadecimal number)
- The floating point numbers w.r.t. assembly language will not be discussed in this unit.

Character and String constants

- Any sequence of characters enclosed either in single or double quotation marks. Embedded quotes are permitted. Eg,
  - ‘A’
  - ‘ABC’
  - “Hello World!”
  - “123” (this is a string, not a number)
  - “This isn’t a test”
  - ‘Say “hello” to him’

Names

- A name identifies either:
  - a label
  - a variable
  - a symbolic constant (name given to a constant)
  - a keyword (assembler-reserved word).
- The first character must be a letter or any one of ‘@’, ‘_’, ‘$’, ‘?’
  - subsequent characters can include digits
  - A programmer chosen name must be different from an assembler reserved word or predefined symbol.
  - avoid using ‘@’ as the first character since many predefined symbols start with it
- By default, the assembler is case-insensitive

Example

- A variable is a symbolic name for a location in memory that was allocated by a data allocation directive. Eg,
  - count db 50 ; allocates 1 byte to variable count
- A label is a name that appears in the code area. Must be followed by ‘:’

Label1:
  mov ax, 0
  mov bx, 0

Label2:
  jmp Label1 ; jump to Label1
Instructions, Directives and Comments

• An assembly language statement consists of four fields:
  [name[:]] [mnemonic] [operands] [:comments]
• The square brackets indicate that the field is optional – square brackets are not part of the statement
• From the above definition, it is then possible to have a blank line as a valid statement (ie. all parts of the statement are optional).

• The names must not exceed 31 characters.
  • The mnemonic (operation/opcode) indicates the operation the instruction is to perform.
  • Assembly language statements usually fall into two classes:
    – Instructions
    – Directives

Instructions:

• An instruction in assembler is an executable statement that tells the CPU what action to take in processing some data.
• One instruction in assembler is usually interpreted into one instruction in machine code.
• Instructions fall into the following general types:
  transfer of control (eg, call Mysub)
  data transfer (eg, mov ax, 10)
  arithmetic (eg, add ax, 5)
  logical (eg, jnz step2)
  input/output (eg, in al, 50)

Opcodes

• These are the mnemonics of 2 to 6 letters which are verbs that describe what an operation is to do, eg, MOV, ADD
• These are abbreviations which save us the trouble of remembering the binary or hexadecimal code for the operations in machine language.
• The assembly language provides a number of words which act as opcodes.
Operands

- Register names, data fields, character strings or numbers may be included as operands for an instruction.
- These operands are separated by commas, if there is more than one. They supply the data for the instruction which is to be executed.

- If a number is going to be used, unless otherwise stated, the assembler assumes the number is decimal. You may use hexadecimal or binary values if you wish, but you must state this by following the number with a 'h' if hexadecimal or a 'b' if the number is binary.

Directives

- Also called pseudo-operations, directives are statements that tell the assembler to perform a certain function at assemble time.
- They have no effect on the program at run time. Directives are not converted to machine code unless they are definitions of data constants.

- PAGE and TITLE directives are used to control the format of a listing of an assembled program:
  - PAGE [length] [width]
  - TITLE text [comment]
- Eg.
  - PAGE 20,60 ;20 lines, 60 chars/line
  - TITLE Hello World

Examples of directives

- The following dw directive tells the assembler to create storage for a word variable named count and initialise it to 100
  - count dw 100
- The following .stack directive tells the assembler to reserve 1024 bytes of stack space:
  - .stack 1024

- SEGMENT and PROC directives are used to define segments and procedures respectively.
- Similarly, END and ENDP directives are used to end the segment and procedure definitions respectively
Macros

- These are single statements that cause the assembler to place one or more statements in the source code at the position the macro holds.
- Before a macro can be used it must first be defined to show the assembler which statements to replace the macro with.

Comments

- Comments are included in the source code of an assembler program for the sole purpose of making the program’s function clear. They are never compiled into machine code. The assembler ignores comments.
- When you write assembler programs it is common practice to have a comment explaining the function of the program on nearly every line of code. Some people suggest to write the pseudo-code as in the comments.
- The overall aim of the comments is to tell anyone reading the code WHY the program is performing any particular section of code. The code itself can tell them how the operation is being performed.

Segments

- The memory on the IBM personal computer is not organised into a continuous stream of memory locations. Note that other computers do however have a linear memory organisation.
- The memory on the IBM compatible computers is broken up into segments.
- Each of these segment’s addresses are held in the appropriate segment registers.

- As mentioned previously, there are four segment registers, the CS, DS, ES, and SS. Each of these registers holds the address of the first byte within the segment.
- When we set up our assembly programs we must tell the assembler which parts of our programs are to be placed in which segments.
- The CS register holds the address of the start of the code segment. The code segment contains the code or instructions associated with our program.

Segment Directives

- The DS register holds the address of the data segment. The data segment holds the data associated with our program.
- The SS register has the address of the stack segment. The stack segment holds the stack which is to be used with our program.
- The ES register contains the address of the extra segment. The ES segment holds any additional data required by our program.

- A program normally consist of a:
  - code segment that holds the executable code
  - data segment that holds the variables
  - stack segment that holds the stack (used for calling and returning from procedures/subroutines)
- Directives `.code`, `.data`, and `.stack` mark the beginning of the corresponding segments.
- The `.model small` directive indicates that the program uses one code segment and one data segment.

Defining Data

- When there is a need for some variables to be used within an assembler program, which will be nearly always, they are usually defined within the data segment.
- A variable may be defined as bytes (DB), words (DW), double words (DD), six bytes (DF) or quad words (DQ).

- The size of the variables is important when you consider how big a binary number must be to hold a certain value.
- For example, if you needed a variable which needed to hold a value in excess of 1 million, you would need to define a double word variable (32 bits). If you were to use a variable, with the highest expected value of 10, you could expect to use a byte variable.

Most variables declared within an assembler program are declared as bytes or words.
- Eg, the DB (define byte) directive allocates memory for one or more 8-bit (byte) values:

```
<table>
<thead>
<tr>
<th>name</th>
<th>DB</th>
<th>initval</th>
</tr>
</thead>
<tbody>
<tr>
<td>num1</td>
<td>db</td>
<td>25</td>
</tr>
</tbody>
</table>
```
- Multiple initialisers: it is possible for the name of a variable to identify a sequence of bytes enabling multiple initialisers to be used in the same declaration, eg, see the definitions of ‘numbers’ and ‘list’ below:

```
numbers db 1,2,3
list db 10,20,30,40
```

- Most variables declared within an assembler program are declared as bytes or words.
- Eg, the DB (define byte) directive allocates memory for one or more 8-bit (byte) values:

```
<table>
<thead>
<tr>
<th>name</th>
<th>DB</th>
<th>initval</th>
</tr>
</thead>
</table>
| char1 db 'A' ; allocate 1 byte of mem.
| num1 db 25 ; value 25
| numbers db 1,2,3 ; allocate 3 bytes of memory, initialise to values 01 02 03
| list db 10,20,30,40 ; allocate 4 bytes, initialise them to 10 20 30 40
| digits db '1','2','3'; 13 32 33
| string db 'ABCD'; 41 42 43 44
| eights dw 2 dup (1000b); 08 08 08 08
| minus_one dw -1 ;FF FF
| var1 db ? ; ?? - undefined byte value in memory
| var2 dw ? ; ?? - undefined word value in memory
| string8 db 4 dup (?); ?? ?? ?? ?? - undefined four byte string in mem.
```

--- Data ends ---
• Note that when words are stored in memory they are stored as the least significant byte first (little-endian method).
• As with any other language you must declare a variable before you can use it within the code.
• To declare a constant, that is a value which is not to change throughout the code, you define the value using the EQU (equate) directive. This informs the assembler that whenever it finds the name of the constant within the source code, it must replace it with the value given as part of the directive.

Example

```
max_value EQU 256   ; assign the value 256 to the constant max_value
clock_tick EQU 1193180 ; assign this value to a constant clock_tick
```

• The above is similar to #define in C
• After these constants have been declared their values may not be changed from within the program. Remember that directives are like "pre-compiler" instructions which have no affect at run time.

Defining Segments

• There are 2 ways to define segments:
  • Full Segment Definition is the traditional method of defining segments.
  • Simplified Segment Definition, which is easier to understand and use, is supported by TASM (Turbo Assembler version 1.0 and higher) and MASM (Microsoft Assembler version 5.0 and higher).

Example of code segment:

```
CODE SEGMENT
 prog_name PROC far ; this is the program entry pt.
 assume cs:code, ds:data, ss:stack

 prog_name ENDP
 CODE ENDS
 END prog_name
```
• The code above is made up of directives which
  – marks the start of the code segment,
  – tells the assembler to assume certain things about
    the segments which have been set up,
  – gives the name of the program that is about to be
    written,
  – marks the end of the program’s procedures, and
    the end of the code segment as a whole.

Stack Segment

Example of stack segment:
STACK SEGMENT STACK
  db 32 dup (?)
STACK ENDS

• The above directives inform the assembler that the stack segment is being initialised. It is defined to hold bytes, each byte is to be filled with the character which is included in the brackets. The contents of the brackets will be copied to the stack 32 times.

Example

; A simple assembly language program
; Uses full segment definition
; -----stack segment ----- 
name1 SEGMENT
  DB 64 DUP (?)
name1 ENDS

; -----data segment ----- 
name2 SEGMENT
  DATA1 DW 1234H
  DATA2 DW 5678H
  SUM DW ?
name2 ENDS

; -----code segment ----- 
name3 SEGMENT
  MAIN PROC FAR 
    ; this is the program entry pt.
    ASSUME CS:name3, DS:name2, SS:name1
    MOV AX, name2
    ; load the data segment address
    MOV DS, AX
    ; assign value to DS
    MOV AX, DATA1 ; get the first operand
    MOV BX, DATA2 ; get the second operand
    ADD AX, BX ; add operands
    MOV SUM, AX ; store result in SUM
    MOV AH, 4CH
    ; set up to INT 21H
    INT 21H ; return to DOS
  MAIN ENDP
name3 ENDS
END MAIN ; this is the program exit point

[Note: The statements in bold in the above example are required in every program.]

Simplified Segment Definition

• Requires choosing the appropriate memory model from among: SMALL, MEDIUM, LARGE, etc.
• Requires using three simple directives: “.STACK”, “.DATA”, and “.CODE”, which correspond to the SS, DS, and CS registers respectively.

Example

; A simple assembly language program
; Rewritten using simplified segment definition
.MODEL SMALL
.STACK 64
; -----data segment ----- 
.DATA
  DATA1 DW 1234H
  DATA2 DW 5678H
  SUM DW ?
Another Example: The Hello World program in assembly language

Title Hello World Program
; File: hello.asm
; This program displays “Hello, world!”

.MODEL SMALL ; each of data and code segments < 64K
.STACK 100h ; reserve 256 bytes of stack space

; -----data segment-----
.DATA ; indicates beginning of data
message db “Hello, world!”,0dh,0ah,‘$’

; -----code segment-----
.CODE ; indicates beginning of code segment
.MAIN PROC ; proc declares the start of a proc
    MOV AX, @DATA ; load the data segment address
    MOV DS, AX ; assign value to DS
    MOV AX, DATA1 ; get the first operand
    MOV BX, DATA2 ; get the second operand
    ADD AX, BX ; add operands
    MOV SUM, AX ; store result in SUM
    MOV AX, 4C00H ; set up to return
    INT 21H ; return to DOS
.END MAIN

[Note: The statements in bold in the above example are required in every program.]

Assembling, Linking and Executing programs

• The steps involved are:
• Edit
  A source program consisting of assembly language statements is created using a text editor (e.g., Programmer’s File Editor - pfe, which is a public domain editor and can be downloaded from the ICT106 Unit materials site). This results in a source file of ASCII text.

1. Assemble
   The assembler program reads the source file and produces an object file, which is a machine-language translation of the source program.

2. Link
   The object file may contain calls to routines (e.g., operating system service functions) in an external link library. The linker then copies the needed routines from the link library into the object file, creates a special header record at the beginning of the program, and produces an executable program.

3. Load
   When the program is run by entering its name on the DOS command line, the DOS loader decodes the header record of the executable program and loads it into memory

4. Execute
   The CPU begins executing the program
Example

- The table below displays a list of file names that would be created if the Hello World program was assembled and linked.

<table>
<thead>
<tr>
<th>Step</th>
<th>Outcome</th>
<th>Filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit</td>
<td>Source program</td>
<td>Hello.asm</td>
</tr>
<tr>
<td>Assembly</td>
<td>Object program</td>
<td>Hello.obj</td>
</tr>
<tr>
<td></td>
<td>Listing File</td>
<td>Hello.lst</td>
</tr>
<tr>
<td>Link</td>
<td>Executable program</td>
<td>Hello.exe</td>
</tr>
<tr>
<td></td>
<td>Map file</td>
<td>Hello.map</td>
</tr>
</tbody>
</table>