

## **Section 1**

# **ZiLOG Z8 Macro Assembler User's Guide**

A  
s  
s  
e  
m  
b  
l  
e  
r

---

# Execution

# Chapter 1

---

## 1.1 Runtime Interface

The assembler is invoked with the following command line:

**Z8ASM [optionlist] <filename>**

The above command line can be entered with any mixture of lower or upper case characters. The options may be specified in any order, but must precede the file to be assembled. If an extension is not specified on the <filename>, then **.ASM** is assumed.

The assembler can be terminated at any point by typing:

**<CNTRL><BREAK>**

or

**<CNTRL><C>**

at the keyboard. The assembler exits with a status of less than 0 if one or more errors were detected and greater than 0 if any warnings were detected; otherwise it exits with a 0.

## 1.2 Options

The following section describes the options allowed by the assembler. All options must be preceded by a - or /. Options may be abbreviated but the abbreviation must be unique.

### 1.2.1 DEBUG

Create <filename>.**DEB**, which contains debug information for the symbolic debugger. The default setting is **NODEBUG**.

### 1.2.2 GENOBJ

Generate an object file with the **.OBJ** extension. This is the default setting.

### 1.2.3 IGCASE

Suppress case sensitivity of user-defined symbols. When this option is used, the assembler converts all symbols to uppercase. This is the default setting.

### 1.2.4 LIST

Generate an output listing with the **.LST** extension. This is the default setting.

### 1.2.5 LISTMAC

Expand macros in the output listing. This is the default setting.

### 1.2.6 NODEBUG

Do not create a debug information file for the symbolic debugger. This is the default setting.

### 1.2.7 NOGENOBJ

Do not generate an object file with the **.OBJ** extension. The default setting is **GENOBJ**.

### 1.2.8 NOIGCASE

Enable case sensitivity of user-defined symbols. The default setting is **IGCASE**.

### 1.2.9 NOLIST

Do not create a list file. The default setting is **LIST**.

### 1.2.10 NOLISTMAC

Do not expand macros in the output listing. The default setting is **LISTMAC**.

### 1.2.11 NOSDIOPT

Do not perform span-dependent optimizations. All size optimizable instructions use the largest instruction size. The default is **SDIOPT**.

### 1.2.12 NOWARNS

Suppress the generation of warning messages to the screen and listing file. A warning count is still maintained. The default is to generate warning messages.

### 1.2.13 PAGELENGTH

Set the new page length for the list file. The page length must immediately follow the = (with no space between). The default is 56.

Example:

**-PAGELENGTH=60**

### 1.2.14 PAGEWIDTH

Set the new page width for the list file. The page width must immediately follow the = (with no space between). The default and minimum page width is 80. The maximum page width is 132.

Example:

**-PAGEWIDTH=132**

### 1.2.15 QUIET

Suppress title information that is normally displayed to the screen. Errors and warnings will still be displayed. The default setting is to display title information.

### 1.2.16 RELIST

Generate an absolute listing by making use of information contained in a linker mapfile. This results in a listing that matches linker generated output.

Syntax:

**-RELIST:<mapfile>**

where **<mapfile>** is the name of the mapfile created by the linker.

Example:

**-RELIST:product.map**

### **1.2.17 SDIOPT**

Perform span-dependent optimizations. The smallest instruction size allowed will be selected for all size optimizable instructions. This is the default setting.

---

# Address Spaces and Segments

---

## Chapter 2

The memory regions of a microprocessor can be accessed at assembly by using proper segment directives. The memory representations created by using these directives are assigned at link time to specify memory areas, as dictated by appropriate linker commands. This section describes these memory representations.

## 2.1 Allocating Memory at Assembly-Time

### 2.1.1 Allocating Processor Memory

A microprocessor may access one or more memory regions to obtain data or code. Memory associated with these regions is allocated by absolute or relocatable representations.

- **Absolute representations**

An absolute memory representation is created by defining a segment as an absolute segment. This is accomplished by using the **ORG** clause within the **DEFINE** segment definition directive. All data and code for that segment will be allocated at the defined physical memory address.

- **Relocatable representations**

Relocatable representations of the physical memory regions are created by using the pre-defined assembler segments, and user defined segments that were not defined with the **ORG** clause. At link time, linker commands are used to specify where these logical representations are to be located. Relocatable representations can be assigned to different physical memory regions at link time without re-assembling.

## 2.1.2 Spaces

The memory regions for the microprocessor are represented by the following assembler spaces:

SPACE ID	MAU SIZE	MAX MAUS PER ADDRESS
ROM	8 bits	1(default)
RDATA	8 bits	1
XDATA	8 bits	1

Code and data is allocated to these spaces by using segments attached to the space.

## 2.1.3 Segments

As described above, segments are used to represent regions of physical memory. Only one segment is considered active at any time during the assembly process. A segment must be defined before setting it as the current segment. A segment is associated with one of the assembler spaces.

### 2.1.3.1 Pre-Defined Segments

For convenience, the following segments are pre-defined by the assembler:

Segment ID	Space	Align	Origin
near_bss	RDATA	1	Relocatable
far_bss	XDATA	1	Relocatable
near_data	RDATA	1	Relocatable
far_data	XDATA	1	Relocatable
code	ROM	1	Relocatable (default)
text	ROM	1	Relocatable

### 2.1.3.2 User defined segments

The **DEFINE** directive defines a new segment and attaches the segment to a space. A segment can also be defined with a boundary alignment and/or origin.

- **Alignment**

Aligning a segment tells the linker to place all modules with that segment identification on the specified boundary.



- **Origin**

When a segment is defined with an origin, the segment becomes an absolute segment: the linker places it at the specified physical address in memory.

### 2.1.3.3 Attaching code/data to segments

Code and data is attached to a segment by using the **SEGMENT** directive. The **SEGMENT** directive makes that segment the current segment, and any code or data following the directive will reside in the segment specified until another segment directive is encountered.

## 2.2 Assigning Memory at Link-Time

At link time, the linker groups together those sections of code and data that have the same segment identifier, and places the resulting segment in the address space attached to the segment. The linker handles relocatable segments and absolute segments differently:

- **Relocatable segments**

If a segment is relocatable, the linker decides where in the address space to place the segment.

- **Absolute segments**

If a segment was defined with an origin, the linker places the segment at the absolute address that the origin specifies.

Segments may be redefined at link time with the appropriate linker commands.

A  
s  
s  
e  
m  
b  
l  
e  
r

---

# Output Files

## Chapter 3

---

The assembler creates the following output files and names them with the root name of the source file:

- **<source>.LST** contains the source and object code the assembler generates. The assembler creates **<source>.LST** unless it is invoked with the **-NOLIST** option.
- **<source>.OBJ** is an object file in relocatable OMF695 format. The assembler creates **<source>.OBJ** unless it is invoked with the **-NOGENOBJ** option.
- **<source>.DEB** is a symbolic debug file that contains symbolic information for debugging. This file is created only when the assembler is invoked with the **-DEBUG** option.

**WARNING:** Do not use source input files with the above extensions. The assembler will not assemble files with these extensions, and the data contained in the files will be **LOST**.

## 3.1 Source Listing Output File

The comprehensive listing output file name is the source file name with the extension **.LST**. Assembly directives allow you to tailor the content and amount of output from the cross-assembler.

### 3.1.1 General Listing File

Each page of the list file contains

- a heading with the assembler version number,
- the source input filename,
- the page number, and
- the date and time of assembly.

Source lines in the list file are preceded by

- the include level,
- a plus sign (+) if the source line contains a macro,
- the line number,
- the location of the object code created, and
- the object code.

The include level starts at level A, and works its way down the alphabet to indicate nested includes.

The format and content of the **.LST** file can be controlled with commands included in the source file:

- **NEWPAGE** starts a new page in the list file.
- **TITLE** specifies a title for the list file.
- **NOLIST** stops output to the list file until the assembler encounters the **LIST** directive. **INCLUDE** files are included in the list file unless **NOLIST** is used within the include file.
- **LIST** resumes output to the list file.
- **MACLIST ON/OFF** turns the listing of macro expansions on and off.
- **CONDLIST ON/OFF** allows the listing of false blocks of conditional assembly to be controlled. If **OFF**, then false blocks are not included in the listing.

**ERROR/WARNING** messages, if any, follow the source line containing the error(s). A count of the errors and warnings detected is included at the end of the listing.

### 3.1.2 Absolute Listing File

The assembler also has the capability to generate an absolute listing file. This type of listing output reflects the actual addresses of a final product. In order to create such an output file, a linker map file must be supplied to the assembler.

The linker map file is a comprehensive output file that contains the location of spaces, segments, and symbols of a final product. The assembler makes use of this information in order to generate an absolute listing file.

An absolute listing can be generated by running the assembler with the **-relist** option.

Example:

### **Z8ASM -relist:product.map main.asm**

When an absolute listing is generated, debug and object files cannot be generated. In addition, if any errors are encountered, then the relist is aborted.

## **3.2 Object Code Output File**

The object code output filename is the source file name with the extension **OBJ**. This file contains the relocatable object code in OMF 695 format and is ready to be processed by the linker and librarian. Refer to IEEE STD 695 Object Module Format for a description of the object module.

## **3.3 Symbolic Debug Output File**

The symbolic debug output file is the source file name with the extension **DEB**. This file contains symbolic information that is used by the symbolic debugger.

A  
s  
s  
e  
m  
b  
l  
e  
r

---

# Source Language Structure

## Chapter 4

---

### 4.1 General Structure

#### 4.1.1 Input Line

Each line of input text consists of ASCII characters terminated by a carriage return (<EOL>). An input line cannot exceed 512 bytes. Blank lines are ignored.

A source line is composed of an optional label field, followed by an instruction field or a directive field. A source line may contain only a label field. Each field is discussed further in the following sections.

#### 4.1.2 Comments

A semicolon (;) terminates the scanning action and any text following the semicolon is treated as source code comments. A semicolon appearing as the first character causes the entire line to be treated as source code comments.

#### 4.1.3 Case Sensitivity

In the default mode, the assembler treats all symbols as case sensitive. Invoke the assembler with the **IGCASE** option to ignore the case of user defined identifiers. Assembler reserved words are not case sensitive.

#### 4.1.4 Label Field

If included, the label field must be the first item in a source line. The first character of a label can be a letter or an underscore (\_), a dollar sign (\$), or a question mark (?). Following characters can include letters, digits, underscores, dollar signs (\$), question marks (?), or pound signs (#). The label may be followed by a colon (:), which completes the label definition. A label may only be defined once.

Labels can be from 1 to 33 characters in length.

#### 4.1.5 Instruction Field

The instruction field contains one valid assembler instruction. The field is terminated by a semicolon or <EOL>. Separate the instruction from its operands by white space (spaces or tabs). The operands are separated by commas.

#### 4.1.6 Directive Field

The directive field contains one valid assembler directive. The field is terminated by a semicolon or <EOL>. Separate the directive from its operands with white space.

#### 4.1.7 Symbols and Variable Assembler Values

The following characters are available for symbols and variables:

A-Z	Uppercase letters
a-z	Lowercase letters
:	Colon (used to end a label definition)
_	Underscore
0-9	Numbers
\$	Dollar sign
?	Question mark
#	Pound sign

## 4.2 Assembler Rules

### 4.2.1 Reserved Words

The following list contains reserved words used by the assembler. These words cannot be used as symbol names. Reserved words are not case sensitive.

C, EQ, F, GE, GT, LE, LT, MI, NC, NE, NZ, NOV, OV, PL, UGE, UGT, ULE, ULT, Z, r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, rr0, rr2, rr4, rr6, rr8, rr10, rr12, rr14, FLAGS, IMR, IPR, IRQ, RP, SPH, SPH,SIO, P01M, P3M, P2M, PRE0, PRE1, T0, T1, TMR, P0, P1, P2, P3

Additionally, do not use the instruction mnemonics or assembler directives as symbol names.

### 4.2.2 Assembler Number Representation

Numbers are represented internally as signed 32-bit integers. Floating-point numbers are represented as double precision 64-bit numbers and are automat-



ically converted to single precision floating-point numbers if required. The cross-assembler will detect if an expression operand is too large for the intended field and generate an appropriate error message.

### 4.2.3 Decimal Numbers

Decimal numbers are signed 32-bit integers consisting of the characters 0-9 inclusive between -2147483648 and 2147483647. Positive numbers are indicated by the absence of a sign. Negative numbers are indicated by a (-) preceding the number.

Examples:

```
1234           ; decimal
-123456        ; negative decimal
```

### 4.2.4 Hexadecimal Numbers

Hexadecimal numbers are signed 32-bit integers beginning with the character \$ or ending with the 'h' or 'H' suffix, and consisting of the characters 0-9 and A-F. A hexadecimal number may have 1 to 8 characters. Positive numbers are indicated by the absence of a sign. Negative numbers are indicated by a (-) preceding the number.

Examples:

```
$ABCDEFFF      ; hexadecimal
-0FFFFh        ; negative hexadecimal
```

### 4.2.5 Binary Numbers

Binary numbers are signed 32-bit integers beginning with the character % or ending with the character 'b' or 'B' and consisting of the characters 0 and 1. A binary number may have 32 characters. Positive numbers are indicated by the absence of a sign. Negative numbers are indicated by a (-) preceding the number.

Examples:

```
%010100101    ; binary number
-0101b         ; negative binary number
```

### 4.2.6 Octal Numbers

Octal numbers are signed 32-bit integers ending with the character 'o' or 'O', and consisting of the characters 0-7. An octal number may have 1 to 11

characters. Positive numbers are indicated by the absence of a sign. Negative numbers are indicated by a (-) preceding the number.

Examples:

<b>1234o</b>	<b>; octal number</b>
<b>-1234o</b>	<b>; negative octal number</b>

## 4.2.7 Floating-Point Numbers

Floating-point numbers are 64-bit IEEE standard double precision values. Hexadecimal and decimal floating-point numbers are supported. Only the +, -, \*, and / operators are allowed in expressions involving floating-point values.

## 4.2.8 Character Constants

A single printable ASCII character enclosed by single quotes (') can be used to represent an ASCII value. This value may be used as an operand value.

Examples:

<b>'A'</b>	<b>; ASCII code of "A"</b>
<b>'3'</b>	<b>; ASCII code of "3"</b>

## 4.2.9 Fractional Numbers

Fractional numbers are signed values between  $-1 \leq x < 1$  or unsigned values between  $0 \leq x < 1$ , that are represented by 8, 16, or 32 bit integers. A fractional number begins with a 0, followed by a decimal point, followed by one or more decimal numbers. Positive numbers are indicated by the absence of a sign. Negative numbers are indicated by a (-) preceding the number.

Examples:

<b>0.1234</b>	<b>; positive fractional number</b>
<b>-0.1234</b>	<b>; negative fractional number</b>

## 4.2.10 Character Strings

Character strings consist of printable ASCII characters enclosed by double (") or single (') quotes. A double quote used within a string delimited by double quotes and a single quoted use within a string delimited by single quotes, must be preceded by a back-slash (\). A single quoted string consisting of a single character will be treated as a character constant. The assembler does not insert

null bytes (0's) at the end of a text string, and a character string cannot be used as an operand.

Examples:

```
DB  "STRING"      ; a string
DB  'STRING',0    ; C printable string
DB  "STRING\"S"   ; embedded quote
DB  'a','b','c'   ; character constants
```

A  
s  
s  
e  
m  
b  
l  
e  
r

---

# Expressions

## Chapter 5

---

In most cases, where a single integer or float value can be used as an operand, an expression can also be used. The assembler evaluates expressions in 32-bit signed arithmetic or 64-bit floating-point arithmetic. Logical expressions are bitwise operators.

Discussed below are the arithmetic, relational, and logical operators and the allowable operands. The assembler detects overflow and division by zero errors.

## 5.1 Expression Operators

### 5.1.1 Arithmetic operators

<<	Left Shift
>>	Arithmetic Right Shift
**	Exponentiation
*	Multiplication
/	Division
%	Modulus
+	Addition
-	Subtraction

Note: The modulus operator must have spaces before and after it to separate it from the rest of the expression.

### 5.1.2 Relational Operators

For use only in conditional assembly expressions.

==	Equal
!=	Not Equal
>	Greater Than
<	Less Than
>=	Greater Than or Equal
<=	Less Than or Equal

### 5.1.3 Boolean Operators

&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
~	Complement
!	Boolean NOT

The table below shows the operator precedence in descending order, with operators of equal precedence on the same line. Operators of equal precedence are evaluated left to right. Parentheses can be used to alter the order of evaluation.

Level 1	(	)					
Level 2	~	unary -	!	high	low		
Level 3	**	*	/	%			
Level 4	+	-	&		^	>>	<<
Level 5	<	>	<=	>=	==	!=	

## 5.2 Expression Operands

An expression operand is either a number, symbol, label, or another expression. Only the +, -, \*, and / operators can be used with floating-point numbers.

**NOTE:** When used as an operand, the contents of the location counter (\$) is the value of the location of the current instruction.

## 5.3 Expression Conversions

### Integer to float

The assembler implicitly performs all conversions from integer to float.

### Float to double precision float

If an expression contains a float value, the assembler converts all operands to double precision float values, and the expression result is of type double precision.

**Float to binary representation**

The **HIGH** and **LOW** keywords can be used to access the upper or lower part of a 64-bit float to be accessed as a 32-bit value or the low/high portion of an address. The result of a high or low operator is a 32-bit integer value.

**HIGH** and **LOW** can only be applied to the result of a floating-point expression or a floating-point number. A space must be used to separate the operator from a floating-point number.

Examples:

```
HIGH 1.23E45  
LOW 12.345  
LOW(12.345+67.89)
```

## 5.4 HIGH and LOW Operators

The **HIGH** and **LOW** operators can be used to extract specific bytes from an integer expression. The **LOW** operator extracts the byte starting at bit 0 of the expression, while the **HIGH** operator extracts the byte starting at bit 8 of the expression.

**HIGH** and **LOW** can also be used to extract portions of a floating-point value, as described above.

## 5.5 Operator Compatibility

The table on the following page shows the operators which are supported for compatibility with other assemblers.

Z8ASM	Compatible Syntax
+	+
-	-
%	^mod
&	^& or ^\$
	^
^	^X
~	^C
*	*
/	/
!	!
>	>
<	<
>=	>=
==	==
!=	<>
&&	&&
LOW	^LB
HIGH	^HB
OFFSET	<none>

Example:

The following expressions are equally acceptable to **Z8ASM**.

**a \*(b %(d >> 4))**

**a \*(b ^mod(d ^> 4))**



---

# Directives

## Chapter 6

---

Directives control the assembly process by providing the assembler with commands and information. These directives are instructions to the assembler itself and are not part of the microprocessor instruction set. The following sections provide details for each of the supported assembler directives.

Syntax:

<b>&lt;directive&gt;</b>	<b>=&gt; &lt;keyword&gt;[&lt;args&gt;]@</b>
	<b>=&gt; &lt;arg&gt;&lt;keyword&gt;&lt;arg&gt;</b>
<b>&lt;keyword&gt;</b>	<b>=&gt; valid assembler directive</b>
<b>&lt;arg&gt;</b>	<b>=&gt; valid directive argument</b>

### 6.1 ALIGN

**<align\_directive>    => ALIGN<int\_const>**

The **ALIGN** directive allows you to adjust the alignment of the current program counter. The program counter is aligned on the defined boundary.

Example:

**ALIGN 4**            ; align the PC on a 4-byte boundary.

## 6.2 COMMENT

**<comment\_directive>   => COMMENT<terminating\_char>**

The **COMMENT** directive allows a block of text to be inserted as comments. Text following the directive will be ignored until the terminating character is encountered. The line following the source line containing the terminating character is considered the next input line.

Example:

**COMMENT \***

This text, and all following text is ignored until this asterisk(\*) is encountered.

## 6.3 CONDLIST

**<condlist\_directive> => CONDLIST (ON|OFF)**

The **CONDLIST** directive controls the listing of conditional assembly blocks. If **CONDLIST ON**, then all conditional assembly blocks are included in the listing file. If **CONDLIST OFF**, false conditional assembly blocks are not included in the listing file. The default is **CONDLIST ON**.

## 6.4 CPU

**<cpu\_directive>       => CPU=<target\_processor>**

The **CPU** directive defines to the assembler which member of the Z8 family is targeted. From this directive **Z8ASM** can determine which instructions are legal as well as the location of the interrupt vectors within the **CODE** space.

The CPU names supported are: Z8601, Z8611, Z8681, Z8682. If no CPU directive is given Z8601 is assumed.

Example:

**CPU=Z8601**

## 6.5 Data Directives

**<data\_directive>    => <type><value>**

**<type>                => BFRACT  
                      => FRACT  
                      => LFRACT  
                      => UBFRACT  
                      => UFRACT  
                      => ULFRACT  
                      => BLKB  
                      => BLKL  
                      => BLKP  
                      => BLKW  
                      => DB  
                      => DD  
                      => DF  
                      => DL  
                      => DPTR  
                      => DS  
                      => DW**

**<value\_list>        => <value>  
                      => <value\_list>,<value>**

**<value>              => <expression>|<string\_const>**

The data directives allow you to reserve space for specified types of data. All data is aligned on a boundary of 1 mau.

### 6.5.1 BFRACT, UBFRACT Declaration type

BFRACT	signed fractional (8 bits)
UBFRACT	unsigned fractional (8 bits)

#### Examples:

BFRACT	[3]0.1,[2]0.2	; Reserve space for 5 8 bit signed ; fractional numbers. Initialize first ; 3 with a 0.1, and last 2 with a 0.2.
UBFRACT	[50]0.1,[50]0.2	; Reserve space for 100 8 bit unsigned ; fractional numbers. Initialize first ; 50 with a 0.1, second 50 with a 0.2.
BFRACT	0.5	; Reserve space for 1 8 bit signed ; fractional number and initialize it ; to 0.5

### 6.5.2 FRACT, UFRACT Declaration type

FRACT	signed fractional (16 bits)
UFRACT	unsigned fractional (16 bits)

#### Examples:

FRACT	[3]0.1,[2]0.2	; Reserve space for 5 16 bit signed ; fractional numbers. Initialize first 3 ; with a 0.1, and last 2 with a 0.2.
UFRACT	[50]0.1,[50]0.2	; Reserve space for 100 16 bit unsigned ; fractional numbers. Initialize first 50 ; with a 0.1, second 50 with a 0.2.
FRACT	0.5	; Reserve space for 1 16 bit signed ; fractional number and initialize it to ; 0.5.

### 6.5.3 LFRACT, ULFRACT Declaration type

LFRACT      signed fractional (32 bits)  
ULFRACT    unsigned fractional (32 bits)

#### Examples:

LFRACT      [3]0.1,[2]0.2      ; Reserve space for 5 32 bit signed  
   ; fractional numbers. Initialize first  
   ; 3 with a 0.1, and last 2 with a 0.2.

ULFRACT    [50]0.1,[50]0.2    ; Reserve space for 100 32 bit unsigned  
   ; fractional numbers. Initialize first  
   ; 50 with a 0.1, second 50 with a 0.2.

LFRACT      0.5                    ; Reserve space for 1 32 bit signed  
   ; fractional number and initialize it  
   ; to 0.5.

### 6.5.4 BLKB Declaration type

BLKB      bytes data (8 bits) [, <init\_value>]

#### Examples:

BLKB      16                    ; Allocate 16 uninitialized bytes.

BLKB      16, -1                ; Allocate 16 bytes and initialize them to  
   ; -1.

### 6.5.5 BLKL Declaration type

BLKL      bytes data (32 bits) [, <init\_value>]

#### Examples:

BLKL      16                    ; Allocate 16 uninitialized longs.

BLKL      16, -1                ; Allocate 16 longs and initialize them to  
   ; -1.

### 6.5.6 BLKP Declaration type

BLKP     bytes data (16 bits) [, <init\_value>]

**Examples:**

BLKP     16                     ; Allocate 16 uninitialized pointers.

BLKP     16, -1                ; Allocate 16 pointers and initialize them  
                                 ; to -1.

### 6.5.7 BLKW Declaration type

BLKW     bytes data (16 bits) [, <init\_value>]

**Examples:**

BLKW     16                     ; Allocate 16 uninitialized words.

BLKW     16, -1                ; Allocate 16 words and initialize them to  
                                 ; -1.

### 6.5.8 DB Declaration type

DB       bytes data (8 bits)

**Examples:**

DB       "Hello World" ; Reserve and initialize 11 bytes.

DB       1,2                    ; Reserve 2 bits. Initialize the first  
                                 ; byte with a 1, and the second with a 2.

DB       [50]1,[50]2           ; Reserve 100 bytes. Initialize first  
                                 ; 50 with a 1, second 50 with a 2.

DB       #12                    ; Reserve 1 byte and initialize it with  
                                 ; #12.

### 6.5.9 DD Declaration type

DD double float (64 bits)

#### Examples:

DD 0.1,0.2 ; Reserve space for 2 double float numbers.  
; Initialize the first with a 0.1, and last with  
; a 0.2.

DD 0.5 ; Reserve space for 1 double float number  
; and initialize it to 0.5.

### 6.5.10 DF Declaration type

DF float (32 bits)

#### Examples:

DF 0.1,0.2 ; Reserve space for 2 float numbers. Initia-  
; lize the first with a 0.1, and last with a 0.2.

DF 0.5 ; Reserve space for 1 float number and  
; initialize it to 0.5.

### 6.5.11 DL Declaration type

DL long (32 bits)

#### Examples:

DL [3]1,[2]2 ; Reserve 5 long words. Initialize first  
; 3 with a 1, and last 2 with a 2.

DL [50]1,[50]2 ; Reserve 100 long words. Initialize  
; first 50 with a 1, second 50 with a 2.

DL #12 ; Reserve space for 1 long word and  
; initialize it to #12.

### 6.5.12 DPTR Declaration type

DPTR      pointer data (16 bits)

#### Examples:

DPTR      LABEL                      ; Reserve 16 bit pointer with  
   ; address of label.

DPTR      LABEL1, LABEL2           ; Reserve 2 16 bit locations to hold  
   ; LABEL1 and LABEL2.

### 6.5.13 DS Declaration type

DS          bytes data (8 bits)

#### Examples:

DS          10                        ; Reserve 10 bytes without initializing them.

### 6.5.14 DW Declaration type

DW          word data (16 bits)

#### Examples:

DW          1,2                       ; Reserve 2 words. Initialize the first  
   ; word with a 1, and the second with a 2.

DW          [50]1,[50]2             ; Reserve 100 words. Initialize first  
   ; 50 with a 1, second 50 with a 2.

DW          #12                       ; Reserve 1 word and initialize it with  
   ; #12.



## 6.6 DEFINE

```
<segment_definition>    => DEFINE<ident>[<space_clause>]
                           [<align_clause>][<org_clause>]
```

The **DEFINE** directive defines a segment and its associated address space, alignment, and origin. A segment must be defined before it can be used, unless it is a pre-defined segment. If a clause is not given, the default for that definition is used. See the **SEGMENT** directive for further information and a listing of pre-defined segments.

### 6.6.1 ALIGN clause

```
<align_clause>          => ,ALIGN=<int_const>
```

The **ALIGN** clause allows you to select the alignment boundary for a segment. The linker will place modules in this segment on the defined boundary. The boundary is a multiple of the **MAU** defined for the space identifier. The multiple must be a power of two (1, 2, 4, 8, etc.).

The default alignment is “1” MAU.

### 6.6.2 ORG clause

```
<org_clause>            => ,ORG=<int_const>
```

An **ORG** clause allows you to specify where the segment is to be located, making the segment an absolute segment. The linker will place the segment at the memory location specified by the **ORG**.

The default is **no ORG** and thus the segment is relocatable.

Examples:

```
DEFINE near_code
; Uses the defaults of CODE, byte alignment and relocatable.
```

```
DEFINE my_data,SPACE=DATA,ALIGN=2
; Aligns on a word boundary, relocatable.
```

```
DEFINE irq_table,ORG=$FFF8
; Uses CODE, byte alignment, and absolute starting address at
; memory location $FFF8.
```

### 6.6.3 SPACE clause

**<space\_clause>           => ,SPACE=<ident>**

A **SPACE** clause defines the address space in which the segment resides. The linker groups together segments with the same space identification. The following are the allowed space identifications:

SPACE ID	MAU SIZE	MAX MAUS PER ADDRESS
ROM	8 bits	1 (default)
RDATA	8 bits	1
XDATA	8 bits	1

## 6.7 END

**<end\_directive>           => END[<expression>]**

The **END** directive informs the assembler of the end of the source input file. If the operand field is present, it defines the start address of the program. During the linking process, only one module may have a start address, otherwise an error results. The **END** directive is optional for those modules not needing a start address.

Any text found after an **END** directive is ignored.

Example:

**END start**       ; Use the value of start as the program start address.

## 6.8 EQU

**<equate\_directive>       => <ident>EQU<expression>**

Symbolic names are given numeric or string values by the **EQU** statement. Any name used to define an equate must not have been previously defined. Other equates and label symbols are allowed in the expression, provided they are previously defined.

Examples:

```
pi      EQU      3.141      ; set to the value of pi
area    EQU      pi*7*7    ; area of a circle
reg     EQU      "X"       ; symbolic name of a register
```

## 6.9 ERROR

**<error\_directive>           => ERROR <string\_const>**

The **ERROR** directive informs the assembler to generate an error using **<string\_const>** as the error message. The error will be associated with error number 475, which is reserved for user errors and warnings.

Example:

```
ERROR "User Error"
```

## 6.10 EXIT

**<exit\_directive>           => EXIT <string\_const>**

The **EXIT** directive informs the assembler to generate an error using **<string\_const>** as the error message, and halt all further source line processing. The generated error will be associated with error number 475, which is reserved for user errors and warnings.

Example:

```
EXIT       "Fatal User Error"
```

## 6.11 EXTERN

**<extern\_directive>       => EXTERN<ident list>**

The **EXTERN** directive specifies that a list of labels in the operand field are defined in another module. The reference will be resolved by the linker. The

labels must not be defined in the current module. A label can be attached to a specific space by following the label with a colon (:) and the name of the space.

Examples:

```
EXTERN      label
EXTERN      label1, label2, label3
EXTERN      label4 : ROM
```

## 6.12 FCALL

**<fcall\_directive>           => .FCALL<module\_name>**

The **FCALL** directive should be used within a frame definition to identify the modules that are called by the frame. This directive is normally used by the compiler for a static frame mechanism.

Example:

```
      .FCALL _func      ; Identifies _func as being called by the
                        ; frame containing this directive.
```

## 6.13 FILE

**<file\_directive>           => FILE <string\_const>**

The **FILE** directive allows the file containing the high level source code associated with the assembly code file to be passed to the debugger and linker. This directive is normally used by the compiler.

Example:

```
      FILE      "test.c"
```

## 6.14 FRAME

```
<frame_directive>      => .FRAME<frame_name>,
                        <module_name>,<space>
```

The **FRAME** directive is used to define and enter a frame that will be statically allocated by the linker by constructing a call graph. This directive is normally used by the compiler for a static frame mechanism.

All data and code that follows this directive will be attached to the frame, until the next segment directive is encountered. The **FCALL** directive should be used to identify the modules called by the function associated with the frame.

Example:

```
.FRAME _seg__func, _func, DATA
DS 1
.FCALL _func1
```

The above example defines a frame called **\_seg\_\_func**. This frame is associated with the module **\_func**, and is to be located in the DATA memory space. The frame allocates one byte of data, and the module **\_func1** is called by **\_func**.

## 6.15 GLOBALS

```
<globals_directive>    => GLOBALS (ON | OFF)
```

All label definitions and equates are considered public if the **GLOBALS** directive is on. If globals is off, then the **XDEF** directive must be used. The default is **GLOBALS OFF**.

## 6.16 INCLUDE

**<include\_directive>      => INCLUDE <string\_const>**

The **INCLUDE** directive allows the insertion of source code from another file into the current source file during assembly. The included file is assembled into the current source file immediately after the directive. When the **EOF** (End of File) of the included file is reached, the assembly resumes on the line after the **INCLUDE** directive.

The file to include is named in the string constant after the **INCLUDE** directive. A file name may contain a path. If the file does not exist, an error results and the assembly is aborted. Recursive **INCLUDE**s also result in an error.

**INCLUDE** files are contained in the list file unless a **NOLIST** directive is active.

Examples:

<b>INCLUDE</b>	<b>"calc.h"</b>	; include calc header file
<b>INCLUDE</b>	<b>"\test\calc.h"</b>	; contains a path name
<b>INCLUDE</b>	<b>calc.h</b>	; ERROR, use string constant

## 6.17 LIST

**<list\_directive>      => LIST (ON|OFF)**

The **LIST** directive instructs the assembler to produce output for the listing file. This mode stays in effect unless a **NOLIST** directive is processed. **LIST ON** is the same as **LIST**. **LIST OFF** is the same as **NOLIST**. **LIST ON** is the default mode.

## 6.18 MACDELIM

**<macdelim\_directive>   => MACDELIM <char\_const>**

The **MACDELIM** directive allows the MACRO delimiter pair to be changed. The selection of a delimiter implies the use of the corresponding closing delimiter. Since a character constant is expected, the new macro delimiter must be enclosed with single quotes. All other character constants will result in an error.

Delimiter Character	Macro Delimiters
{	{,}
[	[,]
(	(,)

Examples:

**MACDELIM**    ‘[’     ; Use [..] pair for macro delimiter.  
**MACDELIM**    ‘,’     ; ERROR, illegal macro delimiter

## 6.19 MACLIST

**<maclist\_directive>     => MACLIST (ON|OFF)**

The **MACLIST** directive controls the listing of macro expansions. If **ON**, then macro expansions are included in the listing file. If **OFF**, then macro expansions are suppressed. The default is **MACLIST ON**.

## 6.20 NEWPAGE

**<newpage\_directive>     => NEWPAGE**

The **NEWPAGE** directive causes the assembler to start a new page in the output listing. This directive has no effect if **NOLIST** is in effect. No operand field is allowed.

## 6.21 NOLIST

**<nolist\_directive>            => NOLIST**

The **NOLIST** directive turns off the output listing of the assembly. This mode remains in effect until a **LIST** directive is processed. No operand field is allowed. **LIST OFF** is the same as **NOLIST**.

## 6.22 ORG

**<org\_directive>            => ORG<int\_const>**

An **ORG** directive resets the assembler's internal program counter to the integer constant following the **ORG**, and causes absolute code/data to be generated.

The **ORG** directive must be followed by an integer constant.

Examples:

```
ORG     $1000     ; generate absolute code at $1000  
ORG     LOOP     ; ERROR, use an integer constant
```

**WARNING:** If the **ORG** directive is used within a relocatable segment, that segment will become an absolute segment, with 0 as the base physical address of the segment. It is recommended that segments requiring the use of an **ORG** should be declared as absolute segments.

## 6.23 PL

**<pl\_directive>            => PL <int\_const>**

The **PL** directive sets the page length for the listing file. The directive must be followed by an integer constant. The default page length is 56.

Examples:

```
PL       60  
PL       LOOP     ; ERROR, use an integer constant
```



## 6.24 PUBLIC

**<public\_directive>      => PUBLIC<ident list>**

The **PUBLIC** directive defines a list of labels in the current module as an external symbol that are to be made publicly visible to other modules at link time. The operands must be labels which are defined somewhere in the assembly file.

Examples:

<b>PUBLIC</b>	<b>label</b>
<b>PUBLIC</b>	<b>label1, label2, label3</b>

## 6.25 PW

**<pw\_directive>      => PW <int\_const>**

The **PW** directive sets the page width for the listing file. The directive must be followed by an integer constant. The default page width is 80.

Examples:

<b>PW</b>	<b>128</b>	
<b>PW</b>	<b>LOOP</b>	; ERROR, use an integer constant

## 6.26 SCOPE

**<scope\_directive>      => SCOPE**

The **SCOPE** directive directs the assembler to start a new local label scope. Any local labels currently active will be removed, and thus can be reused.

## 6.27 SEGMENT

**<segment\_directive>    => SEGMENT <ident>**

A **SEGMENT** directive defines what segment to attach the following code/data. The segment identifier may be user defined, by the **DEFINE** directive, or a pre-defined segment.

A **SEGMENT** directive must be followed by the segment identifier. The default segment is used until the assembler encounters a **SEGMENT** directive. The internal assembler program counter is reset to the previous program counter of the segment when a **SEGMENT** directive is encountered. The following is the list of pre-defined segments:

Segment ID	Space	Align	Origin
near_bss	RDATA	1	Relocatable
far_bss	XDATA	1	Relocatable
near_data	RDATA	1	Relocatable
far_data	XDATA	1	Relocatable
code	ROM	1	Relocatable (default)
text	ROM	1	Relocatable

Refer to Chapter 2, for further information.

Examples:

```

SEGMENT      CODE_DEFAULT ; pre-defined segment
SEGMENT      mycode         ; user defined

```

## 6.28 SUBTITLE

**<subtitle\_directive>    => SUBTITLE <string\_const>**

The **SUBTITLE** directive results in a user defined subtitle being displayed in the assembly listing. The subtitle remains in effect until the next subtitle directive. The operand must be a string constant.

## 6.29 TITLE

**<title\_directive>           => TITLE <string\_const>**

The **TITLE** directive causes a user defined **TITLE** to be displayed in the assembly listing. The new title remains in effect until the next **TITLE** directive. The operand must be a string constant.

## 6.30 VAR

**<var\_directive>           => <ident> VAR <expression>**

The **VAR** directive allows a symbolic value to be modified. The directive follows the same rules as the **EQU** directive, except the symbolic name can be previously defined. The symbol itself may be used in the expression, assuming that it has been previously defined. The value of the equate is undefined if the symbol is used in the expression without having been defined previously.

Example:

```

LOOP   VAR   LOOP+1       ; Loop defined previously, add 1
                          ; to its current value

```

## 6.31 VECTOR

**<vector\_directive>           => VECTOR<vector name>=**  
**<expression>**

The **VECTOR** directive is used to initialize an interrupt or reset vector to a program address. **<vector name>** must be IRQ0...IRQ5, or RESET and specifies which vector is being specified.

If **RESET** is used **Z8ASM** inserts a **JMP** instruction to **<expression>** at the proper reset address.

**<expression>** is the address, usually a label, of the program code to handle the interrupt or reset.

**NOTE:**

If no reset vector is defined within the program, **Z8LINK** generates a warning message.

**NOTE:** The **CPU** directive is used to determine the physical location of the interrupt vectors.

Examples:

```
VECTOR    IRQ0=irq0_handler
VECTOR    IRQ1=irq1_handler
```

## 6.32 WARNING

**<warning\_directive>       => WARNING <string\_const>**

The **WARNING** directive informs the assembler to generate a warning using **<string\_const>** as the warning message. The warning will be associated with warning number 475, which is reserved for user errors and warnings.

Example:

```
WARNING "User warning"
```

## 6.33 Compatibility

The following table shows the equivalences between **Z8ASM** directives and those of other assemblers that are also supported by **Z8ASM**.

Z8ASM	Compatible with
ALIGN	.align
DB	.byte or .ascii
DD	<none>
DF	<none>
DL	.long
DR	<none>
DS	.block
DW	.word
END	.end
ENDIF	.endif
ENDMAC	.endm
EQU	.equ
IF	.if
INCLUDE	.include
LIST	.list <on> or <off>
MACRO	.macro
NEWPAGE	.page [<len>] [<width>]
SEGMENT	.section
TITLE	.title
VECTOR	<none>
XDEF	.global
XREF	.extern

A  
s  
s  
e  
m  
b  
l  
e  
r

---

# Conditional Assembly

## Chapter 7

---

Conditional assembly is used to control the assembly of blocks of code. Entire blocks of code may be enabled or disabled by use of proper conditional statements.

### 7.1 Conditional Assembly Syntax

<code>&lt;if_statement&gt;</code>	<code>=&gt; &lt;if_section&gt;</code> <code>    [&lt;else_statement&gt;]</code> <code>    ENDIF</code>
<code>&lt;if_section&gt;</code>	<code>=&gt; &lt;if conditional&gt;</code> <code>    &lt;code-body&gt;</code>
<code>&lt;if_conditional&gt;</code>	<code>=&gt; (IF&lt;cond_expression&gt;) </code> <code>    (IFDEF&lt;ident&gt;) </code> <code>    (IFSAME&lt;string_const&gt;, &lt;string_const&gt;) </code> <code>    (IFMA&lt;int_const&gt;)</code>
<code>&lt;else_statement&gt;</code>	<code>=&gt; (ELSE &lt;code_body&gt;) </code> <code>    (ELIF&lt;cond_expression&gt; &lt;code_body&gt;</code> <code>    [&lt;else_statement&gt;])</code>
<code>&lt;cond_expression&gt;</code>	<code>=&gt; &lt;expression&gt; </code> <code>    (&lt;expression&gt;&lt;relop&gt;&lt;expression&gt;)</code>
<code>&lt;relop&gt;</code>	<code>=&gt; == &lt; &gt; &lt; = &gt; !=</code>

### 7.1.1 Conditional Assembly Expressions

Any symbol used in a conditional expression must be previously defined by an **EQU** directive. Relational operators may be used in the expression. Relational expressions evaluate to 1 if true, and 0 if false.

If a conditional is true the code body is processed. Otherwise, the code body after an **ELSE** is processed if included.

The **ELIF** keyword allows a case-like structure to be implemented.

**IF/ENDIF** structures may be nested.

Examples:

```
IF FLOAT           ; process code body if FLOAT is not 0
  (code body)
ENDIF

IFDEF FLOAT        ; process code body if FLOAT is defined
  (code body)
ENDIF

IF FLOAT           ; code body 1 if FLOAT is not 0
  (code body 1)
ELIF INTEGER        ; FLOAT=0 and INTEGER is not 0, code body 2
  (code body 2)
ELSE               ; otherwise code body 3
  (code body 3)
ENDIF
```

## 7.2 Conditionals

The following lists the **IF** conditionals and a brief description of their use.

### 7.2.1 IF

Use the **IF** keyword to evaluate integer boolean expressions. If the expression evaluates to 0, the result is false, otherwise the result is true.



### 7.2.2 IFDEF

Use the **IFDEF** keyword to check if a label has been defined. Only a single label may be used with this conditional. If the label is defined, the result is true, otherwise the result is false.

### 7.2.3 IFSAME

Use the **ISAME** keyword to check if two strings constants are equivalent. If the strings are the same, the result is true, otherwise the result is false. If the strings are not enclosed by quotes, the comma is used as the separator.

### 7.2.4 IFMA

The **IFMA** keyword can only be used within a macro, and is used to check if a macro argument has been defined. If the argument is defined, the result is true, otherwise the result is false. If the argument to check is 0, the result is TRUE if no arguments were provided, otherwise the result is FALSE.

A  
s  
s  
e  
m  
b  
l  
e  
r

---

# Macro Assembly

## Chapter 8

---

Macros allow a sequence or series of assembler source lines to be represented by a single assembler symbol. In addition, arguments can be supplied to the macro in order to specify or alter the assembler source lines generated. This section describes how to define and invoke macros.

### 8.1 MACRO definition

```
<macro_def> => <macname>[:]MACRO[<arg>(<arg>)]  
                <code_body>  
                ENDMAC[RO]<macname>
```

A macro definition must precede the use of the macro. The macro name, **<macname>**, must be the same for both the definition and the **ENDMACRO** line. The argument list contains the formal arguments that will be substituted with actual arguments when the macro is invoked.

During the invocation of the macro, a token substitution is performed, replacing the formal arguments with the actual arguments.

Macro Definition:

```
store:    MACRO reg1,reg2,reg3  
  
          ADD      reg1,reg2  
          MOV      (reg3),reg1  
  
          ENDMACstore
```

## 8.2 Macro Invocation

**<mac\_invoc>    => <macname>[<arg>(<arg>)]@]**

A macro is invoked by specifying the macro name, and following the name with the desired arguments. The arguments should be separated by commas.

Macro Invocation:

**store R1,R2,R3**

Result:

**ADD     R1,R2  
MOV     (R3),R1**

This macro invocation causes register R1 and R2 to be added and the result stored at the address in register R3.

## 8.3 Local Macro Labels

So that unique label definitions can be generated by a macro call, the assembler has a local macro label feature. When used within the body of a macro, symbols preceded by two dollar signs (\$\$) are considered local to the scope of the macro, and therefore, are guaranteed to be unique. The two dollars signs are replaced by an underscore followed by the macro invocation number.

Macro Definition:

**LJMP:    MACRO     cc,label  
  
         SJMP       cc,\$\$lab  
         JP          label  
         \$\$lab:  
  
ENDMACLJMP**

Invocation:

**LJMP     GT, done**

Result:

```

        SJMP    GT, _1lab
        JP      done
_1lab:

```

## 8.4 Optional Macro Arguments

A macro can be defined to handle missing arguments by using the **IFMA** (if macro argument) conditional within the macro. The conditional can be used to detect if an argument was supplied with the invocation.

Macro Definition:

```

MISSING_ARG:  MACRO  ARG1,ARG2,ARG3

        IFMA 2
            MOV    ARG1,ARG2
        ELSE
            MOV    ARG1,ARG3
        ENDIF

ENDMACRO MISSING_ARG

```

Invocation:

```
MISSING_ARG    R1, ,R3    ; missing second arg
```

Result:

```
MOV    R1,R3
```

## 8.5 MACEXIT

The **MACEXIT** directive may be used to immediately exit a macro. No further code/directive processing is performed, however, the assembler will check for proper **if-then** conditionals. A **MACEXIT** is normally used to terminate a recursive macro.

Macro Definition:

```
RECURSIVE:  MACRO    ARG1,ARG2  
  
    IFMA 0  
        MACEXIT  
    ELSE  
        RECURS_MAC    ARG2  
        DB    ARG1  
    ENDIF  
  
ENDMACRORECURS_MAC
```

The above example is a recursive macro that demonstrates the use of **MACEXIT** in order to terminate the macro.

## 8.6 Delimiting Arguments

Macro arguments can be delimited by use of the current macro delimiter characters. The delimiters can be used to include commas and spaces which are not normally allowed as part of an argument. The default delimiters are brackets { }, but braces [ ] and parenthesis are also allowed.

Macro Definition:

```
LJMP:      MACRO    label  
  
        JMP      label  
  
ENDMACLJMP
```

Invocation:

```
LJMP      {dummy,X}
```

Result:

```
JMP      dummy,X
```

# Assembler Symbols

## Chapter 9

### 9.1 Equates

Equates are considered symbolic values, and allow you to assign an integer value, floating-point value, or string substitution to a symbol. Integer equates can be exported. When an equate is encountered with the source code, the appropriate value is substituted. An integer equate may be modified during assembly by the use of the **VAR** directive. It is important to note that equates that are imported are treated as labels, and not equates, since the equated value is unknown.

Examples:

<b>PI</b>	<b>EQU</b>	<b>3.141</b>	; Floating-point equate
<b>ANGLE</b>	<b>EQU</b>	<b>45</b>	; Integer equate
<b>REG</b>	<b>EQU</b>	<b>"X"</b>	; String substitution

### 9.2 Labels

Labels are considered representations of memory locations, and can be used to reference that specific memory location within an expression.

#### 9.2.1 Local Labels

Any label beginning with a question mark (?) is considered to be a local label. The scope of a local label ends when a **SCOPE** directive is encountered, thus allowing the label name to be reused. A local label cannot be imported or exported.

Examples:

<b>?LOOP:</b>	<b>BRA</b>	<b>?LOOP</b>	; Infinite branch to ?LOOP
<b>SCOPE</b>			; New local label scope
<b>?LOOP:</b>	<b>BRA</b>	<b>?LOOP</b>	; Reuse ?LOOP

#### 9.2.2 Importing/Exporting Labels

Labels can be imported from other modules by use of the **EXTERN** directive. A space may be provided in the directive to indicate the labels's location, otherwise the space of the current segment is used as the location of the label.

Labels can be exported to other modules by use of the **PUBLIC** directive.

### 9.2.3 Label Locations

The assembler makes use of a label's location when checking the validity of instruction operands. Certain instruction operands require that a label be located in a specific space, since that instruction may only be able to operate on data located in that space. A label is assigned to a space by one of the following methods:

1. The space of the segment in which the label is defined.
2. The space provided in the **EXTERN** directive.
3. If no space is provided with the **EXTERN** directive, the space of the segment where the **EXTERN** directive was encountered is used as the location of the label.

### 9.2.4 Label Checks

The assembler performs location checks when a label is used as an operand, including forward referenced labels. Thus, when a label not located in the proper space is used as an operand, the assembler flags a warning.

Example:

```

SEGMENT  data_default

EXTERN  label1:ROM      ; Label1 is located in ROM
EXTERN  label2          ; Label2 and label3 are located in
label3:          ; the DATA space

LOAD    label2          ; Valid use of page0 label
LOAD    label1          ; Generates assembler warning,
                        ; cannot load from ROM

```



# Addressing Modes

## Chapter 10

This chapter describes the addressing mode syntax for the instruction operands.

### 10.1 Register Name

#### 10.1.1 Working Registers

Name	Description
r0...r15	Working registers
rr0...rr14	Working registers pairs

#### 10.1.2 Special Function Registers

Name	Description	Address
SPL	Stack Pointer (Bits 7-0)	%FF
SPH	Stack Pointer (Bits 15-8)	%FE
RP	Register Pointer	%FD
FLAGS	Program Control Flags	%FC
IMR	Interrupt Mask Register	%FB
IRQ	Interrupt Request Register	%FA
IPR	Interrupt Priority Register	%F9
P01M	Ports 0-1 Mode	%F8
P3M	Port 3 Mode	%F7
P2M	Port 2 Mode	%F6
PRE0	To Prescaler	%F5
T0	Timer/Counter 0	%F4
PRE1	T1 Prescaler	%F3
T1	Timer/Counter 1	%F2
TMR	Timer Mode	%F1
SIO	Serial I/O	%F0
P3	Port 3	%03
P2	Port 2	%02
P1	Port 1	%01

P0

Port 0

%00

## 10.2 Condition Codes

This section defines the condition codes supported by the ZiLOG Z8 assembler.

Name	Description
<b>C</b>	Carry
<b>EQ</b>	Equal
<b>GE</b>	Greater than or equal
<b>GT</b>	Greater than
<b>LE</b>	Less than or equal
<b>LT</b>	Less than
<b>MI</b>	Minus
<b>NC</b>	No carry
<b>NE</b>	Not equal
<b>NOV</b>	No overflow
<b>NZ</b>	Not zero
<b>OV</b>	Overflow
<b>PL</b>	Plus
<b>UGE</b>	Unsigned greater than or equal
<b>ULT</b>	Unsigned less than
<b>UGT</b>	Unsigned greater than
<b>ULE</b>	Unsigned less than or equal
<b>Z</b>	Zero

## 10.3 Addressing Modes

This section discusses the syntax of the addressing modes supported by the ZiLOG Z8 assembler. The symbolic name is used in the discussion of instruction syntax in Section 8.4.

Symbolic Names	Syntax	Description
<cc>	See Condition	Condition Codes. Codes above.
<r>	rn	Working register.
<R>	rn	Working register <const exp> or register.
<RR>	<const exp>	Register pair <b>rrn</b> or working register pair.
<Ir>	@rn	Indirect working register

---

<IR>	@<const exp>	Indirect register @rn or indirect working register.
<Irr>	@rrn	Indirect working register pair.
<IRR>	@<const exp>	Indirect register pair or @rrn Indirect working Register pair.
<X>	<const exp>(rn)	Indexed.
<DA>	<const exp>	Direct address (0..65535).
<RA>	<const exp>	Relative address (-128..+127).
<IM>	#<const exp>	Immediate constant (0..255).

A  
s  
s  
e  
m  
b  
l  
e  
r

### 11.1 Instruction Formats

This section lists each of the instructions supported by the ZiLOG Z8 assembler along with the legal addressing modes for each operand.

#### 11.1.1 ADC

Description:	Add with carry
Synopsis:	ADC      <dst>,<src>
Operands:	<r>,<r> <r>,<Ir> <R>,<R> <R>,<IR> <R>,<IM> <IR>,<IM>
Example:	ADC      SUM,@R10

#### 11.1.2 ADD

Description:	Addition
Synopsis:	ADD      <dst>,<src>
Operands:	<r>,<r> <r>,<Ir> <R>,<R> <R>,<IR> <R>,<IM> <IR>,<IM>
Example:	ADD      SUM,AUGEND

### 11.1.3 AND

Description:	Logical
Synopsis:	AND     <dst>,<src>
Operands:	<r>,<r> <r>,<Ir> <R>,<R> <R>,<IR> <R>,<IM> <IR>,<IM>
Example:	AND     TARGET, %%7B

### 11.1.4 CALL

Description:	Call Procedure
Synopsis:	CALL     <dst>
Operands:	<DA> <IRR>
Example:	CALL     %3521

### 11.1.5 CCF

Description:	Complement carry flag
Synopsis:	CCF
Operands:	None
Example:	CCF

### 11.1.6 CLR

Description: Clear

Synopsis: CLR <dst>

Operands: <R>  
<IR>

Example: CLR R6

### 11.1.7 COM

Description: Complement

Synopsis: COM <dst>

Operands: <R>  
<IR>

Example: COM R8

### 11.1.8 CP

Description: Compare

Synopsis: CP <dst>, <src>

Operands: <r>, <r>  
<r>, <Ir>  
<R>, <R>  
<R>, <IR>  
<R>, <IM>  
<IR>, <IM>

Example: CP TEST, @R0

### 11.1.9 DA

Description: Decimal adjust

Synopsis: DA <dst>

Operands: <R>  
<IR>

Example: DA R0

### 11.1.10 DEC

Description: Decrement

Synopsis: DEC <dst>

Operands: <R>  
<IR>

Example: DEC R10

### 11.1.11 DECW

Description: Decrement Word

Synopsis: DECW <dst>

Operands: <RR>  
<IR>

Example: DECW @R0

### 11.1.12 DI

Description: Disable Interrupts

Synopsis: DI

Operands: None

Example: DI



### 11.1.13 DJNZ

Description:	Decrement and Jump if Nonzero	
Synopsis:	DJNZ	<r>,<dst>
Operand:	<RA>	
Example:	DJNZ	R6,LOOP

### 11.1.14 EI

Description:	Enable Interrupts	
Synopsis:	EI	
Operand:	None	
Example:	EI	

### 11.1.15 HALT

Description:	Wait for Interrupt	
Synopsis:	HALT	
Operand:	None	
Example:	HALT	

### 11.1.16 INC

Description:	Increment	
Synopsis:	INC	<dst>
Operands:	<r> <R> <IR>	
Example:	INC	R10

### 11.1.17 INCW

Description: Increment Word

Synopsis: INCW <dst>

Operands: <RR>  
<IR>

Example: INCW RR0

### 11.1.18 IRET

Description: Interrupt Return

Synopsis: IRET

Operand: None

Example: IRET

### 11.1.19 JP

Description: Jump

Synopsis: JP <cc>,<dst>

Operands: <DA>  
<IRR>

Examples: JP C,%1520

### 11.1.20 JR

Description: Jump Relative

Synopsis: JR <cc>,<dst>

Operand: <RA>

Examples: JR JR MI,\$+9  
JR L0

**11.1.21 LD**

Description: Load

Synopsis: LD &lt;dst&gt;,&lt;src&gt;

Operands:

- <r>,<IM>
- <r>,<R>
- <R\*>,<r>
- <r>,<Ir>
- <Ir>,<r>
- <R>,<R>
- <R>,<IR>
- <R>,<IM>
- <IR>,<IM>
- <IR>,<R>
- <r>,<X>
- <X>,<r>

**NOTE:** \*In this instance only a full 8-bit register address can be used.

Example: LD 240(R0),R10

**11.1.22 LDC**

Description: Load Constant

Synopsis: LDC &lt;dst&gt;,&lt;src&gt;

Operands:

- <r>,<Irr>
- <Irr>,<r>

Example: LDC @R2, @RR6

### 11.1.23 LDCI

Description: Load Consant Autoincrement

Synopsis: LDCI <dst>,<src>

Operands: <Ir>,<Irr>  
<Irr>,<Ir>

Example: LDCI @R2, @RR6

### 11.1.24 LDE

Description: Load External Data

Synopsis: LDE <dst>,<src>

Operands: <r>,<Irr>  
<Irr>,<Ir>

Example: LDE @RR6,R2

### 11.1.25 LDEI

Description: Load External Data Autoincrement

Synopsis: LDEI <dst>,<src>

Operands: <Ir>,<Irr>  
<Irr>,<Ir>

Example: LDEI @RR6,@R2

### 11.1.26 NOP

Description: No operation

Synopsis: NOP

Operand: None

Example: NOP

### 11.1.27 OR

Description: Logical OR

Synopsis: OR    <dst>,<src>

Operands:    <r>,<r>  
              <r>,<Ir>  
              <R>,<R>  
              <R>,<IR>  
              <R>,<IM>  
              <IR>,<IM>

Example: OR    TARGET,##7B

### 11.1.28 POP

Description: Pop

Synopsis: POP    <dst>

Operands:    <R>  
              <IR>

Example: POP    @R6

### 11.1.29 PUSH

Description: Push

Synopsis: PUSH    <src>

Operands:    <R>  
              <IR>

Example: PUSH    FLAGS

### 11.1.30 RCF

Description: Reset Carry Flag

Synopsis: RCF

Operand: None

Example: RCF

### 11.1.31 RET

Description: Return

Synopsis: RET

Operand: None

Example: RET

### 11.1.32 RL

Description: Rotate Left

Synopsis: RL <dst>

Operands: <R>  
<IR>

Example: RL SHIFTER

### 11.1.33 RLC

Description: Rotate Left Through Carry

Synopsis: RLC <dst>

Operands: <R>

Example: RLC SHIFTER

### 11.1.34 RR

Description: Rotate Right

Synopsis: RR <dst>

Operands: <R>  
<IR>

Example: RR R6

### 11.1.35 RRC

Description: Rotate Right Through Carry

Synopsis: RRC <dst>

Operands: <R>  
<IR>

Example: RRC SHIFTER

### 11.1.36 SBC

Description: Subtract With Carry

Synopsis: SBC <dst>, <src>

Operands: <r>, <r>  
<r>, <Ir>  
<R>, <R>  
<R>, <IR>  
<R>, <IM>  
<IR>, <IM>

Example: SBC MINUEND, @R10

**11.1.37 SCF**

Description: Set Carry Flag

Synopsis: SCF

Operand: None

Example: SCF

**11.1.38 SRA**

Description: Shift Right Arithmetic

Synopsis: SRA <dst>

Operands: <R>  
<IR>

Example: SRA SHIFTER

**11.1.39 SRP**

Description: Set Register Pointer

Synopsis: SRP <src>

Operand: <IM>

Example: SRP #%70

**11.1.40 STOP**

Description: Wait for Reset

Synopsis: STOP

Operand: None

Example: STOP



**11.1.41 SUB**

Description:	Subtract
Synopsis:	SUB      <dst>,<src>
Operands:	<r>,<r> <r>,<Ir> <R>,<R> <R>,<IR> <R>,<IM> <IR>,<IM>
Example:	SUB      MINUENT, %%11

**11.1.42 SWAP**

Description:	Swap Nibbles
Synopsis:	SWAP    <dst>
Operands:	<R> <IR>
Example:	SWAP    BCD_Operands

**11.1.43 TCM**

Description:	Test Complement Under Mask
Synopsis:	TCM      <dst>,<src>
Operands:	<r>,<r> <r>,<Ir> <R>,<R> <R>,<IR> <R>,<IM> <IR>,<IM>
Example:	TCM      TESTER, MASK

#### 11.1.44 TM

Description: Test Under Mask

Synopsis: TM <dst>,<src>

Operands: <r>,<r>  
<r>,<Ir>  
<R>,<R>  
<R>,<IR>  
<R>,<IM>  
<IR>,<IM>

Example: TM TESTER, MASK

#### 11.1.45 WDH

Description: Watch Dog Timer During HALT

Synopsis: WDH

Operand: None

Example: WDH

#### 11.1.46 WDT

Description: Watch Dog Timer

Synopsis: WDT

Operand: None

Example: WDT

### 11.1.47 XOR

Description:	Logical Exclusive OR
Synopsis:	XOR     <dst>,<src>
Operands:	<r>,<r> <r>,<Ir> <R>,<R> <R>,<IR> <R>,<IM> <IR>,<IM>
Example:	XOR     TARGET, #7B

A  
s  
s  
e  
m  
b  
l  
e  
r

---

The syntax description that follows is given to outline the general assembler syntax. It does not define assembly language instructions. Refer to the reference material for the microprocessor for specific information.

```
<source_line>    => <if_statement>
                  => [<label_field>]<instruction_field><EOL>
                  => [<label_field>]<directive_field><EOL>
                  => <label_field><EOL>
                  => <EOL>

<if_statement>    => <if_section>
                  => [<else_statement>]
                  => ENDIF

<if_section>      => <if_conditional>
                  <code-body>

<if_conditional> => (IF<cond_expression>)|
                  (IFDEF<ident>)|
                  (IFSAME<string_const>,<string_const>)|
                  (IFMA<int_const>)

<else_statement> => (ELSE <code_body>)|
                  (ELIF<cond_expression>
                   <code_body>
                   [<else_statement>])

<cond_expression>=><expression>|
                  (<expression><relop><expression>)

<relop>          => ==|<|>|<=|>|=|!=

<code_body>      => <source_line>@

<label_field>    => <ident>:
```

<instruction\_field> => <mnemonic>[<operand>]@

<directive\_field> => <directive>

<mnemonic> => valid instruction mnemonic

<operand> => <addressing\_mode>  
=> <expression>

<addressing\_mode> => valid instruction addressing mode

<directive> => ALIGN<int\_const>  
=> <array\_definition>  
=> BANKED(ON|OFF)  
=> CONDLIST(ON|OFF)  
=> CPU=<target\_microprocessor>  
=> END[<expression>]  
=> <ident>EQU<expression>  
=> ERROR<string\_const>  
=> EXIT<string\_const>  
=> .FCALL<ident>  
=> FILE<string\_const>  
=> .FRAME<ident>,<ident>,<space>  
=> GLOBALS (ON|OFF)  
=> INCLUDE<string\_const>  
=> LIST (ON|OFF)  
=> <macro\_def>  
=> <macro\_invoc>  
=> MACDELIM<char\_const>  
=> MACLIST (ON|OFF)  
=> NEWPAGE  
=> NOLIST  
=> ORG<int\_const>  
=> PL<int\_const>  
=> PW<int\_const>  
=> <public\_definition>  
=> <scalar\_definition>  
=> SCOPE  
=> <segment\_definition>  
=> SEGMENT<ident>  
=> SUBTITLE<string\_const>  
=> SYNTAX=<target\_microprocessor>  
=> TITLE<string\_const>  
=> <ident>VAR<expression>  
=> WARNING<string\_const>

<array_definition>	=> <type>['<elements>'] => [<initvalue>(<initvalue>)]
<type>	=> BFRACT => FRACT => LFRACT => UBFRACT => UFRACT => ULFRACT => DB => DL => DPTR => DW
<elements>	=> [<int_const>]
<initvalue>	=> ['<instances>']<value>
<instances>	=> <int_const>
<value>	=> <expression> <string_const>
<expression>	=> '('<expression>')' => <expression><binary_op><expression> => <unary_op><expression> => <int_const> => <float_const> => <label> => PC => HIGH<expression> => LOW<expression> => OFFSET<expression>
<binary_op>	=> + => - => * => / => >> => << => & =>   => ^
<unary_op>	=> - => ~ => !

<int_const>	=> digit(digit '_' )@ => \$hexdigit(hexdigit '_' )@ => %bindigit(bindigit '_' )@ => <char_const>
<char_const>	=> 'any'
<float_const>	=> <decfloat> => <hexfloat>
<decfloat>	=> <float_frac> <float_power>
<float_frac>	=> <float_const>[<exp_part>]
<frac_const>	=> digit '_' ) . (digit '_' )@
<exp_part>	=> E['+' -]digit+
<float_power>	=> digit(digit '_' )@<exp_part>
<hexfloat>	=> \$hexdigit(hexdigit '_' )@ . (hexdigit '_' )@ [^[ '+' -]hexdigit+]
<label>	=> <ident>
<string_const>	=> "('\ " any)@"
<ident>	=> (letter '_' )(letter '_'  digit '.' )@
<ident_list>	=> <ident>(<ident> )@
<macro_def>	=> <ident>MACRO[<arg>(<arg> )] <code_body> ENDMAC[RO]<macname>
<macro_invoc>	=> <macname>[<arg> ](<arg> )]
<arg>	=> macro argument
<public_definition>	=> PUBLIC<ident list> => EXTERN<ident list>
<scalar_definition>	=> <type>[<value>]
<segment_definition>	=> DEFINE<ident>[<space_clause>] [<align_clause> ][<org_clause> ]



---

<space_clause>	=> ,SPACE=<space>
<align_clause>	=> ,ALIGN=<int_const>
<org_clause>	=> ,ORG=<int_const>
<space>	=> Valid microprocessor memory space
<target_microprocessor>	=> ZiLOG

A  
s  
s  
e  
m  
b  
l  
e  
r