Assembly Language Warm up

Addressing Modes

The 3 addressing modes used by the TMS320LF2407 instruction set are -

Immediate Addressing Mode

Direct Addressing Mode

Indirect Addressing Mode

Immediate Addressing Mode

In this mode, the instruction word contains a constant to be manipulated by the instruction. There are 2 types of immediate addressing mode:

Short immediate addressing - Instructions that use this mode take an 8-bit, 9-bit or 13-bit constant as an operand. These instructions require a single word with the constant embedded in that word.

Example 2.1:

```
LACC #99        ; Load the number 99 into the accumulator
```

Long immediate addressing - Instructions that use long-immediate addressing take a 16-bit constant as an operand and require two instruction words. The constant is sent in the second instruction word. This 16-bit value can be used as an absolute constant or as a 2's complement value.

Example 2.2:

```
ADD #16384,2    ; Shift the value 16384 left by two bits
                ; and add the result to the accumulator
```

Direct Addressing Mode

In this mode, the data memory is addressed in blocks of 128 words called data pages. Thus the entire 64K of data memory can be addressed by 512 pages, which are labeled from 0 to 511 (000000000b to 111111111b). The value in the 9-bit data page pointer

(DP) in the status register ST0 determines the current data page. The particular being referenced within a page is determined by a 7-bit offset, which is specified by the seven LSBs of the instruction register.

When using the direct addressing mode, the steps to be followed are-

1. *Set the data page* - Load the appropriate value between 0-511 in the DP register using the LDP instruction. For example, to set the current data page to 2 i.e. addresses 0100h-017Fh the following instruction should be used -

```
LDP   #2        ; Initialize data page pointer
```

2. *Specify the Offset* - Supply the 7-bit offset as an operand of the instruction. For example, if you want to use the ADD instruction for the second value in the current page, the command is

```
ADD   1h        ; Add to accumulator the value in the current
                ; data page, offset of 1
```

Indirect Addressing Mode

As mentioned earlier, the eight auxiliary registers (AR0-AR7) are employed for indirect addressing. The address of the operand is contained in the currently selected auxiliary register. A specific auxiliary register is selected by loading a 3-bit value in the auxiliary register pointer (ARP) of the status register ST0. The register pointed to by the ARP is referred to as the *current auxiliary register* or the *current AR*. The data address (i.e. contents of AR) is passed either to the data-read bus or data-write bus by the ARAU depending on the instruction. The ARAU performs arithmetic operation on the contents of the AR during the decode phase depending upon the mode of addressing used in the instruction.

There are seven indirect addressing modes:

Operand	Option	Example
*	No increment or decrement	**LT*** loads the temporary register (TREG) with the contents of the data memory

		address referenced by the current AR.
*+	Increment by 1 (Auto-increment)	**LT*+** loads the temporary register (TREG) with the contents of the data memory address referenced by the current AR and then adds 1 to the contents of the current AR.
*-	Decrement by 1 (Auto-decrement)	**LT*-** loads the temporary register (TREG) with the contents of the data memory address referenced by the current AR and then subtracts 1 to the contents of the current AR.
*0+	Increment by index amount (Post-indexing by adding contents of AR0)	**LT*0+** loads the temporary register (TREG) with the contents of the data memory address referenced by the current AR and then adds the contents of AR0 to the contents of the current AR
*0-	Decrement by index amount (Post-indexing by subtracting contents of AR0)	**LT*0-** loads the temporary register (TREG) with the contents of the data memory address referenced by the current AR and then subtracts the contents of AR0 from the contents of the current AR

***BRO+**	Increment by index amount, adding with reverse carry (used in FFTs)	**LT *BRO+** loads the temporary register (TREG) with the contents of the data memory address referenced by the current AR and then adds the content of AR0 to the content of the current AR, adding with reverse carry propagation
***BRO-**	Decrement by index amount, subtracting with reverse carry (used in FFTs)	**LT *BRO-** loads the temporary register (TREG) with the contents of the data memory address referenced by the current AR and then subtracts the content of AR0 to the content of the current AR, subtracting with reverse carry propagation

Many instructions also specify a value of next AR, in addition to the current AR. This is current AR after the instruction is complete. Then the APR is loaded with the value of next AR, the previous value is loaded into the auxiliary register pointer buffer (ARB).

Modifying the Auxiliary Register Content

The LAR, ADRK, SBRK and MAR are specialized instructions for changing the contents of an auxiliary register.

- The LAR instruction loads an AR.
- The ADRK instruction adds an immediate value to an AR; SBRK subtracts an immediate value.

- The MAR instruction can increment or decrement an AR value by 1 or by an index amount.

Assembly Language Instructions

Before we start with the instruction set, here are a few tips on how to use the instruction descriptions.

Syntax

The notations used in the syntax expressions are -

italic symbols	Italic symbols in an instruction syntax represent variables. Example : For the syntax **ADD** *dma* Any value can be used for *dma* such as `ADD DAT, ADD 21` etc.
boldface characters	Boldface characters in an instruction syntax must be typed as shown Example : For the syntax **ADD** *dma ,* **16** A variety of values may be used for *dma*, but **ADD** and **16** must be typed shown. `ADD 7h, 16` or `ADD X, 16`

[,x]	Operand x is optional
	Example : For the syntax
	ADD *dma*, [,*shift*]
	dma must be supplied as in the instruction:
	`ADD 7h`
	There is an option to provide a shift value as in the instruction:
	`ADD 7h, 5`

[,x1 [,x2]]	Operands x1 and x2 are optional. However, x2 cannot be included without including x1.

Example : For the syntax

ADD *ind*, [,*shift*[,**AR***n*]]

i*nd* has to be supplied as in the instruction

```
ADD *+
```

Including of *shift* in the instruction is optional.

```
ADD *+, 5
```

Once the *shift* is included, you have an option of including **AR***n* too

```
ADD *+, 5 ,AR1
```

\# The # is a prefix for constants used in immediate addressing.

Example :

RPT #15 causes the next instruction to be repeated 16 times

RPT 15 causes the next instruction to be repeated a number of times determined by the value in that memory location.

The instruction set summary is attached. As we progress through the various chapters, the relevant instructions will be discussed in detail.

The assembly language source files are translated into machine language COFF (common object file format) files. Apart from the various instructions discussed above, the source files also contain assembler directives, which control various aspects of the assembly process such as source listing format, data alignment listing and section content. A source statement can contain four ordered fields and has the general syntax as follows ;

[*label*] [:] *mnemonic* [*operand list*] [;*comment*]

Example:

```
SYM1      .set      2         ;Set SYM1 = 2
Start:    LDPK      SYM1      ;Load DP with 2
```

Label Field : A label can contain up to 32 alpha-numeric characters, should not begin with a number and is case-sensitive. The value of a label is the current value of the section program counter. The section program counter (SPC) represents the current address within a section of code or data. Thus in the example, `Start` points to the instruction `LDPK   SYM1`.

Mnemonic Field: The mnemonic field can contain machine instructions (LDPK in example above) or assembler directives (.set in example above). This field should not start in column 1 or it will be interpreted as a label.

Operand Field: This is the list of operands that follow the mnemonic field. An operand can be a constant, a symbol or a combination of the two depending on the mnemonic preceding it.

Comment Field: A comment can begin in any column and extends to the end of the source line. A comment can contain any ASCII character. Comments are printed in the assembly source listing, but do not affect the assembly.

Assembler Directives

A summary of the various assembler directives is attached. The most commonly used directives will be discussed in this chapter.

Directives that define sections

The smallest unit of an object file is called a section. A section is a block of data or code that occupies a contiguous space in the memory map. COFF files have three default sections:

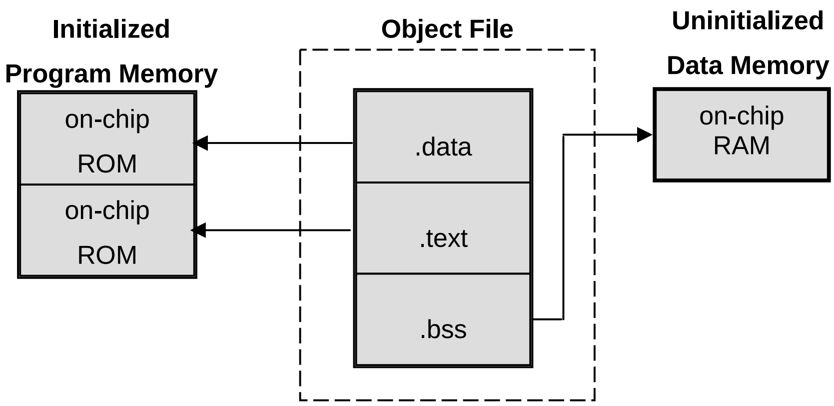
.text section	usually contains executable code
.data section	usually contains initialized data
.bss section	usually reserves space for uninitialized variables

There are 2 basic types of sections-

Initialized sections	contain data or code. .text and .data sections and named section created with the .sect directive lie in this category.

Uninitialized sections reserve space in the memory map for uninitialized data. .bss sections and named sections created with the .usect directive lie in this category

Since all sections are independently relocatable, they enable efficient use of the target memory since any section can be placed in any allocated block of target memory. This partitioning of memory into logical blocks is illustrated in the figure below.



The assembler has 6 directives for handling sections-

- .bss
- .usect
- .text
- .data
- .sect
- .asect

The .bss and. usect create uninitialized sections while the others create initialized sections. If no directive is used, the assembler assembles everything into the .text directive.

Uninitialized sections-

These reserve space in the RAM, which the program can use at runtime for creating and storing variables. The syntax for the relevant directives is:

.bss *symbol, size in words* [*blocking flag*] *symbol* **.usect** "*section name*", *size in words,* [*blocking flag*]

symbol This corresponds to the name of the variable for which the space is being reserved. It can be referenced by any other section and can also be declared as global.

size in words It is an absolute value which determines the words to be reserved in the section.

blocking flag This is an optional parameter. If a value greater than 0 is specified, the assembler associates size words contiguously; the allocated space will not cross a page boundary, unless size is greater than a page in which case the object will start on a page boundary.

section name This is a 8 character name that tells the assembler what named section to reserve space in. A named section is created by the user and can be used like the default .test, .data and .bss sections except that they are assembled separately.

Initialized sections-

These contain executable code or initialized data. The contents of these sections are stored in the object file and placed in the device memory where the program is stored. The syntax for the relevant directives is:

.text

.data

.sect "*section name*"

.asect "*section name*" , *address*

When the assembler encounters any of these directives, it stops assembling in the current section and assembles the subsequent code into the designated section until it again encounters any of the above 4 directives. It is important to note here that the .bss and the

.usect directives do not end the current section or begin a new one. They merely reserve the specified amount of space and the assembler resumes assembly of code or data in the current section.

Example:

```
     .text
     .word     1,2          ; Initialize words with values 1
                            ; and 2 in the .text section
     .sect     "sect1"
     .word     3,4          ; Initialize words with values 3
                            ; 4 in the named section sect1.
     .data
     .word     5,6          ; Initialize words with values 1
                            ; and 2 in the .data section
     .bss     sym,20        ;Reserve 20 words in .bss
     .word     7,8          ;Initialize words with values 7
                            ;and 8 in the .data section
     .text                  ;Resume assembly in .text sect
usym .usect    "sect2", 25  ; Reserve 20 words in named
                            ; section sect2
     .word     9,10         ; Initialize words with values 9
                            ; and 10 in the .text section
```

Directives that initialize constants

The various directives that assemble values for the current section are as follows:

.word places one or more consecutive 16-bit values into words of the current section

.int same as .word

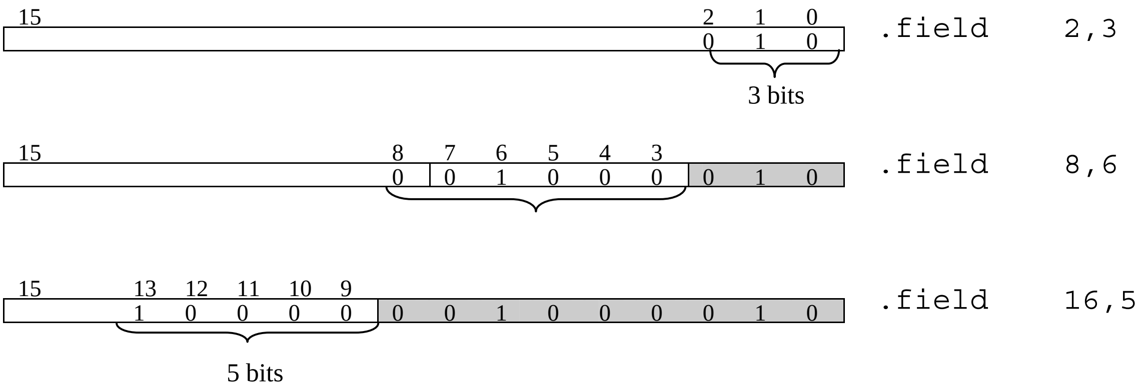
.byte places one or more consecutive 8-bit values into words of the current section

.string similar to .byte, except that two characters are packed into each word.

.field places a specified value into a specified number of bits in the current word.

The assembler does not increment the SPC until the entire word is filled.

Example:

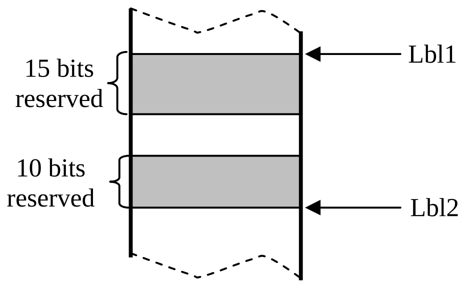


.space reserves a specified no. of bits in the current section(i.e. fills them with 0s) When a label is used with this directive, it points to the first word that contains the reserved bits.

.bes reserves a specified no. of bits in the current section(i.e. fills them with 0s) When a label is used with this directive, it points to the last word that contains the reserved bits.

Example:

```
Lbl1:     .space    15
          .word     20
Lbl2:     .bes      10
          .word     30
```



.float calculates the single-precision 32-bit ieee floating-point representation of a single floating-point value and stores it n two consecutive words in the current section

.bfloat same as .float except that it guarantees the object will not span a page boundary.

.long places 32-bit values into consecutive two-word blocks in the current section current.

.blong same as .long except that it guarantees that the object will not span the page boundary

Directives that align the section program counter

.align Aligns the SPC at a 128-word boundary thus ensuring that the code following this directive begins on a new page boundary.

.even Aligns the SPC so that it points to the next full word. This can be used after a .field directive. If the .field directive does not fill the word, the .even directive fills the unused bits with 0s.

Conditional Assembly Directives

The **.if/.elseif/.else/.endif** directives tell the assembler to conditionally assemble a block of code.

.if *expression* marks the beginning of a conditional block and assembles code if the .if condition is true

.elseif *expression* marks a block of code to be assembled if the .if condition is false and the .elseif condition is true

.else marks the block of code to be executed to be assembled if the .if condition is false

.endif marks the end of a conditional block and terminates the block

Example:

```
var1       .set 2
```

```
var2        .set 3
var3        .set 6
lbl_if :    .if  var3 = var1 * var2  ;Set the value equal
                                     ;to var1*var2
            .byte  var3
            .else
            .byte  var1*var2
            .endif
```

The **.loop/.break/.endloop** directives make the assembler repeatedly assemble a block of code according to the evaluation of a particular expression.

.loop *expression* marks the beginning of a repeatable block of code

.break *expression* tells the assembler to repeatedly assemble the block of code if the *expression* is false and to jump to the code immediately after the .endloop in case the *expression* is false.

.endloop marks the end of a repeatable block.

Example:

```
            .eval     0,x       ; Set x=0. Initialize count
loop1:      .loop
            .word     x*100     ; store x*100 at the current
                                ; location
            .eval     x+1,x     ; Increment x i.e count
            .break    x=6       ; If x=6 quit else goto loop1
            .endloop            ; the word has a value 500
                                ;when program quits the loop
```

Miscellaneous Directives

.asg assigns a character string to a substitution symbol. The value is stored in the substitution symbol table so that whenever the assembler encounters this symbol, it substitute it with the character string. Substitution symbols can be redefined

Example:

```
        .asg "1, 2, 3, 4, 5", char_sym
```

.set sets a constant value to a symbol and cannot be redefined.

Example:

```
bval    .set 0020h
```

.equ same as .set

.global a symbol defined as global in a current module allows it to be accessed from an external module. If the symbol is not defined in an external module, then the current module can access it by defining it as global.

.end it is optional and terminates assembly. It should be the last source statement of a program.