



*SC100 Application Binary Interface Reference Manual*





MNSC100ABI/D  
Rev. 1.8, 4/2000

# SC100 Application Binary Interface

---

## Reference Manual



microelectronics group

**Lucent Technologies**  
Bell Labs Innovations



This document contains information on a new product. Specifications and information herein are subject to change without notice.

© Copyright Lucent Technologies Inc., 2000. All rights reserved.

© Copyright Motorola Inc., 2000. All rights reserved.

LICENSOR is defined as either Motorola, Inc. or Lucent Technologies Inc., whichever company distributed this document to LICENSEE. LICENSOR reserves the right to make changes without further notice to any products included and covered hereby. LICENSOR makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does LICENSOR assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation incidental, consequential, reliance, exemplary, or any other similar such damages, by way of illustration but not limitation, such as, loss of profits and loss of business opportunity. "Typical" parameters which may be provided in LICENSOR data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. LICENSOR does not convey any license under its patent rights nor the rights of others. LICENSOR products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support life, or for any other application in which the failure of the LICENSOR product could create a situation where personal injury or death may occur. Should Buyer purchase or use LICENSOR products for any such unintended or unauthorized application, Buyer shall indemnify and hold LICENSOR and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that LICENSOR was negligent regarding the design or manufacture of the part.

Motorola and  are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer. Lucent, Lucent Technologies, and (the Lucent Technologies insignia) are trademarks of Lucent Technologies Inc. Lucent Technologies Inc. is an Equal Opportunity/Affirmative Action Employer.

StarCore is a trademark of Motorola, Inc. It is used by Lucent Technologies with the authorization of Motorola.

All other tradenames, trademarks, and registered trademarks are the property of their respective owners.

# Table of Contents

---

## Chapter 1 Introduction

1.1	Purpose . . . . .	1-1
1.2	References . . . . .	1-1
1.3	Revision History . . . . .	1-2
1.4	Overview . . . . .	1-2
1.5	Conformance Levels . . . . .	1-3
1.6	Future Standards . . . . .	1-3

## Chapter 2 Low-Level Binary Interface

2.1	Underlying Processor Primitives . . . . .	2-1
2.2	Fundamental Data Types . . . . .	2-1
2.2.1	Compound Data Type . . . . .	2-3
2.2.1.1	Arrays . . . . .	2-3
2.2.1.2	Structures and Unions . . . . .	2-3
2.2.1.3	Bit Fields . . . . .	2-3
2.3	Function Calling Conventions . . . . .	2-4
2.3.1	Stack . . . . .	2-4
2.3.2	Stack-Based Calling Convention . . . . .	2-4
2.3.3	Optimized Calling Sequences . . . . .	2-6
2.3.4	Interrupt Handlers . . . . .	2-6
2.3.5	Stack Frame Layout . . . . .	2-6
2.3.6	Frame and Argument Pointers . . . . .	2-7
2.3.7	Dynamic Memory Allocation . . . . .	2-8
2.3.8	Hardware Loops . . . . .	2-8
2.3.9	Operating Modes . . . . .	2-8

## Chapter 3 High-Level Languages Issues

3.1	C Preprocessor Predefines . . . . .	3-1
3.2	C In-Line Assembly Syntax . . . . .	3-1
3.3	C Name Mapping . . . . .	3-1
3.4	Fractional Arithmetic Support . . . . .	3-2
3.4.1	Optional Prefix . . . . .	3-3
3.5	Libraries . . . . .	3-3
3.5.1	Compiler Assist Libraries . . . . .	3-3
3.5.2	Floating-Point Routines . . . . .	3-4
3.5.3	Integer Routines . . . . .	3-5

3.6	Function Argument and Return Type Checking in C . . . . .	3-5
3.6.1	Signature Symbols . . . . .	3-5
3.6.2	Return Value . . . . .	3-6
3.6.3	Using Signature Symbols . . . . .	3-6

## Chapter 4 SC100 ELF Object File Format

4.1	Formats . . . . .	4-1
4.2	Definitions . . . . .	4-1
4.3	Interface Descriptions . . . . .	4-1
4.3.1	The ELF Header . . . . .	4-2
4.3.2	Sections . . . . .	4-3
4.3.3	Reserved Names . . . . .	4-4
4.3.4	Relocation . . . . .	4-5
4.3.4.1	Relocation Expression Format . . . . .	4-5
4.3.4.2	Functions . . . . .	4-6
4.3.4.3	Special Identifiers . . . . .	4-8
4.3.4.4	Reserved Names . . . . .	4-8
4.3.4.5	Constants . . . . .	4-9
4.3.4.6	Operators . . . . .	4-9
4.3.4.7	Relocation Forms . . . . .	4-10
4.3.5	NOTE Section . . . . .	4-23
4.3.6	Program Headers . . . . .	4-24

## Chapter 5 Endian Support

5.1	Memory Organization . . . . .	5-2
5.1.1	SC140 Architecture . . . . .	5-3
5.1.2	Data Move . . . . .	5-4
5.1.3	Instruction Word Transfers . . . . .	5-6
5.2	Memory Access Behavior in Endian Modes . . . . .	5-7
5.3	Comments . . . . .	5-18
5.3.1	MOVE Multiple Registers . . . . .	5-18
5.3.2	MOVE.L for the Extension Registers . . . . .	5-18
5.3.3	PUSH/POP Instructions . . . . .	5-18
5.3.4	BSR/JSR . . . . .	5-18
5.3.5	Control instructions . . . . .	5-18

## Chapter 6 Assembler Syntax and Directives

6.1	Assembler-Significant Characters . . . . .	6-1
6.2	Assembler Directives . . . . .	6-2
6.2.1	Assembly Control . . . . .	6-2
6.2.2	Symbol Definition . . . . .	6-3
6.2.3	Data Definition/Storage Allocation . . . . .	6-3

6.2.4	Object File Control . . . . .	6-4
6.2.5	Macros and Conditional Assembly . . . . .	6-4
6.2.6	Assembler Syntax . . . . .	6-4
6.2.6.1	Input File Format . . . . .	6-4
6.2.6.2	Symbol Names . . . . .	6-5
6.2.6.3	Strings . . . . .	6-5
6.2.6.4	Source Statement Format . . . . .	6-6
6.2.6.5	Labels . . . . .	6-6
6.2.6.6	Operation Field . . . . .	6-6
6.2.6.7	Operand Field . . . . .	6-7
6.2.6.8	Comment Fields . . . . .	6-7





# List of Tables

---

2-1	Mapping of C Fundamental Data Types to SC100 . . . . .	2-2
2-2	Mapping of C Fractional Types to SC100 . . . . .	2-2
2-3	Register Usage in the Stack-Based Calling Convention . . . . .	2-5
3-1	Predefined Macros . . . . .	3-1
3-2	Required Intrinsic for Fractional Types . . . . .	3-2
3-3	Floating-Point Routines . . . . .	3-4
3-4	Integer Routines . . . . .	3-5
3-5	Italicized Fields in the Symbol Names . . . . .	3-5
3-6	Basetype Values . . . . .	3-6
4-1	SC100 ELF Sections . . . . .	4-3
4-2	Reserved Symbol Names . . . . .	4-4
4-3	Reserved Names for Relocation . . . . .	4-8
4-4	Group F1 Form . . . . .	4-10
4-5	Group F2 Forms . . . . .	4-10
4-6	Group F3 Forms . . . . .	4-11
4-7	Group F4 Forms . . . . .	4-11
4-8	Group F5 Forms . . . . .	4-12
4-9	Group F6 Forms . . . . .	4-12
4-10	Group F7 Forms . . . . .	4-13
4-11	Group F8 Forms . . . . .	4-13
4-12	Group F9 Forms . . . . .	4-14
4-13	Group F10 Forms . . . . .	4-14
4-14	Group F11 Forms . . . . .	4-15
4-15	Group F12 Forms . . . . .	4-16
4-16	Group F13 Forms . . . . .	4-16
4-17	Group F14 Forms . . . . .	4-17
4-18	Group F15 Forms . . . . .	4-18
4-19	Group F16 Forms . . . . .	4-19
4-20	Group F17 Forms . . . . .	4-19
4-21	Group F18 Forms . . . . .	4-20
4-22	Group F19 Forms . . . . .	4-21
4-23	Group F20 Forms . . . . .	4-21
4-24	Group F21 Forms . . . . .	4-22
4-25	Group F22 Forms . . . . .	4-22

4-26	Group F23 Forms .....	4-23
5-1	MOVE Instructions .....	5-7
5-2	Stack Support Instructions .....	5-10
5-3	Bit-Mask Instructions .....	5-11
5-4	Change of Flow Instructions .....	5-12
5-5	Control Instructions .....	5-13
5-6	Memory Access Instructions .....	5-14

# List of Figures

---

2-1	Fundamental Data Types .....	2-1
2-2	Stack Frame Layout .....	2-7
4-1	Object File Format .....	4-2
4-2	Vendor Identification Note Format.....	4-23
4-3	User (Application-Specific) Note Format.....	4-24
5-1	Memory Organization in Big/Little Endian Architecture.....	5-2
5-2	SC140 Basic Architecture.....	5-3
5-3	Data Transfer in Big/Little Endian .....	5-4
5-4	Multiple Data Transfer in Big/Little Endian.....	5-5
5-5	Program Memory Organization .....	5-6
5-6	Instruction Moves .....	5-6



# List of Examples

---

2-1	Structure Determined by Underlying Type . . . . .	2-3
2-2	1-Byte Field Offset . . . . .	2-4
4-1	ELF Header Structure . . . . .	4-2
4-2	SC140 Specifics . . . . .	4-2
4-3	Section Header Defined . . . . .	4-3
4-4	Relocation Entry Defined with Elf32_Rel . . . . .	4-5
4-5	Relocation Expression . . . . .	4-5
4-6	Relocation Form . . . . .	4-5
4-7	Relocation Attribute Setting . . . . .	4-6
4-8	Program Header . . . . .	4-24
6-1	Column 1 Labels . . . . .	6-6
6-2	Multiple-Line Instruction Group . . . . .	6-6



# Chapter 1

## Introduction

---

This manual defines the SC100 Application Binary Interface (ABI). The ABI is a set of interface standards that writers of compilers, assemblers, and debugging tools must use when creating tools for the SC100 architecture. These standards cover run-time aspects as well as object formats to be used by compatible tools chains. The standard definition ensures that all SC100 tools are compatible and can interoperate.

Although compiler support routines are provided, this manual does not describe how to develop SC100 tools, nor does it define the services provided by an operating system or a set of libraries. These components must be defined by tools, libraries, and operating system suppliers.

### 1.1 Purpose

The standards defined in this manual are intended to ensure that all development tool chains for the SC100 architecture are fully compatible. This ensures that the tools can interoperate, thus making it possible to select the best tool for each component of the application development tools chain. The standards in this manual also ensure that compatible libraries of binary components can be created and maintained. Developers can build up libraries over time with the assurance of continued compatibility.

### 1.2 References

The following documents provide useful reference information:

- *Executable and Linking Format Specification*, UNIX Systems Laboratories, Portable Formats Specification, Version 1.1
- *DWARF Debugging Information Format*, Unix International, Revision 2.0.0, July 27, 1993
- *ANSI/IEEE Std 754-1985*, IEEE standard for binary floating-point arithmetic data types
- *ELF-Executable and Linking Format* specification (the ELF spec)

The following documents provide StarCore-specific information:

- *SC100 Assembly Language Tools User's Manual* (order # MNSC100ALT/D)
- *SC140 DSP Core Reference Manual* (order # MNSC140CORE/D)
- *SC100 C Compiler User's Manual* (order # MNSC100CC/D)

## 1.3 Revision History

The following updates were made to Chapter 6, “Assembler Syntax and Directives,” for the April 2000 release of this manual:

- Added four new directives: SECFLAGS, SECTYPE, SIZE, and TYPE
- Removed the modulo storage directives: BSM and DSM
- Noted those directives that are not currently supported
- Changed the maximum length of symbol names and Assembler source statements to 4000 characters.

## 1.4 Overview

Standards in this manual cover the following SC100 tools components:

- Object modules generated by different tools chains
- Object modules and the SC100 architecture family of cores
- Object modules and source level debugging tools
- Debugging APIs

Current definitions include the following types of standards:

- Low level run-time binary interface standards
  - Processor-specific binary interface (the instruction set, representation of fundamental data types, and exception handling)
  - Function calling conventions (how arguments are passed and results are returned, how registers are assigned, and how the calling stack is organized)
- Object-file binary interface standards
  - Header convention
  - Section layout
  - Symbol table format
  - Relocation information format
  - Debugging information format

The SC100 object-file binary interface standards are based on ELF, as described in Section 4, “SC100 ELF Object File Format,” on page 4-1. All development tools for SC100 are required to use the following ELF-based standards.

- Source-level standards
  - C language (preprocessor predefines, in-line assembly and name mapping)
  - Assembler syntax and directives
- Library standards
  - Compiler run-time libraries



## 1.5 Conformance Levels

The ABI interface standards define two levels of conformance:

- Level 0                      Features that are classified as Level 0 are mandatory. Features described in this document are classified as Level 0, unless specifically stated otherwise.
- Level 1                      Features that are classified as Level 1 are optional. SC100 development tools are not required to implement Level 1 features. Those development tools that do provide functionality that is covered by Level 1 ABI features are required to conform to the standards.



# Chapter 2

## Low-Level Binary Interface

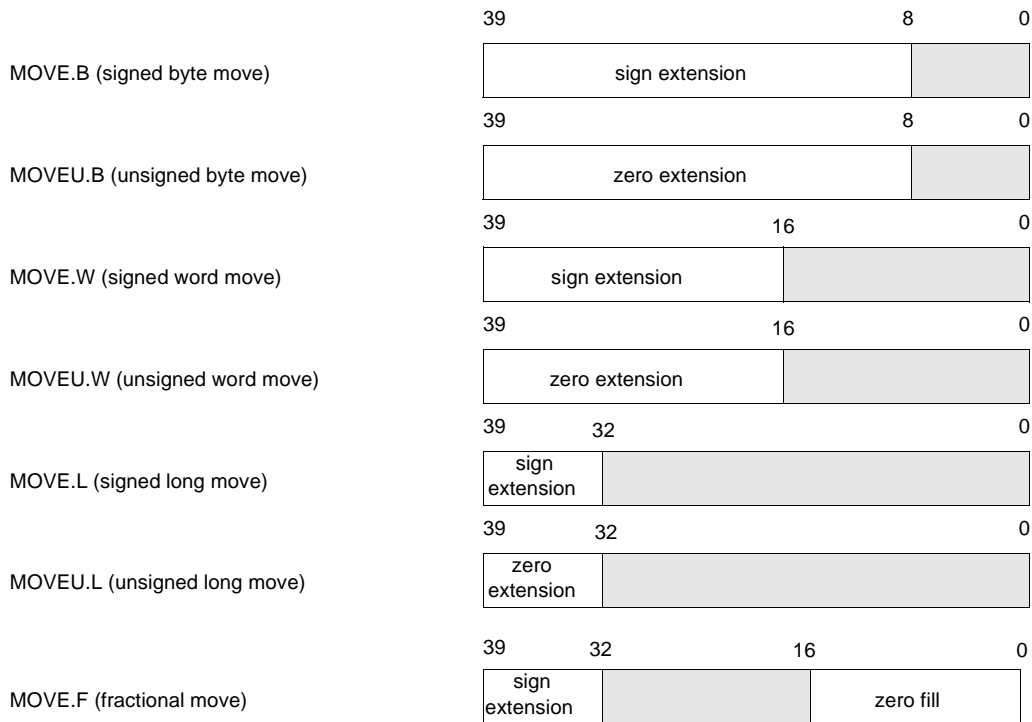
---

### 2.1 Underlying Processor Primitives

For a complete description of the SC100 architecture, refer to the *SC140 DSP Core Reference Manual*.

### 2.2 Fundamental Data Types

The SC100 architecture supports the fundamental data types shown in Figure 2-1. The mapping between these data types and the C language fundamental data type is shown in Table 2-1. Fractional types are supported in C using intrinsic functions; Table 2-2 shows the fractional types supported.



**Figure 2-1. Fundamental Data Types**

**Table 2-1. Mapping of C Fundamental Data Types to SC100**

Type	Size (mem)	Size (Dreg)	Size (Rreg)	Align	Limits
char	8	40	32	8	-128..127
unsigned char	8	40	32	8	0..255
short	16	40	32	16	-0x8000..0x7fff
unsigned short	16	40	32	16	0.0xffff
int	32	40	32	32	-0x80000000..0x7fffffff
unsigned	32	40	32	32	0..0xffffffff
long	32	40	32	32	-0x80000000..0x7fffffff
unsigned long	32	40	32	32	0..0xffffffff
float double long double (24-bit mantissa)	32	40	32	32	-1.17e-38..1.17e+38
pointer	32	40	32	32	0..0xffffffff

**Table 2-2. Mapping of C Fractional Types to SC100**

Type	C Type Definition	Size (mem)	Size (Dreg)	Size (Rreg)	Align	Limits
fractional	short	16	40	32	16	-1.0 .. 0.99999
long fractional	long or int	32	40	32	32	-1.0 .. 0.999999
long fractional with extension bits	typedef struct { int lsb; char guard; } xfrac;	64	40	NA	32	-16.0 .. 15.999999
double precision fractional	typedef struct { int lsb; int msb; } dfrac;	64	2x40	NA	32	-1.0 .. 0.999999

The SC100 architecture uses little-endian memory representation byte ordering. The lowest addressable byte of a memory location always contains the least significant bits of the value. Fundamental data is always naturally aligned: a long word is 4-byte aligned, and a short word is 2-byte aligned.

## 2.2.1 Compound Data Type

Arrays, structures, unions and bit fields have different alignment characteristics, as described in the following sections.

### 2.2.1.1 Arrays

Arrays have the same alignment restriction as their individual elements.

### 2.2.1.2 Structures and Unions

Members of unions and structures have the most restrictive alignments. For example, a structure containing a char, a short, and a long word must have a 4-byte alignment to match the alignment of the long field. In addition, the size of a union or structure must be an integral multiple of its alignment. Padding must be applied to the end of a union or structure to make its size a multiple of the alignment. To meet this alignment requirement, padding must be introduced between members as necessary.

### 2.2.1.3 Bit Fields

Members of structures are always allocated on byte boundaries. However, bit fields in structures can be allocated at any bit and can be of any length that does not exceed the size of a long word (32-bits). Signed and unsigned bit fields are permitted and are sign-extended when fetched. A bit field of type int is considered signed. Bit fields will be allocated from the low-order end of a word (right to left, or little endian). Bit field sizes are not allowed to cross a long word boundary.

In Example 2-1, the structure `more` has 4-byte alignment and will have the size of 4-bytes. This is because the fundamental type of the bit fields is long, which requires a 4-byte alignment. The second structure `less` requires only a 1-byte alignment because that is the requirement of the fundamental type (char) used in that structure. The alignments are driven not by the width of the fields but by the underlying type. These alignments are to be considered along with any other structure members. Structure `careful` requires a 4-byte alignment; its bit fields only require 1-byte alignment, but the field `fluffy` requires a 4-byte alignment.

#### Example 2-1. Structure Determined by Underlying Type

---

```
struct more {
    long first : 3;
    unsigned int second : 8;
};

struct less {
    unsigned char third : 3;
    unsigned char fourth : 8;
};

struct careful {
    unsigned char third : 3;
    unsigned char fourth : 8;
    long fluffy;
};
```

---

Fields within structures and unions begin on the next possible suitably-aligned boundary for their data type. For non-bit fields, this is a suitable byte alignment. Bit fields begin at the next available bit offset with the following exception: the first bit field after a non-bit field member will be allocated on the next available byte boundary.

In Example 2-2, the offset of the field `c` is 1 byte. The structure itself has 4-byte alignment and is 4 bytes in size because of the alignment restrictions introduced by using the underlying `int` data type for the bit field.

---

**Example 2-2. 1-Byte Field Offset**

---

```
struct s {  
    int bf: 5;  
    char c;  
}
```

---

## 2.3 Function Calling Conventions

SC100 compilers must support a stack-based calling convention. Additional calling conventions may be supported. Calling conventions can be mixed within a single application.

A pragma interface should be used to set the calling convention for a function to one that is different from the stack-based calling convention. This change will affect both the calling sequence and prologue/epilog of these functions, if a function body is present. This will allow both calling and being called by functions compiled with different compilers.

### 2.3.1 Stack

The SP register serves as the stack pointer. SP will point to the first available location, with the stack direction being towards higher addresses (i.e., a push will be implemented as “(sp)+”). The stack pointer must be 8-byte aligned.

### 2.3.2 Stack-Based Calling Convention

The calling conventions described in the following paragraphs must be supported.

The first (left-most) function parameter, regardless of size, will be passed in `d0` (if a numeric scalar) or in `r0` (if an address parameter). The second function parameter, regardless of size, will be passed in `d1` (if a numeric scalar) or in `r1` (if an address parameter). The rest of the parameters will be pushed into the stack. Parameters will be pushed on the stack using little-endianess (least significant bits in lower addresses).

Structures and union objects that can fit in a register are treated as numeric parameters and therefore are candidates to be passed in a register.

Numeric return values will be returned in `d0`. Numeric address return values will be returned in `r0`. Functions returning large structures (i.e., structures that do not fit in a single register) will receive and return the returned structure address in `r2`. The caller allocates the space for the returned object.

Functions with a variable number of parameters will allocate all parameters on the stack.

All parameters of size smaller or equal to 4 bytes will be aligned in memory to a 4-byte boundary.

The following registers will be saved by the caller: `d0-d5`, `d8-d15`, `r0-r5`, `n0-n3`.

The following registers will be saved by the callee, if actually used: d6-d7,r6-r7.

The compiler should assume that the current settings of the following operating control bits are correct:

- Saturation mode
- Round mode
- Scale bits

Setting these mode bits is under the application responsibility. For example, for the function call:

```
foo(int a1, struct fourbytes a2, struct eightbytes a3, int *a4)
```

parameters will be allocated as follows:

```
a1 - in register d0.
a2 - in register d1.
a3 - on the stack.
a4 - on the stack.
```

For the call

```
bar(long *b1, int b2, int b3[])
```

parameters will be allocated as follows:

```
b1 - in r0
b2 - in d1
b3 - in stack.
```

The stack-based calling convention must be used when calling functions that are required to maintain a calling stack.

The compiler is free to use optimized calling sequences for functions that are not exposed to external calls.

Locals and formals will be allocated on the stack and in registers.

Table 2-3 summarizes register usage in the stack-based calling convention.

**Table 2-3. Register Usage in the Stack-Based Calling Convention**

Register	Used As	Caller Saved	Callee Saved
d0	First numeric parameter Return numeric value	+	
d1	Second numeric parameter	+	
d2–d5		+	
d6–d7			+
d8–d15		+	
r0	First address parameter Return address value	+	
r1	Second address parameter	+	
r2	Big return object address	+	
r3–r5		+	
r6	Optional argument pointer		+

**Table 2-3. Register Usage in the Stack-Based Calling Convention (Continued)**

Register	Used As	Caller Saved	Callee Saved
r7	Optional frame pointer		+
n0–n3 m0–m3		+	

### 2.3.3 Optimized Calling Sequences

A stackless convention may be used when calling functions that are not re-entrant, if this technique generates more efficient code than other conventions. However, this convention can be used only if the function is not visible to external code. When using the stackless convention, locals may be allocated statically (i.e., not on a stack). Functions whose lifetime is mutually exclusive may share space for their locals.

The calling function places actual parameters at locations allocated in the called function for the formal parameters. The compiler is free to use registers and memory locations when allocating locations for the formal parameters. Under this calling convention, all registers are classified as caller-saved. Return values from functions will be placed in the space allocated in the calling function for the function return value. The compiler is free to use a register or a memory location as the space for the function return value.

### 2.3.4 Interrupt Handlers

Using an implementation-dependent pragma, functions requiring no parameters and returning no result can be designated as interrupt functions. The interrupt handler function will always follow the stack-based calling convention. When an interrupt function is called, the interrupt handler will save all registers and other resources that are modified by the function. Upon returning from the function, all registers and hardware loop state saved at entry will be restored to their original state.

Locals will be saved on the stack. Interrupt handlers that are known to be noninterruptible may allocate data statically as well. Return from interrupt will be implemented using an RTE instruction.

### 2.3.5 Stack Frame Layout

The stack pointer points to the top (high address) of the stack frame. Space at higher addresses than the stack pointer is considered invalid and may actually be unaddressable. The stack pointer value must always be a multiple of eight.

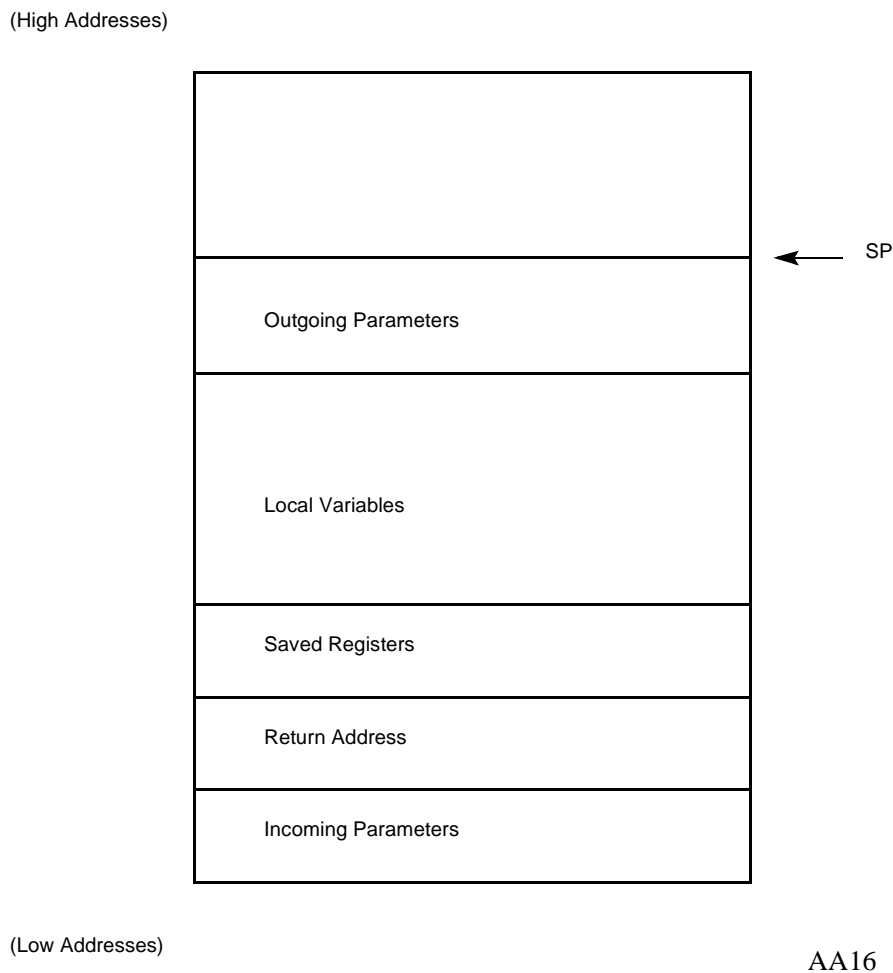
Figure 2-2 shows typical stack frames for a function and indicates the relative position of local variables, parameters, and return addresses. The outbound argument block is located at the top (higher addresses) of the frame. Any incoming argument spill generated for varargs and stdags processing must be at the bottom (lower addresses) of the frame.

The caller puts argument variables that do not fit in registers into the outbound argument overflow area. If all arguments fit in registers, this area is not required. A caller may allocate argument overflow space sufficient for the worst-case call, use portions of it as necessary, and not change the stack pointer between calls.

Local variables that do not fit into the local registers are allocated space in the local variables area of the stack. If there are no such variables, this area is not required.



For return variables that do not fit in registers, the caller must reserve stack space. This return buffer area is typically located with the local variables. This space is typically allocated only in functions that make calls returning structures. Beyond these requirements, a function is free to manage its stack frame in any way desired.



**Figure 2-2. Stack Frame Layout**

## 2.3.6 Frame and Argument Pointers

The ABI standard does not require the use of a frame pointer or an argument pointer. If, however, the use of a frame pointer or an argument pointer is necessary, a compiler may allocate R7 as a frame pointer and R6 as an argument pointer. When these registers are allocated for this purpose, they should be saved and restored as part of the function prologue/epilog code.

## 2.3.7 Dynamic Memory Allocation

Dynamic allocations are implemented using a heap structure managed by the standard library functions `malloc()` and `free()`. The heap shall be allocated statically by the linker. In typical configurations, the stack is allocated above (higher addresses) the static and global variables, growing towards higher addresses. The heap is allocated at the top of the available memory, growing towards lower memory addresses.

## 2.3.8 Hardware Loops

All hardware loop resources are available for the compiler's use. As it is assumed that no nesting occurs when entering a function, a function may use all four nesting levels for its own use. An additional side effect of this assumption is that loops with a function call as part of the loop code cannot be implemented using hardware loops. A compilation switch will be available to disable the use of loop 3. As in interrupts, this will enable the user to allocate this loop counter to a different execution thread.

Loops will be nested beginning loop counter 0 in the outermost nesting level.

## 2.3.9 Operating Modes

Compilers should make two assumptions about run-time operating modes and machine state:

1. All M registers (m0–m3) should be assumed to contain the value -1 (linear addressing). Should the use of an M register be required, the using function must restore the M register to the value -1 before returning or before calling another function.
2. No particular operating mode settings in the OMR register are assumed. It is expected that the user will set the default settings in the start-up code (saturation modes, rounding mode, and scale bits). These operating modes may change during the application execution under user control.

# Chapter 3

## High-Level Languages Issues

---

### 3.1 C Preprocessor Predefines

All C/C++ language compilers must have predefined macros as in Table 3-1, in addition to the predefined macros required by the C and C++ language standards.

Table 3-1. Predefined Macros

Macro	Explanation
<code>__SC100__</code>	Defined for SC100-based compilers.

### 3.2 C In-Line Assembly Syntax

A C in-line assembly syntax must be provided. At a minimum, a single line assembly has to be supported:

```
asm("wait");
```

### 3.3 C Name Mapping

Externally visible names in the C language are prefixed by “\_” when generating assembly language symbol names. For example, the following:

```
void testfunc()
{
    return;
}
```

generates assembly code similar to the following fragment:

```
_testfunc:
    rts
```

## 3.4 Fractional Arithmetic Support

The compiler must support the intrinsic functions listed in Table 3-2.

**Table 3-2. Required Intrinsics for Fractional Types**

Intrinsic Function	Description
Fractional Arithmetic	
int add(int,int)	Short add
int sub(int,int)	Short sub
int mult(int,int)	Short multiplication
int div_s(int,int)	Short Div
int add_r(int,int)	Short add with round
int sub_r(int,int)	Short sub with round
int mult_r(int,int)	Multiply with round
long L_mac(long,int,int)	Multiply Accumulate
long L_macNs(long,int,int)	Multiply accumulate with no saturation
int mac_r(long,int,int)	Multiply accumulate with round
long L_msu(long,int,int)	Multiply Subtract
long L_msuNs(long,int,int)	Multiply subtract with no saturation
int msu_r(long,int,int)	Multiply subtract with round
int abs_s(int)	Short abs
int negate(int)	Short negate
int round(long)	Round
int shl(int,int)	Short shift left
int shr(int,int)	Short shift right
int shr_r(int,int)	Short shift right with round
int norm_s(int)	Normalize any fractional value
Long Fractional Arithmetic	
long L_add(long,long)	Long add
long L_sub(long,long)	Long subtract
long L_mult(int,int)	Long multiplication

**Table 3-2. Required Ininsics for Fractional Types (Continued)**

Intrinsic Function	Description
int extract_h(long)	Extract high
int extract_l(long)	Extract Low
long L_deposit_h(int)	Deposit short in MSB
long L_deposit_l(int)	Deposit short in LSB
long L_abs(long)	Long Abs
long L_negate(long)	Long negate
int norm_l(long)	Normalize any long fractional value
long L_shl(long,int)	Long shift left
long L_shr(long,int)	Long shift right
long L_shr_r(long,int)	Long shift right with round
long L_sat(long)	Long saturation

### 3.4.1 Optional Prefix

In supporting the intrinsic functions listed in Table 3-2, the compiler can either recognize the function names as listed in the table, or recognize them using the “\_\_” (a double underscore) prefix. A compiler that supports this prefix must provide a header file that maps the names as listed in the table to the prefixed names.

## 3.5 Libraries

The following sections provide details on support libraries.

### 3.5.1 Compiler Assist Libraries

The SC100 architecture does not provide hardware support for floating-point data types, nor for divide functionality for integer types. Compilers should provide the functionality for some of these operations through the use of support library routines.

The functions to be provided through support library routines include the following:

- Floating-point math routines
- Integer divide routines
- Integer modulo routines

Compilers that generate in-line code to provide these functions must make no reference to the library functions. Compilers that provide these functions by generating function calls to the support libraries must use the stack-based calling convention when calling them.

To ensure the ability to link code produced by different compilers into a single executable, it is required that names of compiler support library functions match those listed in Table 3-3 and Table 3-4.

Routines in support libraries must satisfy the following constraints:

- The only external state information used is floating-point operation mode (rounding mode, flush to zero, etc.).
- No other global state can be modified.
- Identical results must be returned when a routine is reinvoked with the same input arguments.
- Multiple calls with the same input arguments can be collapsed into a single call with a cached result.

These properties permit a compiler to make assumptions about variable lifetimes across library function calls: values in memory will not change, previously dereferenced pointers need not be referenced again.

## 3.5.2 Floating-Point Routines

These routines should comply with the stack-based calling conventions.

The data formats are as specified in IEEE-754. The math routines are not required to compute results as specified in IEEE-754. Implementation of these routines must document the degree to which operations conform to the IEEE standard. Not all users of floating point require IEEE-754 precision and exception handling, and may not want to incur the overhead that complete conformance requires.

SpFloat is used in Table 3-3 and Table 3-4 to represent a 32-bit basic type.

**Table 3-3. Floating-Point Routines**

Function Prototype	Description
SpFloat _f_add( SpFloat a, SpFloat b );	Returns the value of a+b
SpFloat _f_sub( SpFloat a, SpFloat b)	Returns the value of a-b.
SpFloat _f_mul( SpFloat a, SpFloat b)	Returns the value of a*b.
SpFloat _f_div ( SpFloat a, SpFloat b)	Returns the value of a/b.
void _f_feq ( SpFloat a, SpFloat b)	Sets the T bit in the status register if a=b.
void _f_fge ( SpFloat a, SpFloat b)	Sets the T bit in the status register if a>=b.
void _f_fgt ( SpFloat a, SpFloat b)	Sets the T bit in the status register if a>b.
SpFloat _f_itof( long a)	Converts a 32-bit signed integer value to floating-point representation.
SpFloat _f_ufou( unsigned long a)	Converts a 32-bit unsigned integer value to floating-point representation.
long _f_ftoi( SpFloat a)	Converts a floating-point value to a 32-bit signed integer representation.
unsigned long _f_ftou( SpFloat a)	Converts a floating-point value to a 32-bit unsigned integer representation.

### 3.5.3 Integer Routines

The integer routines listed in Table 3-4 should comply with the ABI calling conventions. The routines have no side effects.

**Table 3-4. Integer Routines**

Function Prototype	Description
<code>int __div16(short a, short b);</code>	Returns the value of $a/b$ .
<code>int __udiv16(unsigned short a, unsigned short b);</code>	Returns the value of $a/b$ (unsigned).
<code>int __div32(long a, long b);</code>	Returns the value of $a/b$ .
<code>int __udiv32(unsigned long a, unsigned long b);</code>	Returns the value of $a/b$ (unsigned).
<code>int __rem16(short a, short b);</code>	Returns the value of $a \bmod b$ .
<code>int __urem16(unsigned short a, unsigned short b);</code>	Returns the value of $a \bmod b$ .
<code>int __rem32(long a, long b);</code>	Returns the value of $a \bmod b$ .
<code>int __urem32(unsigned long a, unsigned b);</code>	Returns the value of $a \bmod b$ (unsigned).

## 3.6 Function Argument and Return Type Checking in C

Level 1 conforming implementations support the following mechanism for checking that arguments and return types of function calls match the called functions' signatures.

### 3.6.1 Signature Symbols

For every direct call to a non-static function in a source file (i.e., a call using the function name as opposed to a call through a function pointer), the compiler system produces in the ELF object file a symbol of the following convention:

```
__caller.name.return_type.parameter_types
```

For every non-static function definition, the compiler system produces a symbol of the following convention:

```
__callee.name.return_type.parameter_types
```

Table 3-5 explains the construction of the italicized fields in the symbol names:

**Table 3-5. Italicized Fields in the Symbol Names**

Field	Value	Description
<code>name</code>	ASCII string	The name of the called function
<code>return_type</code>	<code>basetype</code>	
<code>parameter_types</code>	<code>basetype[basetype[...]]</code>	

Table 3-6 explains the possible values for `basetype`.

**Table 3-6. Basetype Values**

Code	Definition
i	scalar type (e.g. char, short, int) of size <= 32 bits, passed in register
l	scalar type of size = 64 bits, passed in register
p	Pointer, passed in a register
f	Float, passed in a register
d	Double float, passed in a register
<i>snum</i>	Struct, passed in a data register
<i>anum</i>	Struct, passed in an address register
n	A parameter passed on the stack
v	Void
x	Start of a variable argument list (...)

Example:

Definition:

```
int foo(struct { int a,b; } parm1, double parm2);
```

Call:

```
struct { int a,b; } tmp;
foo(tmp, 1.0);
```

Special Symbols:

```
__callee.foo.i.s2f
__caller.foo.v.s2f
```

## 3.6.2 Return Value

In generating a signature symbol for a call to a function defined as returning a (non-void) value, if the return value is ignored by the caller, then the compiler may specify `i` as the return value type for the function.

## 3.6.3 Using Signature Symbols

The caller/callee match verification using signature symbols is implementation-dependent. The implementation must accept object modules that do not contain signature symbols.



# Chapter 4

## SC100 ELF Object File Format

---

### 4.1 Formats

The executable and linking format (ELF) is used for representing the binary application to the system. For a complete description of ELF, see the *ELF-Executable and Linking Format* specification (the ELF spec). This section highlights differences between the ELF v1.1 definition and the SC100 ELF implementation.

### 4.2 Definitions

This section describes the interface for relocatable and executable programs. A relocatable program contains code suitable for linking to create another relocatable program or executable program. An executable program contains binary information suitable for loading and execution on a target processor.

### 4.3 Interface Descriptions

ELF presents two views of binary data, as shown in Figure 4-1:

- The *linking* view provides data in a format suitable for incremental linking into a relocatable file or final linking to an executable file.
- The *execution* view provides binary data in a format suitable for loading and execution.

An ELF header is always present in either view of the ELF file. For the linking view, sections are the main entity in which information is presented. A section header table provides information for interpretation and navigation for each section. For the execution view, segments are the primary sources of information. Sections may be present but are not required. A program header table provides information for interpretation and navigation through each segment. For exact details, see the ELF spec.

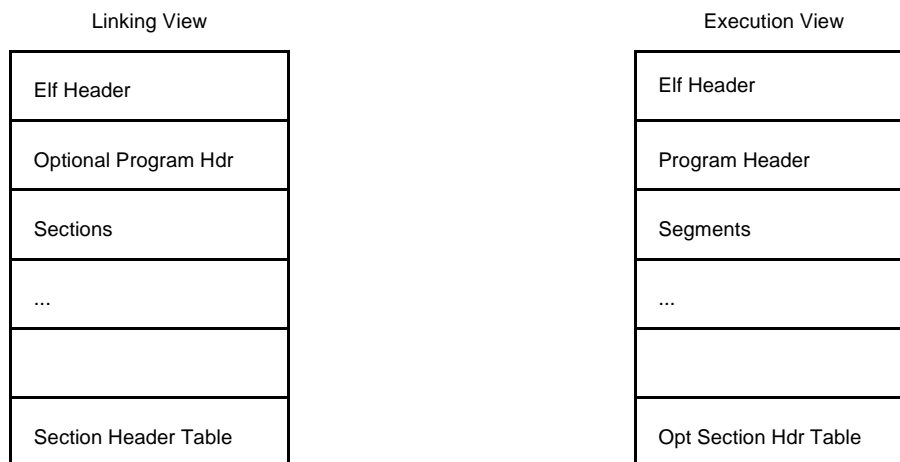


Figure 4-1. Object File Format

### 4.3.1 The ELF Header

The ELF header structure is shown in Example 4-1. This structure is defined by the ELF spec, and definitions for each field can be found in the ELF spec. Example 4-2 shows SC140-specific code.

#### Example 4-1. ELF Header Structure

```
typedef struct {
    unsigned char e_ident[EI_NIDENT];
    Elf32_Half e_type;
    Elf32_Half e_machine;
    Elf32_Word e_version;
    Elf32_Addr e_entry;
    Elf32_Off e_phoff;
    Elf32_Off e_shoff;
    Elf32_Word e_flags;
    Elf32_Half e_ehsize;
    Elf32_Half e_phentsize;
    Elf32_Half e_phnum;
    Elf32_Half e_shentsize;
    Elf32_Half e_shnum;
    Elf32_Half e_shstrndx;
} Elf32_Ehdr;
```

#### Example 4-2. SC140 Specifics

```
e_ident[EI_CLASS] = ELFCLASS32
e_ident[EI_DATA] = ELFDATA2LSB (Little-endian memory mode)
e_ident[EI_DATA] = ELFDATA2MSB (Big-endian memory mode)
e_machine: 0x3a (EM_STARCORE)
```

## 4.3.2 Sections

Sections are the main components of the ELF file. Section headers define all the information about a section. A section header is defined in Section 4-3. It is identical to the ELF V1.1 definition.

### Example 4-3. Section Header Defined

```
typedef struct {
    Elf32_Word sh_name;
    Elf32_Word sh_type;
    Elf32_Word sh_flags;
    Elf32_Addr sh_addr;
    Elf32_Off  sh_offset;
    Elf32_Word sh_size;
    Elf32_Word sh_link;
    Elf32_Word sh_info;
    Elf32_Word sh_addralign;
    Elf32_Word sh_entsize;
} Elf32_Shdr;
```

Sections used in SC100 ELF binaries are shown in Table 4-1.

**Table 4-1. SC100 ELF Sections**

Name	Type	Flags	Purpose	Comments
<section_name>	PROGBITS	ALLOC/EXEC	Named program section	
<section_name>	NOBITS	ALLOC/EXEC	Overlay section component	Defined run time overlay section characteristics. sh_info contains overlay count.
<section_name>	OVERLAY <sup>1</sup>	ALLOC	Overlay section component	Defines load time overlay section characteristics. sh_info contains overlay count.
.text	PROGBITS	ALLOC/EXEC	Section data	Assembler physical section containing executable instructions.
.rel.text	REL	ALLOC	Relocation info	See Section 4.3.4, "Relocation."
.data	PROGBITS	ALLOC	Initialized data section	
.rodata	PROGBITS	ALLOC	Read-only, initialized data	
.bss	NOBITS	ALLOC	Uninitialized data section	
.symtab	SYMTAB	ALLOC	Symbol table	
.shstrtab	STRTAB	NONE	Section header string table	
.strtab	STRTAB	ALLOC	General string table	

**Table 4-1. SC100 ELF Sections (Continued)**

Name	Type	Flags	Purpose	Comments
.note	NOTE	ALLOC	File identification	
.line	PROGBITS	NONE	Assembly debug line number info	This information in DWARF format.
.rel.line	REL	NONE	Assembler line number relocation info	
.debug_macro	PROGBITS	NONE	Assembly debug macro information	This information in DWARF format.
.debug_info	PROGBITS	NONE	C/C++ debug information	This information in DWARF format.
.debug_abbrev	PROGBITS	NONE	Abbreviation tables	This information in DWARF format.
.debug_line	PROGBITS	NONE	Line number information	This information in DWARF format.
.debug_aranges	PROGBITS	NONE	Address range table	This information in DWARF format.
.debug_pubname	PROGBITS	NONE	Global names table	This information in DWARF format.
.hash	HASH	NONE	Symbol hash table information	

1. Processor-specific section type.

### 4.3.3 Reserved Names

Symbol names listed in Table 4-2 (upper- and lower-case) are reserved for the system.

**Table 4-2. Reserved Symbol Names**

Names	
.bss	.global
.data	.hash
.debug	.line
.debug_abbrev	.note
.debug_aranges	.rel.text
.debug_info	.rodata
.debug_line	.shstrtab
.debug_macro	.strtab
.debug_pubnames	.symtab
.etext	.text

## 4.3.4 Relocation

Each section which contains relocatable data has a corresponding relocation section of SHT\_REL. The sh\_info field of the relocation section defines a section header index of the section to which the relocation applies. The sh\_link field of the rel section defines the associated symbol table. The relocation entry definition is modified for the SC140. The r\_offset field defines an address to which the relocation applies. The r\_info field specifies an offset into the string table of an expression, which, when evaluated, yields the proper relocation data at the address given by r\_offset.

A relocation entry is defined by Elf32\_Rel in Example 4-4.

### Example 4-4. Relocation Entry Defined with Elf32\_Rel

---

```
typedef struct {
    Elf32_Addr r_offset;
    Elf32_Word r_info;
} Elf32_Rel;
```

---

#### 4.3.4.1 Relocation Expression Format

Link file data expressions are generated when external or relocatable operands are encountered during assembly or incremental link processing. For example, consider the assembly source in Example 4-5:

### Example 4-5. Relocation Expression

---

```
move.w (sp+LA),d0
```

The SC100 Assembler produces the following relocation expression for this line in the object file:

```
"check(sym(7),u,16), op | F1(sym(7))"          (1)
"check(sym(7),u,16), op | F2(sym(7))"          (2)
```

---

Since the value of symbol LA is not known to the assembler, it generates a two-word instruction and places a relocation reference to the strings in (1) and (2) in the relocation entry section data. The strings in (1) and (2) are called relocation expressions or relocations. A relocation is composed of legal C operators, keyword identifiers defined by this document, and constant values. Note that the quotes are shown here to represent a sequence of printable characters in the object file's string table. The quotes are not required in this context. The sequence of characters are not terminated by any special characters.

Relocations are case-sensitive. Symbols are referenced by the function "sym," whose sole argument is the symbol table index of the symbol from its respective compilation unit. Hence, "sym(7)" indicates that "LA" is symbol number 7 in the symbol table for this example. "F1" and "F2" are architecture-dependent encoding forms, whose definitions are specific to the SC140 DSP core.

The general form of a relocation is described by Example 4-6.

### Example 4-6. Relocation Form

---

```
"field,field,...,field"
```

---

The final field in the sequence, described in Example 4-6, is the value field. The value field, after it is evaluated, determines the contents of the memory location. The other fields of the relocation may set attributes of the memory location and may do general or architecture-based checking as shown in Example 4-7.

#### **Example 4-7. Relocation Attribute Setting**

---

```
"check(sym(5),s,8),op|F5(sym(5))"
```

---

The relocation in Example 4-7 performs a check of symbol number 5 to make sure that its signed value will fit into a bit field 8 bits wide. The memory location value is calculated by bit-wise ORing the section data (represented by the pre-defined identifier “op”) and the value generated by the architecture-dependent function, “F5.”

### **4.3.4.2 Functions**

Functions are categorized as generic or architecture-specific. Within the architecture-specific category, functions may be further categorized as encoding or other.

Following are descriptions of the functions used for relocations.

#### **4.3.4.2.1 Check Function**

Category: generic

Arguments: value, (s | u), <len> [,<align>=0]

where:

s=signed range check

u=unsigned range check

Description: This function checks that the value passed will fit into the number of bits specified by <len>. The second argument specifies the type of value, signed or unsigned integer. The optional parameter, <align>, specifies the number of constant least significant bits in value. These constant bits are disregarded when calculating whether the value will fit into <len> bits. The <align> argument, if omitted, defaults to 0 (no alignment).

Returns: 0, generates error is value is out of range.

#### **4.3.4.2.2 Pack Function**

Category: generic

Arguments: value, (s | u | n), <len> [,<align>=0]

where:

s=signed range check

u=unsigned range check

n=truncate to range

Description: This function is similar to the check function. It returns the significant bits which can be placed in a <len> size bit field. If the second argument is ‘n’, the value is truncated to the number of <len> bits and no error is generated.

Returns: <len> significant bits of value, generates error is value is out of range (s or u) or truncated value if second argument is ‘n’.

#### 4.3.4.2.3 Sym Function

Category: generic  
Argument: <index>  
Description: This function simply returns the symbol table value of the symbol at <index>.  
Returns: value of symbol

#### 4.3.4.2.4 Size Function

Category: generic  
Argument: <num\_bytes>  
Description: This function sets the size of the memory target word to which the relocation applies to <num\_bytes>. The memory target word defaults to the architecture word size if no size function is specified. The SC100 default word size is 2 bytes.  
Returns: 0

#### 4.3.4.2.5 Line Function

Category: generic  
Arguments: <line> [,<file>]  
Description: This function generates line number information for the relocation. An optional string literal, <file>, may be passed which identifies the file from which the instruction comes, such as an included file.  
Returns: 0

#### 4.3.4.2.6 Hi Function

Category: generic  
Arguments: value  
Description: This function returns the high bits of a data value. The data value is masked and shifted into the lower bits. The number of bits masked is determined by the size function or the default target memory word size.  
Returns: masked and shifted value  
Example: hi(0x1234) is equivalent to 0x0012 for size(2).

#### 4.3.4.2.7 Lo Function

Category: generic  
Arguments: value  
Description: This function returns the low bits of a data value. The data value is masked, the remaining value remains in the lower bits. The number of bits masked is determined by the size function.  
Returns: masked value  
Example: lo(0x12345678) is equivalent to 0x00005678 for size(4).

#### 4.3.4.2.8 Memcheck Function

Category: SC100-specific

Arguments: value, <memid>

Description: This function verifies certain memory constraints of the symbol value. <memid> may be 0 (no memory constraints) or 1 (target memory location must be located in peripheral address space).

Returns: 0, generates errors if memory constraint violation

#### 4.3.4.3 Special Identifiers

The following special identifiers are used by relocations:

Op Identifier This identifier contains the base opcode value. This is value of the memory location (section data value) to which the relocation applies before any fixups are performed.

PC Identifier This identifier contains the current value of the program counter. This allows calculation of program counter relative offsets.

PA Identifier Relative branches are based on the first address of the packet. This identifier provides the first of address of the current execution packet.

#### 4.3.4.4 Reserved Names

The following list of names are reserved for use in relocation. ABI-conforming applications should not use these names except for their defined purposes.

**Table 4-3. Reserved Names for Relocation**

Names	
check	op
memcheck	pc
pack	pa
sym	s
hi	u
lo	n



### 4.3.4.5 Constants

Integer constants are valid in relocations. Integer constants are recognized as a sequence of the following characters:

[0-9]	Decimal integer
0x[0-f]	Hexadecimal integer
0[0-7]	Octal integer
0b[0-1]	Binary integer

String literals are valid in relocations. String literals are enclosed in double-quotes. All characters within double-quotes are treated literally; there is no provision for escape sequences.

### 4.3.4.6 Operators

The operators following operators are legal in relocations (from highest to lowest precedence):

( )	parentheses
!, ~, -	unary
*, /, %	multiplicative
+, -	additive
<<, >>	shift
<, <=, >, >=	relational
==, !=	equality
&	bitwise and
^	bitwise exclusive or
	bitwise or
&&	logical and
	logical or
?:	conditional
=, +=, -=, *=, /=, %=	assignment
,	comma

All binary operators associate from left to right, except for conditional and assignment operators which associate from right to left. The operators' functions are the same as the C language.

### 4.3.4.7 Relocation Forms

Device-specific *forms* describe how a memory location should be patched by the linker. Forms are a combination of functions, operators, and constants which completely describe how a memory location's value will be calculated from relocatable symbols. A conforming assembler should generate the relocation forms described in this section.

Typically, it takes more than one form to encode an average instruction. For example, the group F5 requires four member forms to completely describe how the instruction in (5) should be patched when both of its operands are external.

```
'313 BMCLR.W #{iU16B0}, <({AU16W0})' (5)
```

**Table 4-4. Group F1 Form**

Group Name	<b>F1</b>
Arguments	value, (s u), <len>, <align>
Member	F1W1
Description	Encoding function for word 1 of instruction
Definition	"value & mask(0,10)"
Applicable Instructions	'BRA{D} <+{AS10W1}'

**Table 4-5. Group F2 Forms**

Group Name	<b>F2</b>	
Arguments	value, (s u), <len>, <align>	
Member	<b>F2W3</b>	
	Description	Encoding function for word 3 of the instruction
	Definition	"0xffff & mask(0,13)"
Member	<b>F2W2</b>	
	Description	Encoding function for word 2 of the instruction
	Definition	"value & mask(0,12)"
Member	<b>F2W1</b>	
	Description	Encoding function for word 1 of the instruction
	Definition	"(value & mask(13,15))>>8   (0xffff & mask(14,15))>>11"
Applicable Instruction(s)	'628 NOT.W <({AU16W0})'	

**Table 4-6. Group F3 Forms**

Group Name	<b>F3</b>	
Arguments	value, (s u), <len>, <align>	
Member	<b>F3W1</b>	
	Description	Encoding function for word 1 of the instruction
	Definition	"(value & mask(13,15))>>8"
Member	<b>F3W2</b>	
	Description	Encoding function for word 2 of the instruction
	Definition	"value & mask(0,12)"
Applicable Instruction(s)	'435 DOEN{0..3} #{iU16B0}'	
	'DOENSH{0..3} #{iU16B0}'	

**Table 4-7. Group F4 Forms**

Group Name	<b>F4</b>	
Arguments	value, (s u), <len>, <align>	
Member	<b>F4W1O1</b>	
	Description	Encoding function for word 1 operand 1 of the instruction
	Definition	"(~value & mask(14,15))>>11"
Member	<b>F4W1O2</b>	
	Description	Encoding function for word 1 operand 2 of the instruction
	Definition	"(value & mask(13,15))>>8"
Member	<b>F4W2O2</b>	
	Description	Encoding function for word 2 operand 2 of the instruction
	Definition	"value & mask(0,12)"
Member	<b>F4W3O1</b>	
	Description	Encoding function for word 3 operand 1 of the instruction
	Definition	"~value & mask(0,13)"
Applicable Instruction(s)	'616 AND.W #{iU16B0}, <({AU16W0})'	

**Table 4-8. Group F5 Forms**

Group Name	<b>F5</b>	
Arguments	value, (s u), <len>, <align>	
Member	<b>F5W101</b>	
	Description	Encoding function for word 1 operand 1 of the instruction
	Definition	"(~value & mask(14,15))>>11"
Member	<b>F5W102</b>	
	Description	Encoding function for word 1 operand 2 of the instruction
	Definition	"(value & mask(13,15))>>8"
Member	<b>F5W202</b>	
	Description	Encoding function for word 2 operand 2 of the instruction
	Definition	"value & mask(0,12)"
Member	<b>F5W301</b>	
	Description	Encoding function for word 3 operand 1 of the instruction
	Definition	"value & mask(0,13)"
Applicable Instruction(s)	<pre> \ 313 BMCLR.W #{iU16B0},&lt;({AU16W0})' \ 315 BMSET.W #{iU16B0},&lt;({AU16W0})' \ 317 BMCHG.W #{iU16B0},&lt;({AU16W0})' \ 319 BMTSTC.W #{iU16B0},&lt;({AU16W0})' \ 321 BMTSTS.W #{iU16B0},&lt;({AU16W0})' \ 324 MOVE.W #{iS16B0},&lt;({AU16W0})' \ 620 OR.W #{iU16B0},&lt;({AU16W0})' \ 624 EOR.W #{iU16B0},&lt;({AU16W0})' </pre>	

**Table 4-9. Group F6 Forms**

Group Name	<b>F6</b>	
Arguments	value, (s u), <len>, <align>	
Member	<b>F6W101</b>	
	Description	Encoding function for word 1 operand 1 of the instruction
	Definition	"(~value & mask(14,15))>>11"
Member	<b>F6W102</b>	
	Description	Encoding function for word 1 operand 2 of the instruction
	Definition	"(value & mask(13,15))>>8"
Member	<b>F6W202</b>	
	Description	Encoding function for word 2 operand 2 of the instruction
	Definition	"value & mask(0,12)"

**Table 4-9. Group F6 Forms (Continued)**

Member	<b>F6W3O1</b>	
	Description	Encoding function for word 3 operand 1 of the instruction
	Definition	"value & mask(0,13)"
Applicable Instruction(s)	<pre> `446 BMCLR.W  #{iU16B0}, (SP+{AS16W0})' `447 BMSET.W  #{iU16B0}, (SP+{AS16W0})' `448 BMCHG.W  #{iU16B0}, (SP+{AS16W0})' `449 BMTSTC.W #{iU16B0}, (SP+{AS16W0})' `450 BMTSTS.W #{iU16B0}, (SP+{AS16W0})' `451 BMTSET.W #{iU16B0}, (SP+{AS16W0})' `460 MOVE.W   #{iS16B0}, (SP+{AS16W0})' `635 AND.W    #{iU16B0}, (SP+{AS16W0})' `636 OR.W     #{iU16B0}, (SP+{AS16W0})' `637 EOR.W   #{iU16B0}, (SP+{AS16W0})' `638 NOT.W   (SP+{AS16W0})' </pre>	

**Table 4-10. Group F7 Forms**

Group Name	<b>F7</b>	
Arguments	value, (s u), <len>, <align>	
Member	<b>F7W1</b>	
	Description	Encoding function for word 1 of the instruction
	Definition	"(value & mask(13,15))>>8"
Member	<b>F7W2</b>	
	Description	Encoding function for word 2 of the instruction
	Definition	"value & mask(0,12)"
Applicable Instruction(s)	<pre> `262 AND  #{iS16B0}0000, DJ, DF' `265 AND  #{iU16B0}, DJ, DF' </pre>	

**Table 4-11. Group F8 Forms**

Group Name	<b>F8</b>	
Arguments	value, (s u), <len>, <align>	
Member	<b>F8W1</b>	
	Description	Encoding function for word 1 of the instruction
	Definition	"(value & mask(0,6))"
Applicable Instruction(s)	<pre> `099 MOVE.W  #{iS7B0}, HHHH' </pre>	

**Table 4-12. Group F9 Forms**

Group Name	<b>F9</b>	
Arguments	value, (s u), <len>, <align>	
Member	<b>F9W202</b>	
	Description	Encoding function for word 2 operand 2 of the instruction
	Definition	"value & mask(0,5)"
Member	<b>F9W201</b>	
	Description	Encoding function for word 2 operand 1 of the instruction
	Definition	"(value & mask(0,5))<<6"
Applicable Instruction(s)	<pre> `259 EXTRACT #{IU6B0},#{iU6B0},Dj,DF' `260 EXTRACTU #{IU6B0},#{iU6B0},Dj,DF' `261 INSERT #{IU6B0},#{iU6B0},Dj,DF' </pre>	

**Table 4-13. Group F10 Forms**

Group Name	<b>F10</b>	
Arguments	value, (s u), <len>, <align>	
Member	<b>F10W1</b>	
	Description	Encoding function for word 1 of the instruction
	Definition	"(value & mask(13,14))>>8"
Member	<b>F10W2</b>	
	Description	Encoding function for word 2 of the instruction
	Definition	"value & mask(0,12)"
Applicable Instruction(s)	<pre> `206.w MOVE.W HHHH, (R+{AS15W0})' `206.r MOVE.W (R+{AS15W0}), HHHH' `208.w MOVE.L HHHH, (R+{AS15L0})' `208.r MOVE.L (R+{AS15L0}), HHHH' `207.w MOVE.B HHHH, (R+{AS15B0})' `207.r MOVE.B (R+{AS15B0}), HHHH' `209 MOVEU.W (R+{AS15W0}), HHHH' `210 MOVE.F (R+{AS15W0}), Dj' `211 MOVES.F Dj, (R+{AS15W0})' `212.w MOVE.W DDDDD, (SP+{AS15W0})' `212.r MOVE.W (SP+{AS15W0}), DDDDD' `213 MOVE.B HHHH, (SP+{AS15B0})' `217.w MOVE.L DDDDD, (SP+{AS15L0})' `217.r MOVE.L (SP+{AS15L0}), DDDDD' `214 MOVEU.B (SP+{AS15B0}), HHHH' `216 MOVE.B (SP+{AS15B0}), HHHH' `218 MOVE.F (SP+{AS15W0}), Dj' `219 MOVES.F Dj, (SP+{AS15W0})' `220 MOVEU.W (SP+{AS15W0}), DDDDD' </pre>	

**Table 4-14. Group F11 Forms**

Group Name	<b>F11</b>	
Arguments	value, (s u), <len>, <align>	
Member	<b>F11W1</b>	
	Description	Encoding function for word 1 of the instruction
	Definition	"(value & mask(30,31))>>27   (value & mask(13,15))>>8"
Member	<b>F11W2</b>	
	Description	Encoding function for word 2 of the instruction
	Definition	"value & mask(0,12)"
Member	<b>F11W3</b>	
	Description	Encoding function for word 3 of the instruction
	Definition	"(value & mask(16,29))>>16"
Applicable Instruction(s)	<pre> `311 MOVE.L #{iS32B0},DDDD' `312 MOVEU.L #{iU32B0},Dj' `300.w MOVE.W HHHH,({AU32W0})' `300.r MOVE.W ({AU32W0}),HHHH' `301.w MOVE.B HHHH,({AU32B0})' `301.r MOVE.B ({AU32B0}),HHHH' `302.w MOVE.L HHHH,({AU32L0})' `302.r MOVE.L ({AU32L0}),HHHH' `303 MOVEU.W ({AU32W0}),HHHH' `305 MOVES.F Dj,({AU32W0})' `304 MOVE.F ({AU32W0}),Dj' `326.0 JMPD {AU32W0}' `326.1 JMP {AU32W0}' `327.0 JSRD {AU32W0}' `327.1 JSR {AU32W0}' `328.0.0 JTD {AU32W0}' `328.0.1 JT {AU32W0}' `328.1.0 JFD {AU32W0}' `328.1.1 JF {AU32W0}' `407 MOVE.L #{iU32B0},CCC' `466 MOVE.L Df1.e:Df2.e,({AU32L0})' `465 MOVE.L ({AU32L0}),DQ.e' `467 MOVE.L ({AU32L0}),Dq.e' </pre>	

**Table 4-15. Group F12 Forms**

Group Name		<b>F12</b>
Arguments		value, (s u), <len>, <align>
Member	<b>F12W1</b>	
	Description	Encoding function for word 1 of the instruction
	Definition	"(value & mask(30,31))>>27   (value & mask(13,15))>>8"
Applicable Instruction(s)		<pre> `102 CMPEQ.W #{iU5B0},DF' `103 CMPGT.W #{iU5B0},DF' `100 ADD #{iU5B0},DF' `101 SUB #{iU5B0},DF' `104 ASLL #{iU5B0},DF' `105 ASRR #{iU5B0},DF' `428 LSRR #{iU5B0},DF' `130 ADDA #{iU5B0},RRRR' `131 SUBA #{iU5B0},RRRR' `602 INCA RRRR' `603 DECA RRRR' </pre>

**Table 4-16. Group F13 Forms**

Group Name		<b>F13</b>
Arguments		value, (s u), <len>, <align>
Member	<b>F13W1</b>	
	Description	Encoding function for word 1 of the instruction
	Definition	"(~value & mask(13,15))>>8"
Member	<b>F13W2</b>	
	Description	Encoding function for word 2 of the instruction
	Definition	"~value & mask(0,12)"
Applicable Instruction(s)		<pre> `640 AND #{iU16B0},HHHH.H' `614 AND.W #{iU16B0},(R)' `276a AND.W #{iU16B0},CCC.L' </pre>



Table 4-17. Group F14 Forms

Group Name		<b>F14</b>
Arguments		value, (s u), <len>, <align>
Member	<b>F14W1</b>	
	Description	Encoding function for word 1 of the instruction
	Definition	"(value & mask(13,15))>>8"
Member	<b>F14W2</b>	
	Description	Encoding function for word 2 of the instruction
	Definition	"value & mask(0,12)"
Applicable Instruction(s)	<pre> `245 ADDNC.W #{iS16B0},DJ,DF' `247 MAC #{iS16B0},DJ,DF' `641 AND #{iU16B0},HHHH.L' `642 OR #{iU16B0},HHHH.H' `643 OR #{iU16B0},HHHH.L' `644 EOR #{iU16B0},HHHH.H' `645 EOR #{iU16B0},HHHH.L' `234 ADDA #{iS16B0},rrrr,R' `242 CMPEQ.W #{iS16B0},DF' `243 CMPGT.W #{iS16B0},DF' `240 SUBNC.W #{iS16B0},DF' `244 IMPY.W #{iS16B0},DF' `279 BMCLR.W #{iU16B0},(R)' `280 BMSET.W #{iU16B0},(R)' `281 BMCHG.W #{iU16B0},(R)' `291 BMTSTC.W #{iU16B0},(R)' `292 BMTSTS.W #{iU16B0},(R)' `457 BMTSET.W #{iU16B0},(R)' `618 OR.W #{iU16B0},(R)' `622 EOR.W #{iU16B0},(R)' `269 BMCLR #{iU16B0},HHHH.H' `270 BMCLR #{iU16B0},HHHH.L' `271 BMSET #{iU16B0},HHHH.H' `272 BMSET #{iU16B0},HHHH.L' `273 BMCHG #{iU16B0},HHHH.H' `274 BMCHG #{iU16B0},HHHH.L' `284 BMTSTC #{iU16B0},HHHH.H' `285 BMTSTC #{iU16B0},HHHH.L' `286 BMTSTS #{iU16B0},HHHH.H' `287 BMTSTS #{iU16B0},HHHH.L' `276 BMCLR #{iU16B0},CCC.L' `277 BMSET #{iU16B0},CCC.L' `278 BMCHG #{iU16B0},CCC.L' `289 BMTSTC #{iU16B0},CCC.L' `290 BMTSTS #{iU16B0},CCC.L' `413 BMTSTC #{iU16B0},CCC.H' </pre>	

**Table 4-17. Group F14 Forms (Continued)**

F14 Applicable Instructions (Continued)	<pre> `414 BMTSTS #{iU16B0},CCC.H' `410 BMCLR #{iU16B0},CCC.H' `411 BMSET #{iU16B0},CCC.H' `412 BMCHG #{iU16B0},CCC.H' `276b OR.W #{iU16B0},CCC.[HL]' `276c EOR.W #{iU16B0},CCC.[HL]' `231 MOVE.W #{iS16B0},DDDDD' `232 MOVE.F #{iS16B0},Dj' `295 MOVEU.W #{iU16B0},Dj.H' `296 MOVEU.W #{iU16B0},Dj.L' `299 MOVE.W #{iS16B0},(R)' `221.w MOVE.W DDDDD,&lt;({AU16W0})' `221.r MOVE.W &lt;({AU16W0}),DDDDD' `222 MOVE.B HHHH,&lt;({AU16B0})' `226.w MOVE.L DDDDD,&lt;({AU16L0})' `226.r MOVE.L &lt;({AU16L0}),DDDDD' `228 MOVES.F Dj,&lt;({AU16W0})' `227 MOVE.F &lt;({AU16W0}),Dj' `223 MOVEU.B &lt;({AU16B0}),HHHH' `225 MOVE.B &lt;({AU16B0}),HHHH' `229 MOVEU.W &lt;({AU16W0}),DDDDD' </pre>
--	--

**Table 4-18. Group F15 Forms**

Group Name	<b>F15</b>	
Arguments	value, (s u), <len>, <align>	
Member	<b>F15W1</b>	
	Description	Encoding function for word 1 of the instruction
	Definition	"(pack(value,s u,len,align) & mask(0,7))<<1"
Applicable Instruction(s)	<pre> `156.0.0 BTD &lt;*+{AS8W1}' `156.0.1 BT &lt;*+{AS8W1}' `156.1.0 BFD &lt;*+{AS8W1}' `156.1.1 BF &lt;*+{AS8W1}' `157.0 BSRD &lt;*+{AS8W1}' `157.1 BSR &lt;*+{AS8W1}' </pre>	

**Table 4-19. Group F16 Forms**

Group Name	<b>F16</b>	
Arguments	value, (s u), <len>, <align>	
Member	<b>F16W101</b>	
	Description	Encoding function for word 1 operand 1 of the instruction
	Definition	"(value & mask(13,15))>>8"
Member	<b>F16W102</b>	
	Description	Encoding function for word 1 operand 2 of the instruction
	Definition	"pack(value,s u,len,align) & mask(0,4)"
Member	<b>F16W201</b>	
	Description	Encoding function for word 2 operand 1 of the instruction
	Definition	"value & mask(0,12)"
Applicable Instruction(s)	<pre>'294 MOVE.W #{iS16B0},(SP-{AU5W1})'</pre> <pre>'294 alias 3 MOVE.W #{iS16B0},(SP)'</pre>	

**Table 4-20. Group F17 Forms**

Group Name	<b>F17</b>	
Arguments	value, (s u), <len>, <align>	
Member	<b>F17W101</b>	
	Description	Encoding function for word 1 operand 1 of the instruction
	Definition	"(~value & mask(13,15))>>8"
Member	<b>F17W102</b>	
	Description	Encoding function for word 1 operand 2 of the instruction
	Definition	"pack(value,s u,len,align) & mask(0,4)"
Member	<b>F17W201</b>	
	Description	Encoding function for word 2 operand 1 of the instruction
	Definition	"~value & mask(0,12)"
Applicable Instruction(s)	<pre>'615 AND.W #{iU16B0},(SP-{AU5W1})'</pre> <pre>'266a alias 3 #{iU16B0},(SP)'</pre>	

**Table 4-21. Group F18 Forms**

Group Name	<b>F18</b>	
Arguments	value, (s u), <len>, <align>	
Member	<b>F18W101</b>	
	Description	Encoding function for word 1 operand 1 of the instruction
	Definition	"(~value & mask(13,15))>>8"
Member	<b>F18W102</b>	
	Description	Encoding function for word 1 operand 2 of the instruction
	Definition	"pack(value,s u,len,align) & mask(0,4)"
Member	<b>F18W201</b>	
	Description	Encoding function for word 2 operand 1 of the instruction
	Definition	"value & mask(0,12)"
Applicable Instruction(s)	<pre> `266 BMCLR.W #{iU16B0},(SP-{AU5W1})' `266 alias 3 BMCLR.W #{iU16B0},(SP)' `266c alias 3 EOR.W #{iU16B0},(SP)' `282 BMTSTC.W #{iU16B0},(SP-{AU5W1})' `282 alias 3 BMTSTC.W #{iU16B0},(SP)' `283 BMTSTS.W #{iU16B0},(SP-{AU5W1})' `283 alias 3 BMTSTS.W #{iU16B0},(SP)' `267 BMSET.W #{iU16B0},(SP-{AU5W1})' `267 alias 3 BMSET.W #{iU16B0},(SP)' `268 BMCHG.W #{iU16B0},(SP-{AU5W1})' `268 alias 3 BMCHG.W #{iU16B0},(SP)' `452 BMTSET.W #{iU16B0},(SP-{AU5W1})' `452 alias 3 BMTSET.W #{iU16B0},(SP)' `619 OR.W #{iU16B0},(SP-{AU5W1})' `266b alias 3 OR.W #{iU16B0},(SP)' `623 EOR.W #{iU16B0},(SP-{AU5W1})' </pre>	

**Table 4-22. Group F19 Forms**

Group Name	<b>F19</b>	
Arguments	value, (s u), <len>, <align>	
Member	<b>F19W1</b>	
	Description	Encoding function for word 1 of the instruction
	Definition	"(pack(value,s u,len,align)&mask(19,19)>>8)   (pack(value,s u,len,align)&mask(16,18)>>16)   (pack(value,s u,len,align)&mask(12,14)>>7)"
Member	<b>F19W2</b>	
	Description	Encoding function for word 2 of the instruction
	Definition	"(pack(value,s u,len,align)&mask(15,15)>>8)   (pack(value,s u,len,align)&mask(0,11)<<1)  "
Applicable Instruction(s)	<pre> `236.0 BRAD &gt;+{AS20W1}' `236.1 BRA &gt;+{AS20W1}' `237.0 BSRD &gt;+{AS20W1}' `237.1 BSR &gt;+{AS20W1}' `238.0.0 BTD &gt;+{AS20W1}' `238.0.1 BT &gt;+{AS20W1}' `238.1.0 BFD &gt;+{AS20W1}' `238.1.1 BF &gt;+{AS20W1}' </pre>	

**Table 4-23. Group F20 Forms**

Group Name	<b>F20</b>	
Arguments	value, (s u), <len>, <align>	
Member	<b>F20W1</b>	
	Description	Encoding function for word 1 of the instruction
	Definition	"pack(value,s u,len,align)&mask(0,4)"
Applicable Instruction(s)	<pre> `123.w MOVE.W HHHH, (R+{AU3W1})' `123.r MOVE.W (R+{AU3W1}), HHHH' `123.w alias 4 MOVE.W HHHH, (R)' `124.w MOVE.L HHHH, (R+{AU3L2})' `124.w MOVE.L (R+{AU3L2}), HHHH' `124.w alias 4 MOVE.L HHHH, (R)' </pre>	

**Table 4-24. Group F21 Forms**

Group Name	<b>F21</b>	
Arguments	value, (s u), <len>, <align>	
Member	<b>F21W1</b>	
	Description	Encoding function for word 1 of the instruction
	Definition	"pack(value,s u,len,align)&mask(0,5)"
Applicable Instruction(s)	<pre> `162 DOEN{0..3} #{iU6B0}' `421 DOENSH{0..3} #{iU6B0}' `154.w MOVE.W HHHH,(SP-{AU6W1})' `154.w alias 2 MOVE.W HHHH,(SP)' `154.r MOVE.W (SP-{AU6W1}),HHHH' `154.r alias 2 MOVE.W (SP),HHHH' `155.w MOVE.L HHHH,(SP-{AU6L2})' `155.w alias 2 MOVE.L HHHH,(SP)' `155.r MOVE.L (SP-{AU6L2}),HHHH' `155.r alias 2 MOVE.L (SP),HHHH' </pre>	

**Table 4-25. Group F22 Forms**

Group Name	<b>F22</b>	
Arguments	value, (s u), <len>, <align>	
Member	<b>F22W1</b>	
	Description	Encoding function for word 1 of the instruction
	Definition	"(pack(value,s u,len,align) & mask(12,14))>>7"
Member	<b>F22W2</b>	
	Description	Encoding function for word 2 of the instruction
	Definition	"(pack(value,s u,len,align) & mask(15,15))>>15   (pack(value,s u,len,align) & mask(0,11))<<1"
Applicable Instruction(s)	<pre> `405 BREAK *+{AS16W1}' `418 SKIPLS *+{AS16W1}' `416.0 CONTD *+{AS16W1}' `416.1 CONT *+{AS16W1}' `329 DOSETUP{0..3} *+{AS16W1}' </pre>	

**Table 4-26. Group F23 Forms**

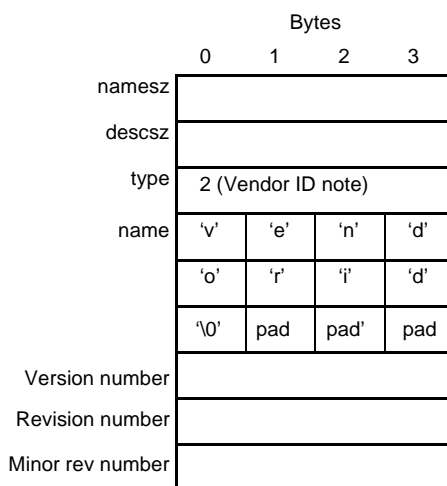
Group Name	<b>F23</b>		
Arguments	value, (s u), <len>, <align>		
Member	<b>F23W1</b>		
	Description	Encoding function for word 1 of the instruction	
	Definition	$(0xffff \& \text{mask}(13,15)) \gg 8 \mid$ <code>pack(value, s u, len, align) &amp; mask(0,4)</code>	
Applicable Instruction(s)	'627 NOT.W (SP-{AU5W1})'		

### 4.3.5 NOTE Section

The note section is optional and contains object file vendor identification and application-specific object file comments. If included, it follows the described format.

Vendor identification format is shown in Figure 4-2. It consists of the following:

namesz	The string length (not counting null terminator) of the name. It is a 4 byte unsigned integer.
descsz	The size of the description entries. This is 12 bytes for the vendor id note. The description fields contain the version, revision, minor revision numbers of the producing entity (assembler or linker). Data is an unsigned 4-byte integer.
type	Type equals 2 for the vendor identification note. It is a 4-byte unsigned integer in little-endian order.
name	Null terminated string and padded, if necessary, to achieve a 4-byte boundary alignment which represents the vendor's identification.

**Figure 4-2. Vendor Identification Note Format**

Object file comments generated by the user through an assembler directive are placed in the note section. This is typically for users to identify their object code. The same string termination and padding restrictions apply to object file comments as apply to vendor identification notes. The field contains a user-specified comment. A null comment ( \0 ) is not a valid comment.

The object file comment format is shown in Figure 4-3.

	0	1	2	3
namesz				
descsz	0			
type	1			
name	'c'	'o'	'm'	'm'
	'e'	'n'	't'	\0

**Figure 4-3. User (Application-Specific) Note Format**

## 4.3.6 Program Headers

Program headers are used to build an executable image in memory and are only useful for executable files. While section headers may or may not be included in executable files, program headers are always present in executable files. See Example 4-8 for a sample program header.

### Example 4-8. Program Header

```
typedef struct {
    Elf32_Word p_type;
    Elf32_Off  p_offset;
    Elf32_Addr p_vaddr;
    Elf32_Addr p_paddr;
    Elf32_Word p_filesz;
    Elf32_Word p_memsz;
    Elf32_Word p_flags;
    Elf32_Word p_align;
} Elf32_Phdr;
```

Refer to the following list for a description of program header members.

- `p_type`—describes the type of program header. Only `PT_LOAD` and `PT_NOTE` are recognized as types.
- `p_offset`—offset from beginning of file to first byte of segment.
- `p_vaddr`—virtual address in memory of the first byte of the segment.
- `p_paddr`—physical address in memory of the first byte of the segment.
- `p_filesz`—gives the number of bytes in segment's memory image. (May be zero.)
- `p_memsz`—gives the number of bytes in segment's memory image. (May be zero.)
- `p_flags`—gives flags relevant to the segment. Defined flags are `PF_R`, `PF_W`, `PF_X`.
- `p_align`—segment alignment requirements in file and memory.



# Chapter 5

## Endian Support

---

This chapter describes the different behavior of the SC140 instructions in the Big-Endian and Little-Endian memory system modes.

- **Little-Endian:**

*“A computer architecture in which, within a given multi-byte numeric representation, bytes at lower addresses have lower significance (the word is stored “little-end-first”).”*

For example, the instruction `MOVE.W D0, (r0)` will store bits 0-7 of D0 into address (r0) and bits 15-8 into address (r0+1).

- **Big-Endian:**

*“A computer architecture in which, within a given multi-byte numeric representation, the most significant byte has the lowest address (the word is stored “big-end-first”).”*

For example, the instruction `MOVE.W D0, (r0)` will store bits 15-8 of D0 into address (r0) and bits 0-7 into address (r0+1).

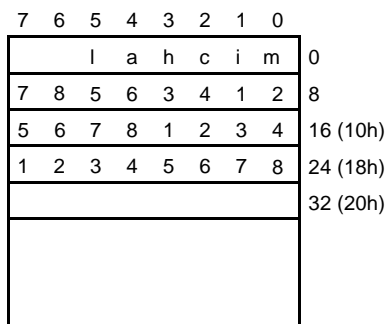
The SC140 DSP core supports Big and Little Endian architecture through a mode bit in its Exception and Mode Register. This bit samples a core input signal when exiting the reset state and cannot be changed during normal operation.

# 5.1 Memory Organization

Different types of data will be stored differently in the memory in the different modes. Consider this example of data in the memory shown in Figure 5-1:

A0='m'	A8=0102h (16-bit Number)
A1='i'	A10=0304h
A2='c'	A12=0506h
A3='h'	A14=0708h
A3='a'	
A5='l'	A16=01020304h (32-bit Number)
	A20=0506078h
	A24=0102030405060708h (64-bit Number)

### Little Endian



### Big Endian

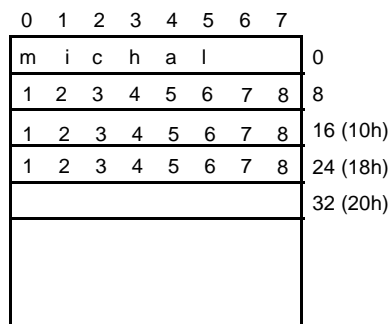


Figure 5-1. Memory Organization in Big/Little Endian Architecture

## 5.1.1 SC140 Architecture

The entire memory space of the SC140 core is unified. The memory supports two parallel 64-bit data accesses issued by the core at the same time, one 128-bit program bus, and external port (usually for DMA accesses).

The two data busses that connect between the Data ALU register file and the memory are 64-bits wide each. Load and Store instructions utilize the maximum width of the bus according to the application requirement, by means of having different versions of the instructions for different bandwidth:

- MOVE.B loads or stores bytes (8-bit)
- MOVE.W or MOVE.F loads or stores integer or fractional words (16-bit)
- MOVE.2W, MOVE.2F and MOVE.L loads or stores double-integers, double-fractions and long words respectively (32-bit)
- MOVE.4W, MOVE.4F loads or stores quad-integers and quad-fractions respectively (64-bit)
- MOVE.2L loads or stores double-long words (64-bit)



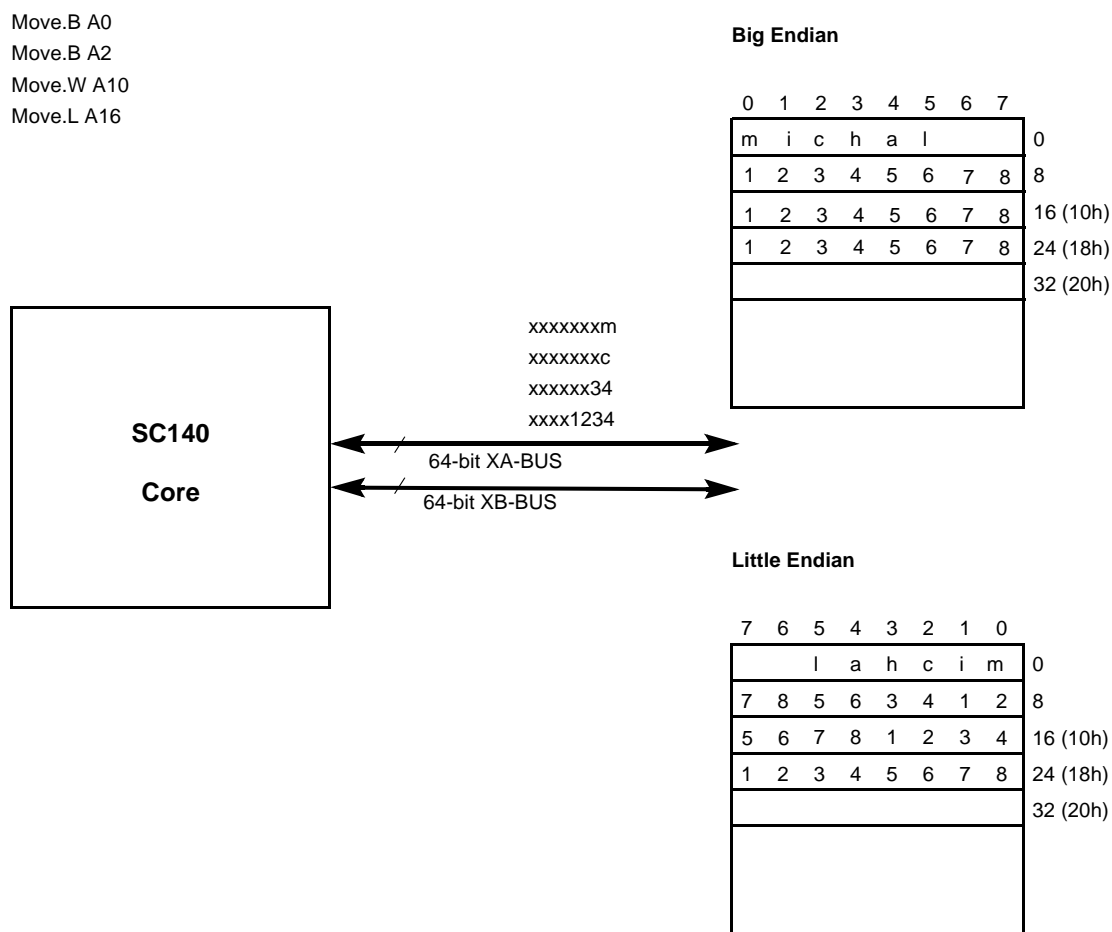
Figure 5-2. SC140 Basic Architecture

## 5.1.2 Data Move

Data moves are done by moving DALU register/ Memory over one of the data buses, XDBA or XDBB. Data registers can be accessed with three types of data:

- A long type access, writing or reading 32-bit operands
- A word type access, writing or reading 16-bit operands
- A byte type access, writing or reading 8-bit operands

Figure 5-3 and Figure 5-4 illustrates a single data transfer and multiple data transfer, respectively, in big/little endian modes.



**Figure 5-3. Data Transfer in Big/Little Endian**

Move.2W A8  
 Move.4W A16  
 Move.2L A16

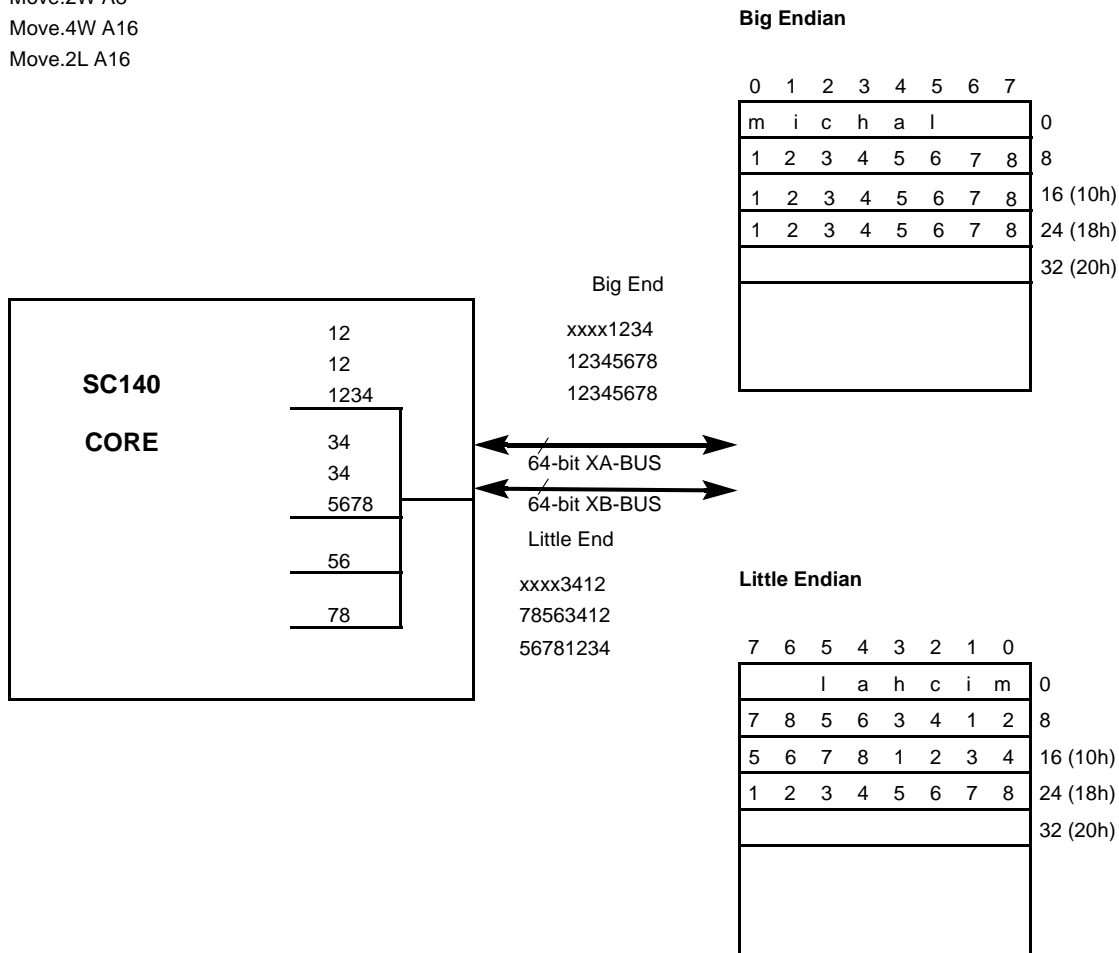


Figure 5-4. Multiple Data Transfer in Big/Little Endian

### 5.1.3 Instruction Word Transfers

Instruction words are transferred to the core from the memory over the Program Data Bus (PDB), to special instruction registers in the Program Dispatch Unit.

The instruction registers can be accessed only with 128-bit width (8 instructions).

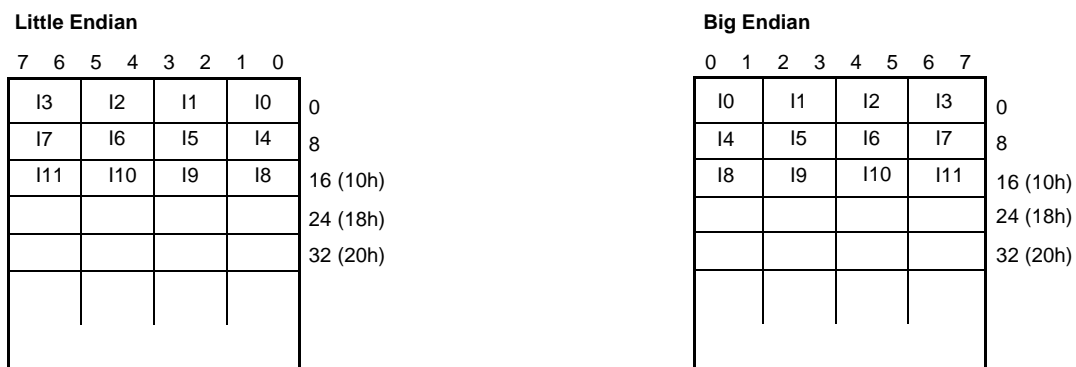


Figure 5-5. Program Memory Organization

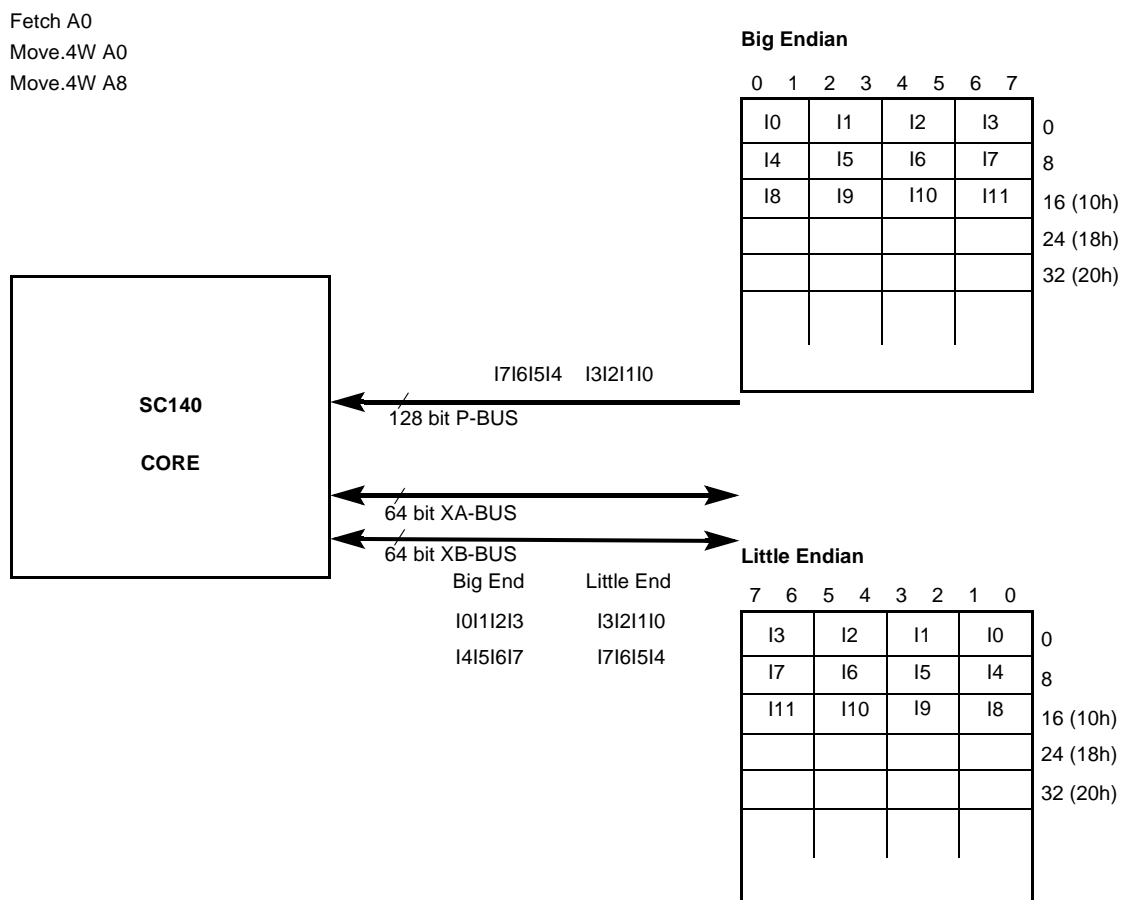


Figure 5-6. Instruction Moves

## 5.2 Memory Access Behavior in Endian Modes

Table 5-1. MOVE Instructions

Instruction	Register Operands	Big Endian	Little Endian
MOVE.B		A0=A	A0=A
MOVEU.B		A0=A	A0=A
MOVE.W		A0=A A1=B	A0=B A1=A
MOVEU.W		A0=A A1=B	A0=B A1=A
MOVE.2W		A0=A A1=B A2=C A3=D	A0=B A1=A A2=D A3=C
MOVE.4W		A0=A A1=B A2=C A3=D A4=E A5=F A6=G A7=H	Add0=B Add1=A Add2=D Add3=C Add4=F Add5=E Add6=H Add7=G
MOVE.L		A0=A A1=B A2=C A3=D	A0=D A1=C A2=B A3=A
MOVE.L (Extension)		A0=Lb A1=B A2=La A3=A	A0=A A1=La A2=B A3=Lb

**Table 5-1. MOVE Instructions (Continued)**

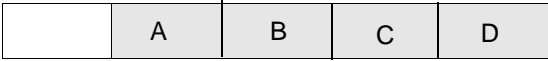
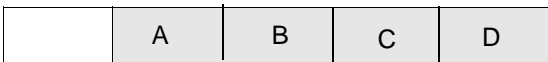
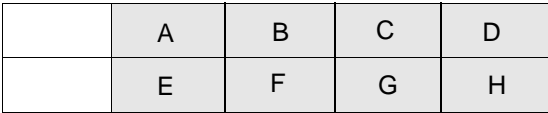


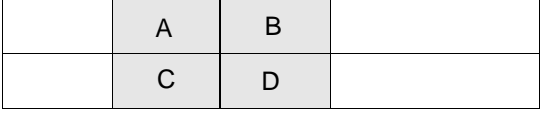
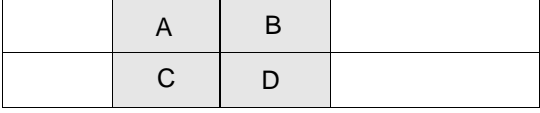
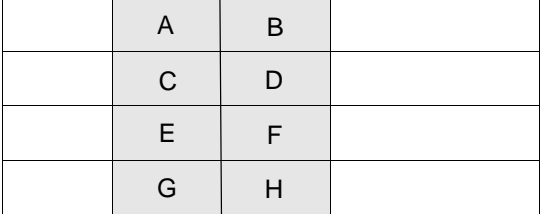
Instruction	Register Operands	Big Endian	Little Endian
MOVEU.L		A0=A A1=B A2=C A3=D	A0=D A1=C A2=B A3=A
MOVES.L		A0=A A1=B A2=C A3=D	A0=D A1=C A2=B A3=A
MOVE.2L		A0=A A1=B A2=C A3=D A4=E A5=F A6=G A7=H	A0=D A1=C A2=B A3=A A4=H A5=G A6=F A7=E
MOVE.F		A0=A A1=B	A0=B A1=A
MOVES.F		A0=A A1=B	A0=B A1=A
MOVE.2F		A0=A A1=B A2=C A3=D	A0=B A1=A A2=D A3=C
MOVES.2F		A0=A A1=B A2=C A3=D	A0=B A1=A A2=D A3=C
MOVE.4F		A0=A A1=B A2=C A3=D A4=E A5=F A6=G A7=H	A0=B A1=A A2=D A3=C A4=F A5=E A6=H A7=G



Table 5-1. MOVE Instructions (Continued)

Instruction	Register Operands	Big Endian	Little Endian																
MOVES . 4F	<div style="display: flex; justify-content: space-between; margin-bottom: 5px;"> <span>39</span> <span>32</span> <span>16</span> <span>0</span> </div> <table border="1" style="width: 100%; border-collapse: collapse; text-align: center;"> <tr> <td style="width: 25%;"></td> <td style="width: 12.5%;">A</td> <td style="width: 12.5%;">B</td> <td style="width: 50%;"></td> </tr> <tr> <td></td> <td>C</td> <td>D</td> <td></td> </tr> <tr> <td></td> <td>E</td> <td>F</td> <td></td> </tr> <tr> <td></td> <td>G</td> <td>H</td> <td></td> </tr> </table>		A	B			C	D			E	F			G	H		A0=A A1=B A2=C A3=D A4=E A5=F A6=G A7=H	A0=B A1=A A2=D A3=C A4=F A5=E A6=H A7=G
	A	B																	
	C	D																	
	E	F																	
	G	H																	
VSL . 4W	<div style="display: flex; justify-content: space-between; margin-bottom: 5px;"> <span>39</span> <span>16</span> <span>0</span> </div> <table border="1" style="width: 100%; border-collapse: collapse; text-align: center;"> <tr> <td style="width: 66.6%;"></td> <td style="width: 16.6%;">A</td> <td style="width: 16.6%;">B</td> </tr> <tr> <td></td> <td>C</td> <td>D</td> </tr> <tr> <td></td> <td>E</td> <td>F</td> </tr> <tr> <td></td> <td>G</td> <td>H</td> </tr> </table>		A	B		C	D		E	F		G	H	A0=C A1=D A2=A A3=B A4=G A5=H A6=E A7=F	A0=B A1=A A2=D A3=C A4=F A5=E A6=H A7=G				
	A	B																	
	C	D																	
	E	F																	
	G	H																	
VSL . 4F	<div style="display: flex; justify-content: space-between; margin-bottom: 5px;"> <span>39</span> <span>32</span> <span>16</span> <span>0</span> </div> <table border="1" style="width: 100%; border-collapse: collapse; text-align: center;"> <tr> <td style="width: 25%;"></td> <td style="width: 12.5%;">A</td> <td style="width: 12.5%;">B</td> <td style="width: 50%;"></td> </tr> <tr> <td></td> <td>C</td> <td>D</td> <td></td> </tr> <tr> <td></td> <td>E</td> <td>F</td> <td></td> </tr> <tr> <td></td> <td>G</td> <td>H</td> <td></td> </tr> </table>		A	B			C	D			E	F			G	H		A0=C A1=D A2=A A3=B A4=G A5=H A6=E A7=F	A0=B A1=A A2=D A3=C A4=F A5=E A6=H A7=G
	A	B																	
	C	D																	
	E	F																	
	G	H																	
VSL . 2W	<div style="display: flex; justify-content: space-between; margin-bottom: 5px;"> <span>39</span> <span>16</span> <span>0</span> </div> <table border="1" style="width: 100%; border-collapse: collapse; text-align: center;"> <tr> <td style="width: 66.6%;"></td> <td style="width: 16.6%;">A</td> <td style="width: 16.6%;">B</td> </tr> <tr> <td></td> <td>C</td> <td>D</td> </tr> </table>		A	B		C	D	A0=C A1=D A2=A A3=B	A0=B A1=A A2=D A3=C										
	A	B																	
	C	D																	
VSL . 2F	<div style="display: flex; justify-content: space-between; margin-bottom: 5px;"> <span>39</span> <span>32</span> <span>16</span> <span>0</span> </div> <table border="1" style="width: 100%; border-collapse: collapse; text-align: center;"> <tr> <td style="width: 25%;"></td> <td style="width: 12.5%;">A</td> <td style="width: 12.5%;">B</td> <td style="width: 50%;"></td> </tr> <tr> <td></td> <td>C</td> <td>D</td> <td></td> </tr> </table>		A	B			C	D		A0=C A1=D A2=A A3=B	A0=B A1=A A2=D A3=C								
	A	B																	
	C	D																	

**Table 5-2. Stack Support Instructions**

Instruction	Register operands	Big Endian	Little Endian								
POP	<div style="text-align: center;"> <span style="margin-right: 100px;">32</span> <span>0</span> <table border="1" style="margin: auto;"> <tr> <td style="padding: 5px;">A</td> <td style="padding: 5px;">B</td> <td style="padding: 5px;">C</td> <td style="padding: 5px;">D</td> </tr> <tr> <td style="padding: 5px;">E</td> <td style="padding: 5px;">F</td> <td style="padding: 5px;">G</td> <td style="padding: 5px;">H</td> </tr> </table> </div>	A	B	C	D	E	F	G	H	A0=A A1=B A2=C A3=D A4=E A5=F A6=G A7=H	A0=D A1=C A2=B A3=A A4=H A5=G A6=F A7=E
A	B	C	D								
E	F	G	H								
POPN	<div style="text-align: center;"> <span style="margin-right: 100px;">32</span> <span>0</span> <table border="1" style="margin: auto;"> <tr> <td style="padding: 5px;">A</td> <td style="padding: 5px;">B</td> <td style="padding: 5px;">C</td> <td style="padding: 5px;">D</td> </tr> <tr> <td style="padding: 5px;">E</td> <td style="padding: 5px;">F</td> <td style="padding: 5px;">G</td> <td style="padding: 5px;">H</td> </tr> </table> </div>	A	B	C	D	E	F	G	H	A0=A A1=B A2=C A3=D A4=E A5=F A6=G A7=H	A0=D A1=C A2=B A3=A A4=H A5=G A6=F A7=E
A	B	C	D								
E	F	G	H								
PUSH	<div style="text-align: center;"> <span style="margin-right: 100px;">32</span> <span>0</span> <table border="1" style="margin: auto;"> <tr> <td style="padding: 5px;">A</td> <td style="padding: 5px;">B</td> <td style="padding: 5px;">C</td> <td style="padding: 5px;">D</td> </tr> <tr> <td style="padding: 5px;">E</td> <td style="padding: 5px;">F</td> <td style="padding: 5px;">G</td> <td style="padding: 5px;">H</td> </tr> </table> </div>	A	B	C	D	E	F	G	H	A0=A A1=B A2=C A3=D A4=E A5=F A6=G A7=H	A0=D A1=C A2=B A3=A A4=H A5=G A6=F A7=E
A	B	C	D								
E	F	G	H								
PUSHN	<div style="text-align: center;"> <span style="margin-right: 100px;">32</span> <span>0</span> <table border="1" style="margin: auto;"> <tr> <td style="padding: 5px;">A</td> <td style="padding: 5px;">B</td> <td style="padding: 5px;">C</td> <td style="padding: 5px;">D</td> </tr> <tr> <td style="padding: 5px;">E</td> <td style="padding: 5px;">F</td> <td style="padding: 5px;">G</td> <td style="padding: 5px;">H</td> </tr> </table> </div>	A	B	C	D	E	F	G	H	A0=A A1=B A2=C A3=D A4=E A5=F A6=G A7=H	A0=D A1=C A2=B A3=A A4=H A5=G A6=F A7=E
A	B	C	D								
E	F	G	H								

Table 5-3. Bit-Mask Instructions

Instruction	Register Operands	Big Endian	Little Endian
BMCHG.W		A0=A A1=B	A0=B A1=A
BMCLR.W		A0=A A1=B	A0=B A1=A
BMSET.W		A0=A A1=B	A0=B A1=A
BMTSTS.W		A0=A A1=B	A0=B A1=A
BMTSTC.W		A0=A A1=B	A0=B A1=A
BMTSET.W		A0=A A1=B	A0=B A1=A
NOT.W		A0=A A1=B	A0=B A1=A
AND.W		A0=A A1=B	A0=B A1=A
OR.W		A0=A A1=B	A0=B A1=A
EOR.W		A0=A A1=B	A0=B A1=A

**Table 5-4. Change of Flow Instructions**

Instruction	Register Operands	Big Endian	Little Endian
BSR	$PC = \begin{array}{ c c c c } \hline & 32 & & 0 \\ \hline A & B & C & D \\ \hline \end{array}$	A0=A A1=B A2=C A3=D	A0=D A1=C A2=B A3=A
BSRD	$PC = \begin{array}{ c c c c } \hline & 32 & & 0 \\ \hline A & B & C & D \\ \hline \end{array}$	A0=A A1=B A2=C A3=D	A0=D A1=C A2=B A3=A
JSR	$PC = \begin{array}{ c c c c } \hline & 32 & & 0 \\ \hline A & B & C & D \\ \hline \end{array}$	A0=A A1=B A2=C A3=D	A0=D A1=C A2=B A3=A
JSRD	$PC = \begin{array}{ c c c c } \hline & 32 & & 0 \\ \hline A & B & C & D \\ \hline \end{array}$	A0=A A1=B A2=C A3=D	A0=D A1=C A2=B A3=A
RTE	$\begin{array}{l} PC = \begin{array}{ c c c c } \hline & 32 & & 0 \\ \hline A & B & C & D \\ \hline \end{array} \\ SR = \begin{array}{ c c c c } \hline E & F & G & H \\ \hline \end{array} \end{array}$	A0=A A1=B A2=C A3=D A4=E A5=F A6=G A7=H	A0=D A1=C A2=B A3=A A4=H A5=G A6=F A7=E
RTED	$\begin{array}{l} PC = \begin{array}{ c c c c } \hline & 32 & & 0 \\ \hline A & B & C & D \\ \hline \end{array} \\ SR = \begin{array}{ c c c c } \hline E & F & G & H \\ \hline \end{array} \end{array}$	A0=A A1=B A2=C A3=D A4=E A5=F A6=G A7=H	A0=D A1=C A2=B A3=A A4=H A5=G A6=F A7=E
RTS	$PC = \begin{array}{ c c c c } \hline & 32 & & 0 \\ \hline A & B & C & D \\ \hline \end{array}$	A0=A A1=B A2=C A3=D	A0=D A1=C A2=B A3=A

Table 5-4. Change of Flow Instructions (Continued)

Instruction	Register Operands	Big Endian	Little Endian
RTSD	$PC = \begin{array}{ c c c c } \hline & 32 & & 0 \\ \hline A & B & C & D \\ \hline \end{array}$	A0=A A1=B A2=C A3=D	A0=D A1=C A2=B A3=A
RTSTK	$PC = \begin{array}{ c c c c } \hline & 32 & & 0 \\ \hline A & B & C & D \\ \hline \end{array}$	A0=A A1=B A2=C A3=D	A0=D A1=C A2=B A3=A
RTSTKD	$PC = \begin{array}{ c c c c } \hline & 32 & & 0 \\ \hline A & B & C & D \\ \hline \end{array}$	A0=A A1=B A2=C A3=D	A0=D A1=C A2=B A3=A

Table 5-5. Control Instructions

Instruction	Register operands	Big Endian	Little Endian
TRAP	$\begin{array}{l} PC = \begin{array}{ c c c c } \hline & 32 & & 0 \\ \hline A & B & C & D \\ \hline \end{array} \\ SR = \begin{array}{ c c c c } \hline E & F & G & H \\ \hline \end{array} \end{array}$	A0=A A1=B A2=C A3=D A4=E A5=F A6=G A7=H	A0=D A1=C A2=B A3=A A4=H A5=G A6=F A7=E

Table 5-6 lists all the mnemonics available for instructions that are endian sensitive. In a future version, a pointer will be added from each line of the table to the correct table above that describes the way it behaves in the different endian modes.

**Table 5-6. Memory Access Instructions**

Instruction ID#	Instruction Mnemonic	
614	AND.W	#{iU16B0},(R)
635	AND.W	#{iU16B0},(SP+{AS16W0})
615	AND.W	#{iU16B0},(SP-{AU5W1})
616	AND.W	#{iU16B0},<({AU16W0})
281	BMCHG.W	#{iU16B0},(R)
448	BMCHG.W	#{iU16B0},(SP+{AS16W0})
268	BMCHG.W	#{iU16B0},(SP-{AU5W1})
317	BMCHG.W	#{iU16B0},<({AU16W0})
279	BMCLR.W	#{iU16B0},(R)
446	BMCLR.W	#{iU16B0},(SP+{AS16W0})
266	BMCLR.W	#{iU16B0},(SP-{AU5W1})
313	BMCLR.W	#{iU16B0},<({AU16W0})
280	BMSET.W	#{iU16B0},(R)
447	BMSET.W	#{iU16B0},(SP+{AS16W0})
267	BMSET.W	#{iU16B0},(SP-{AU5W1})
315	BMSET.W	#{iU16B0},<({AU16W0})
457	BMTSET.W	#{iU16B0},(R)
451	BMTSET.W	#{iU16B0},(SP+{AS16W0})
452	BMTSET.W	#{iU16B0},(SP-{AU5W1})
461	BMTSET.W	#{iU16B0},<({AU16W0})
291	BMTSTC.W	#{iU16B0},(R)
449	BMTSTC.W	#{iU16B0},(SP+{AS16W0})
282	BMTSTC.W	#{iU16B0},(SP-{AU5W1})
319	BMTSTC.W	#{iU16B0},<({AU16W0})
292	BMTSTS.W	#{iU16B0},(R)
450	BMTSTS.W	#{iU16B0},(SP+{AS16W0})
283	BMTSTS.W	#{iU16B0},(SP-{AU5W1})
321	BMTSTS.W	#{iU16B0},<({AU16W0})
157	BSR{D}	<*+{AS8W1}

**Table 5-6. Memory Access Instructions (Continued)**

Instruction ID#	Instruction Mnemonic	
237	BSR{D}	>*+{AS20W1}
622	EOR.W	#iU16B0),(R)
637	EOR.W	#iU16B0),(SP+{AS16W0})
623	EOR.W	#iU16B0),(SP-{{AU5W1}}
624	EOR.W	#iU16B0),<({AU16W0})
327	JSR{D}	{AU32W0}
115	MOVE.2F	{ea_MMM},Dh
608	MOVE.2L	Dh,{ea_MMM}
113	MOVE.2W	Dh,{ea_MMM}
606	MOVE.4F	{ea_MMM},Dk
605	MOVE.4W	Dk,{ea_MMM}
159	MOVE.B	HHHH,{ea_0MM}
161	MOVE.B	{ea_0MM},HHHH
216	MOVE.B	(SP+{AS15B0}),HHHH
225	MOVE.B	<({AU16B0}),HHHH
207	MOVE.B	HHHH,(R+{AS15B0})
213	MOVE.B	HHHH,(SP+{AS15B0})
301	MOVE.B	HHHH,({AU32B0})
222	MOVE.B	HHHH,<({AU16B0})
210	MOVE.F	(R+{AS15W0}),Dj
218	MOVE.F	(SP+{AS15W0}),Dj
304	MOVE.F	({AU32W0}),Dj
227	MOVE.F	<({AU16W0}),Dj
172	MOVE.F	Dj,{ea_0MM}
112	MOVE.F	{ea_MMM},Dj
462	MOVE.L	(SP+{AS15L0}),DQ.E
464	MOVE.L	(SP+{AS15L0}),Dq.E
465	MOVE.L	({AU32L0}),DQ.E
467	MOVE.L	({AU32L0}),Dq.E
165	MOVE.L	DDDD,(R)
217	MOVE.L	DDDDD,(SP+{AS15L0})
226	MOVE.L	DDDDD,<({AU16L0})

**Table 5-6. Memory Access Instructions (Continued)**

Instruction ID#	Instruction Mnemonic	
463	MOVE.L	Df1.E:Df2.E,(SP+{AS15L0})
466	MOVE.L	Df1.E:Df2.E,({AU32L0})
122	MOVE.L	HHHH,(R+r)
208	MOVE.L	HHHH,(R+{AS15L0})
124	MOVE.L	HHHH,(R+{AU3L2})
155	MOVE.L	HHHH,(SP-{{AU6L2})
302	MOVE.L	HHHH,({AU32L0})
110	MOVE.L	HHHH,{ea_MMM}
299	MOVE.W	#{{iS16B0}},(R)
460	MOVE.W	#{{iS16B0}},(SP+{AS16W0})
294	MOVE.W	#{{iS16B0}},(SP-{{AU5W1})
324	MOVE.W	#{{iS16B0}},<({AU16W0})
164	MOVE.W	DDDD,(R)
212	MOVE.W	DDDDD,(SP+{AS15W0})
221	MOVE.W	DDDDD,<({AU16W0})
121	MOVE.W	HHHH,(R+r)
206	MOVE.W	HHHH,(R+{AS15W0})
123	MOVE.W	HHHH,(R+{{AU3W1})
154	MOVE.W	HHHH,(SP-{{AU6W1})
300	MOVE.W	HHHH,({AU32W0})
109	MOVE.W	HHHH,{ea_MMM}
114	MOVES.2F	Dh,{ea_MMM}
607	MOVES.4F	Dk,{ea_MMM}
211	MOVES.F	Dj,(R+{AS15W0})
219	MOVES.F	Dj,(SP+{AS15W0})
305	MOVES.F	Dj,({AU32W0})
228	MOVES.F	Dj,<({AU16W0})
111	MOVES.F	Dj,{ea_MMM}
117	MOVES.L	Dj,{ea_MMM}
160	MOVEU.B	{ea_0MM},HHHH
403	MOVEU.B	(R+{AS15B0}),HHHH
214	MOVEU.B	(SP+{AS15B0}),HHHH



**Table 5-6. Memory Access Instructions (Continued)**

Instruction ID#	Instruction Mnemonic	
401	MOVEU.B	{{AU32B0}},HHHH
223	MOVEU.B	<{{AU16B0}},HHHH
209	MOVEU.W	(R+{AS15W0}),HHHH
220	MOVEU.W	(SP+{AS15W0}),DDDDD
303	MOVEU.W	{{AU32W0}},HHHH
229	MOVEU.W	<{{AU16W0}},DDDDD
116	MOVEU.W	{ea_MMM},HHHH
626	NOT.W	(R)
638	NOT.W	(SP+{AS16W0})
627	NOT.W	(SP-{AU5W1})
628	NOT.W	<{{AU16W0}}
618	OR.W	#iU16B0),(R)
636	OR.W	#iU16B0),(SP+{AS16W0})
619	OR.W	#iU16B0),(SP-{AU5W1})
620	OR.W	#iU16B0),<{{AU16W0}}
170	POP	EEEEEE
443	POP	eeeeee
171	POPN	EEEEEE
444	POPN	eeeeee
166	PUSH	EEEEEE
441	PUSH	eeeeee
168	PUSHN	EEEEEE
442	PUSHN	eeeeee
194	RTE{D}	
439	RTSTK{D}	
193	RTS{D}	
186	TRAP	
473	VSL.2F	D1:D3,(R)+N0
472	VSL.2W	D1:D3,(R)+N0
471	VSL.4F	D2:D6:D1:D3,(R)+N0
470	VSL.4W	D2:D6:D1:D3,(R)+N0

## 5.3 Comments

The following are some comments related to some of the instructions. Based on these comments, a more detailed definition of these cases will be described in a future version of this document.

### 5.3.1 MOVE Multiple Registers

In case of instructions that access more than one register (such as MOVE.2W, etc.), notice should be paid to which register of the pair uses what address. For example, in the following instruction, d0 uses A0+A1 and d1 uses A2+A3 in little endian mode.

```
MOVE.2W d0:d1, (r0)
```

For all instructions except the VSL instructions, the relevant registers are consecutive starting at an “aligned” number.

### 5.3.2 MOVE.L for the Extension Registers

MOVE.L for data extensions has also single-extension variants.

The location of this extension in memory depends on the parity of the register number.

### 5.3.3 PUSH/POP Instructions

PUSH/POP instructions are 32-bit operations. In case two such instructions are used, the identity of which operand ends where in memory should be defined.

PUSH/POP can also be done on single or paired extensions—the same as MOVE.L for extension. In case of a paired extension operand, it is *not* like grouping to PUSH/POP together.

### 5.3.4 BSR/JSR

BSR/JSR, etc. also push the SR, so the comment on PUSH instructions also apply here.

### 5.3.5 Control instructions

ILLEGAL and “interrupt service” will be added to the table of Control Instructions (Table 5-5).

# Chapter 6

## Assembler Syntax and Directives

---

This chapter identifies the directives and special characters that must be recognized by SC100 assemblers. Directives are commands that instruct the assembler to carry out some action during assembly, rather than instructions to be directly translated into object code.

A detailed description of the SC00 Assembler, its syntax, and directives can be found in the *SC100 Assembly Language Tools User's Manual*.

### 6.1 Assembler-Significant Characters

Several one- and two-character sequences are significant to the assembler. Some have multiple meanings depending on the context in which they are used. These characters are as follows:

;	Comment delimiter
::	Unreported comment delimiter
\	Line continuation character or macro dummy argument concatenation operator
?	Macro value substitution operator
%	Macro hex value substitution operator
^	Macro local label override operator
"	Macro string delimiter or quoted string DEFINE expansion character
@	Function delimiter
*	Location counter substitution
++	String concatenation operator
[ ]	Substring delimiter or instruction grouping delimiter
<<	I/O short addressing mode force operator
<	Short addressing mode force operator
>	Long addressing mode force operator
#	Immediate addressing mode operator
#<	Immediate short addressing mode force operator
#>	Immediate long addressing mode force operator

## 6.2 Assembler Directives

Assembler directives can be grouped by function into the following types:

1. Assembly control
2. Symbol definition
3. Data definition/storage allocation
4. Listing control and options
5. Object file control
6. Macros and conditional assembly

### 6.2.1 Assembly Control

The directives used for assembly control are the following:

COMMENT	Start comment lines
DEFINE	Define substitution string
END	End of source program
FAIL	Programmer generated error message
FORCE <sup>1</sup>	Set operand forcing mode
HIMEM	Set high memory bounds
INCLUDE	Include secondary file
LOMEM	Set low memory bounds
MODE	Change relocation mode
MSG	Programmer generated message
ORG	Initialize memory space and location counters
RADIX	Change input radix for constants
RDIRECT <sup>1</sup>	Remove directive or mnemonic from table
UNDEF	Undefine DEFINE symbol
WARN	Programmer generated warning

---

1. Not currently supported.

## 6.2.2 Symbol Definition

The directives used to control symbol definition are the following:

ENDSEC	End section
EQU	Equate symbol to a value
GLOBAL	Global section symbol declaration
GSET	Set global symbol to a value
SECFLAGS	Set ELF section flags
LOCAL <sup>1</sup>	Local section symbol declaration
SECTION	Start section
SECTYPE	Set ELF section type
SET	Set symbol to a value
SIZE	Set size of symbol in the ELF symbol table
TYPE	Set symbol type in the ELF symbol table
XDEF <sup>1</sup>	External section symbol definition
XREF <sup>1</sup>	External section symbol reference

## 6.2.3 Data Definition/Storage Allocation

The directives used to control constant data definition and storage allocation are the following:

ALIGN	Set address to modulo boundary
BADDR	Set buffer address
BSB	Block storage bit-reverse
BSC	Block storage of constant
BUFFER	Start buffer
DC, DCW	Define constant (16-bits)
DCB	Define constant byte (8-bits)
DCL	Define constant long word (32-bits)
DS	Define storage
DSR	Define reverse carry storage
ENDBUF	End buffer

---

1. Not currently supported.

## 6.2.4 Object File Control

The directives used for control of the object file are the following<sup>1</sup>:

COBJ	Comment object code
IDENT	Object code identification record
SYMOBJ	Write symbol information to object file

## 6.2.5 Macros and Conditional Assembly

The directives used for macros and conditional assembly are the following:

DUP	Duplicate sequence of source lines
DUPA	Duplicate sequence with arguments
DUPC	Duplicate sequence with characters
DUPF	Duplicate sequence in loop
ENDIF	End of conditional assembly
ENDM	End of macro definition
EXITM	Exit macro
IF	Conditional assembly directive
MACLIB	Macro library
MACRO	Macro definition
PMACRO	Purge macro definition

## 6.2.6 Assembler Syntax

The following sections provide details on assembler syntax.

### 6.2.6.1 Input File Format

Programs written in assembly language consist of a sequence of source statements. Any source statement can be extended to one or more lines by including the line continuation character (\) as the last character on the line to be continued. A source statement (first line and any continuation lines) can be a maximum of 4000 characters long. Upper and lower case letters are equivalent for assembler mnemonics and directives, but are distinct for labels, symbols, directive arguments, and literal strings.

---

1. Object file control directives are not currently supported.

## 6.2.6.2 Symbol Names

Symbol names can be from one to 4000 characters long. The first character of a symbol must be alphabetic (upper or lower case); any remaining characters can be either alphanumeric (A-Z, a-z, 0-9) or the underscore character (`_`). Upper and lower case letters in symbols are considered distinct unless the IC option is in effect. Symbol names and other identifiers beginning with a `'.` are legal but reserved for the system.

a) Valid Symbol Names	b) Invalid Symbol Names
<code>loop_1</code>	<code>1_loop</code>
<code>ENTRY</code>	<code>loop.e</code>
<code>a_B_c</code>	

Certain identifiers are reserved by the assembler and cannot be used. These identifiers are the upper- or lower-case name of any SC140 DSP core processor register.

## 6.2.6.3 Strings

One or more ASCII characters enclosed by single quotes (`'`) constitute a literal ASCII string. In order to specify an apostrophe within a literal string, two consecutive apostrophes must appear where the single apostrophe is intended. Strings are used as operands for some assembler directives and also can be used to a limited extent in expressions.

A string may also be enclosed in double quotes (`"`) in which case any `DEFINE` directive symbols contained in the string would be expanded. The double quote should be used with care inside macros since it is used as a dummy argument string operator. In that case, the macro concatenation operator can be used to escape the double-quoted string if desired.

Two strings separated by the string concatenation operator (`++`) will be recognized by the assembler as equivalent to the concatenation of the two strings. For example, the following two strings are equivalent:

```
'ABC'++'DEF' = 'ABCDEF'
```

The assembler has a substring extraction capability using the square brackets (`[ ]`). Refer to the following example:

```
['abcdefg',1,3] = 'bcd'
```

Substrings may be used wherever strings are valid and can be nested. There are also functions for determining the length of a string and the position of one string within another.

### 6.2.6.4 Source Statement Format

Each source statement may include several fields (for a single instruction assembly line) separated by one or more spaces or tabs: a label field, an operation field, an operand field or data transfer fields, and a comment field. Only fields preceding the comment field are considered significant to the assembler; the comment field is ignored. Opcode, directives, or pseudo ops may not begin in column 1 (first character location). Labels must begin in column 1 unless followed by a colon, as shown in Example 6-1.

#### Example 6-1. Column 1 Labels

---

```
lab      move.2w (r0),d0d1      ; Two word memory to register move
```

---

SC140 allows instruction groupings in which multiple instructions are executed in parallel. Instruction groups are delimited with []. An instruction group may not begin in column 1. An instruction group may span multiple lines without the use of a continuation character, as shown in Example 6-2.

#### Example 6-2. Multiple-Line Instruction Group

---

```
[
  move.w (sp-14),r2
  clr d5          clr d6 clr d7          clr d4
]
```

---

### 6.2.6.5 Labels

The label field is the first field of a source statement. A space or tab as the first character on a line ordinarily indicates that the label field is empty. Labels are subject to the following rules:

- The first character of a label is an alphabetic character.
- A label that has an underscore (`_`) as the first character is a *global label*.
- Labels may be indented if the label symbol is immediately followed by a colon (`:`) with no intervening spaces. In this case, all characters preceding the label on the line must be whitespace characters—spaces or tabs.
- A label may occur only once in the label field of an individual source file unless it is used as a local label, a label local to a section, or is used with the SET directive. If a non-local label occurs more than once in a label field, each reference to that label after the first will be flagged as an error.
- A line consisting only of a label only is valid and has the effect of assigning the value of the location counter to the label. With the exception of some directives, a label is assigned the value of the location counter of the first word of the instruction or data being assembled.

### 6.2.6.6 Operation Field

The operation field appears after the label field, and must be preceded by at least one space or tab. Entries in the operation field may be one of three types:

Opcode	Mnemonics that correspond directly to DSP machine instructions.
Directive	Special operation codes known to the assembler which control the assembly process.
Macro call	Invocation of a previously defined macro which is to be inserted in place of the macro call.



### **6.2.6.7 Operand Field**

The interpretation of the operand field is dependent on the contents of the operation field. The operand field, if present, must follow the operation field, and must be preceded by at least one space or tab. The operand field may contain a symbol, an expression, or a combination of symbols and expressions separated by commas with no intervening spaces.

### **6.2.6.8 Comment Fields**

Comments are ignored by the assembler, but can be included in the source file for documentation purposes. A comment field is composed of any characters (not part of a literal string) that are preceded by a semicolon (;).



## Symbols

" 6-1  
# 6-1  
#< 6-1  
#> 6-1  
% 6-1  
\* 6-1  
++ 6-1  
; 6-1  
;; 6-1  
< 6-1  
<< 6-1  
> 6-1  
? 6-1  
@ 6-1  
\ 6-1  
^ 6-1

## A

Address Generation Unit (AGU) 5-7  
Argument Pointers 2-7  
Arrays 2-3

## B

Bit Fields 2-3

## C

C In-Line Assembly Syntax 3-1  
C Name Mapping 3-1  
C Preprocessor Predefines 3-1  
Calling Sequences 2-6  
Check Function 4-6  
Compound Data Type 2-3  
Constants 4-9

## D

Data Types 2-1  
Directive  
    assembly control 6-2  
    BADDR 6-3  
    BSB 6-3  
    BSC 6-3  
    BUFFER 6-3  
    COBJ 6-4  
    COMMENT 6-2  
    data definition 6-3

DC 6-3  
DCB 6-3  
DEFINE 6-2  
DS 6-3  
DSR 6-3  
DUP 6-4  
DUPA 6-4  
DUPC 6-4  
DUPF 6-4  
END 6-2  
ENDBUF 6-3  
ENDIF 6-4  
ENDM 6-4  
ENDSEC 6-3  
EQU 6-3  
EXITM 6-4  
FAIL 6-2  
FORCE 6-2  
GLOBAL 6-3  
GSET 6-3  
HIMEM 6-2  
IDENT 6-4  
IF 6-4  
INCLUDE 6-2  
LOCAL 6-3  
LOMEM 6-2  
MACLIB 6-4  
MACRO 6-4  
macro 6-4  
MODE 6-2  
MSG 6-2  
object file 6-4  
ORG 6-2  
PMACRO 6-4  
RADIX 6-2  
RDIRECT 6-2  
SCSJMP 6-2  
SECFLAGS 6-3  
SECTION 6-3  
SECTYPE 6-3  
SET 6-3  
SIZE 6-3  
symbol definition 6-3  
SYMOBJ 6-4  
TYPE 6-3  
UNDEF 6-2  
WARN 6-2  
XDEF 6-3

XREF 6-3  
Dynamic Memory Allocation 2-8

## E

Endian 5-1  
Executable and Linking Format (ELF)  
  Header 4-2  
  Relocation 4-5  
  Sections 4-3

## F

Floating Point Routines 3-4  
Fractional Arithmetic Support 3-2  
Frame Pointers 2-7

## H

Hardware Loops 2-8  
Hi Function 4-7

## I

Identifiers  
  Op 4-8  
  PA 4-8  
  PC 4-8  
Integer Routines 3-5  
Interrupt Handlers 2-6

## L

Label 6-6  
  local 6-6  
Libraries 3-3  
Line Function 4-7  
Lo Function 4-7  
Local label 6-6

## M

Memcheck Function 4-8

## O

Op Identifier 4-8  
Operating Modes 2-8  
Operators 4-9  
Optional Prefix 3-3

## P

PA Identifier 4-8  
Pack Function 4-6  
PC Identifier 4-8  
Program Headers 4-24

## R

Relocation Functions 4-6  
  Check 4-6  
  Hi 4-7  
  Line 4-7  
  Lo 4-7  
  Memcheck 4-8  
  Pack 4-6  
  Size 4-7  
  Sym 4-7  
Reserved Names 4-8  
Reserved Symbol Names 4-4  
Return Value 3-6

## S

Signature Symbols 3-5  
Size Function 4-7  
Stack 2-4  
Stack Frame Layout 2-6  
String  
  concatenation 6-5  
  substring 6-5  
Structures 2-3  
Sym Function 4-7

## U

Unions 2-3





# **STAR CORE**

**BRIGHTER DSP TECHNOLOGY!**

How to reach us:

**Motorola Literature Distribution**

P.O. Box 5405  
Denver, Colorado 80217  
1 (800) 441-2447

**Asia/Pacific**

Motorola Semiconductors H.K. Ltd., Hong Kong  
852-26629298

**Japan**

Motorola Japan, Ltd., Shinagawa-ku, Japan  
81-3-5487-8488

**Motorola Fax Back System (Mfax™)**

1 (800) 774-1848; RMFAX0@email.sps.mot.com

**DSP Helpline**

dsphelp@dsp.sps.mot.com

**Technical Resource Center**

1 (800) 521-6274

**Internet**

<http://www.motorola-dsp.com>

**Lucent Technologies Microelectronics Group**

**Internet**

<http://www.lucent.com/micro/dsp>

**Email**

docmaster@micro.lucent.com

**U.S./Canada**

Lucent Technologies Microelectronics Group  
1-800-372-2447, FAX 610-712-4106  
In CANADA: 1-800-553-2448, FAX 610-712-4106

**Asia/Pacific Microelectronics Group**

Lucent Technologies Singapore Pte. Ltd., Singapore  
Tel. (65) 778 8833, FAX (65) 777 7495

**China Microelectronics Group**

Lucent Technologies (China) Co., Ltd., Shanghai  
Tel. (86) 21 6440 0468, ext. 316, Fax (86)21 6440 0652

**Japan Microelectronics Group**

Lucent Technologies Japan, Ltd., Shinagawa-ku, Japan  
Tel. (81) 3 5421 1600, FAX (81) 3 5421 1700

**Europe Microelectronics Group Dataline**

Tel. (44) 1189 324 299, FAX (44) 1189 328 148



microelectronics group

**Lucent Technologies**  
Bell Labs Innovations

