

## 4 ASSEMBLY LANGUAGE SYNTAX

### 4.1 General

This chapter describes the syntax for the assembly language used by the assembler, which is derived from the GNU assembler. The assembler generates object files in a variant of the a.out format. For topics that are not covered here, please see the GNU assembler manual.

### 4.2 Definitions

Throughout this chapter, "whitespace" means any number and combinations of spaces (ASCII 32) and tabs (ASCII 9).

A simple form of syntax-describing notation will be used:

Any item written without surrounding { } (braces) or < > (brackets) must be written exactly as it stands.

Case is irrelevant when writing instructions.

An item enclosed in < > (brackets) does not have its literal meaning, which is defined elsewhere. For example,

```
MOVE.<size modifier>
```

<size modifier> is described elsewhere, and may be one of B, W, D.

In some instances, the item may be followed by a number as in <operand1>. This means that there are several operands, numbered incrementally, but that there is only one definition of <operand>. An operand, in general, may in this context be specified as <operandn>.

An item enclosed in { } (braces) is optional and may be left out:

```
{<label> :}
```

Indicates that a label is optional. Please note, however, that a label must be followed by a : (colon).

The symbol ... (three periods) indicates that any number of the previous item may follow. For example:

```
{<operand1> {,<operand2> {,...}}}
```

means that any number of <operands> are valid.

A range of characters is indicated by using .. (two periods) inside { } (braces):

```
R{0..15}
```

indicates R0, R1, ... R15

The symbol := (colon, equal sign) indicates a definition:

```
<reg> := R{0..15}
```

The symbol | ("or") indicates that only one of the items may follow:

```
<size modifier> := B | W | D
```

Size modifier may be one of B, W, D.

In many cases, where it is easier to write a description in plain English, this will be done instead.

### 4.3 Files, Lines and Fields

An assembly program may be made up of several files. The assembler assembles each file separately. The linker, derived from the GNU ld, resolves relocations and cross-references, and produces an executable file in a variant of the a.out object format.

Each file may contain zero or more lines of assembly code. Each line consists of a number of characters, followed by a line-feed character (ASCII LF, 0x0a).

Each line of assembly code is made up of several fields. There may be up to four fields on a line: The **label field**, the **opcode field**, the **operands field**, and the **comment field**.

```
{<label>:}{ <opcode>{ <operand1>{,<operand2>{, ...}}}{;<comment>}
```

The **label field** starts in the first column. The label is comprised of symbol characters (as described in section 4.4 *Labels and Symbols*), and ends with a : (colon).

The **opcode field** is exactly one opcode or assembler directive such as MOVE.D or .BYTE. An opcode must be preceded by at least one white space character.

The **operands field** may contain any number of operands separated by commas, and there may be whitespace on either side of the commas. The first operand must be preceded by at least one whitespace character.

The comment field starts with a ; (semi-colon), and ends at the end of the line.

The symbol # (hash) is a special prefix character used as a semi-directive such as #APP and #NO\_APP and line number specification.

### 4.4 Labels and Symbols

A symbol is a set of characters associated with a value, which may be a number or a location counter. A label is a symbol. The value of symbols other than labels may be set using the .SET directive.

```
<label> := <symbol>
```

A symbol is made up of any number of the characters: {0..9} {A..Z} {a..z} . \$ \_ (i.e. a period, dollar sign, or underline space). However, the first character of a symbol may not be a digit (i.e. {0..9}).

It is recommended that symbols that start with the letter 'r' or 'R', followed by a number in the range from {0 ... 15} be avoided, as well as the mnemonic names and register numbers of the special registers (see section 1.1 *Registers*) since they may be interpreted as a register.

Symbols are case sensitive. All characters are significant.

## 4.5 Opcodes

An opcode has the form:

```
<opcode> := <op>{.<size_modifier>}
```

where <op> is one of the instructions described in chapter 1 "Instruction Set Description", and <size\_modifier> := B | W | D

The size modifier indicates whether the operation should be performed as a byte, word or dword operation where a byte is 8 bits, a word is 16 bits, and a dword is 32 bits in length.

Note that only operations which support variable size have the size modifier, and that in this case it is mandatory. On the other hand, the size modifier must not be used for operations that do not support variable size.

The opcode field is not case sensitive. For example, the no-operation instruction may be written "NOP" or "nop" or even "noP".

In some cases, the assembler may have aliases for opcodes meaning that two syntactically different assembly statements may produce the exact same code. For instance, the Branch on Lower (BLO) instruction is implemented as Branch on Carry Set and has, therefore, the acronym (BCS).

Also, although the CRIS has no explicit PUSH or POP instructions, the assembler provides these mnemonics as alternatives for the instructions that perform these operations. For example:

```
PUSH      Rn == MOVE.D Rn, [SP=SP-4]
POP       Rn == MOVE.D [SP+], Rn
```

## 4.6 Operands

### 4.6.1 General

The following syntax applies:

```
<operand> := <addressing_mode> | <expression>
```

<expression> is defined in the GAS manual and will only be outlined here.

<addressing\_mode> is described in section 4.7 *Addressing Modes*.

Register names are not case-sensitive.

### 4.6.2 Expressions

The expression syntax is the same as defined by the GAS, except that some simplifications are in order.

Expression evaluation can only handle integers. The compiler uses integer constants for the bit patterns of floating point numbers as given in the IEEE 754 standard for 32 and 64 bit representation.

White space is allowed in expressions but not in constants or symbols.

All expression evaluation takes place at full precision (32 bits); in other words, there are no different data types (word, byte, etc.). If the result of an expression is too large

for the selected mode, (e.g. `MOVE.B 0xAB3, R0`), it is an error which will be indicated by the assembler. If it is smaller than the indicated size, it will be padded with zeroes.

One must be careful when performing operations on symbols belonging to different segments since the absolute address of the segments is not known at assembly time. Normally, expressions are used to provide the difference between a jump table and its destination (offsets into structs etc.). Expressions involving more than one segment, and which can not be reduced to only one segment at assembly time, are not allowed.

### 4.6.2.1 Expression operands

The following expression operands are supported:

Name	Comment
<hexadecimal_constant>	
<decimal_constant>	
<octal_constant>	
<symbol>	
.	Current location counter
'<character_constant>	

*Table 4-1 Supported operands*

<hexadecimal\_constants> are hexadecimal numbers prefixed with 0x or 0X (i.e. `0xFF80 = 65408`). Either upper or lower case may be used. <octal\_constants> are octal numbers prefixed with 0 (zero) (i.e. `017 = 15`). <decimal\_constants> begin with {1..9}. `5633` is a valid <decimal\_constant>; `083` is not. <symbols> have already been described in section 4.4 *Labels and Symbols*.

```
<character_constant> :=    \{any_printable_ascii_car} |  
                          \\<special_char>
```

<any\_printable\_ascii\_char> is an ASCII character in the range from 33 to 126 (0x21 to 0x7E). The complete list of <special\_char> is:

```
\t (HT), \n (LF), \r (CR), \b (BS), \f (FF), \' ('), \" (")
```

The following are examples of legal <character\_constants>:

```
'a      'A      '%'      '3      '\t      '\n
```

Any character backslashed that is not a special\_char, is treated "as itself" (i.e. \y == y).

Neither <hexadecimal\_constants> nor <octal\_constants> are supported as <character\_constants>.

#### 4.6.2.2 Expression operations

The following binary operations are supported:

\*, /, %, +, - (times, divide, remainder, plus, minus)

&, |, ^ (bitwise and, or, xor)

<<, >> (shift left and right)

The following unary are supported:

- (minus)

~ (logical (bitwise) not)

#### 4.6.2.3 String expressions

A string expression is a special type of expression which may only appear in an .ASCII directive. It has the following form.

```
<string> := "{<any_char1>{<any_char2>{...}}}"
```

where:

```
<any_char> := <any_printable_ascii_char> | \<octal_constant> |
<special_char> | \"
```

Thus, a string expression is made up of zero or more characters. Every character is similar to the character\_constant described above, with the addition that \" means the quote character. For example:

```
"This is a\040string with a \"newline\" at the end\n"
```

## 4.7 Addressing Modes

In order to describe what actually happens in each description below, a form of pseudo-code which is very similar to C is used.

<size\_modifier> refers to the size modifier of the opcode:

```
<reg> := R{0..15} | PC | SP
```

where PC is R15 and SP is R14.

There is also a series of special registers used for such things as storing the return address from a subroutine, etc. However, since these registers can be explicitly referred to only in special MOVE instructions, and then only in the Register addressing mode, they will not be dealt with here.

Mode: Immediate

Written as <expression>

Example: 34404

Explanation: 34404;

Mode: Quick immediate

Written as: <expression>

Example: 12

Explanation: 12;

Mode: Absolute

Written as: [<expression>]

Example: [34404];

Explanation: \*(size\_modifier\*) 34404;

Mode: Register

Written as: <reg>

Example: R5

Explanation: r5;

Mode: Indirect

Written as: [<reg>]

Example: [R5]

Explanation: \* (size\_modifier \*) r5;

Mode: Autoincrement

Written as: [<reg>+]

Example: [R5+]

Explanation: \* (size\_modifier \*) r5++;

(Note: R5 is incremented by a value corresponding to the <size\_modifier> in the opcode.)

Mode: Indexed  
 Written as: [`<reg1>+<reg2>.<size_modifier2>`]  
 Example: [R5+R6.D]  
 Explanation: `*(size_modifier*) (r5 + (r6<<log2(<size_modifier2>)));`

(Note: The value of R6 is shifted one step left for .W and two steps left for .D)

Mode: Indexed with assign  
 Written as: [`<reg1>=<reg2>+<reg3>.<size_modifier2>`]  
 Example: [R4=R5+R6.D]  
 Explanation: `*(size_modifier*) (r4 = r5 + (r6<<log2(<size_modifier2>)));`

Mode: Immediate offset  
 Written as: [`<reg>+<expression>`]  
 Example: [R5 + TABLE]  
 Explanation: `*(r5 + TABLE);`

Mode: Indirect offset  
 Written as: [`<reg1>+[<reg2>].<size_modifier2>`]  
 Example: [R5 + [R6].D]  
 Explanation: `*(r5 + *(size_modifier2*) r6);`

Mode: Autoincrement offset  
 Written as: [`<reg1>+[<reg2>+].<size_modifier2>`]  
 Example: [R5 + [R6+].D]  
 Explanation: `*(r5 + *(size_modifier2*) r6++);`

Mode: Immediate offset with assign  
 Written as: [`<reg1>=<reg2>+<expression>`]  
 Example: [R4 = R5 + TABLE]  
 Explanation: `* (r4 = r5 + TABLE);`

Mode: Indirect offset with assign  
 Written as: [`<reg1>=<reg2>+[<reg3>].<size_modifier2>`]  
 Example: [R4 = R5 + [R6].D]  
 Explanation: `*(r4 = r5 + *(size_modifier2*) r6);`

**Mode:** Autoincrement offset with assign  
**Written as:** [`<reg1>=<reg2>+<reg3>+].<size_modifier2>`]  
**Example:** [`R4 = R5 + [R6+].D`]  
**Explanation:** `*(r4 = r5 + *(size_modifier2*) r6++);`

**Mode:** Double indirect  
**Written as:** [`<reg>`]  
**Example:** [`R5`]  
**Explanation:** `*(size_modifier*) (*(dword*) r5);`

**Mode:** Double indirect with autoincrement  
**Written as:** [`<reg>+`]  
**Example:** [`R5+`]  
**Explanation:** `*(size_modifier*) (*(dword*) r5++);`

**Note 1:** The difference between the Quick immediate addressing mode and the Immediate addressing mode is that the Quick immediate mode is valid only for certain instructions (such as ADDQ) where one of the operands is a small integer. The range of values for this mode varies according to the instruction. Immediate values, on the other hand, can be anything that fits in the size indicated by the instruction.

**Note 2:** The assembler implements the Immediate and Absolute modes in the following ways:

- The Immediate mode is actually the Autoincrement mode using PC
- The Absolute mode is actually the Double indirect with autoincrement mode using PC

**Note 3:** The double Indirect (with or without autoincrement), Offset (with or without assign), Indexed (with or without assign) and Absolute addressing modes are implemented using special addressing mode prefixes.

## 4.8 Assembler Directives

### 4.8.1 Directives controlling the storage of values

```
.BYTE <expression1> {, <expression2> {, ...}}
```

**Example:**

```
.BYTE 0x41, 0x42, 0x43, 0x38, 0x30
```

Insert a byte at the current location, incrementing the location counter by one. Repeat this until the list of expressions has been exhausted.

```
.WORD <expression1> {, <expression2> {, ...}}
```

**Example:**

```
.WORD 34404, 0x2040
```

Insert a word at the current location, incrementing the location counter by two.  
Repeat this until the list of expressions has been exhausted.

```
.DWORD <expression1> {, <expression2> {, ...}}
```

Example:

```
.DWORD 0xbf96a739
```

Insert a dword at the current location, incrementing the location counter by four.  
Repeat this until the list of expressions has been exhausted.

```
.ASCII <string1> {, <string2> {, ...}}
```

Example:

```
.ASCII "Megatroid\n", "AX-Foo\r\n"
```

Insert a string of ASCII characters, and increment the location counter by the size of the string. Repeat this until the list of strings has been exhausted.

### 4.8.2 Directives controlling storage allocation

The assembler only supports text data and bss segments. In fact, values can not be stored in the bss segment either, per definition; however, space can be reserved in this segment.

```
.TEXT
```

Select the text location counter (used for the program text).

```
.DATA
```

Select the data location counter (used for initialized data).

```
.BSS
```

Select the bss location counter (used for uninitialized data).

```
.ORG <expression>
```

Example:

```
.ORG 0
```

Set the current location counter to <expression>.

```
.LCOMM <symbol>, <expression>
```

Example:

```
.LCOMM _screen_width, 2
```

Reserve the indicated number of bytes in the bss segment, and assign the indicated symbol to the start of the area. This is used by the GCC compiler when a default-zero initialized variable is defined. The location counter is increased by <expression>. Note that symbols defined by .LCOMM are default local and need a .GLOBAL directive to be available for other files.

```
.SPACE <expression1>, <expression2>
```

Example:

```
.SPACE 10, '\r'
```

Put the number of bytes indicated by the first expression into the current segment. Each byte has the value indicated by the second expression. The location counter is advanced by one for each byte inserted. The example above puts 10 carriage returns at the current location.

```
.ALIGN <expression>
```

Example:

```
.ALIGN 1
```

Align the location counter so that the <expression> least significant bits of the location counter are zero, or to put it another way, so that the location counter is an even multiple of  $2^{**}<expression>$ . If the location counter is already aligned, nothing happens, otherwise it is incremented until it is aligned.

- Note 1:** In the example .ALIGN 1 above, the location counter is to be word aligned.
- Note 2:** Program code in the text segment must always be word aligned. This means that after data has been inserted into the text segment that might result in an odd number of bytes, such as the result of a .BYTE or .ASCII directive, an .ALIGN 1 should be performed before the next instruction. However, note that data itself may start at odd or even addresses in the text segment.
- Note 3:** Since segment alignment is only guaranteed to word in between files, an <expression> value of two or greater might not have the intended effect in cases where, for example, earlier positioned object files for linking have been word but not dword aligned. So, it is up to the user to make sure files are compiled with '-m32bit' or an other similar option to guarantee the desired alignment.

### 4.8.3 Symbol handling

```
.GLOBAL <symbol>
```

Example:

```
.GLOBAL _start_gate
```

Make the <symbol> available to other modules. Used for global functions and variables.

```
.SET <symbol>, <value>
```

Example:

```
.SET ACIA_DATA, 0x80003a
```

Give the <symbol> a value. Note that writing

```
LABEL:
```

on a line is equivalent to writing

```
.SET LABEL, .
```

A symbol assigned a value by the .SET directive may be changed at any time. (The value of a label may not be changed, however).

### 4.9 Alignment

Program code must always be word aligned. However, it is up to the programmer to ensure that this is done by performing .ALIGN 1 before code that may potentially end up on an odd address. This could happen after a .BYTE, .ASCII, or .SPACE directive.

