

Chapter 3 Assembly Language Program and Programming Tools

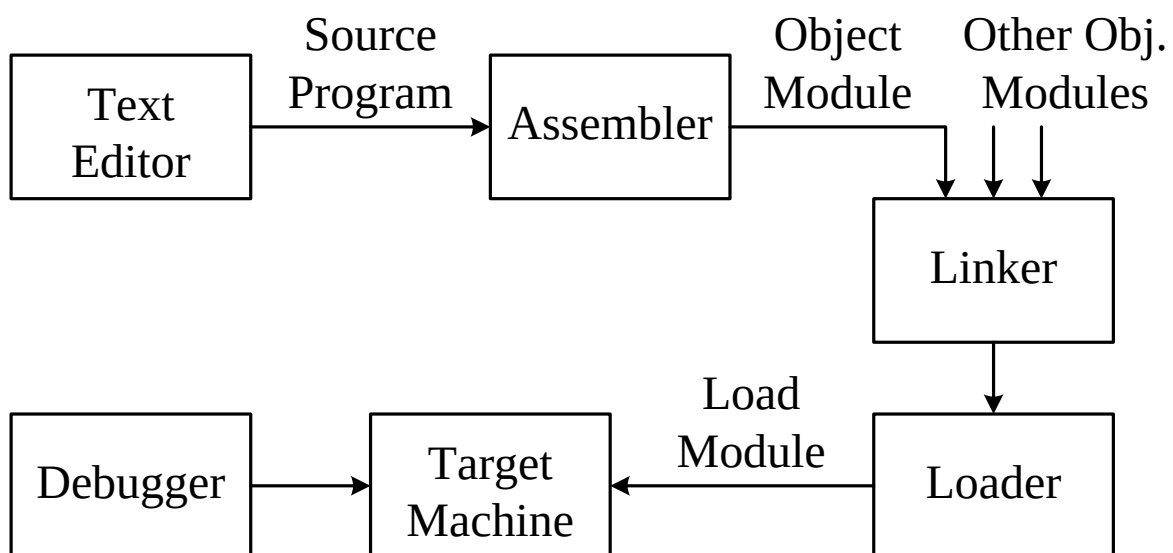
3.1 General

Assembly language is a low level programming language. It comprises with a set of instructions which direct the microprocessor to perform specific functions. Different CPU has its own instruction set.

Example of instructions: MOV, ADD, AND, etc..

Machine language (comprised with “0” and “1”) is the language that microprocessor understands. Assembly language is required to be translated to machine language. Each instruction is represented by a binary sequence and its binary pattern is dependent on the instruction, operand and addressing mode.

3.2 Software Tools



Assembler:

A program that translates an assembly language program (source program) into bit strings (object module) that make up a machine language program, called object module.

Cross Assembler:

An assembler that runs on one machine and produces object module for another machine that with a different CPU.

Macro Assembler:

Macro is sequence of instruction which may occur in a program very often. A macro name can be used to replace such sequence in a program. The macro assembler performs all functions of a standard assembler and it *textually* replaces the macro name with the full instruction sequence.

Linker:

A program resolves external references and concatenates object modules into one contiguous relocatable module.

Loader:

A program that downloads the bit strings to the suitable position of the target machine's memory.

Debugger:

It executes one or a few instructions of a program at a time to allow programmer to see the effects of small pieces of the program and thereby isolate programming errors (bugs).

Simulator:

A program simulates the individual machine instructions. It is usually run on a machine other than the one being simulated. It can be used to test the validity of a program before the hardware is available.

Relocating Assembler***Absolute code:***

Machine code that has pre-defined load address (the first address that the program occupies in memory); and also has specified addresses for temporary storages.

Position-independent code:

Load address is not pre-defined in the program. The program can be loaded anywhere in memory and still run correctly, particularly important for program linked up with multiple object modules. All temporary storages should be made relatively to the program counter (PC).

Relocating Assembler:

It keeps track of what must be changed in a program if the load address is changed and a relocation dictionary containing information for the relocatable loader will be produced. The assembled instructions and relocation information together are called ***relocatable code***.

3.3 Example program:

Example: A source program for ARM state (32-bit instruction format)

20 + 30 => result is stored in memory with label *res*

For GNU Assembler	
<pre>.global _start .equ x = 0x20 .text _start: /* program entry point */ MOV R0, #x /* R0 = 0x20 */ MOV R1, #0x30 /* R1 = 0x30 */ ADD R2, R0, R1 /* R2 = R0 + R1 */ LDR R3, =res /* Set address pointer */ STRB R2, [R3] /* Store results */ loop: B loop .data res: .byte 0xFF .end</pre>	

For ARM Software	
<pre>x EQU 0x20 AREA P_Name, CODE, READONLY ENTRY start MOV R0, #0x20 ; R0 = 0x20 MOV R1, #0x30 ; R1 = 0x30 ADD R2, R0, R1 LDR R3, =res STRB R2, [R3] loop B loop Start_of_tab DCB 0xFF END</pre>	

The listing file

-----Building project: test1 -----

arm-elf-as -a -gdwarf2 C:\EmbestIDE\ASM_test\test1\t1.s -o.\debug\t1.o

ARM GAS C:\EmbestIDE\ASM_test\test1\t1.s

```

1          .global _start
2
3          .text
4
5          _start:          /* program entry point */
6  0000 2000A0E3      MOV     R0, #0x20      /* R0 = 0x20 */
7  0004 3010A0E3      MOV     R1, #0x30      /* R0 = 0x30 */
8  0008 012080E0      ADD     R2, R0, R1      /* R2 = R0 + R1 */
9  000c 04309FE5      LDR     R3, =res        /* Set address pointer */
10 0010 0020C3E5      STRB    R2, [R3]        /* Store results */
11 0014 030000EA      loop:  B      loop
12 0018 00000000      .data          /* data area */
13          res:
14 0000 FF          .byte  0xFF
15          .end

```

DEFINED SYMBOLS

```
C:\EmbestIDE\ASM_test\test1\t1.s:5    .text:00000000 _start  
C:\EmbestIDE\ASM_test\test1\t1.s:13    .data:00000000 res  
C:\EmbestIDE\ASM_test\test1\t1.s:11    .text:00000014 loop
```

ARM 32-bit instruction format → each instruction occupies 4 bytes

The address of an object module starts at 0x0000.

The address of an instruction is 4 plus the address of previous instruction.

The object module contains the machine language program (each instruction is translated to its corresponding machine instruction, which is in a binary form).

Symbol table

example.elf: file format elf32-littlearm

SYMBOL TABLE:

```

0c000000 1      d  .text 00000000
0c00001c 1      d  .glue_7 00000000
0c00001c 1      d  .glue_7t 00000000
0c00001c 1      d  .data 00000000
0c00001d 1      d  .bss 00000000
00000000 1      d  .debug_info 00000000
00000000 1      d  .debug_line 00000000
00000000 1      d  .debug_abbrev 00000000
00000000 1      d  .debug_aranges 00000000
00000000 1      d  *ABS* 00000000
00000000 1      d  *ABS* 00000000
00000000 1      d  *ABS* 00000000
0c00001c 1      .data 00000000 res
0c000014 1      .text 00000000 loop
0c00001d g      O *ABS* 00000000 Image_RW_Limit
0c00001d g      O *ABS* 00000000 __bss_start__
0c00001c g      O *ABS* 00000000 Image_RW_Base
0c00001d g      O *ABS* 00000000 end
0c00001d g      O *ABS* 00000000 __bss_end__
0c000000 g      .text 00000000 __start__
0c00001d g      O *ABS* 00000000 __EH_FRAME_END__
0c00001c g      O *ABS* 00000000 Image_RO_Limit
0c00001d g      O *ABS* 00000000 Image_ZI_Base
0c000000 g      O *ABS* 00000000 Image_RO_Base
0c00001d g      O *ABS* 00000000 __EH_FRAME_BEGIN__
0c00001d g      O *ABS* 00000000 __end__
0c00001d g      O *ABS* 00000000 Image_ZI_Limit

```

The symbol table contains:

- the address of label
- the values of constant

3.4 Assembly Language Program Structure:

For GNU Assembler	
<pre> .global _start .equ x = 0x20 .text _start: MOV R0, #x /* program entry point */ MOV R1, #0x30 /* R0 = 0x20 */ ADD R2, R0, R1 /* R0 = 0x30 */ LDR R3, =res /* R2 = R0 + R1 */ STRB R2, [R3] /* Set address pointer */ loop: B loop /* Store results */ .data /* data area */ res: .byte 0xFF .end </pre>	

label: instruction/assembler_directive operand(s) /* comment */

Label: refers to the line it labels and is optional. Label is followed with “:”.

Instruction or **assembler directive:** an instruction; assembler directive tells the assembler something about the program.

Operand(s): data associated with the instruction.

Comment: for easy understanding of the program and is optional, which is enclosed by /* ... */.

3.5 Assembler Directive (GNU)

Assembler directives tell the assembler what to do with the program.

They are not part of the instruction set and they assign the program to certain areas in memory, define symbols, designate memory for storage, allow references to other programs, etc..

All assembler directives have names that begin with a full-stop “.”.

Most important of all, assembler directive will not occupy any memory space in the object module !!!

Data format description:

Hexadecimal no: starting with 0x or 0X, e.g. **0x**a7fe

Binary no: starting with 0b or 0B, e.g. **0b**11010011

Octal no: starting with 0, e.g. **0**7431

Decimal no: no prefix, e.g. 2398

ASCII character: embraced by ‘’, e.g. ‘A’, ‘g’

Some commonly used assembler directives:

.global symbol

Specify that ***symbol*** is to be made globally visible to all other modules that are part of the executable, and visible to the GNU Linker. The symbol ***_start***, which is required by the GNU Linker to ***specify the first instruction to be executed in a program***, must always be a global one (and only present in one module, of course).

.equ symbol, expression

The following three lines are identical, and set the value of adams to 42:

```
.equ adams, (5 * 8) + 2
.set adams, 0x2A
adams = 0b00101010
```

.text

Switch the following section for instructions (program) of the final executable.

.data

Switch the following section for data of the final executable. All executable programs have at least one **.text** section and many optional **.data** sections.

Directives having those names allow you to switch back and forth between program and data sections.

.end

Mark the end of this source code file. ***Everything*** after this directive is ignored by the assembler.

.byte expression ...

Insert the (8-bit) byte value of the expression into the object file. More than one expression may appear, if separated by commas.

```
.byte 64, 'A'           /* inserts the bytes 0x40 0x41 */
.byte 0x42              /* inserts the byte 0x42 */
.byte 0b1000011, 0104   /* inserts the bytes 0x43 0x44 */
```

Form of memory map (assuming starting address of 0x100000)

Address	+0	+1	+2	+3
0x100000	40	41	42	43
0x100004	44			

.hword *expression* ...

.2byte *expression* ...

.short *expression* ...

Insert the (16-bit) half-word value of the expression into the object file.

.hword **0xAA55**, 12345 /* inserts the bytes 0x55 0xAA 0x39 0x30 */

.2byte **0x55AA**, -1 /* inserts the bytes 0xAA 0x55 0xFF 0xFF */

Form of memory map (assuming starting address of 0x100000)

Address	+0	+1	+2	+3
0x100000	55	AA	39	30
0x100004	AA	55	FF	FF

ARM processor accesses h-word size data at even address

It will return wrong result if that location is an odd address.

It is **IMPORTANT** to make sure that the h-word size data is stored at even address, i.e. with 2 bytes boundary. (refer to the .align directive below)

.word *expression* ...

.4byte *expression* ...

.long *expression* ...

Insert the (32-bit) word value of the expression into the object file.

.word **0xDEADBEEF** /* inserts the bytes 0xEF 0xBE 0xAD 0xDE */

.4byte -42 /* inserts the bytes 0xD6 0xFF 0xFF 0xFF */

Form of memory map (assuming starting address of 0x100000)

Address	+0	+1	+2	+3
0x100000	EF	BE	AD	DE
0x100004	D6	FF	FF	FF

ARM processor accesses word size data at address of multiple of 4.

It is **IMPORTANT** to make sure that word size data is stored at location with 4-byte boundary. Otherwise, wrong result will be returned.

.align

When building up a data segment, data may be in byte, h-word or word formats.

ARM will access h-word data at even address and access word data at address of multiple of 4.

When data of mixed format are being stored in the same data segment, data alignment must be made to ensure correct storage in the memory.

.align directive insert 0 to 3 bytes of 0x00 to memory so that the next location will be on a 4-byte (word) boundary.

Example, the following three lines will insert eight bytes into the object file output, assuming that the first line is already on a word boundary:

```
.byte    0x55          /* inserts the byte 0x55 */
.align   /* inserts 3 bytes of 0x00 */
.word    0xAA55EE11    /* inserts 0x11 0xEE 0x55 0xAA */
.hword   0x7F83
.align   /* inserts 2 bytes of 0x00 */
.word    0x3498A5E
```

Form of memory map (assuming starting address of 0x100000)

Address	+0	+1	+2	+3
0x100000	55	00	00	00
0x100004	11	EE	55	AA
0x100008	83	7F	00	00
0x10000C	5E	8A	49	03

.ascii "*string*" ...

Insert the string literal into the object file exactly as specified, with no trailing NUL character. More than one string may be specified if they are separated by commas. As an example, the following line inserts three bytes into the object file output:

```
.ascii    "JNZ"    /* inserts the bytes 0x4A 0x4E 0x5A */
```

.asciz "*string*" ...

Insert the string literal into the object file, followed by a NUL character (a 0x00 byte). As with .ascii, more than one string may be specified if separated by commas. As an example, the following line inserts four bytes (not three) into the object file output:

```
.asciz    "JNZ"    /*inserts the bytes 0x4A 0x4E 0x5A 0x00 */
```

.skip *expression*

Skip *expression* bytes in the object file output. This directive can be used to define a storage area. The following three lines declare three variables, two that are pre-initialized, one (buffer) that is not:

```
head:     .word     123    /* Head pointer to within buffer m*/  
tail:     .word     789    /* Tail pointer to within buffer */  
buffer:   .skip     512    /* Buffer of 512 bytes, uninitialized */
```

.include "*filename*"

Insert the contents of *filename* into the current source file. This is exactly the same as C's use of #include, and it allows you to include header files full of various definitions. There should be **NO .end** in the included file since the GNU Assembler will stop reading everything after that directive!

.arm

Set the processor to ARM state – 32-bits machine.

.thumb

Set the processor to THUMB state – 16-bits machine.

3.6 Expression

The operand required by the instruction or directive can be defined by an expression. The variables used in the expression can be labels.

```
.equ    x, 23
.equ    y, 10
.equ    z, x+y
.text
```

```
_start:                                /* program entry point */
        mov    r2, #(x+y)
        mov    r3, #(y+z)
.end
```

3.7 A General program structure

```
.global _start
.equ    var1 = constant/expression
.equ    var2 = constant/expression
.
.
.

.text
_start:                                /* program entry point */
    program instructions
    .
    .
    .

.data                                    /* data area */
res:
.byte   0xFF, 0x78

.align                                     /* align the address for word */
                                     /* size data */
.word   0x1231abcd
.word   0x976af
.
.
.skip   128                               /* reserve 128 bytes of memory */
.
.

.end
```