MetaWare Assembling and Linking Essentials

Assembling

An assembler translates assembly language statements or *source* into a file of binary coded machine instructions and data. The translation process generally consists of two parts: 1) identify and associate labels with memory locations and 2) translate assembly instructions into numeric or machine codes. The resultant binary file is generally referred to as an *object* file.

Assembler Directives

Assembler directives direct the assembler during the translations process to perform such operations as:

- Section identification and specification
- Data storage and allocation
- Symbol declaration

Section Identification and Specification

At the most fundamental level, a program is located in two distinct parts of memory: 1) the program or executable portion and 2) the data storage portion. As an assembly programmer, you must distinguish these parts so that the assembler can distinctly locate them in memory. This is accomplished with the *section* assembly directive. The syntax of the directive follows:

.section name, class

Notice that the *section* directive starts with a period. The assembler distinguishes all directives with a leading period. *Name* is arbitrary and identifies the section. Class specifies or determines what kind of section proceeds. There are several classes, but we will be concerned with just two: *text* and *data*.

Text identifies the following section as executable. This section is used for your assembly program. For example, to declare an assembly-coded program section, my_program, you would use the following section directive.

```
.section my_program, text

assembly line 1
assembly line 2
etc
```

An abbreviated section directive for a text section is:

.text

It is equivalent to:

.section text, text

Notice the name of the section is ‘text’.

Data identifies the following section as data storage. This is where you will allocate and initialize memory for arrays, buffers and variables. For example, to declare a data storage section, my_data, use the following section directive.

.section my_data, data

memory allocation and initialization statements

An abbreviated section directive for data is:

.data

It is equivalent to:

.section data, data

Notice the name of the section is ‘data’.

Data Storage and Allocation

There are several assembler directives for allocating and initializing memory in the data section. We will use the following:

.word
.half
.byte
.skip
.align
.asciiz

Word, *half* and *byte* directives allocate and initialize word, half word and byte memory locations. The directive is followed by one or more values to be initialized. For example,

.word 1, 2, 3, 4, 5

initializes 5 consecutive word locations to 1 through 5.

The *skip n* directive will allocate and zero initializes a block of n bytes. For example, the statement:

.skip 100

initializes the a block of 100 consecutive bytes to zero.

These data storage directives are usually used in conjunction with a *label*. A label is not an assembler directive. It is used to associate a memory location with a symbol or label. A label takes the form:

Label:

For example, to allocate a buffer of 20 words called inbuf you would use the following statement:

inbuf: .skip 80

The align directive is used to set the current location to a half or word boundary. The align directive has the syntax:

. align n

where n refers to a 2^n location boundary. This is useful if you are allocating consecutive half or word memory locations. The MPC accesses words or half words more efficiently when they occur on a half and word boundaries. Allocating on an appropriate boundary improves access efficiency.

The asciiz directive is used to allocate and initialize memory locations to corresponding ASCII codes.

. asciz string

where string represents text bounded by quotation marks. For example, '123' represents the text string 123. In addition, the asciiz directive appends a null code or zero to the string. For example, the directive

.asciz '123'

will load the following values in consecutive memory locations.

0x31, 0x32, 0x33, 0x00

Symbol Declaration

There are several directives associated with symbol declaration. We are primarily concerned with:

.global
.equ

The *global* assembler directive exports or lists a symbol for external access. For example, if you have a variable named 'done_flag' that you want accessible by another assembled file, you need to declare the label to that memory location as global. The following statements allocate and initizalize space for the variable 'done_flag' and declare the label global.

```
Done_flag .word 0

.global done_flag
```

Note that the symbol *_Start* must be globally declared to determine a starting point when the linker generates an executable object.

The *equ* directive is used to associate a value with a name. The directive takes the form:

.equ name value

This particularly useful when assigning constant values making your code readable and reducing the possibility of errors. For example,

```
.equ size 10
```

associates the decimal value 10 to the name 'size'.

Additional Assembler Directives

This document focuses on essential assembler directives required to do the 373 labs. There are many more assembler directives at your disposal. Several reference copies are provided in the lab listing all the assembler directives.

Linking

Although the assembler generates and locates a binary representation of the data and assembly instructions, the code is note ready for execution. The binary file generated by the assembler represents the relative location of the data and program space to one another. The locations have yet to be explicitly associated with actual memory locations.

The MPC823 provides memory from 0x0000 0000 to 0x003F FFFF. The space from 0x0000 0000 to 0x0001 0000 is used by the system and a SingleStep monitor that communicates with the SingleStep Debugger running on your PC. We generally use the space from 0x0001 0000 to 0x003F FFFF. The linker file file-link.txt file contains linker commands to associate the text and data sections with specific memory addresses. Specifically,

```
        SECTIONS {

  .text ADDRESS 0x10000:

  .data ADDRESS 0x11000:

}
```

The text or executable code will start at location 0x0001 0000 and the data will start at 0x0001 1000.

Example

The following is an example of a data section and the corresponding memory contents.

	.data	memory location	contents
flag1:	.byte 0	0x11000	0x00
		0x11001	unknown
	.align 1		
flag2:	.byte 1	0x11002	0x01
		0x11003	unknown
	.align 2		
array1:	.word 2, 3456	0x11004	0x00
		0x11005	0x00
		0x11006	0x00
		0x11007	0x02
		0x11008	0x03
		0x11009	0x04
		0x1100A	0x05
		0x1100B	0x06
msg:	asciiz ‘123’	0x1100C	0x31
		0x1100D	0x32
		0x1100E	0x33
		0x1100F	0x00

The following example is identical to the previous example with alignment directives removed.

	.data	memory location	contents
flag1:	.byte 0	0x11000	0x00
flag2:	.byte 1	0x11001	0x01
array1:	.word 2, 3456	0x11002	0x00
		0x11003	0x00
		0x11004	0x00
		0x11005	0x02
		0x11006	0x03
		0x11007	0x04
		0x11008	0x05
		0x11009	0x06
msg:	asciiz ‘123’	0x1100A	0x31
		0x1100B	0x32
		0x1100C	0x33
		0x1100D	0x00

Notice that flag2 in no longer located on a half word boundary and the array1 is no longer located on a word boundary.