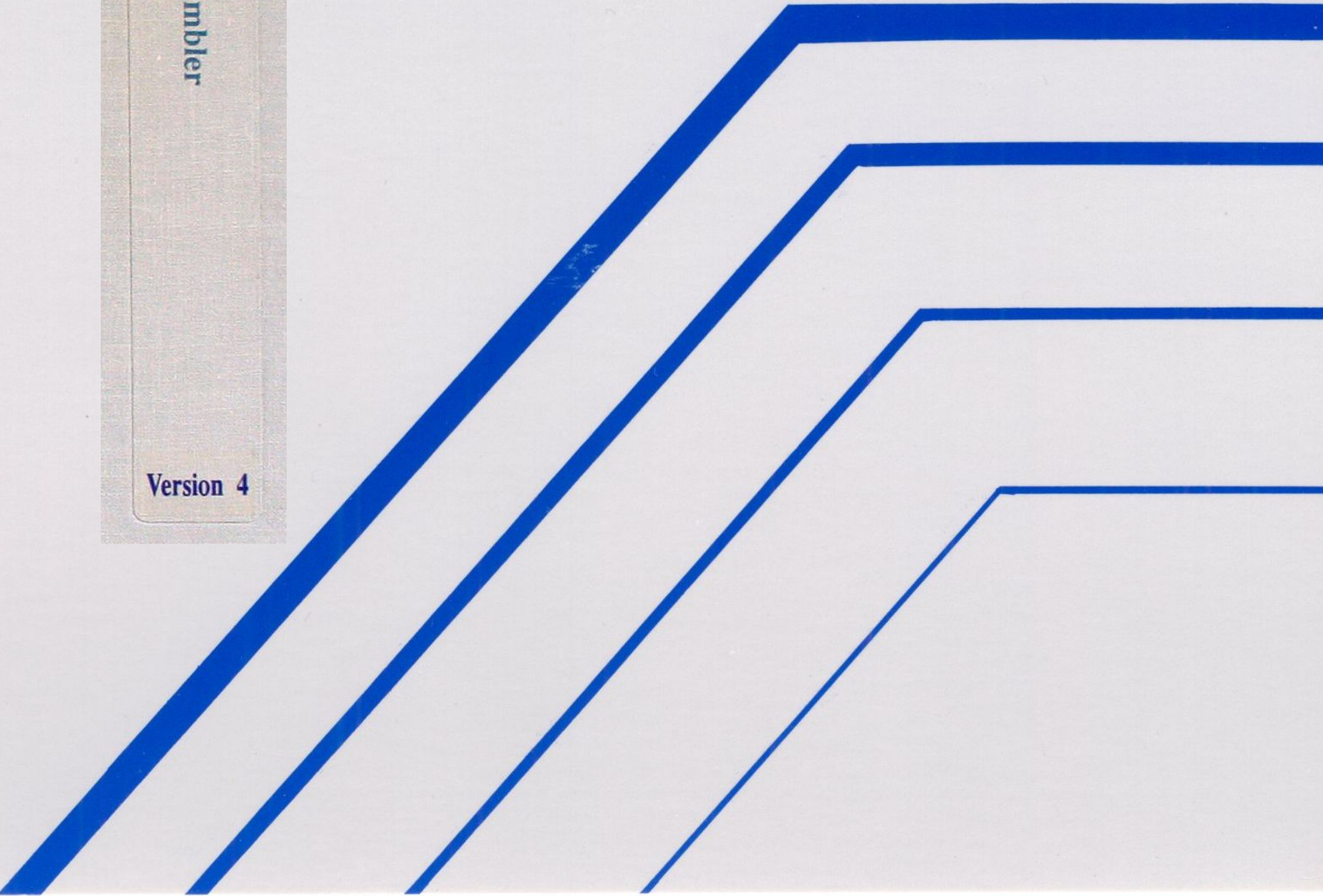
GNX Assembler

Version 4



Series 32000®

GNX — Version 4.4
Assembler Reference Manual

Customer Order Number 424010497-004

June 1992

REVISION RECORD

VERSION	RELEASE DATES	SUMMARY OF CHANGES
4.0	May 1990	First Release. Introduction of the new macro and conditional assembler. Introduction of procedure support. *Series 32000/EP* support.
4.1	Sep 1990	Enhanced macro listing control and symbolic debugging of assembly programs.
4.2	Feb 1991	Synchronization revision. No changes.
4.3	Aug 1991	Minor manual corrections.
4.4	June 1992	MS-DOS support added.

PREFACE

This document describes the GNX *Series 32000* Assembler, a support program that assembles *Series 32000* Assembly language source programs. This manual defines the syntax, format, and function of the following:

- *Series 32000* Assembly Language Elements
- Assembly Language Programs
- Instructions and Instruction Operands
- Assembler Directives
- Assembler Procedure Support
- Macro and Conditional Assembly
- Series 32000 Assembler Invocation

The GNX Assembler is intended for use as a component in the *Series 32000* GNX tools family to create assembly language programs for *Series 32000*-based systems. It may be used in either a native or a cross environment.

A complete list of National Semiconductor's international offices may be found on the inside back cover of this manual.

ISE, SYS32 and GENIX are trademarks of National Semiconductor Corporation.

Series 32000 is a registered trademark of National Semiconductor Corporation.

Portions of this document are derived from AT&T copyrighted material and reproduced under license from AT&T; portions are based in part on the Fourth Berkeley Software Distribution under license from The Regents of the University of California.

UNIX is a registered trademark of AT&T.

VAX, VMS, and DEC are trademarks of Digital Equipment Corporation.

SUN and SunOS are trademarks of SUN Microsystems Inc.

CONTENTS

Chapter 1 INTRODUCTION AND OVERVIEW

Chapter 2 ELEMENTS OF THE GNX ASSEMBLY LANGUAGE

Chapter 3 GNX ASSEMBLER PROGRAMS

Chapter 4 INSTRUCTION OPERANDS

Chapter 5 SERIES 32000 INSTRUCTION SET

Chapter 6 GNX ASSEMBLER DIRECTIVES

Chapter 7 PROCEDURE SUPPORT

Chapter 8 MACRO AND CONDITIONAL ASSEMBLER

Chapter 9 INVOCATION AND OPERATION

Appendix A DIRECTIVE SUMMARY

Appendix B GNX ASSEMBLER RESERVED SYMBOLS

Appendix C PROGRAM EXAMPLES

Appendix D INITIALIZATION OF INTERRUPTS

Appendix E SERIES 32000 STANDARD CALLING CONVENTIONS

Appendix F COMPATIBLY-SUPPORTED MACROS

Appendix G GLOSSARY

FIGURES

TABLES

INDEX

Chapter 1

INTRODUCTION AND OVERVIEW

1.1 INTRODUCTION

The GNX Assembler is a support program that assembles *Series 32000* Assembly Language source programs and generates relocatable object modules. Relocatable object modules may be linked to create executable load modules which may be run on *Series 32000* microprocessor-based systems that support the Common Object File Format (COFF) as implemented by National Semiconductor. The *Series 32000* GNX (GENIX Native and Cross-Support) language tools provide linkage and library maintenance programs.

This manual describes the GNX Assembler in detail and is organized as follows:

Chapter 1, Introduction and Overview
Introduces the GNX Assembler, summarizes its features, and describes the *Series 32000* registers.

Chapter 2, Elements of the GNX Assembly Language
Describes the format of the GNX Assembly Language statements, constants, values, symbols, and expressions.

Chapter 3, GNX Assembler Programs
Describes program segments, linkage, and relocation.

Chapter 4, Instruction Operands
Describes the syntax of the GNX Assembly Language instruction operands.

Chapter 5, Series 32000 Instruction Set
Lists the syntax of the *Series 32000* instruction set.

Chapter 6, GNX Assembler Directives
Defines the syntax and function of the GNX Assembler directives.

Chapter 7, Procedure Support
Provides a review of the GNX procedure support.

Chapter 8, Macro and Conditional Assembly
Describes the new macro-assembler.

Chapter 9, Invocation and Operation
Describes the GNX Assembler, assembly options, output formats, error messages, and the Symbol Table.

Appendix A, Directive Summary
Summarizes the GNX Assembler directive syntax and function.

Appendix B, Reserved Symbols
Lists the GNX Assembler reserved symbols.

Appendix C, Program Examples
Provides GNX Assembly Language program examples.

Appendix D, Initialization of Interrupts
Illustrates interrupt initialization for a *Series 32000* system.

Appendix E, Series 32000 Standard Calling Conventions
Describes elements of the *Series 32000* standard calling sequence.

Appendix F, Compatibly-Supported Macros
Describes the Version 2.0 macro-assembler.

Appendix G, Glossary
Provides a glossary of GNX terms.

1.2 OVERVIEW OF THE GNX ASSEMBLER FEATURES

The GNX Assembler provides a number of features for efficient assembly language programming.

Input and Output Files. The GNX Assembler generates an object code file, an optional listing file, an optional cross-reference listing, and an optional symbol table dump from an assembler source file. The object code file consists of assembled statements suitable for execution after the appropriate linking process. The listing file consists of the source file statements, and the assembled code, if the source file assembles successfully; otherwise, the listing file consists of error messages and source file statements that caused the error. Input and output files, listing file format, cross-reference listing symbol table dump, and error messages are described in Chapter 9.

Instruction Set. The GNX Assembler supports the complete *Series 32000* instruction set, including the integer, quick integer, extended integer, bit, bit field, Boolean, string, packed decimal, array, block, processor control, and processor service instructions. The GNX Assembler also supports memory management and floating-point instruction sets for systems with the optional NS32082 or NS32382 Memory Management Unit, NS32081, NS32181, NS32381 or NS32580 Floating-Point Unit, and NS32CG16, NS32CG160 and NS32FX16 High Performance Graphic Instructions. Chapter 5 defines the syntax of all *Series 32000* instructions. Instruction operation is described in detail in the *Series 32000* and *Series 32000/EP Programmer's Reference Manual.*

Addressing Modes. The GNX Assembler supports eight general addressing modes: register, register relative, memory, memory relative, immediate, absolute, top of stack, external, and also provides scaled indexing for all of these modes except immediate.

Data Types. The GNX Assembler recognizes a variety of operand data types including integers (byte, word, double-word), single- and double-precision floating-point numbers, packed decimal numbers, bits, and bit fields.

GNX Assembler Directives. The GNX Assembler provides directives to create symbolic labels, generate data, allocate storage, control program listings, control linkage, control line number table, control program segments, define module table entry, define symbol table entry, define macros, and define file name.

1.3 SERIES 32000 REGISTERS

The *Series 32000* system has four sets of registers; these are described in Sections 1.3.1 through 1.3.4.

- General Purpose registers
- Dedicated registers
- Floating-Point registers
- Memory Management registers

1.3.1 General Purpose Registers

There are eight General Purpose registers. The register names are: r0, r1, r2, r3, r4, r5, r6, and r7. The General Purpose registers provide temporary storage for address computation, arithmetic operations, and parameter passing.

Each register is 32 bits long and may be used in byte, word, and double-word operations. Byte operations affect the register's low-order eight bits only; word operations affect the low-order 16 bits, and double-word operations affect all 32 bits.

General Purpose registers may be combined to form even/odd register pairs: r0/r1, r2/r3, r4/r5, r6/r7. A register pair is 64 bits in length and may hold word, double-word, and quad-word data. The odd register holds the high-order byte(s); the even register holds low-order byte(s). Register pair names are: r0, r2, r4, and r6.

1.3.2 Dedicated Registers

The eight Dedicated registers store memory addresses and status information needed for CPU operation:

Register	Name	Contents
Program Counter	pc	address of current instruction
Static Base	sb	address of current Static Base Area
User Stack Pointer	sp1	address of top of User Stack
Interrupt Stack Pointer	sp0	address of top of Interrupt Stack
Frame Pointer	fp	address of current Frame
Interrupt Base	intbase	address of Interrupt Dispatch Table
Module	mod	address of current Module Descriptor
Processor Status	psr	Processor Status flags

The pc, sb, sp1, sp0, fp, and intbase registers are each 32 bits long; however, these registers contain memory addresses in which the number of bits used for address representation and calculation depends on the actual CPU used. 24 bits are used for address representation in the NS32Oxx and the NS32CG16 CPUs. These CPUs will be referred to as 24-pin address processors. 32 bits are used for address representation in the NS32332, NS32532, NS32FX16, NS32CG160, NS32GX32, and NS32GX320 CPUs. These CPUs will be referred to as 32-pin address processors. The mod register contains a memory address, and the psr register contains the processor status flags. A program can work only with one of the hardware stack registers at any given time: sp0 or sp1. The CPU selects sp0 as the current stack pointer if the `s` bit in the psr is zero, and sp1 if the `s` bit is one. The GNX Assembler uses the symbol sp to refer to the currently selected stack. A program has no control over which stack will be accessed, except by changing the `s` bit. Consequently, the GNX Assembler does not support the symbols sp0 and sp1, and any reference to "stack" in this manual refers to the currently selected stack.

The psr status flags define the current operational mode of the CPU and the execution results of the previous instruction(s). The psr has the following form:

Supervisor Flags					User Flags							
x x x x	i	p	s	u	n	z	f	x x	l	t	c	
15				8	7						0	

x - reserved for future use

The status flags have the following functions:

User Flags (available to all programs)

c — is the Carry flag. On execution of an add or subtract instruction, `c` flag is set to 1 on a carry or a borrow and is set to 0 when neither occur.

t — is the Trace-Trap flag. The Trace-Trap flag enables/disables the system Trace-Trap (TRC). The TRC stops program execution on completion of each program instruction and permits program single-stepping. When `t` is 1, the trace trap is enabled. Also see `p` flag.

l — is the Low flag. The Low flag signals the result of an unsigned comparison between two integers. The `l` flag is set to 1 if the second integer is less than the first; `l` flag is set to 0 if the second integer is greater than, or equal to, the first.

f — is the General Condition flag. If set to 1, this flag indicates an overflow in arithmetic operations, a set bit in a bit instruction, an until/while condition in a string instruction, or an out-of-bounds subscript in a check instruction. If set to 0, this flag indicates no overflow in arithmetic operations, a clear bit in a bit instruction, no until/while condition in a string instruction, and no out-of-bounds subscript in a check instruction.

z	is the Zero flag. The Zero flag indicates the result of the comparison between two integers. The `z` flag is set to 1 if the integers are equal and is set to 0 if not equal.
n	is the Negative flag. The Negative flag indicates the result of a signed comparison between two integers. The `n` flag is set to 1 if the second integer is less than the first and is set to 0 if the second integer is greater than, or equal to, the first.

Supervisor Flags (available to supervisor programs only)

u	is the User Mode flag. The User Mode flag sets the current mode of system operation. If `u` is 1, the system is in the user mode and no privileged instructions may be executed. If `ua` is 0, the system is in the supervisor mode and all instructions may be executed.
s	is the Stack flag. The Stack flag selects the current stack pointer. If `s` is 1, the User Stack Pointer (sp1) is active. If `s` is 0, the Interrupt Stack Pointer (sp0) is active. On an interrupt or a trap, the `s` bit is automatically cleared.
p	is the Trace Pending flag. The Trace Pending flag, in conjunction with the `t` flag, enables/disables the trace trap. At the start of an instruction, the `t` flag contents are copied to the `p` flag. At the end of the instruction, if `p` is 1, a trace trap is taken. If `p` is 0, no trace trap is taken.
i	is the Interrupt flag. The Interrupt flag enables/disables the vectored and nonvectored interrupts. If `i` is 1, the interrupts are enabled. If `i` is 0, then all interrupts except the Non-Maskable Interrupt (NMI) are disabled.

1.3.3 Floating-Point Registers

The floating-point registers for the NS32081, NS32181, NS32381, and NS32580 are described in this Section. The Floating-Point Status Register (fsr) contains the floating-point status flags. The *Series 32000 Programmer's Reference Manual* describes the flags in detail.

The NS32081 Registers

The NS32081 registers (f0, f1, f2, f3, f4, f5, f6 and f7) provide temporary work space for floating-point operations. Each register is 32 bits long and may hold single-precision floating-point numbers.

These registers may be combined to form even/odd register pairs: f0/f1, f2/f3, f4/f5, or f6/f7. A register pair is 64 bits in length and may hold double-precision floating-point numbers. The odd register contains the high-order bytes of the number; the even register contains the low-order bytes. Register pair names are: f0, f2, f4, and f6.

The NS32181,NS32381 and NS32580 Registers

The NS32181, NS32381 and NS32580 registers (f0, f1, f2, f3, f4, f5, f6, and f7) provide temporary work space for floating-point operations. Each register is 32 bits long and may hold single-precision floating-point numbers.

The NS32181, NS32381 and NS32580 registers (*l*0, *l*1, *l*2, *l*3, *l*4, *l*5, *l*6, and *l*7) are 64 bits long and may hold double-precision floating-point numbers.

1.3.4 Memory Management Registers

The Memory Management registers support virtual memory and program debugging.

Memory Management registers are described in detail in the *Series 32000 Programmer's Reference Manual.*

NS32082 MMU Registers

The following is a comprehensive list of the Memory Management registers for the NS32082. All Memory Management registers, except sc0 and scl, are 32 bits long. Both sc0 and scl are 16 bits long and occupy a single 32-bit register. The assembler refers to sc0 and scl with the symbol sc. The sc0 register is contained in the lower 16-bit field of sc; scl is contained in the upper 16-bit field of sc.

Register	Name	Function
Page Table Register 0	ptb0	Contains the base address of the level 1 Page Table.
Page Table Register 1	ptb1	Contains the base address of the user mode level 1 Page Table (when MMU is in dual space operation).
Error/Invalidate Address	eia	Contains, on an error, the virtual address that caused the error. When written to, causes the removal of invalid Page Table entries from the MMU Translation Buffer.
Breakpoint Register 0	bpr0	Contains a breakpoint address. The system breaks execution when the address is accessed.
Breakpoint Register 1	bpr1	Contains a breakpoint address. The system breaks execution when the address is accessed.

Breakpoint Count	bcnt	Contains a count of the number of bpr0 breakpoint conditions that have been met. If count is 0, a break is taken. Otherwise, no break is taken.
Program Flow 0	pf0	Contains the address of the last nonsequential instruction.

Register	Name	Function
Program Flow 1	pf1	Contains the address of the next to last nonsequential instruction.
Sequential Count 0	sc0	Contains the number of sequential instructions executed since the last nonsequential instruction.
Sequential Count 1	sc1	Contains the number of sequential instructions executed prior to the last nonsequential instruction.
Memory Status Register	msr	Contains the status and control flags of the MMU.

NS32382 MMU Registers

The following is a comprehensive list of memory management registers for the NS32382. All registers except ivar0 and ivar1 can be read by the smr instruction. Ivar0 and ivar1 are write-only pseudo-registers. The tear, bear, and bdr registers are read-only registers and cannot be loaded by the lmr instruction. Writing to a read-only register has no effect on the MMU; however, avoid reading a write-only register since random data patterns may be returned.

Registers	Name	Description
Breakpoint Address Register	bar	Holds a virtual address for breakpoint address comparison during instruction and operand accesses.
Breakpoint Mask Register	bmr	Indicates which bit positions of the virtual address are to be compared when the Breakpoint Address Compare Function is enabled.

Breakpoint Data Register	bdr	Contains the virtual address of the multiplexed address data bus from the CPU when a breakpoint is detected.

Registers	**Name**	**Description**
Invalid Virtual Address Register 0	ivar0	Contains 20-bit physical numbers and 20-bit virtual address tag of the 32 most recently used pages. Used to invalidate entries with AS (Address Space)=0.
Invalid Virtual Address Register 1	ivar1	Contains 20-bit physical numbers and 20-bit virtual address tag of the 32 most recently used pages. Used to invalidate entries with AS=1.
MMU Control Register	mcr	Contains the different features provided by the MMU.
MMU Status Register	msr	Contains the status of the MMU when a translation exception or a bus error is reported to the CPU.
Translation Exception Address Register	tear	Is clocked when a translation exception occurs. The register contains the 32-bit virtual address which caused the translation exception.
Bus Error Address Register	bear	Is clocked when a CPU or MMU error occurs. This register contains the 32-bit virtual address which triggered the bus error.
Page Table Base 0	ptb0	Contains the base address used for address translation (when in superuser mode.)
Page Table Base 1	ptb1	Contains the base address used for address translation (when in user mode.)

NS32532 MMU Registers

The following is a comprehensive list of memory management registers for the NS32532. All registers except ivar0 and ivar1 can be read by the smr instruction. Ivar0 and ivar1 are write-only pseudo-registers. The tear register is a read-only register and cannot be loaded by the lmr instruction. Writing to a read-only register has no effect on the MMU. However, reading a write-only register should be avoided since random data patterns may be returned.

Registers	Name	Description
Invalid Virtual Address Register 0	ivar0	Contains 20-bit physical numbers and 20-bit virtual address tag of the 32 most recently used pages. Used to invalidate entries with AS=0.
Invalid Virtual Address Register 1	ivar1	Contains 20-bit physical numbers and 20-bit virtual address tag of the 32 most recently used pages. Used to invalidate entries with AS=1.
MMU Control Register	mcr	Contains the different features provided by the MMU.
MMU Status Register	msr	Contains the status of the MMU when a translation exception or a bus error is reported to the CPU.
Translation Exception Address Register	tear	Is clocked when a translation exception occurs. The register contains the 32-bit virtual address which caused the translation exception.
Page Table Base 0	ptb0	Contains the base address used for address translation (when in superuser mode.)
Page Table Base 1	ptb1	Contains the base address used for address translation (when in user mode.)

1.4 DEFINITION OF TERMS

The following terms are used throughout this document:

Software Module	A software module is a portion of a program that may be separately compiled or assembled and linked together with other software modules into an executable program image.
Series 32000 Module	A *Series 32000* module is a software module that uses the *Series 32000* architecture support for linkage. Currently, the GNX linker supports only one *Series 32000* module per program.
Relative Value	A relative value is a symbol or expression that specifies an address within one of the Common Object

	File Format (COFF) sections or the corresponding assembly program segment. Because such addresses are not bound to actual memory locations until link time, their value is relative to the base or starting address of the segment. Relative values are called *relocatable addresses.*
Absolute Value	An absolute value is a symbol or expression that specifies a numeric address. An absolute value or *absolute address* is unaffected by linkage.

1.5 DOCUMENTATION CONVENTIONS

The following documentation conventions are used in text, syntax descriptions, and examples in describing commands and parameters.

1.5.1 General Conventions

Nonprinting characters are indicated by enclosing a name for the character in angle brackets <>. For example, <CR> indicates the RETURN key, <ctrl/B> indicates the character input by simultaneously pressing the control key and the `B` key.

Constant-width type is used within text for filenames, directories, command names and program listings; it is also used to highlight individual numbers and letters. For example,

> the C preprocessor, `cpp`, resides in the `GNXDIR/lib` directory.

1.5.2 Conventions in Syntax Descriptions

The following conventions are used in syntax descriptions:

Constant-width boldface type indicates actual user input.

Italics indicate user-supplied items. The italicized word is a generic term for the actual operand that the user enters. For example,

`cc` [[*option*] ... [*filename*] ...] ...

Spaces or blanks, when present, are significant; they must be entered as shown. Multiple blanks or horizontal tabs may be used in place of a single blank.

{ }	Large braces enclose two or more items of which one, and only one, must be used. The items are separated from each other by a logical OR sign " | ."
[ ]	Large brackets enclose optional item(s).

|	Logical OR sign separates items of which one, and only one, may be used.
...	Three consecutive periods indicate optional repetition of the preceding item(s). If a group of items can be repeated, the group is enclosed in large parentheses "( )."
,,,	Three consecutive commas indicate optional repetition of the preceding item. Items must be separated by commas. If a group of items can be repeated, the group is enclosed in large parentheses "( )."
( )	Large parentheses enclose items which need to be grouped together for optional repetition. If three consecutive commas or periods follow an item, only that item may be repeated. The parentheses indicate that the group may be repeated.
␣	Indicates a space. ␣ is only used to indicate a specific number of required spaces.

All other characters or symbols appearing in the syntax must be entered as shown. Brackets, parentheses, or braces which must be entered, are smaller than the symbols used to describe the syntax. (Compare user-entered [ ], with [ ] which show optional items.)

1.5.3 Example Conventions

In interactive examples where both user input and system responses are shown, the machine output is in constant-width regular type; user-entered input is in constant-width boldface type. Output from the machine which varies (*e.g.*, the date) is in italic type. For example,

```
(dbug) <CR>
   Breakpoint 2 reached at filename _main: .3
   .3   printf("hello\r\n");
```

Chapter 2

ELEMENTS OF THE GNX ASSEMBLY LANGUAGE

2.1 INTRODUCTION

This chapter describes the elements of the GNX Assembly Language. The following topics are discussed:

- Character set
- Statements
- Constants
- Symbols, symbol types, and values
- Location counter
- Expressions

2.2 CHARACTER SET

The GNX Assembly Language character set consists of the following subset of the standard ASCII character set:

- Upper- and lower-case letters `A` through `Z` of the English alphabet.
- Digits 0 through 9.
- Blanks (ASCII 32), Tabs (9), Vertical Tabs (11), and Form Feeds (12).
- The following printable characters:

Character	Name	Character	Name
´	Single Quote/Apostrophe	+	Plus Sign
(	Left Parenthesis	/	Slash
)	Right Parenthesis	:	Colon
.	Period	;	Semi-Colon
_	Underscore	@	At Sign
,	Comma	[	Left Square Bracket
-	Minus Sign/Hyphen	]	Right Square Bracket
*	Asterisk	"	Double-Quote
\	Back Slash	%	Percent
~	Tilde	#	Pound Sign
^	Caret	|	Vertical Bar
&	Ampersand	<	Left Angle Bracket
$	Dollar Sign	>	Right Angle Bracket
?	Question Mark		

Carriage Return and Line Feed serve as line terminators; therefore, they cannot be entered directly into source code statements. They can be entered as their ASCII value.

Any other ASCII character may appear only within quoted strings.

The GNX Assembler is case sensitive, *i.e.*, the assembler distinguishes between upper- and lower-case letters. Reserved symbols must be typed in lower-case. User symbols are interpreted exactly as they are typed.

2.3 GNX ASSEMBLER STATEMENTS

The GNX Assembly Language consists of lines of text that contain one or more statements separated by semicolons and an optional comment. A statement is an optional label followed, optionally, by a mnemonic plus its operands. Statements are composed of user-defined symbols (names and labels representing variable quantities or memory locations), reserved symbols, constant values, and delimiters.

GNX Assembly Language statements are of two kinds: GNX assembly language instructions and GNX assembler directives. The GNX assembly language instructions are translated directly into machine instructions so that their meanings are carried out at execution time. The GNX Assembler directives, on the other hand, are commands to the assembler itself to carry out some action during program translation, *e.g.*, allocating a block of memory.

Lines of GNX Assembly Language code have the following form:

Syntax: ([*label* : [:]] [*mnemonic* [*operands*]] [;]),,, [# *comment*]

where:	*label*	is an optional label. The label must be a valid symbol name and must be followed by one or two colons. See the syntax descriptions of GNX assembly language directives in Chapter 6 for those directives that do not allow labels.
	mnemonic	is an optional instruction mnemonic or assembler directive. It must end with a space, tab, end-of-line, or semicolon.
	operands	are the operands of the instruction or of the assembler directive. The number of operands depends on the instruction or directive type. Each operand must be separated from the next operand by a comma. Spaces between operands are ignored. If the statement contains no instruction or directive, the operands must also be omitted.
	comment	is the optional comment. A comment must be preceded by a pound sign (#). If the –c flag (or /CPP in VMS) is given, the comments should not begin in column 1.

Description: A line of GNX Assembly Language code must conform to the following rules:

1. Multiple statements (*i.e.*, label, mnemonic, and operands) must be separated by a semicolon (;).
2. The code line may begin in any column.
3. A line of code may be up to 64K characters (including the end-of-line (EOL) character) in length. However, in the listing, lines longer than 132 characters (including the new-line (NL) character) will be truncated.
4. A code line may consist of zero or more statements, *i.e.*, label, mnemonic, and operands, separated by semicolons, and optionally followed by a comment.

Example:

```
        ret   0      # a return instruction
        jump  START  # a jump instruction and its one operand
        movw  r2, r3 # a move word instruction and two operands
END:                 # a label only
START:  movb  r0, r1 # a label, instruction, and operands
                     # a comment only
```

2.4 STRING AND NUMBER SYNTAX

There are four basic types of constants in GNX Assembly Language statements: integer values, floating-point values, character constants, and strings. The syntax for each type of constant is defined in Sections 2.4.1 through 2.4.4.

2.4.1 Integer Syntax

Integer syntax has the following form:

Syntax: [*sign*] [*base*] *digits*

where:

sign	specifies the sign. By default, the sign is positive. A negative sign may be specified with the minus sign (-).
base	specifies the base. It may be one of the following: Binary — B′ or b′ Octal — O′, o′, Q′, q′ or 0 (leading digit zero) Decimal — D′ or d′ Hexadecimal — H′, h′, X′, x′, 0x (digit zero), or 0X (digit zero) Default is decimal.
digits	specifies the integer. Digits must be compatible with the specified base. Binary — 0 to 1 Octal — 0 to 7 Decimal — 0 to 9 Hexadecimal — 0 to 9 and A to F or a to f

Description: Integer constants may have the following range of values, depending on the context in which the constant is specified: –128 to 255 for byte constants, –32768 to 65535 for word constants, and –2147483648 to 2147483647 (-2^{31} to $2^{31}-1$) for double-word constants.

Decimal constants are sign-extended to double-words. Hexadecimal, octal and binary constants are zero-extended to double-words.

If the first operand of the addp*i* or subp*i* instruction is a constant, the processor expects BCD encoding. The assembler generates only two's complement encoding. However, a valid BCD number preceded by H´ or 0x will be correctly encoded, because both hexadecimal and BCD use the same encodings within the BCD range.

For example, the instruction

```
addpd   $0x123, bcd_int
```

adds the immediate decimal value 123 to the contents of the location bcd_int.

Examples:

Binary	Octal	Decimal	Hexadecimal
`B´11110001`	`O´077`	`D´1492`	`H´12ff`
`-B´11`	`-Q´5077`	`-999`	`-X´302F`
`b´11`	`0123`	`1457`	`0xAB03`

2.4.2 Floating-Point Number Syntax

Floating-point values may be specified in one of two forms: as a decimal number in scientific notation, or as a hexadecimal value. The GNX Assembler expects floating-point numbers specified as hexadecimal values to be correctly encoded in the *Series 32000* internal floating-point format. Therefore, hexadecimal notation is most useful to the writers of compilers or optimizers.

Decimal Floating-Point Syntax

Decimal floating-point syntax has the following form:

Syntax: [*decimal prefix*]*decimal value*

where: *decimal prefix* specifies whether the constant is short or long floating-point format. It may be one of the following:

{0f | 0F} — short format floating-point value (float).

{01 | 0L} — long format floating-point value (long).

decimal value specifies a floating-point value in scientific notation.

Description: A decimal floating-point constant has two parts, an optional prefix that specifies short format (32 bits) or long format (64 bits) and a decimal value expressed in scientific notation.

The *decimal value* format is:

Syntax: [*sign*] *digits* [.*digits*] [{E | e} [*sign*]*digits*]

Mantissa Exponent

where:

sign	specifies the sign. A negative sign may be specified (–); by default, the sign is positive.
digits	specify the value. Only decimal digits are permitted (0 to 9). At least one digit must precede the decimal point.
.	is the decimal point.
E | *e*	is the exponent flag. It is required when specifying an exponent.

Description: The decimal value must be in the appropriate range for the prefix size specified or in the format that is required by the instruction. See note below.

Examples:

Valid	Invalid	Comments
`3.14152`	`.0125`	`# digit before decimal point required`
`971.`	`-0.00FF`	`# decimal digits only`
`0f0.1E-14`	`0.125E999`	`# exponent exceeds limit`

NOTE: The GNX Assembler recognizes two types of floating-point constants: single-precision (float) and double-precision (long). Single-precision numbers occupy four bytes. The most positive single-precision value is 3.40282346 x 10^{38}; the least positive value is 1.17549436 x 10^{-38}. The most negative value is the negative of the most positive value. Double-precision numbers occupy eight bytes. The most positive double-precision number is 1.7976931348623157 x 10^{308}; the least positive value is 2.2250738585072014 x 10^{-308}. The negative range is the negative of the positive value.

Hexadecimal Floating-Point Syntax

Hexadecimal floating-point syntax is of the following form:

Syntax: *hexadecimal prefix* hexadecimal digits

where: *hexadecimal prefix*

is one of the following:

{f′ | F′ | 0y | 0Y} — short format.
{l′ | L′ | 0z | 0Z} — long format.

`f′ | F′ | 0y | 0Y`

specifies an encoded short (32-bit) floating-point value. Must be followed by eight hexadecimal digits, if not, the assembler might generate unpredictable results.

`l′ | L′ | 0z | 0Z`

specifies an encoded long (64-bit) floating-point value. Must be followed by sixteen hexadecimal digits, if not, the assembler might generate unpredictable results.

`hex digits` specify the value. Only hexadecimal digits are permitted (0 to F or f). The encoded value is an exact bit representation of the resultant 32- or 64-bit value.

Examples:

Valid	Invalid	Comments
`f'E01267AC`	`-F'A7261CD5`	`#no sign permitted`
`L'12A945BD4266ECF0`	`L'E596C.4BF5DB46A26`	`#no decimal point permitted`

NOTE: The memory formats for both float and long constants are described in Chapter 3 of the *Series 32000 Programmer's Reference Manual.* The GNX Assembler stores both as long type.

2.4.3 Character Constant Syntax

Character constants have the following form:

Syntax: ´ {*ASCII char* | *escape sequence*}´

where:

ASCII char	is any single ASCII encoded character.
escape sequence	is one of the special escape sequences, described in this section.

Description: A character constant is a single ASCII character enclosed by single quotes, as in `´A´`. If the desired character is a special character, for example, the single quote itself, or if the character is not a printable character, then an escape sequence may be used to represent the character. The following rules apply to escape sequences:

- Except as noted in Table 2-1, any character preceded by the escape character backslash (\) represents that character.
- A backslash followed by one to three octal digits represents the character whose ASCII encoding is the octal value.
- Certain special characters are represented by the escape sequences specified in the escape sequence table below.

If the character constant is itself a single quote, the quote must be escaped, that is, preceded by the escape character backslash (\). Thus, the character constant single quote is (\´). Similarly, if the character constant is a backslash it must be escaped. The character constant backslash is (\\).

Other non-printable or special characters may be generated by the escape sequences in Table 2-1.

Character constants may be used in expressions. The value of the constant is its ASCII encoding. If the character constant is used as an immediate operand or in an expression, it is zero-extended to the appropriate number of bytes.

Table 2-1. Escape Sequences

ESCAPE	VALUE
\n	newline
\t	horizontal tab
\b	backspace
\r	carriage return
\f	form feed
\\	backslash
\´	single quote
\0	ASCII character 0, or null, the C string terminator
ddd	an arbitrary byte-sized bit pattern, where *ddd* is one to three octal digits, *i.e.,* the character constant “ ” represents the character with value zero.

2.4.4 String Syntax

String syntax has the following form:

Syntax: "({*ASCII char* | *escape sequence*})..."

where: *ASCII char* is an ASCII encoded character.

escape sequence is a character sequence used to represent special or non-printable ASCII encoded characters. Refer to Table 2-1.

Description: A string is a sequence of ASCII encoded characters enclosed by double-quotes. The same rules and escape sequence definitions specified in the description of character constants may be used in string constants. Special consideration must be given if a double-quote mark is part of the string. Strings enclosed in double-quote marks which also contain double-quotes are allowed. However, each quote which is a part of the string must be escaped, that is, it must be preceded by the escape character backslash (\ ). It is not necessary to escape the single-quote character in a string constant.

Strings may not be used in expressions.

Examples:	Strings Coded In Source Statements	Generated String
	"This is a string"	This is a string
	"Five O´Clock"	Five O´Clock
	"\"A\" for Ampere"	"A" for Ampere

2.5 SYMBOLS

A symbol is a name that refers to a memory location. Each symbol has a type and a value. The type of a symbol is either the segment in which the symbol is defined, *external* if the symbol is not defined in the assembly file, or *absolute* if the symbol is a numeric address. The value of a symbol is the address of the memory location. A symbol may have the attribute global. A symbol with the global attribute may be referenced from any software module in the program. By default, all symbols referenced but not defined are considered global.

Some symbol names are reserved, *i.e.*, the instruction mnemonics, directive mnemonics, names for the registers, address mode indicators, flags, scaled index qualifiers, the delimiters, and operators. The user may not redefine the reserved symbols. Appendix B contains a list of the reserved symbols in the GNX Assembly Language. The rest of this section and all of Section 2.6 deal with user-defined symbols.

2.5.1 Symbol Names

The name of a user-defined symbol is composed of one or more letters, digits and the characters underscore (_) and period (.). Except for temporary labels, the first character of the name may not be a digit. Symbol names with the initial character period (.) are assumed to be internal names generated by the GNX language tools; for example, compiler labels, Common Object File Format (COFF) section names, and reserved names should not be used. The name's length is limited to 64 characters.

The assembler is case sensitive, that is, it differentiates between upper- and lower-case letters in a user-defined symbol name. Thus, for example, the names ALPHA and Alpha are not identical and can be defined as separate symbols.

Examples:	Valid	Invalid	Comment
	SYMBOL	$YMBOL	# ''$'' dollar-sign character illegal
	_ALPHA	2ALPHA	# first character cannot be number
	REG2	r1	# r1 is reserved symbol

2.5.2 Symbol Types

The type of a symbol specifies the segment of the object file in which it occurs. All labels defined within a segment have the type of that segment. For example, all symbols defined in the .text segment (*i.e.,* following the `.text` directive) are of type text. The address of symbols associated with object file segments must be updated at link time, when the linker associates the object file segment with memory locations.

Undefined symbols are of type external. The value of an undefined symbol is resolved by the linker. Numeric addresses are of type absolute. The value of absolute symbols is unaffected by linkage.

A symbol's type determines the default addressing mode the assembler uses when the symbol is referenced. The following table lists symbol types, the associated object file segment and the default addressing mode for references to the symbol.

Type	Segment	Default Addressing Mode
Text	Text or code segment	PC Relative
Data	Initialized data segment	Absolute
Bss	Uninitialized data segment	Absolute
Static	Static base segment	SB Relative
Link	Link table segment	Absolute
External	-	Absolute
Absolute	-	Absolute
< user-defined>	<defined by attributes>	Absolute

The type of a symbol delimits the places where the symbol may be used as an operand and the way its value may be manipulated in expressions. Expressions also have one of the above types. The type of an expression is determined by the types of the symbols it contains.

Following are descriptions of each of the symbol types:

1. Symbols of type text.

   All symbols defined in the .text segment, *i.e.*, labels following a `.text` directive, are of type text. All symbols or expressions of type text represent addresses within the text segment of the program's object code. The text segment contains program code and read-only data. The GNX Assembler uses the Program Counter (PC) Relative addressing mode for symbols and expressions of type text.

2. Symbols of type data.

   All symbols defined in the .data segment, *i.e.*, labels following a `.data` directive, are of type data. All symbols or expressions of type data represent addresses within the initialized data segment of the program's object code. The GNX Assembler uses the Absolute addressing mode for symbols and expressions of type data.

3. Symbols of type bss.

   All symbols defined in the uninitialized data (.bss) segment are of type bss. Symbols defined by the `.bss` directive are of type bss, as are labels defined after a `.udata` directive. All symbols or expressions of type bss represent addresses within the uninitialized data segment of the program's object code. The GNX Assembler uses the Absolute addressing mode for symbols and expressions of type bss.

4. Symbols of type static.

   All symbols defined in the .static segment, *i.e.*, labels following a `.static` directive, are of type static. All symbols or expressions of type static represent addresses within the .static segment of the program's object code. The .static segment is used to store static base relative data. The GNX Assembler uses the Static Base Register (SB) Relative addressing mode for symbols and expressions of type static.

5. Symbols of type link.

   All symbols defined in the .link segment, *i.e.*, labels following a `.link` directive are of type link. All symbols or expressions of type link represent addresses within the .link segment of the program's object code. The .link segment is used to store the link table for a *Series 32000* module. The GNX Assembler uses the Absolute addressing mode for symbols and expressions of type link.

6. Symbols of type external.

   All undefined symbols are of type external. Symbols defined using the `.comm` directive are also of type external. The GNX Assembler uses the Absolute addressing mode for symbols and expressions of type external.

7. Symbols of type absolute.

   All symbols assigned numeric values are of type absolute. Absolute symbols specify an absolute numeric address. They are not relative to any segment of the object file. Symbols of type absolute may only be defined using the `.set` directive. The GNX Assembler uses the Absolute addressing mode for symbols and expressions of type absolute.

8. Symbols of user-defined type.

   All symbols defined in a section, following the .section definition, are the type of the section. All symbols are allowed via absolute addressing mode.

2.5.3 Global Symbols

Global symbols are used by multiple software modules. The symbol must be defined exactly once. The defining module exports the symbol, that is, makes the symbol available for import by one or more additional software modules. Global symbols must be declared for export by the defining module with the `.globl` directive. Undefined symbols intended to be imported from other software modules should also be declared with the `.globl` directive, although this is not required.

Except for temporary labels, every user-defined symbol must be defined exactly once. A symbol definition assigns a value and type to a symbol name. There are several formats for defining symbols. The formats form four groups:

- Labels.
- Symbols defined by the `.set` directive.
- Uninitialized symbols defined by the `.bss` directive.
- Common symbols defined by the `.comm` directive.

External, or undefined, user symbols may be declared for import with the `.globl` directive. Such a declaration does not define the symbol. Any symbol that is referenced in an assembler statement but not defined within the assembly is assigned type external.

Labels

The formats permitted for label definitions are:

Syntax: *symbol name* :

or

symbol name ::

or

symbol name : *assembly statement*

or

symbol name :: *assembly statement*

where: *assembly statement*

may be any assembly statement except those directives that do not accept labels. See Chapter 6 for detailed descriptions of the syntax of all the GNX assembly language directives.

Description: In each case, the current value and the type of the location counter is assigned to the symbol, see Section 2.7. The second construction (using "::") also sets the global attribute on the symbol, see Section 6.6.

Temporary Labels

Syntax: *temporary label*:

where: *temporary label*

consists of a digit from 1 to 9.

Description: A temporary label consists of a digit from 1 to 9, followed by a colon. Reference to the label is via the symbols *nf* and *nb*, where *n* specifies temporary label *n*, where *f* means forward, and *b* means backwards. All referenced temporary labels must be defined somewhere within the program. Temporary labels may not be exported. There is no limit on the number of times that a temporary label may be redefined. The following symbols are reserved:

1f 2f 3f 4f 5f 6f 7f 8f 9f 1b 2b 3b 4b 5b 6b 7b 8b 9b

Temporary labels are most useful in conjunction with macros.

Example:

```
1                          9:
2  T00000000  a2a2a2a2         .space 10
              a2a2a2a2
              a2a2
3  T0000000a  ea06             br 7f     # branch to line 6
4  T0000000c  ea74             br 9b     # branch to line 1
5  T0000000e  ea02             br 9f     # branch to line 7
6                          7:
7                          9:
8                          7:
9  T00000010  ea00             br 7b     # branch to line 8
```

In this program, the branch on line 3 refers to label 7 on line 6, the branch on line 4 refers to label 9 on line 1, the branch on line 5 refers to label 9 on line 7, and the branch on line 9 refers to label 7 on line 8.

Defining Symbols with the .set Directive

The format for symbol definition using the `.set` directive is:

Syntax:	**`.set`** *symbol name, expression*
Description:	The statement assigns the value and type of *expression* to the symbol. The expression may not be of type external (undefined), nor a forward reference.

Defining Uninitialized Symbols with the .bss Directive

The format for the definition of uninitialized symbols using the `.bss` directive is:

Syntax:	**`.bss`** *symbol name, expression1, expression2*
Description:	This form is used only for uninitialized data (bss) symbols. The symbol is assigned type bss and the value of the current bss location counter after it is aligned to a multiple of expression2. For a complete description of the `.bss` directive see Section 6.7.4.

Defining Common Symbols

The format for the definition of uninitialized, common symbols using the `.comm` directive is:

Syntax:	**`.comm`** *symbol name, expression*
Description:	The type of common symbols is external. If no software module defines a global symbol by this name, then the linker will allocate an uninitialized storage area whose size is the largest *expression* specified by any `.comm` directive for this *symbol.* See Section 6.6.2 for a description of the `.comm` directive.

2.6 LOCATION COUNTER

The GNX Assembler manages a location counter that keeps track of the current relocatable memory address. The current location counter is set to the type of the segment that is being assembled and the value of the next available address within the segment. The current location counter is initialized to the TEXT segment, address 0 at the start of assembly.

The assembler re-initializes the current location counter to a new value (*i.e.*, a new type and offset) each time a segment control directive is encountered. The segment control directives determine the segment into which the following code should be assembled. On encountering a segment control directive, the assembler saves the next available address in the previous segment before entering the new segment, so that it is able to restore the previous address if the previous segment is re-opened. The assembler maintains a saved location counter for each object file segment (text, data, bss, static, user-defined sections, dsects and link) as well as each user-defined segment.

When a statement is processed, the assembler increments or decrements the location counter by the number of bytes of object code generated or by the amount of data storage allocated.

The location counter symbol, (.) period, is a special token which may be used in expressions or instruction operands to specify the location counter's current value. The symbol may appear alone or as a term in an arithmetic expression (addition or subtraction only).

Examples:

```
1.    .set  A, .
2.    bne   .-8
```

In example 1, (.) specifies the current address. The symbol `A` is assigned the current location counter address.

In example 2, the expression .-8 specifies the current address minus 8.

2.7 EXPRESSIONS

An expression is a combination of terms and operators which evaluate to a single value and type. Valid expressions include addresses and integer expressions. Floating-point expressions are not valid.

Terms in expressions may be constants or symbols, including the location counter symbol (.), see Sections 2.4, 2.5, and 2.6. The type of the term determines the way in which the term may be combined with other terms and operators. Section 2.7.2 defines the effect the type of a term has on the result of an expression.

Operators in expressions are the special symbols which define arithmetic and logical operations. An operator has the following characteristics:

- An operator has a level of precedence which affects the order in which the GNX Assembler evaluates an expression containing the operator.
- An operator defines the type of the term(s) that may be used with the operator and the location of the term(s) relative to the operator.

Table 2-2 lists all GNX Assembly Language operators in order of precedence.

Table 2-3 defines the type and order of the terms that may be used with the operators.

Table 2-2. Operator Precedence

PRECEDENCE	OPERATOR	NAME	OPERATION
Unary Operator			
1	-	Unary minus	Two's complement.
1	~	Unary complement	One's complement.
Binary Operator			
2	*	Multiply	Multiply 1st term by 2nd.
2	/	Divide	Divide 1st term by 2nd.*
2	%	Modulus	Remainder from 1st term divided by 2nd.**
2	<<	Shift left	Shift 1st term by 2nd; emptied bits are zero-filled.
2	>>	Shift right	Shift 1st term by 2nd; emptied bits are zero-filled.
2	~	Logical OR / complement	Bit-wise OR of 1st term and one's complement of 2nd term.
3	&	Logical AND	Bit-wise AND of 1st and 2nd terms.
3	|	Logical OR	Bit-wise OR of 1st and 2nd terms.
3	^	Logical XOR	Bit-wise XOR of 1st and 2nd terms.
4	+	Add	Add 1st and 2nd terms.
4	-	Subtract	Subtract 2nd term from 1st term.

* Rounds toward 0, *e.g.*, -7/3 = -2 and 7/3 = 2

** *e.g.*, -7%3 = -1 and 7%3 = 1.

Table 2-3. Types and Operators

UNARY OPERATORS			
Operator		Term1	Operation
-		abs	Type abs.
~		abs	Type abs.
BINARY OPERATORS			
Term1 Type	Operator	Term2 Type	Result Type
abs	*	abs	Type abs.
abs	/	abs	Type abs.
abs	%	abs	Type abs.
abs	<<	abs	Type abs.
abs	>>	abs	Type abs.
abs	~	abs	Type abs.
abs	&	abs	Type abs.
abs	|	abs	Type abs.
abs	^	abs	Type abs.
abs	+	abs	Type abs.
abs	-	abs	Type abs.
rel	+	abs	Type rel.*
rel	-	abs	Type rel.*
rel	-	rel	Type abs.**
ext	+	abs	Type ext.
ext	-	abs	Type ext.

NOTE:

abs	Any term of type absolute.
rel	Any term of relative type, *i.e.,* text, data, etc.
ext	Any term of type external, undefined.

* The type of the result matches the type of the relative term in the expression.
** Term1 and Term2 must be the same type, the result is type absolute.

2.7.1 Rules for Expressions

The rules for forming and evaluating expressions are as follows:

1. All unary operators must precede a single term and cannot be used to separate two terms.
2. All binary operators must separate two terms. For example, the expression `8*4` is legal, but `8**4` is not.
3. Compound expressions are valid. An expression may be constructed from other expressions using unary and binary operators. For example, the two individual expressions `A+1` and `B+2` may be combined with a multiply operator and parentheses to form the single expression `(A+1)*(B+2)`. Note that the parentheses override the default precedence rules.
4. Evaluation of an expression is governed by three factors:
    - *Parentheses* – expressions enclosed in parentheses are always evaluated first. For example, the expression `8/4/2` evaluates to 1, but the expression `8/(4/2)` evaluates to 4.
    - *Precedence Groups* – an operation of a higher precedence group is evaluated before an operation of a lower precedence whenever parentheses do not otherwise determine the evaluation order. For example, the expression `8+4/2` is evaluated as 10, but the expression `8/4+2` is evaluated as 4.
    - *Left to Right Evaluation* – expressions are evaluated from left to right whenever parentheses and precedence groups do not determine evaluation order. For example, the expression `8*4/2` is evaluated as 16, but the expression `8/4*2` is evaluated as 4.

2.7.2 Types in Expressions

The type of the result of an expression depends on the type of the terms and the operations performed. The rules for types in expressions are as follows:

1. Expressions with terms having absolute type.

   Terms with absolute type may be added, subtracted, multiplied, etc. All operators are allowed. The result is always an absolute type.

Examples:

```
 1. 21 * 5  # result is 105
 2. 21 / 5  # result is 4
 3. 21 % 5  # result is 1
 4. 21 & 5  # result is 5
 5. 21 << 5 # result is 672
 6. 21 >> 5 # result is 0
 7. 21 + 5  # result is 26
 8. 21 - 5  # result is 16
 9. 21 | 5  # result is 21
10. 21 ^ 5  # result is 16
```

2. Expressions combining terms having relative and absolute types.

   The only valid operations between terms with relative types and terms with absolute type are addition and subtraction. The operations take place between the values of the first and the second terms and the result is assigned the type of the relative term.

   Addition is commutative. An absolute term may be added to a relative term or a relative term may be added to an absolute term, the result is the same in either case.

   Subtraction is not commutative. An absolute term may be subtracted from a relative term. A relative term may not be subtracted from an absolute term.

Example:

```
 1          .set    ZERO, 0
 2          .set    TEN, 10
 3          .set    COUNT, 30
 4
 5          .udata
 6  Size:   .blkd
 7  Start:  .space  (COUNT * 4)
 8  End:    .blkd
 9
10          .text
11          movb    $ZERO, Start + ZERO
12          movb    $TEN, TEN + Start
13          movd    (End - TEN), r0
```

In the preceding example several symbols and expressions are used. The symbols ZERO, TEN, and COUNT are of type absolute. The symbols Size, Start, and End are of type bss, refer to Section 2.5.2.

The expression "(COUNT * 4)" in line 7 combines two absolute terms, the result is absolute.

The expression "Start + ZERO" in line 11 adds a relative type to an absolute type. The result is type bss.

The expression "TEN + Start" in line 12 adds an absolute type to a relative type. The result is type bss.

The expression "End - TEN" in line 13 subtracts an absolute type from a relative type. The result is type bss.

3. Expressions combining terms having relative types.

   Terms with relative or absolute type may be subtracted from terms with the same type. No other operator is allowed. The result is always an absolute type.

   Example:

```
1          .set    COUNT, 30
2
3          .udata
4   Size:  .blkd
5   Start: .space  (COUNT * 4)
6   End:   .blkd
7
8          .text
9          movd    $(End - Start)/4, Size
```

   The expression "End - Start" in line 9 subtracts a relative term from another relative term. Since both symbols are of the same type (bss), this is a legal expression. The result is of type absolute, *i.e.,* the absolute number of bytes between the two labels. The result of the subtraction is then divided by 4, both terms are type absolute and the result is type absolute.

   Note that "(End - Start)/4" is not a legal expression without parentheses. Division is of higher precedence than subtraction, but a relative term may not be divided.

4. Expressions with terms having external and absolute type.

   Terms with absolute type may be added to or subtracted from terms with external type. No other operations are allowed. The result always has external type. A term of type absolute may be subtracted from a term of type external, but a term of type external may not be subtracted from a term of type absolute. The first term of the subtraction must be the term of type external.

Example:

```
 1     .set   ZERO, 0
 2     .set   TEN, 10
 3     .set   COUNT, 30
 4
 5     .globl Start
 6     .globl End
 7
 8     .text
 9     movb   $ZERO, Start + ZERO
10     movb   $TEN, TEN + Start
11     movd   (End - TEN), r0
```

The expression “Start + ZERO” in line 9 adds an absolute type to an external (undefined) type. The result is type external.

The expression “TEN + Start” in line 10 adds an external type to an absolute type. The result is type external.

The expression “End - TEN” in line 11 subtracts an absolute type from an external type. The result is type external.

5. Expressions with character constants.

   Character constants may appear as terms in expressions. When a character constant is used this way, it is converted to an integer constant. Integer constants are stored in four bytes; the assembler fills the higher order bytes with zero.

   Examples:

```
1. .set UPCASE,     'A' - 'a# result is -32
2. .set LOWCASE,    'a' - 'A# result is 32
```

2.7.3 Size of Expressions

Expressions are stored in 4 bytes, with the higher order bytes filled with zero by the assembler.

Chapter 3

GNX ASSEMBLER PROGRAMS

3.1 INTRODUCTION

This chapter describes the structure of GNX Assembly Language programs and how the GNX Assembler assigns memory addresses to symbols, instructions, and data. In particular, it describes:

- Program Structure
- Program Segments
- *Series 32000* Module Segments
- User-Defined Dummy and Comment Segments
- Linkage and Relocation Modes

3.2 GNX ASSEMBLER PROGRAM STRUCTURE

The structure of a GNX Assembly Language program reflects the structure of the object file and the layout of the program image in memory. The structure allows instructions and data to be grouped into logical segments that occupy contiguous memory. Each segment is an atomic unit, that is, segments may be combined together into larger units but not broken into smaller units.

Every object file contains at least three program segments: text, data and bss. These segments correspond to the .text, .data, and .bss sections of a Common Object File Format (COFF). The text segment contains program instructions and constant data, the data segment contains writable, initialized data, and the bss segment contains uninitialized data. No object file space is allocated for the bss segment.

In addition to the default program segments, there are several special segments that support *Series 32000* modules. The module table segment contains the module table entries and corresponds to the .mod section of the COFF file. The link segment contains the module's Link Table and corresponds to the .link section of the COFF file. The static segment contains Static Base Relative data and corresponds to the .static section of the COFF file.

The programmer is also allowed to create user-defined segments through the assembler directives `.dsect` and `.section` or comment segment through the assembler directive `.ident`. The GNX Assembler maintains a location counter for each object file segment.

3.3 PROGRAM SEGMENTS

Every assembly program consists of one or more program segments. A program segment is a block of sequential statements which are placed in contiguous memory and treated as a unit with common properties, for example, access protection. Every program contains the following types of segments:

- Text or Program Code Segment
- Initialized Data Segment
- Uninitialized Data Segment (bss)

A segment begins and ends with one of the segment control directives (Section 6.7) and contains any number of statements. The following illustrates the form of a program segment:

```
.text          # specifies the start of a program code segment
statement-1    # assembler statements
statement-2
     .
     .
     .
statement-n
.data          # specifies start of a data segment
               # a segment terminates with another segment
               # control directive or EOF
```

3.3.1 Text Segment

The text segment contains *Series 32000* instructions and constant data. After every statement, the text segment location counter is incremented by the number of bytes generated for that statement. The location counter of the text segment may not be decremented.

The text segment is written to the .text section of the object file. Each text address maps to a location in the .text section of the object file. When the text segment is loaded into memory, it is protected for read-only access.

NOTE: This statement is only true for GNX native environments and if the MMU is on a development board. It may not be true in other applications.

All symbols defined in the text segment are of type text. References to locations in the text segment are addressed with the Program Counter Relative addressing mode.

Related directive: `.text` (Section 6.7.2).

3.3.2 Initialized Data Segment

The initialized data segment contains writable, initialized data. After every statement, the data segment location counter is incremented by the number of bytes generated for that statement. The location counter of the data segment may not be decremented.

The data segment is written to the .data section of the object file. Each data address maps to a location in the .data section of the object file. When the data segment is loaded into memory, it is protected for read-write access.

All symbols defined in the .data section are of type data. References to locations in the data segment are addressed with the Absolute addressing mode.

Related directive: `.data` (Section 6.7.3).

3.3.3 Uninitialized Data (bss) Segment

The uninitialized data or bss segment consists of storage allocated for uninitialized data. After every statement following the .udata section control directive, the bss segment location counter is incremented by the number of bytes allocated by that statement. The bss location counter is also updated by the `.bss` directive. No code or data may be generated in the bss segment. The location counter of the bss segment may not be decremented.

Each bss address maps to a location in the .bss section of the object file, although the .bss section of the COFF file contains no actual data. Storage space is allocated and zeroed at load time. When the .bss section is loaded into memory, it is protected for read-write access.

All symbols defined in the .bss section are of type bss. References to locations in the bss segment are addressed with the Absolute addressing mode.

Related directives: `.udata` (Section 6.7.5), `.bss` (Section 6.7.4).

3.4 SERIES 32000 MODULE SEGMENTS

The GNX Assembler supports three additional segments for building *Series 32000* modules. A *Series 32000* module uses the *Series 32000* hardware support for linkage.

The following segment types support *Series 32000* modules:

- Module Table Segment
- Link Table Segment
- Static Base Relative Segment

The *Series 32000* hardware support for linkage requires a Module Table for the program and a Link Table for each *Series 32000* module.

The Module Table records the Program Base, the Static Data Base, and the Link Table Base for each module. The Program Base is the base address for the text (or program code) segment of the module. The Static Data Base is the base address for the static data segment of the module. The Static Data Base may be defined once for each module with the `.module` or `.modentry` directive. If the Static Data Base address is not explicitly defined, it defaults to zero. The Link Table Base is the base address for the link segment. The Link Table Base may be defined once for each module with the `.module` or `.modentry` directive. If the Link Table Base address is not explicitly defined, it defaults to zero. The Module Table is built by the linker.

The Link Table contains an address for each external data reference and an external procedure descriptor for each external function entry point. It must be built by the programmer in the link segment using the `.xdd` directive to define each external data address and the `.xpd` directive to define an external procedure descriptor for each function entry point that uses the cxp/rxp calling discipline. The addresses in the Link Table are generated at link time.

3.4.1 Module Table Segment

The module table segment is one of three special segments whose function is to support *Series 32000* modules. The module table segment, if one is present, contains the module table for the program. Each module table entry consists of four 32-bit entries corresponding to each component of a module:

- The Static Base (sb) entry contains the base address for the module's static local data.
- The Link Base (lb) entry contains the base address for the module's link table.
- The Program Base (pb) entry contains the base address for the module's program code.
- A fourth entry is currently unused but reserved.

Each base address is a standard *Series 32000* address.

Module table entries may be generated with the `.module` and the `.modentry` directives.

The module segment location counter is incremented by the number of bytes generated for the directive. The location counter of the module segment may not be decremented. The module segment is written to the .mod section of the object file.

Related directives: `.module` (Section 6.8.1) and `.modentry` (Section 6.8.2).

3.4.2 Link Table Segment

The link table segment is one of three special segments whose function is to support *Series 32000* modules. The link table segment, if one is present, contains the link table for the *Series 32000* module. The link table consists of one 4-byte entry for each variable or function that is accessed with the External addressing mode, see Section 4.2.13. If the link table entry is for a data item, the entry contains the absolute memory address of the variable. If the link table entry is for the entry point of a function, the link table entry contains an external procedure descriptor. Each link table entry is filled with the appropriate address or procedure descriptor by the linker at link time.

An external procedure descriptor must be generated for any function called with the cxp or cxpd instruction. An external procedure descriptor consists of a 16-bit module table offset and a 16-bit program code offset. The module table offset is the distance in bytes from the base of the program's module table to the module table entry for this module. The program code offset is the distance from the program code base of the module to the entry point of the function. External procedure descriptors may be generated with the `.xpd` directive.

Link table entries for data items may be generated with the `.xdd` directive.

After every statement, the link segment location counter is incremented by the number of bytes generated for that statement. The location counter of the link segment may not be decremented. The link segment is written to the .link section of the object file. Each link address maps to a location in the .link section of the object file. When the link segment is loaded into memory, it is protected for read-only access.

All symbols defined in the link segment are of type link. References to locations in the link segment are addressed with the Absolute addressing mode.

Related directives: `.xpd` (Section 6.3.8), `.xdd` (Section 6.3.9), `.link` (Section 6.7.7), `.module` (Section 6.8.1), and `.modentry` (Section 6.8.2).

3.4.3 Static Base Relative Segment

The static segment should be used instead of the data segment when building a program of *Series 32000* modules. The static segment contains Static Base Relative data. If the static segment is present, the linker assigns the Static Base Register the value of the base address of the static segment. After every statement, the static segment location counter is incremented by the number of bytes generated for that statement. The location counter of the static segment may not be decremented. The static segment is written to the .static section of the object file. Each static address maps to a location in the .static section of the object file. When the static segment is loaded into memory, it is protected for read-write access.

All symbols defined in the static segment are of type static. References to locations in the static segment are addressed with the Static Base Relative addressing mode.

Related directives: `.static` (Section 6.7.6), `.module` (Section 6.8.1) and `.modentry` (Section 6.8.2).

3.5 USER-DEFINED, DUMMY AND COMMENT SEGMENTS

This section describes user-defined, dummy and comment segments.

3.5.1 User-Defined Segments

User-defined segments are generated with the `.section` directive. These segments occupy real space in the object file and, depending on the attributes selected, may appear in the linked file. Symbols declared in these segments are addressed via the absolute addressing mode.

Related directive: `.section` (Section 6.7.8).

3.5.2 Dummy Segments

The "dummy" segments are generated with the `.dsect` directive. These segments do not allocate storage, nor do they contain generated code or data. If the dummy segment is of a relative type, it will overlay some portion of that type of segment. For example, a user-defined dummy segment might be used to overlay one or more structured data types on a pool of storage. Dummy segments of type absolute may be used to generate symbolic positive or negative offsets from the frame pointer register for function arguments or local variables.

Every statement following a `.dsect` directive increments or decrements the location counter for the dummy segment by the number of bytes specified by that statement.

Related directive: `.dsect` (Section 6.7.1).

3.5.3 Comment Segments

Comment segments are generated with the `.ident` directive and corresponds to the .comment section of the COFF file.

3.6 LINKAGE

Linkage is the combination of the output of several assemblies or compilations into a single program. A linker must also resolve all external references and all references to relocatable addresses within each program segment.

The GNX Linker combines all input segments of the same type into a single output segment and assigns the resultant output segments to specific memory addresses. The linker also updates all references to addresses within the segment if the base address of the segment has changed.

3.6.1 Relocatable Addresses

The GNX Assembler assigns a relocatable memory address to each instruction and each byte of data storage defined in an assembly language program. A relocatable memory address is one which is relative to the start of the segment. If the linker moves the base address of the segment, the linker must update every address within the segment by the same amount. At link time each relocatable address is resolved to an absolute address, *i.e.*, to the actual address in system memory where the instruction or data is stored.

A relocatable memory address consists of a type that specifies the segment in which the symbol is defined and an address that specifies the location of the instruction or data in memory, relative to the beginning of its segment. If the base address of the segment is changed at link time, all relocatable addresses within the segment must be modified accordingly.

3.6.2 Linking Program Segments

An assembly language program segment is the smallest unit the GNX Linker manipulates. Within a segment all code or data remains contiguous throughout the linkage process. By default, the linker combines all input segments of the same type and module according to the linker's combining rule, for example, all text segments, into a single output segment of the same type. The linker binds the output segment to a section of memory within the program's address space. The linker may function differently depending upon the programmer's instructions.

Each program segment the assembler outputs has associated relocation entries for every undefined symbol or relocatable address referenced within the segment. The linker uses these entries to generate absolute memory addresses for the references.

A program segment is relocatable if the segment may be combined with other segments of the same type and if there are relocation entries for all undefined or relocatable addresses the segment references.

3.6.3 Linking Series 32000 Modules

Series 32000 modules use special hardware support provided by the *Series 32000* chip family to resolve external references. A *Series 32000* module has three components, a program base relative code segment, a static base relative data segment, and a link table. The program base relative portion of the module corresponds to the text segment of the assembly program. The static base relative component is the static segment. All data references should use the Static Base Register Relative addressing mode. See Section 9.3 and Table 9-2 for the command line option that defaults data segment addresses to the Static Base Register Relative addressing mode. The link table corresponds to the link segment of the assembly program.

A *Series 32000* module built with all code in the text segment, all data in the static base relative segment, and all external references resolved through the link table is position independent. All memory references in the module are relative to base addresses stored in its Module Table entry. Only the module table and possibly the link table require updating if the module is moved to a different memory location.

To link *Series 32000* modules, a module table entry must be built for each module. The link table of each module must be filled with the address of each external data variable and an external procedure descriptor for each external procedure.

Chapter 4

INSTRUCTION OPERANDS

4.1 INTRODUCTION

This chapter defines the syntax of instruction operands. Instruction operands identify the participants in the operation specified by an opcode or a directive.

Instruction operands may be constants, memory addresses, symbols, and/or expressions. The type of operand required in an instruction is determined by the instruction itself. These are the following operand types:

Operand Type	**Section**
General Operands	Section 4.2
Expression Operands	Section 4.2.1
Register Operands	Section 4.2.2
Register Relative Operands	Section 4.2.3
Frame Memory Operands	Section 4.2.4
Frame Memory Relative Operands	Section 4.2.5
Stack Memory Operands	Section 4.2.6
Stack Memory Relative Operands	Section 4.2.7
Static Memory Operands	Section 4.2.8
Static Memory Relative Operands	Section 4.2.9
Program Memory Operands	Section 4.2.10
Immediate Operands	Section 4.2.11
Absolute Operands	Section 4.2.12
External Operands	Section 4.2.13
Top-of-Stack Operands	Section 4.2.14
Scaled-Index Byte Operands	Section 4.2.15
Scaled-Index Word Operands	Section 4.2.16
Scaled-Index Double-Word Operands	Section 4.2.17
Scaled-Index Quad-Word Operands	Section 4.2.18
Displacement Operands	Section 4.2.19
Immediate Subrange Operands	Section 4.3
Quick Operands	Section 4.3.1
Block Length Operands	Section 4.3.2
Bit-Field Length Operands	Section 4.3.3
Bit-Field Offset Operands	Section 4.3.4
Displacement Operands	Section 4.3.5

Operand Type	**Section**
Program Memory Operands	Section 4.4
General Register Operands	Section 4.5
Register List Operands	Section 4.6
Configuration List Operands	Section 4.7
Processor Register Operands	Section 4.8
Memory Management Register Operands	Sections 4.9 – 4.11
External Register Operands	Section 4.12

About 90 percent of the instructions use one or more general operands.

The following sections define the syntax of the instruction operands.

4.2 GENERAL OPERANDS

Syntax: *gen*

where: *gen* is one of the following general operand types:

Expression
Register
Register Relative
Frame Memory
Frame Memory Relative
Stack Memory
Stack Memory Relative
Static Memory
Static Memory Relative
Program Memory
Immediate
Absolute
External
Top-of-stack
Scaled-index Byte
Scaled-index Word
Scaled-index Double-word
Scaled-index Quad-word
Displacement (:b, :w, :d)

Description: Each of the general operand types corresponds to a GNX Assembler general addressing mode.

Many general operands use displacements (*disp*) to specify the offset from a base address to a particular memory location. General operand displacements must be within the range $-2^{24}+1$ to $2^{24}-1$ (−16777215 to 16777215) if the CPU is a 24-pin address CPU. Although the 32-pin address processors have a full 32-bit address space, displacements are limited to the range$-(2^{29}-2^{24})$ to $2^{29}-1$ (−536870912 to 536870911) because of the four byte displacement format (see the *Series 32000 Programmer's Reference Manuals* for more details). Displacements that are expressions must be enclosed in parentheses.

NOTE: The GNX Assembler uses the range -2^{29} to $2^{29}-1$. It is up to the user to limit this range if a 24-pin address processor is used.

24-pin address CPUs include the NS320*xx* and the NS32CG16.

32-pin address CPUs include the NS32332, NS32532, NS32FX16, NS32CG160, NS32GX32, and the NS32GX320.

Sections 4.2.1 through 4.2.19 define the syntax and function of each of the general operand types.

4.2.1 Expression Operands

Syntax:	*expression*
Description:	When an expression is used as an operand for a general class instruction, the addressing mode, or operand type, of the operand depends on the type of the expression. Expressions of type text generate the Program Counter Relative addressing mode. Expressions of type static generate the Static Base Relative addressing mode. All other expression types generate the Absolute addressing mode.

4.2.2 Register Operands

Syntax: *register*

where: *register* is one of the General-purpose or Floating-point registers. (See Sections 1.3.1 and 1.3.3.) The specified register contains the operand.

Description: A register operand specifies a General-purpose or Floating-point register. In some instructions, the specified General-purpose register points to the location of the operand, *i.e.*, the register contents are the address of the operand. In such cases, the contents of the register are not affected by the instruction operation.

Floating-point registers may be specified only in floating-point instructions.

Example:

```
1  T00000000  be4500  movf   f1, f2
2  T00000003  c101    addw   r0, r7
```

The movf instruction copies a single-precision floating-point number from Floating-point register f1 to Floating-point register f2. The addw instruction adds the low-order word of r0 to the low-order word of r7.

4.2.3 Register Relative Operands

Syntax: *expression(register)*

where:

expression is a displacement or expression which evaluates to an absolute value within the range −16777215 to 16777215 if the CPU is a 24-pin address CPU; or the range −536870912 to 536870911 if the CPU is a 32-pin address CPU; or to a relative value of type text, data, bss, static, or link.

(register) is one of the general registers, r0 to r7 (see Section 1.3.1). Parentheses are required.

NOTE: The GNX Assembler uses the range −536870912 to 536870911 for displacement. It is up to the user to limit this range if a 24-pin address CPU is used.

Description: A Register Relative operand specifies an operand at a memory address. The address is the sum of the displacement *expression* and the contents of the General-purpose register *rn*.

Example:

```
1                           .set    TEN, 10
2
3   T00000000   81aac000    addw    INTEG, 0(r2)
                000000
4   T00000007   435514c0    addd    (TEN*2)(r2), INTEG
                000000
```

In line 3, 0(r2) is a Register Relative operand. The instruction adds the word at the address specified by the symbol INTEG to the word at the memory address specified by 0(r2). The result is stored at 0(r2).

In line 4, (TEN*2)(r2) is a Register Relative operand. The expression "TEN * 2" evaluates to the absolute value 20. This value is added to the contents of register r2 to yield the operand's address. The instruction adds the double-word at this address to the double-word at the address specified by the symbol INTEG. The result is stored at INTEG.

NOTE: 24-pin address CPUs include the NS320*xx* and the NS32CG16.

32-pin address CPUs include the NS32332, NS32532, NS32FX16, NS32CG160, NS32GX32, and the NS32GX320.

4.2.4 Frame Memory Operands

Syntax: *disp* **`(fp)`**

where: *disp* is a displacement or expression which evaluates to an absolute base value within the range –16777215 to 16777215 if the CPU is a 24-pin address CPU, or the range –536870912 to 536870911 if the CPU 32-pin address CPU.

`(fp)` specifies the Frame-pointer register. Parentheses are required.

NOTE: The GNX Assembler uses the range –536870912 to 536870911 for displacement. It is up to the user to limit this range if a 24-pin address CPU is used.

Description: A Frame Memory operand specifies an operand at a memory address. The address is the sum of the displacement *disp* and the contents of the Frame-pointer register.

Example:

```
1                            .set   TEN, 10
2
3  T00000000   01aec000   addw   INTEG, 31(fp)
               00001f
4  T00000007   43c514c0   addd   (TEN*2)(fp), INTEG
               000000
```

In line 3, 31(fp) is a Frame Memory operand. The instruction adds the word at the address specified by the symbol INTEG to the word at the memory address specified by 31(fp). The result is stored at 31(fp).

In line 4, (TEN*2)(fp) is a Frame Memory operand. The expression "(TEN*2)" evaluates to the absolute value 20. This value is added to the contents of the Frame-pointer register (fp) to yield the operand's address. The instruction adds the double-word at this address to the double-word at the address specified by the symbol INTEG. The result is stored at INTEG.

NOTE: 24-pin address CPUs include the NS320*xx* and the NS32CG16.

32-pin address CPUs include the NS32332, NS32532, NS32FX16, NS32CG160, NS32GX32, and the NS32GX320.

4.2.5 Frame Memory Relative Operands

Syntax: *disp2*`(`*disp1*`(fp))`

where:

disp1	is an expression with absolute type with a value in the range –16777215 to 16777215 if the CPU is a 24-pin address CPU, or the range –536870912 to 536870911 if the CPU is a 32-pin address CPU.
disp2	is an expression with absolute type with a value absolute value within the range –16777215 to 16777215 if the CPU is a 24-pin address CPU, or the range –536870912 to 536870911 if the CPU is a 32-pin address CPU.
`(fp)`	specifies the Frame-pointer register. Parentheses are required. Parentheses are required around *disp1*`(fp)`. NOTE: The GNX Assembler uses the range –536870912 to 536870911. It is up to the user to limit this range if a 24-pin address CPU is used.

Description: A Frame Memory Relative operand specifies an operand at a memory address which is relative to the contents of a double-word in memory. The address is the sum of *disp2* and the double-word at the address specified by the Frame-pointer relative value *disp1*`(fp)`.

Example:

```
1                            .set   TEN, 10
2                            .set   FIFTY, 50
3
4  T00000000   1404330f   movb   r0, 15(FIFTY+1(fp))
5  T00000004   d7801400   movd   0((TEN*2)(fp)), r3
```

In the example, 15(FIFTY+1(fp)) is a Frame Memory Relative operand. The instruction copies the low-order byte of register r0 to the specified address. The address is the sum of 15 and the double-word at the address (FIFTY+1(fp)). The address (FIFTY+1(fp)) is the sum of the symbol FIFTY and one, which evaluates to the absolute value 51, and the current contents of the fp register.

Line 4 moves the double-word pointed to by 20(fp) to r3.

NOTE: 24-pin address CPUs include the NS320*xx* and the NS32CG16.

32-pin address CPUs include the NS32332, NS32532, NS32FX16, NS32CG160, NS32GX32, and the NS32GX320.

4.2.6 Stack Memory Operands

Syntax: *disp* `(sp)`

where:

disp	is a displacement or expression which evaluates to an absolute base value within the range –16777215 to 16777215 if the CPU is a 24-pin address CPU, or the range –536870912 to 536870911 if the CPU is a 32-pin address CPU.
`(sp)`	specifies the current stack pointer. The current stack pointer may be the User Stack Pointer (sp1) or the Interrupt Stack Pointer (sp0). The `s` bit in the *psr* specifies which pointer is currently active. Parentheses are required. NOTE: The GNX Assembler uses the range –536870912 to 536870911. It is up to the user to limit this range if a 24-pin address CPU is used.

Description: A Stack Memory operand specifies an operand at a memory address. The address is computed as the sum of the displacement *disp* and the contents of the current stack pointer register.

Example:

```
1                         .set   TEN, 10
2
3  T00000000   41aec000  addw   INTEG, 31(sp)
               00001f
4  T00000007   43cd14c0  addd   (TEN*2)(sp), INTEG
               000000
```

In line 3, 31(sp) is a Stack Memory operand. The instruction adds the word at the address specified by the symbol INTEG to the word at the memory address specified by 31(sp). The result is stored at 31(sp).

In line 4, (TEN*2)(sp) is a Stack Memory operand. The expression "(TEN*2)" evaluates to the absolute value 20. This value is added to the contents of the Stack-pointer register to yield the operand's address. The instruction adds the double-word at this address to the double-word at the address specified by the symbol INTEG. The result is stored at INTEG.

NOTE: 24-pin address CPUs include the NS320*xx* and the NS32CG16.

32-pin address CPUs include the NS32332, NS32532, NS32FX16, NS32CG160, NS32GX32, and the NS32GX320.

4.2.7 Stack Memory Relative Operands

Syntax: *disp2*`(`*disp1*`(`**`sp`**`))`

where:

disp2 is an expression with absolute type with a value in the range –16777215 to 16777215 if the CPU is a 24-pin address CPU, or the range –536870912 to 536870911 if the CPU is a 32-pin address CPU.

disp1 is a displacement or expression which evaluates to an absolute base value within the range –16777215 to 16777215 if the CPU is a 24-pin address CPU, or the range –536870912 to 536870911 if the CPU is a 32-pin address CPU.

`(sp)` specifies the current stack pointer. The current stack pointer may be the User Stack Pointer `(sp1)` or the Interrupt Stack Pointer `(sp0)`. The `s` bit in the *psr* specifies which pointer is currently active. Parentheses are required.

Parentheses are required around the stack memory value *disp1*`(sp)`.

NOTE: The GNX Assembler uses the range –536870912 to 536870911. It is up to the user to limit this range if a 24-pin address CPU is used.

Description: A Stack Memory Relative operand specifies an operand at a memory address which is relative to the contents of a double-word in memory. The address is the sum of *disp2* and the double-word at the address specified by the Stack-pointer relative value, *disp1*`(sp)`.

Example:

```
1                               .set   TEN, 10
2                               .set   FIFTY, 50
3
4  T00000000   5404350f   movb   r0, 15(FIFTY+3(sp))
5  T00000004   17881400   movd   0((TEN*2)(sp)), r0
```

In the above example, 15(FIFTY+3(sp)) is a Stack Memory Relative operand. The instruction copies the low-order byte of register r0 to the specified address. The address is the sum of 15 and the double-word at address FIFTY+3(sp). The address FIFTY+3(sp) is the sum of the symbol FIFTY, which evaluates to the absolute value 50, 3, and the contents of the current stack pointer.

In line 5, 0((TEN*2)(sp)) is a Stack Memory Relative operand. The instruction copies the double-word at the address 0((TEN*2)(sp)) into register r0. The address 0((TEN*2)(sp)) is the sum of 0 and (TEN*2)(sp). (TEN*2)(sp) is the sum of 20 and the current contents of the Stack-pointer register.

NOTE: 24-pin address CPUs include the NS320*xx* and the NS32CG16.

32-pin address CPUs include the NS32332, NS32532, NS32FX16, NS32CG160, NS32GX32, and the NS32GX320.

4.2.8 Static Memory Operands

Syntax: *disp*`(sb)`

or

^*expression*

where:	*disp*	is a displacement or expression which evaluates to an absolute value within the range –16777215 to 16777215 if the CPU is a 24-pin address CPU, or the range –536870912 to 536870911 if the CPU is a 32-pin address CPU.
	`(sb)`	specifies the Static Base register. Parentheses are required.
	expression	is a legal expression of any type.

NOTE: The GNX Assembler uses the range –536870912 to 536870911. It is up to the user to limit this range if a 24-pin address CPU is used.

Description: A Static Memory operand specifies an operand at a memory address. The address is the sum of the displacement *disp* and the contents of the Static Base register.

If the Static Memory operand is of the form ^*expression*, the address specified by *expression* is converted to an offset from the *Series 32000* module's Static Base.

Static Memory Operands (Cont)

Example:

```
1                              ..eet TEN, 10
2
3  T00000000  81aec000         addw  INTEG, 31(sb)
              00001f
4  T00000007  43d514c0         addd  (TEN*2)(sb), INTEG
              000000
5  T0000000e  97aec000         movd  INTEG, ^s_val
              0000c000
              018
6                              .data
7  D00000000  00000000 s_val:.double 0
```

In line 3 of the example, 31(sb) is a Static Memory operand. The instruction adds the word at the address specified by the symbol INTEG to the word at the memory address specified by 31(sb). The result is stored at 31(sb).

In line 4 of the example, (TEN*2)(sb) is a Static Memory operand. The expression "(TEN*2)" evaluates to the absolute value 20. This value is added to the contents of register sb to yield the operand's address. The instruction adds the double-word at this address to the double-word at the address specified by the symbol INTEG. The result is stored at INTEG.

In line 5 of the example, "^s_val" is a Static Memory operand. The displacement value is the distance from the Static Base to the address specified by the "s_val" label. Section 3.4 explains how the Static Base is determined. The instruction moves the value stored at the location INTEG to the s_val location.

NOTE: 24-pin address CPUs include the NS320*xx* and the NS32CG16.

32-pin address CPUs include the NS32332, NS32532, NS32FX16, NS32CG160, NS32GX32, and the NS32GX320.

4.2.9 Static Memory Relative Operands

Syntax:	*disp2*`(`*disp1*`(sb))`	
where:	*disp2*	is an expression of absolute type with a value in the range −16777215 to 16777215 if the CPU is a 24-pin address CPU, or the range −536870912 to 536870911 if the CPU is a 32-pin address CPU.
where:	*disp1*	is a displacement or expression which evaluates to an absolute base value within the range −16777215 to 16777215 if the CPU is a 24-pin address CPU, or the range −536870912 to 536870911 if the CPU is a 32-pin address CPU.
	`(sb)`	specifies the Static Base register. Parentheses are required.

Parentheses are required around the static memory value, *disp1*`(sb)`.

NOTE: The GNX Assembler uses the range −536870912 to 536870911. It is up to the user to limit this range if a 24-pin address CPU is used.

Description: A Static Memory Relative operand specifies an operand at a memory address which is relative to the contents of a double-word in memory. The address is the sum of *disp* and the double-word at the address specified by the sb relative value, *disp1*`(sb)`.

Example:

```
1                          .set   TEN, 10
2                          .set   FIFTY, 50
3
4 T00000000   9404320f    movb   r0, 15(FIFTY(sb))
5 T00000004   17901400    movd   0((TEN*2)(sb)), r0
```

In the above example, 15(FIFTY(sb)) is a Static Memory Relative operand. The instruction copies the low-order byte of register r0 to the specified address. The address is the sum of 15 and the double-word contents of the address FIFTY(sb). The address FIFTY(sb) is the sum of the symbol FIFTY, which evaluates to the absolute value 50, and the current contents of the sb register. In line 5 of the example, 0((TEN*2)(sb)) is a Static Memory Relative operand. The statement moves the double-word pointed to by (TEN*2)(sb) to r0. (TEN*2)(sb) is the sum of 20 and the current contents of the Static Base register.

NOTE: 24-pin address CPUs include the NS320*xx* and the NS32CG16.

32-pin address CPUs include the NS32332, NS32532, NS32FX16, NS32CG160, NS32GX32, and the NS32GX320.

4.2.10 Program Memory Operands

Syntax: * { + | - } *disp*

or

%expression

where:	*	is the current contents of the Program Counter register.
	disp	is a displacement or expression which evaluates to an absolute value within the range −16777215 to 16777215 if the CPU is a 24-pin address CPU, or the range −536870912 to 536870911 if the CPU is a 32-pin address CPU.
	expression	is a legal expression of any type. NOTE: The GNX Assembler uses the range −536870912 to 536870911. It is up to the user to limit this range if a 24-pin address CPU is used.

Description: A Program Memory operand specifies an operand at a memory address. The address is the sum of the displacement *disp* and the current contents of the Program Counter register.

If the Program Memory operand is of the form *%expression*, the address specified by *expression* is converted to an offset from the current location, *i.e.*, the contents of the Program Counter register.

Example:

```
1                                  .set     TEN, 10
2
3  T00000000  c1aec000             addw     INTEG, *+31
              0000c000
              001f
4  T0000000a  43ddffff             addd     *-(TEN*2), INTEG
              ffecc000
              0000
5  T00000014  d7aec000             movd     INTEG, %data
              0000c000
              000c
6                                  .data
7  D00000000  00000000   data:     .double 0
```

In line 3 of the example, "*+31" is a Program Memory operand. The instruction copies the word at the address specified by the symbol INTEG to the word at the memory address specified by the contents of the Program Counter register plus 31. The result is stored at the "*+31" address.

In line 4 of the example, "*-(TEN*2)" is a Program Memory operand. The expression "(TEN*2)" evaluates to the absolute value 20. This value is subtracted from the contents of the Program Counter register. The instruction adds the double-word at this address to the double-word at the address specified by the symbol INTEG. The result is stored at INTEG.

In line 5 of the example "%data" is a Program Memory operand. It is interpreted as the distance from the current location, *i.e.*, the contents of the Program Counter register and the address specified by the "data" label. The instruction moves the value stored at the location INTEG to the location data.

NOTE: 24-pin address CPUs include the NS320*xx* and the NS32CG16.

32-pin address CPUs include the NS32332, NS32532, NS32FX16, NS32CG160, NS32GX32, and the NS32GX320.

4.2.11 Immediate Operands

Syntax: *$expression*

where: *expression* is one of the following:

- A short or long format floating-point value (refer to Section 2.4.2).
- A character constant (refer to Section 2.4.3).
- A legal expression of any type (refer to Section 2.7).

Description: Immediate operands are encoded into the Immediate addressing mode; thus, the operand's value is stored in the instruction stream. If the expression is a relative type or an external, undefined type, the assembler generates a relocation entry for the operand. The linker uses the relocation entry to update the operand address at link time.

The range of immediate operands is limited by the length specifier of the instruction as follows:

- For expressions of type absolute, the ranges are:

–128..255	for byte instructions.
–32768..65535	for word instructions.
–2147483648..2147483647	for double-word instructions.

- For floating-point expressions, the positive ranges are:

$1.18 \times 10^{-38}..3.40 \times 10^{38}$	for single-precision instructions.
$2.23 \times 10^{-308}..1.80 \times 10^{308}$	for double-precision instructions.

If a character constant is shorter than the length required by the instruction, the high-order bytes are zero-filled. If the relocated address of a relative type is too large for the instruction, an error occurs at link time.

Immediate Operands (Cont)

Example:

```
 1                                 .set    NUMB, 5
 2  T00000000   55a50005    movw    $NUMB, TEMP
                c0000000
 3
 4  T00000008   be01a06b    addf    $0f3.14152e26, f0
                81ee23
 5
 6  T0000000f   57a50000    movd    $'?', LAST
                003fc000
                0000
 7
 8                          .set    ONE, 1
 9                          .set    THREE, ONE+2
10  T00000019   04a003      cmpb    $THREE, r0
```

Example line 2 copies the constant 5 to the memory address specified by TEMP.

Example line 4 adds the floating-point number 3.14152e26 to the contents of register f0.

Example line 6 copies the character constant '?' to the double-word at the address specified by LAST.

Example line 10 compares the value of the expression "ONE+2" with the low-order byte of register r0. The expression must evaluate to an absolute value in the range –128 to 255, in this case the value is 3.

4.2.12 Absolute Operands

Syntax: *@expression*

where: *expression* is a legal expression of any type.

Description: An Absolute operand specifies the absolute memory address of an operand. Regardless of the type of the expression that specifies the address, or the default addressing mode that the assembler would otherwise use, an Absolute operand always implies the Absolute addressing mode.

Examples:

```
1. addw   $H'1234, @9

2. addw   @TWELVE, r0

3. .set   BASE, 100
   addw   @BASE, r0
```

In example 1, @9 is an Absolute operand. The instruction adds the immediate operand H'1234 to the word starting at absolute address 9. The result is stored at the absolute address.

In example 2, @TWELVE is an Absolute operand. The symbol TWELVE may be of any type. The instruction adds the word at the absolute address specified by TWELVE to the low-order word of register r0. If TWELVE is a segment relative symbol (*e.g.*, text, data, etc.) the assembler generates a relocation entry so that the correct address can be inserted at link time. The result is stored in r0.

In example 3, BASE is location 100, type absolute. The addw instruction adds the word at the absolute address specified by BASE to the low-order word of register r0. The result is stored in the low-order word of r0. The upper word of r0 is undisturbed.

4.2.13 External Operands

Syntax: *disp* `(`*link offset*`(`**`ext`**`))`

where:	*disp*	is an expression with absolute type with a value in the range –16777215 to 16777215 if the CPU is a 24-pin address CPU, or the range –536870912 to 536870911 if the CPU is a 32-pin address CPU.
	link offset	specifies the byte offset from the base of the Link Table. The link offset must be an expression of type link or type absolute, and a multiple of four bytes.
	`(ext)`	is a literal that represents the base address of the Link Table. The *Series 32000* processor gets this address from the Module Table entry for the current module. Parentheses are required. Parentheses are required around the Link Table entry value, *link offset*`(ext)`. NOTE: The GNX Assembler uses the range –536870912 to 536870911. It is up to the user to limit this range if a 24-pin address CPU is used.

Description: An External operand specifies a Link Table entry and, possibly, an offset. During execution the contents of the Link Table entry and the offset are added together to produce an address. External operands should be used only in *Series 32000* modules.

Examples:

```
1. movb   r0, 12(ext)
2. movb   LAST, THREE(TWELVE(ext))
```

In example one, 12(ext) is an External operand. The instruction copies the low-order byte of register r0 to the byte specified by 12(ext). The external address is the sum of the double-word contents of the third Link Table entry and 0 (the default offset).

In example two, THREE(TWELVE(ext)) is an External operand. The instruction copies the byte at the address specified by LAST to the byte specified by THREE(TWELVE(ext)). The symbol THREE must evaluate to an absolute value. The symbol TWELVE must evaluate to an absolute value, or a value of type link, that is a multiple of four.

NOTE: 24-pin address CPUs include the NS320*xx* and the NS32CG16.

32-pin address CPUs include the NS32332, NS32532, NS32FX16, NS32CG160, NS32GX32, and the NS32GX320.

4.2.14 Top-of-Stack Operands

Syntax: *tos*

where: *tos* is the required keyword.

Description: A Top-of-Stack operand (*tos*) specifies an operand at the top of the stack. The top of the stack is the address specified by the stack pointer. The stack pointer will be either the User Stack Pointer (sp1), or the Interrupt Stack Pointer (sp0) depending on the value of the `s` bit in the *psr*. (Refer to Section 1.3.2.)

Top-of-Stack operands are encoded as *tos* addressing mode. (See the *Series 32000 Programmer's Reference Manual.*)

If a Top-of-Stack operand is read, the stack pointer is incremented by the length (in bytes) of the operand after the read is performed. If the operand is to be written, the operation decrements the stack pointer by the length (in bytes) of the operand before the write is performed. When an operand is both read and written, the stack pointer is not modified.

Examples:

```
1. movb   r0, tos
2. addb   tos, tos
```

Example 1 decrements the stack pointer by 1 and then copies the low-order byte of register r0 to the top of the stack.

Example 2 reads the byte at the top of the current stack, increments the stack pointer by 1, and adds this byte to the byte at the new top of stack.

4.2.15 Scaled-Index Byte Operands

Syntax: *gen* [*register*:b]

where:

gen	specifies a general operand. It must not be an immediate operand or another Scaled-Index operand.
register	specifies a General-purpose register r*n* where *n* must be a decimal digit in the range of 0 to 7.
`:b`	is the byte-scaling flag; it specifies a multiplier of 1. The colon (:) is required.
`[ ]`	are the required brackets.

Description: A Scaled-Index Byte operand specifies an operand at a memory address which is relative to the address specified by a general operand. The address is the sum of the contents of the General-purpose register r*n* multiplied by 1 and the address given by the general operand *gen*.

Example:

```
movb   r0, 5(sp)[r1:b]
```

In this example, 5(sp)[r1:b] is a Scaled-Index Byte operand. The instruction copies the low-order byte of register r0 to the specified address. The address is the sum of the contents of the register r1 multiplied by 1 and the address specified by 5(sp).

4.2.16 Scaled-Index Word Operands

Syntax: *gen* [*register*:w]

where:	*gen*	operand or another Scaled-Index operand.
	register	specifies a General-purpose register r*n* where *n* must be a decimal digit in the range of 0 to 7.
	`:w`	is the word-scaling flag; it specifies a multiplier of 2. The colon (:) is required.
	`[ ]`	are the required brackets.

Description: A Scaled-Index Word operand specifies an operand at a memory address which is relative to the address specified by a general operand. The address is the sum of the contents of the General-purpose register r*n* multiplied by 2 and the address of the general operand *gen*.

Example:

```
movb   r2, 10(sb)[r6:w]
```

In this example, 10(sb)[r6:w] is a Scaled-Index Word operand. The instruction copies the low-order byte of register r2 to the specified address. The address is the sum of the contents of the register r6 multiplied by 2 and the address specified by 10(sb).

4.2.17 Scaled-Index Double-Word Operands

Syntax: *gen* `[`*register*`:d]`

where:

gen	specifies a general operand. It must not be an absolute operand or another Scaled-Index operand.
register	specifies a General-purpose register r*n* where *n* must be a decimal digit in the range of 0 to 7.
`:d`	is the double-word scaling flag; it specifies a multiplier of 4. The colon (:) is required.
`[ ]`	are the required brackets.

Description: A Scaled-Index Double-Word operand specifies an operand at a memory address which is relative to the address specified by a general operand. The address is the sum of *gen* and the contents of the General-purpose register r*n* multiplied by 4 and the address of the general operand *gen*.

Example:

```
movb   r5, 10(r6)[r7:d]
```

In this example, 10(r6)[r7:d] is a Scaled-Index Double-Word operand. The instruction copies the low-order byte of register r5 to the specified address. The address is the sum of the contents of register r7 multiplied by 4 and the address specified by 10(r6).

4.2.18 Scaled-Index Quad-Word Operands

Syntax: *gen* [*register*:q]

where:

gen	specifies a general operand. It must not be an Absolute operand or another Scaled-Index operand.
register	specifies a General-purpose register r*n* where *n* must be a decimal digit in the range of 0 to 7.
:q	is the quad-word scaling flag; it specifies a multiplier of 8. The colon (:) is required.
[ ]	are the required brackets.

Description: A Scaled-Index Quad-Word operand specifies an operand at a memory address which is relative to the address specified by a general operand. The address is the sum of the contents of the General-purpose register r*n* multiplied by 8 and the address of the general operand *gen*.

Example:

```
movb   r0, 8(fp)[r1:q]
```

In this example, 8(fp)[r1:q] is a Scaled-Index Quad-Word operand. The instruction copies the low-order byte of register r0 to the specified address. The address is the sum of the contents of register r1 multiplied by 8 and the address specified by 8(fp).

4.2.19 Displacement Operands

Syntax: *addressing_mode*

where: *addressing_mode*

is not Register, Top-of-Stack, External, Immediate, or Scaled Indexed.

displacement_size

is an optional field. It can be one of the following:

`:b`	specifies 1 byte displacement.
`:w`	specifies 2 byte displacement.
`:d`	specifies 4 byte displacement.

Description: The *displacement_size* option allows the programmer to specify the size of the displacement field. If the desired displacement is too small, the assembler will issue an error. This feature is allowed only in an operand position and only if the operand involves a displacement.

Example:

```
1  T00000000  ea34         br L1:b
2  T00000002  a2a2a2a2     .space 10
              a2a2a2a2
              a2a2
3  T0000000c  ea8028       br L1:w
4  T0000000f  a2a2a2a2     .space 10
              a2a2a2a2
              a2a2
5  T00000019  eac00000     br L1:d
              1b
6  T0000001e  a2a2a2a2     .space 10
              a2a2a2a2
              a2a2
7  T00000028  ea0c         br L1
8  T0000002a  a2a2a2a2     .space 10
              a2a2a2a2
              a2a2
9                        L1:
```

In this program, the branch on line 1 is requested to be placed in a byte-displacement field, the branch on line 3 is placed in a word-displacement field, the branch on line 5 is placed in a double-word-displacement field, and the branch on line 7 is placed in the smallest single displacement field in which it will fit.

4.3 IMMEDIATE SUBRANGE OPERANDS

All Immediate Subrange operands are expressions of absolute type with values within subranges of the full double-word range.

4.3.1 Quick Operands

Syntax: [$]*quick*

Description: A signed constant or expression which evaluates to a constant immediate value within the range of -8 to 7 inclusive.

A Quick operand is encoded as a 4-bit signed integer in a field of the instruction. Quick operands are used in the quick-integer instructions. (See Section 5.3.) A Quick operand is the first operand in the following instructions:

Move Quick Integer	movq*i*
Compare Quick Integer	cmpq*i*
Add Quick Integer	addq*i*
Add, Compare, and Branch	acb*i*

Examples:

```
1.              movqd   -2, FIRST(r1)
2.              cmpqw   5, TEMP
3.              addqb   -1, r1
4.  LOOP:       muld    r2, r1
                acbb    -1, r0, LOOP
```

Example 1 copies the Quick operand –2 to the Double-word operand at the address specified by FIRST(r1).

Example 2 compares the Quick operand 5 with the Word operand specified by TEMP.

Example 3 adds the Quick operand –1 to the low-order byte of register r1.

In Example 4, the acbb instruction adds –1 to the low-order byte of register r0 and passes execution to the muld statement labelled LOOP as long as the result is not zero.

4.3.2 Block Length Operands

Syntax: [$]*integer_cons*

Description: An unsigned constant or expression which evaluates to an absolute value within the range of 1 to 16 (see below).

A Block Length operand is an unsigned constant which specifies the length of a block of integers. The block may be no more than 16 bytes in length. Therefore, the range of values the constant may have depends on the instruction's length specifier (`b`, `w`, or `d`) as shown by the following:

Length Specifier	**Integer Size (Bytes)**	**Range**
b	1	1 to 16
w	2	1 to 8
d	4	1 to 4

At assembly-time, the operand is multiplied by the corresponding integer size and then decremented by one, before being encoded in the instruction.

A Block Length operand is the last operand in the following instructions:

Move Multiple	movm*i*
Compare Multiple	cmpm*i*

Examples:

```
1. movmd    GEN1,    GEN2,    3
2. cmpmw    GEN1,    GEN2,    NELEMENTS
```

In example 1, the block length 3 specifies the number of double-words to move from the address specified by GEN1 to the address specified by GEN2.

In example 2, the block length is the expression "NELEMENTS." It specifies the number of elements to be compared. NELEMENTS must evaluate to a number in the range of 1 to 8.

4.3.3 Bit-Field Length Operands

Syntax:	[$]*integer_cons*

Description: An unsigned constant or an expression which evaluates to an absolute value within the range of 1 to 32. At assembly-time, the number is decremented by 1 before being encoded into the instruction.

A Bit-Field Length operand is an unsigned number. It specifies the length of a bit field in a bit-field instruction (Section 5.7). The length operands of the short bit-field instructions are encoded into a 5-bit field and into a byte in the other bit-field instructions. A Bit-Field Length operand is the last operand in the following instructions:

Extract Field	ext*i*
Insert Field	ins*i*
Extract Field Short	exts*i*
Insert Field Short	inss*i*

Examples:

```
1. extsw  BASE, DEST, 3, 14
2. inssb  SRC, BASE, 7, 10
```

In example one, 14 is the bit-field length constant. The instruction computes the location of a bit field that is 14 bits in length at 3 bits offset from the address specified by BASE and then copies the field to the address specified by the symbol DEST.

In example two, 10 is the bit-field length constant. The instruction computes the destination of a bit field that is 10 bits in length by adding a bit offset 7 to the address referenced by BASE. It then moves the field from the address specified by SRC to the destination.

4.3.4 Bit-Field Offset Operands

Syntax: [$]*integer_cons*

Description: An unsigned constant or expression which evaluates to an absolute value within the range of 0 to 7.

A Bit-Field Offset operand is an unsigned number. It specifies an offset which is used to compute the location of the first bit in a bit field. A Bit-Field Offset operand is the third operand in the following instructions:

Extract Field Short	exts*i*
Insert Field Short	inss*i*

Examples:

```
1. extsw  BASE, DEST, 3, 14
2. inssb  SRC, BASE, 7, 10
```

In example one, 3 is a Bit-Field Offset constant. The instruction copies a field, 14 bits in length, to the address specified by the symbol DEST and zero-fills the high-order two bits. The field's location is specified by adding a bit offset of 3 to the address BASE.

In example two, 7 is a Bit-Field Offset constant. The instruction copies SRC into the bit field addressed by adding a bit offset 7 to the address referenced by BASE. The length of the field to be written to is 10 bits.

4.3.5 Displacement Operands

Syntax: [$]*disp*

Description: A constant or expression which evaluates to an absolute base value within the range of –16777215 to 16777215 if the CPU is a 24-pin address CPU, or the range of –536870912 to 536870911 if the CPU is a 32-pin address CPU.

NOTE: The GNX Assembler uses the range –536870912 to 536870911. It is up to the user to limit this range if a 24-bit CPU is used.

A Displacement operand specifies a signed integer. A Displacement operand is the last (or only) operand in the following instructions:

Extract Field	ext*i*
Insert Field	ins*i*
Return From External Procedure	rxp
Return from Trap	rett
Enter New Context	enter
Return from Subroutine	ret

Examples:

```
1. extw   r2, BASE, DEST, 4
2. insb   r0, r2, 0(r1), 7
3. rxp    5
4. rett   1
5. enter  [r0], 2
6. ret    0
```

In example 1, the displacement is 4. It specifies the length of a field.

In example 2, the displacement is 7. It specifies the number of bits in the field to be written to.

In example 3, the displacement is 5. It specifies the number of bytes to be removed from the stack on the return from an external procedure.

In example 4, the displacement is 1. It specifies the number of bytes to be removed from the stack on the return from a trap.

In example 5, the displacement is 2. It specifies the number of bytes to allocate on the stack on entry to a procedure.

In example 6, the displacement is 0. It specifies the number of bytes to remove from the stack on the return from a local procedure.

Displacement operands are encoded the same way as memory displacements are. This format is described in the *Series 32000 Programmer's Reference Manual.*

NOTE: 24-pin address CPUs include the NS320*xx* and the NS32CG16.

32-pin address CPUs include the NS32332, NS32532, NS32FX16, NS32CG160, NS32GX32, and the NS32GX320.

4.4 PROGRAM MEMORY OPERANDS

Syntax: *label*

where: *label* is a legal expression of any type.

Description: A Program Memory operand specifies the destination of a branch instruction. The operand is interpreted as the address of the destination of the branch. The assembler converts this address to an offset from the current location counter. The Label operand must specify the address of the first byte of an instruction.

The Program Memory operand *label* may also use the syntax of the general Program Memory class operands, refer to Section 4.2.10.

A Label operand is the last (or only) operand in the following instructions:

Branch on Condition	b*cond*
Unconditional Branch	br
Add, Compare, and Branch	acb*i*
Branch to Subroutine	bsr

Examples:

```
1. beq   EQUAL
2. br    * + 8
3. acbb  -1, r0, * - 12
4. bsr   SUBROUTINE
```

In example 1, the Label operand is the label EQUAL. If the `z` flag in the psr is set, the instruction passes control to the location EQUAL.

In example 2, the Label operand is the "*+8" expression. The "*" symbol specifies the current location counter as a pc-relative address. The expression evaluates to a pc-relative address which is 8 bytes from the current location.

In example 3, the Label operand is "* - 12" which evaluates to a PC-relative value. As long as the contents of register r0 remains non-zero, the program continues its execution at a point which is 12 bytes lower than the current program code address.

In example 4, the Label operand is the label SUBROUTINE. When executed, the instruction transfers program control to the subroutine referenced by the label SUBROUTINE.

4.5 GENERAL REGISTER OPERANDS

Syntax: *register*

Description: A General-purpose register specified by r*n*, where *n* must be a decimal digit in the range of 0 to 7.

A General Register operand specifies one of the General-purpose registers. The value of the operand is the value contained in the register. A General Register operand is the first operand in the following instructions:

Extract Field	ext*i*
Insert Field	ins*i*
Convert to Bit Pointer	cvtp
Check Array Index	check*i*
Calculate Array Index	index*i*

General Register operands differ from Register operands (Section 4.2.2) only in that Register operands can be used with floating-point instructions to specify floating-point registers.

Examples:

```
1. extw    r2, BASE.A, DEST.A, 5
2. insb    r0, r2, 0(r1), 7
3. cvtp    r1, BASE.B, DEST.B
4. checkw  r3, BOUND1, K
5. indexd  r5, 3, J
```

In example 1, the General Register operand is r2. The contents of register r2 (together with the values of BASE.A and the displacement 5) is used as an offset to compute the location of a bit field which is five bits long. The field is copied to the address specified by DEST.A.

In example 2, the General Register operand is r0. The contents of register r0 is used as an offset from the base 0(r1) to specify the starting bit of the desired bit field.

In example 3, the General Register operand is r1. The contents of register r1 (together with the value of BASE.B) is used to compute the bit-address of a bit of memory. The bit-address is copied to DEST.B.

In example 4, the General Register operand is r3. The contents of register r3 reflects the difference between an array's lower bound, addressed by BOUND1, and an index `K` into the array.

In example 5, the General Register operand is r5. The register contains the accumulated index into an array. This index is generated by adding 1 to 3, multiplying this result by the previous contents of register index, and then adding the value referenced by `J` to this product.

4.6 REGISTER LIST (reglist) OPERAND

Syntax: [[*register*,,,]]

where:

[ ]	are the required brackets. The brackets are required even if no registers are specified.
register	specifies a symbol with register type.

Description: A Register List operand specifies one or more General-purpose registers. A Register List operand may be used in the following instructions:

Save Registers	save
Restore Registers	restore
Enter New Context	enter
Exit Context	exit

Examples:

```
1. save    [r1]
2. restore [r0, r1]
3. enter   [r0, r1, r2, r3, r4, r5, r6, r7], 5
4. exit    [ ]
```

In example 1, the operand specifies a single register r1. The contents of register r1 are saved on stack.

In example 2, the operand specifies two registers: r0 and r1. The instruction pops two consecutive double-words from the stack to the registers.

In example 3, the operand specifies all eight General-purpose registers. The instruction copies the contents of the eight General-purpose registers to consecutive double-words on stack. The Displacement operand 5 is then used to allocate five bytes of the stack for storage.

In example 4, the operand specifies no General-purpose registers and none are popped off the stack when the exit instruction is executed.

4.7 CONFIGURATION LIST (cfglist) OPERAND

Syntax: `[[c][i][de] [m][f] [ff][fm][fc][p]]`

where:	`[ ]`	are the required brackets. The brackets are required even if no configuration bit is specified.
	`c`	specifies the Clock Scaling bit for NS32CG16, NS32FX16, and NS32CG160.
		specifies the Custom Slave bit for the rest of the *Series 32000* processors.
	`i`	specifies the Interrupt Control Unit bit.
	`de`	specifies the direct exception bit in the NS32CG160.
	`m`	specifies the Clock Scaling Factor bit for the NS32CG16, NS32FX16, and NS32CG160.
		illegal for the NS32008, NS32GX32, and NS32GX320.
		specifies the Memory Management Unit bit for the NS32016, NS32032, NS32332, and NS32532.
	`f`	specifies the Floating-Point Unit bit.
	`ff`	specifies the Fast FPU Protocol bit, NS32332 and NS32532 only.
	`fm`	specifies the Fast MMU Protocol bit, NS32332 and NS32532 only.
	`fc`	specifies the Fast Custom Slave Protocol bit, NS32332 and NS32532 only.
	`p`	specifies the 4096 bytes page size bit, NS32332 and NS32532 only.

If a configuration option is specified, the corresponding bit in the *Series 32000* Configuration register is set (see the data sheet for the appropriate *Series 32000* processor); otherwise, the bit is cleared. Any combination is allowed; however, if more than one bit is specified, they must be separated by commas.

Description: A Configuration List operand specifies Configuration register bits. The list may specify any combination of the bits, depending on which devices are present in the system.

A Configuration List operand may be used in the `setcfg` instruction.

Examples:

```
1. setcfg [ ]
2. setcfg [f]
3. setcfg [de]
```

In example 1, the operand specifies no register bits. The instruction clears all bits in the Configuration register.

In example 2, the operand specifies the `f` bit. The instruction sets the `f` bit and clears the `i`, `m`, and `c` bits.

In example 3, the operand specifies the `de` bit of the NS32CG160.

4.8 PROCESSOR REGISTER OPERANDS

Syntax: *procreg*

Description: Specifies a Dedicated register. It must be one of the following register names:

upsr	User Processor Status Register
fp	Frame Pointer
sp	Stack Pointer
sb	Static Base
psr	Processor Status Register
intbase	Interrupt Base
mod	Module

The function of the Dedicated registers is described in Section 1.3.2.

A Processor Register operand specifies a Dedicated register. A Processor Register operand is the first operand in the following instructions:

Load Processor Register	lpr*i*
Store Processor Register	spr*i*

Examples:

```
1. lprw  mod, r1
2. sprb  upsr, TEMP
```

In example 1, mod is the Processor Register operand. The instruction copies the low-order word in register r1 to the Module register.

In example 2, upsr is the Processor Register operand. The instruction copies the low-order byte of the Processor Status register (*i.e.*, the user's part of the psr) to the address specified by the symbol TEMP.

4.9 NS32082 MEMORY MANAGEMENT REGISTER OPERAND

Syntax: *mmureg*

Description: Specifies a NS32082 Memory Management register. It must be one of the following register names:

bpr0	Breakpoint Register 0
bpr1	Breakpoint Register 1
pf0	Program Flow 0
pf1	Program Flow 1
bcnt	Breakpoint Count
ptb0	Page Table Base 0
ptb1	Page Table Base 1
sc	Sequential Count 0 and Sequential Count 1
msr	Memory Management Status Register
eia	Error/Invalidate Address

Memory Management registers are described in Section 1.3.4 and in the NS32082 Data Sheet.

A Memory Management Register operand specifies a Memory Management register. A Memory Management Register operand is the first operand in the following instructions:

Load Memory Management Register	lmr
Store Memory Management Register	smr

Examples:

```
1. lmr   bpr0,  SETBPR0
2. smr   msr,   SAVEMSR
```

In example 1, bpr0 is the Memory Management Register operand. The instruction loads the double-word at the address specified by SETBPR0 into the Breakpoint Register 0.

In example 2, msr is the Memory Management Register operand. The instruction stores the double-word contents of the msr at the address specified by the symbol SAVEMSR.

4.10 NS32382 MEMORY MANAGEMENT REGISTER OPERAND

Syntax: *mmureg*

Description: Specifies an NS32382 Memory Management register. It must be one of the following register names:

bar	Breakpoint Address Register
bmr	Breakpoint Mask Register
bdr	Breakpoint Data Register
ivar0	Invalid Virtual Address Register 0
ivar1	Invalid Virtual Address Register 1
mcr	MMU Control Register
msr	MMU Status Register
tear	Translation Exception Address Register
bear	Bus Error Address Register
ptb0	Page Table Base 0
ptb1	Page Table Base 1

Memory Management registers are described in Section 1.3.4 and in the NS32382 Data Sheet.

A Memory Management Register operand specifies a Memory Management register. A Memory Management Register operand is the first operand in the following instructions:

Load Memory Management Register	lmr
Store Memory Management Register	smr

Examples:

```
1. lmr   bpr0,  SETBPR0
2. smr   msr,   SAVEMSR
```

In example 1, bpr0 is the Memory Management Register operand. The instruction loads the double-word at the address specified by SETBPR0 into the Breakpoint Register 0.

In example 2, msr is the Memory Management Register operand. The instruction stores the double-word contents of the msr at the address specified by the symbol SAVEMSR.

4.11 NS32532 MEMORY MANAGEMENT REGISTER OPERAND

Syntax: *mmureg*

Description: Specifies an NS32532 Memory Management register. It must be one of the following register names:

ivar0	Invalid Virtual Address Register 0
ivar1	Invalid Virtual Address Register 1
mcr	MMU Control Register
msr	MMU Status Register
tear	Translation Exception Address Register
ptb0	Page Table Base 0
ptb1	Page Table Base 1

Memory Management registers are described in Section 1.3.4 and in the NS32532 Data Sheet.

A Memory Management Register operand specifies a Memory Management register. A Memory Management Register operand is the first operand in the following instructions:

Load Memory Management Register	lmr
Store Memory Management Register	smr

Examples:

```
1. lmr   bpr0,  SETBPR0
2. smr   msr,   SAVEMSR
```

In example 1, bpr0 is the Memory Management Register operand. The instruction loads the double-word at the address specified by SETBPR0 into the Breakpoint Register 0.

In example 2, msr is the Memory Management Register operand. The instruction stores the double-word contents of the msr at the address specified by the symbol SAVEMSR.

4.12 EXTERNAL PROCEDURE OPERANDS

Syntax: *external*

Description: An operand of the general operand type External with no offset, or a procedure label for which a Link Table entry has been defined.

An External Procedure operand specifies an entry in the Link Table for an external procedure descriptor. Link Table entries for external procedures, called external procedure descriptors, consist of the 16-bit address of an entry in the Module Table and a 16-bit offset. During execution, the address of the external procedure is calculated by adding the offset to the address in the Module Table entry. (See the section on software modules in the *Series 32000 Programmer's Reference Manual.*)

An External operand is the first operand in the following instructions:

Call External Procedure	cxp

Example:

```
.globl OUT
.link
.xpd   OUT

.text
cxp    OUT
```

In this example, OUT is an External operand. OUT references a Link Table entry address. The Link Table entry offset of OUT is divided by four to obtain the Link Table entry number. The double-word at this Link Table entry number specifies a Module Table entry and an offset from the address contained in the Module Table entry. The address plus offset is the start address of a procedure.

4.13 LENGTH OF DISPLACEMENTS

The GNX Assembler attempts to determine the optimal number of bytes to allocate for each displacement it assembles, *i.e.*, the smallest number of bytes into which the displacement value will fit. If an expression involves a forward reference, the displacement size cannot be determined until the reference is defined. If an expression is composed of one undefined symbol plus or minus a constant, the Assembler allocates one byte and makes an entry in the span-dependent instruction link list. The actual size required to hold the displacement is determined at the completion of the first pass. If the size of the displacement cannot be determined at assembly time, the GNX Assembler uses the largest displacement size available by default.

The Assembler determines displacement length by the following rules:

1. If the expression evaluates to a defined absolute value, the Assembler uses the smallest displacement that will fit.
2. If the expression can be evaluated and the type is relocatable, that is, relative to the memory location into which an object file segment will be loaded, then the Assembler allocates the maximum number of bytes and generates a relocation entry. The relocation entry will be used to determine the actual address at link time.
3. If the expression contains an external, undefined term, the maximum displacement is generated.
4. Optimizing of displacement may be overridden by using the displacement operands, refer to Section 4.2.19, or the –n flag (or /nosdi flag for VMS).

A programmer may set the maximum length displacement using a command line argument. If the maximum size chosen is too small, an error will result at link time. See Chapter 9.

NOTE: Displacements which span an `.align` directive use the maximum length displacement.

For example:

```
        br foo
        .align  4
    foo:
```

uses 4 bytes or a user-specified *disp* for displacement.

Chapter 5

SERIES 32000 INSTRUCTION SET

5.1 INTRODUCTION

This chapter presents the syntax of the *Series 32000* assembly language instruction set. The syntax describes the following:

- Opcode Mnemonic
- Operands

The opcode mnemonic specifies the operation to be performed by the instruction. In most cases, the opcode mnemonic consists of three or more letters and one of the following:

i — length of integer operands — must be `b` (byte), `w` (word), or `d` (double-word)

f — length of floating-point operands — must be `f` (float) or `l` (long)

When encoding an instruction, the *i* and *f* must be replaced by the appropriate operand length specifier.

The operand syntax specifies the number, type, and access class of the operands. The first line of the operand description indicates the use for the operand, *e.g.*, the first operand of mov*i* is marked src; therefore, that operand is the source for the move. The second line, when present, indicates the access class for the operand. For a complete discussion of the access classes, see the *Series 32000 Programmer's Reference Manual*. The operand type is indicated by a combination of the access class and, when present, a third line. Most of the operands can use any of the general operand types. The general operands use a particular set of access classes, so the access class alone is used to identify a general operand. For nongeneral operands the access class is not sufficient; the third line of the operand description states the specific type of operand for nongeneral operands.

The general operand access classes are:

read — the operand is read.
write — the operand is written.
rmw — the operand is read, modified, and written.
addr — the address of the memory location designated by the operand is calculated. Whether or not the address is accessed depends upon the instruction.

regaddr — the operand designates either a memory location or a general register which is in turn used as a base for a bit address calculation.

The other access classes used are:

quick — 4-bit constant is read
reg — double-word from register is read
short — 4-bit condition code is read
imm — a) 8-bit register mask is read
    b) concatenated 5-bit and 3-bit constants are read
disp — 1-, 2-, or 4-byte displacement is read

NOTE: The GNX Assembler groups quick, *imm* type `b` and disp operands together as immediate subrange operands (Section 4.3), the 5-bit field of imm type `b`. The GNX Assembler uses bit-field offset operands to store the 3-bit field of imm type `b`. The GNX Assembler uses block-length operands to store the 4-bit constant of the movm*i* and cmpm*i* instructions. (The constant is encoded in a disp operand.)

Instead of being a member of one of the access classes an operand may be:

cond = eq — equal: z=1
ne — not equal: z=0
cs — carry set: c=1
cc — carry clear: c=0
lo — lower: z=0 and l=0
hs — higher or the same: z=1 or l=1
lt — less than: z=0 and n=0
ge — greater than or equal: z=1 or n=1
fs — flag set: f=1
fc — flag clear: f=0
hi — higher: l=0 and z=0
ls — lower or the same: l=1
gt — greater than: n=1
le — less than or equal: n=0

cfglist = [[i][f][de][m][c] [ff][fm][fc][p]]
[ ] — no slaves
i — ICU (Interrupt Control Unit)
de — Direct exception
f — FPU (Floating-Point Unit)
m — MMU (Memory Management Unit)
    or Clock scaling factor
c — Custom slave clocking scale or clock scaling list
ff — Fast FPU

fm — Fast MMU (32332 and 32532 only)
fc — Fast Custom (32332 and 32532 only)
p — 4 Kbytes (4096 byte) page (32332 and 32532 only)

procreg = upsr — User psr (low byte in psr)
fp — Frame Pointer
sp — Stack Pointer
sb — Static Base
psr — Processor Status Register
intbase — Interrupt Base
mod — Module

mmureg (NS32082) = bpr0 — Breakpoint Register 0
bpr1 — Breakpoint Register 1
pf0 — Program Flow 0
pf1 — Program Flow 1
sc — Sequential Count Registers sc0 and sc1
msr — Memory Status Register
bcnt — Breakpoint Count
ptb0 — Page Table Base 0
ptb1 — Page Table Base 1
eia — Error/Invalidate Address

mmureg (NS32382) = bar — Breakpoint Address Register
bmr — Breakpoint Mask Register
bdr — Breakpoint Data Register
ivar0 — Invalid Virtual Address Register 0
ivar1 — Invalid Virtual Address Register 1
mcr — MMU Control Register
msr — MMU Status Register
tear — Translation Exception Address Register
bear — Bus Error Address Register
ptb0 — Page Table Base 0
ptb1 — Page Table Base 1

mmureg (NS32532) = ivar0 — Invalid Virtual Address Register 0
ivar1 — Invalid Virtual Address Register 1
mcr — MMU Control Register
msr — MMU Status Register
tear — Translation Exception Address Register
ptb0 — Page Table Base 0
ptb1 — Page Table Base 1

[b[,]][u | w] = b — backwards
w — while

b, w — backwards and while
u — until
b, u — backwards and until

When encoding an instruction, the above operand names must be replaced with appropriate operands. Operand types and syntax are described in detail in Chapter 4. Commas, if shown, are required.

Some instructions are privileged and may be executed only when the system is in the supervisor-mode (*i.e.*, the u bit in the *psr* is clear). In the following sections, the symbol "§" specifies a privileged instruction.

Instruction operations are defined in the *Series 32000 and Series 32000/EP Programmer's Reference Manuals*.

NOTE: The MMU can generate an ABT trap for any instruction; this occurs when the instruction or operand is stored in, or tries to write to, an address not currently in memory or in a protected memory location. Some instructions may cause an ABT without themselves causing a page fault. Only this latter group of instructions are marked as capable of taking an ABT trap.

The following notations are used in the description of the action of the instructions:

OPERATIONS

Operators	*Definition*
:=	replace value on left with value on right.
+, -, *	addition, subtraction, multiplication.
div	division with truncation.
/	division with rounding toward negative infinity.
mod	modulus (or remainder after “div”).
**	exponential operator.
AND	bit-wise logical AND.
NOT(_name_)	bit-wise complement of name.
OR	bit-wise logical OR.
XOR	bit-wise logical exclusive OR.
SIGN(_name_)	evaluates to sign bit of name.
BIT(_base, offset_)	evaluates to bit at bit address 8*_base+offset_.
FIELD(_base, offset, length_)	evaluates to field at bit address 8*_base+offset_; _length_ is field length in bits.
WORD(_address_)	word operand from address.
DOUBLEWORD(_address_)	double-word operand from address.
ADDR(_name_)	evaluates to address of name.
return address	address of next sequential instruction after a Branch, Jump, Call instruction.
PUSHi(_name_)	push argument name, of length i, onto the stack.
POPi(_name_)	pop item, of length i, from the stack into name.

TRAPS

Trap Name		*Taken on*
DVZ	=Divide by Zero Trap	a zero divisor in a Divide, Modulus, Quotient, Remainder, or Divide Extended Integer instruction.
ILL	=Illegal Instruction Trap	a Privileged instruction when u=1
UND	=Undefined Instruction Trap	a Memory Management instruction when cfg m=0, or a Floating-point instruction when cfg f=0, or any undefined operation codes.
FPU	=Floating-Point Error Trap	a Floating-point instruction on: Underflow Overflow Invalid Division Illegal Instruction Reserved Operand Inexact Result
SVC	=Supervisor Trap	a Supervisor Call instruction.
FLG	=Flag Trap	a Flag instruction when f=1.
BPT	=Breakpoint Trap	a Breakpoint instruction.
ABT	=Instruction Abort Trap	a Page fault.
TRC	=Trace Trap	instruction completion while in trace mode (Series 32000 family only)
SLAVE	=Slave Trap	exceptional condition detected during slave instruction execution.
OVF	=Integer Overflow Trap	detected overflow during integer instruction execution (NS32532, NS32GX32, and NS32GX320 only).
DBG	=Debug Trap	a condition selected by a DSR bit is detected (NS32532, NS32GX32, and NS32GX320 only).

5.2 INTEGER INSTRUCTIONS

SYNTAX			**OPERATIONS**	**FLAGS AFFECTED**	**TRAPS TAKEN**
Arithmetic Instructions					
add*i*	*src*, read.*i*	*dest* rmw.*i*	Add *dest* := *dest* + *src* c := 1 on carry; c := 0 on no carry f := 1 on overflow; f := 0 on no overflow	c f	—
addc*i*	*src*, read.*i*	*dest* rmw.*i*	Add with Carry *dest* := *dest* + *src* + c c := 1 on carry; c := 0 on no carry f := 1 on overflow; f := 0 on no overflow	c f	—
cmp*i*	*src1*, read.*i*	*src2* read.*i*	Compare z := 1 if *src1*=*src2*; z := 0 otherwise n := 1 if *src1*>*src2*; n := 0 otherwise (signed operands) l := 1 if *src1*>*src2*; l := 0 otherwise (unsigned operands)	z n l	—
sub*i*	*src*, read.*i*	*dest* rmw.*i*	Subtract *dest* := *dest* - *src* c := 1 on borrow; c := 0 on no borrow f := 1 on overflow; f := 0 on no overflow	c f	—
subc*i*	*src*, read.*i*	*dest* rmw.*i*	Subtract with Borrow *dest* := *dest* - (*src* + c) c := 1 on borrow; c := 0 on no borrow c := 1 on overflow; f := 0 on no overflow	c f	—
neg*i*	*src*, read.*i*	*dest* write.*i*	Negate *dest* := 0 - *src* c := 1 on carry; c := 0 on on carry f := 1 on overflow; f := 0 on no overflow	c f	—

abs*i*	*src*,
	read.*i*
		dest
		write.*i*
			Absolute Value
			if *src*< 0, then
			dest := 0 - *src*; f := 1 on overflow
			f := 0 on no overflow
			else
			dest := *src*; f := 0
				f	—

INTEGER INSTRUCTIONS (Cont)

SYNTAX			OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
Arithmetic Instructions					
mul*i*	*src*, read.*i*	*dest* rmw.*i*	Multiply *dest* := *src* * *dest*	—	—
div*i*	*src*, read.*i*	*dest* rmw.*i*	Divide if *src*=0, then TRAP(DVZ) else *dest* := *dest* DIV *src* (signed division; *dest* DIV *src* rounded toward negative infinity)	—	DVZ
mod*i*	*src*, read.*i*	*dest* rmw.*i*	Modulus if *src*=0, then TRAP(DVZ) else *dest* := *dest* - *src**(*dest* DIV *src*) (signed division; *dest* DIV *src* rounded toward negative infinity)	—	DVZ
quo*i*	*src*, read.*i*	*dest* rmw.*i*	Quotient if *src*=0, then TRAP(DVZ) else *dest* := *dest/src* (signed division; *dest/src* round toward zero)	—	DVZ
rem*i*	*src*, read.*i*	*dest* rmw.*i*	Remainder if *src*=0, then TRAP(DVZ) else *dest* := *dest* - *src**(*dest/src*) (signed division; *dest/src* rounded toward zero)	—	DVZ
Move Instructions					
mov*i*	*src*, read.*i*	*dest* write.*i*	Move *dest* := *src*	—	—
movxbw	*src*, read.b	*dest* write.w	Move Sign-Extending Byte to Word *dest* (low-order byte) := *src* *dest* (high-order bits) := SIGN(*src*)	—	—
movxbd	*src*, read.b	*dest* write.d	Move Sign-Extending Byte to Double-Word *dest* (low-order byte) := *src* *dest* (high-order bits) := SIGN(*src*)	—	—

SYNTAX			OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
Move Instructions					
movxwd	*src*, read.w	*dest* write.d	Move Sign-Extending Word to Double-Word *dest* (low-order byte) := *src* *dest* (high-order bits) := SIGN(*src*)	—	—
movzbw	*src*, read.b	*dest* write.w	Move Zero-Extending Byte to Word *dest* (low-order byte) := *src* *dest* (high-order bits) := 0	—	—
movzbd	*src*, read.b	*dest* write.d	Move Zero-Extending Byte to Double-Word *dest* (low-order byte) := *src* *dest* (high-order bits) := 0	—	—
movzwd	*src*, read.w	*dest* write.d	Move Zero-Extending Word to Double-Word *dest* (low-order word) := *src* *dest* (high-order bits) := 0	—	—
addr	*src*, addr	*dest* write.d	Compute Effective Address *dest* := ADDR(*src*)	—	—
Shift Instructions					

ash*i*	*count*,
	read.B
		dest
		rmw.*i*
			Arithmetic Shift (Left or Right)
			if *count*< 0, then
			dest := *dest* shifted right by ∣ *count* ∣ bits, emptied bit positions filled from original sign bit.
			else
			dest := *dest* shifted left by ∣ *count* ∣ bits, emptied bit positions filled with zero.
				—	—

lsh*i*	*count*,
	read.B*i*
		dest
		rmw.*i*
			Logical Shift (Left or Right)
			if *count*< 0, then
			dest := *dest* shifted right by ∣ *count* ∣ bits, emptied bit positions filled with zeroes.
			else
			dest := *dest* shifted left by ∣ *count* ∣ bits, emptied bit positions filled with zeroes.
				—	—

rot*i*	*count*,
	read.B*i*
		dest
		rmw.*i*
			Rotate (Left or Right)
			if *count*< 0, then
			dest := *dest* shifted right by ∣ *count* ∣ bits, end-around.
			else
			dest := *dest* shifted left by ∣ *count* ∣ bits, end-around.
				—	—

SYNTAX			OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
	Logical Instructions				
and*i*	*src*, read.*i*	*dest* rmw.*i*	Logical AND *dest* := *dest* AND *src*	—	—
or*i*	*src*, read.*i*	*dest* rmw.*i*	Logical OR *dest* := *dest* OR *src*	—	—
bic*i*	*src*, read.*i*	*dest* rmw.*i*	Bit Clear *dest* := *dest* AND NOT(*src*)	—	—
xor*i*	*src*, read.*i*	*dest* rmw.*i*	Exclusive OR *dest* := *dest* XOR *src*	—	—
com*i*	*src*, read.*i*	*dest* write.*i*	Complement *dest* := NOT(*src*)	—	—

5.3 QUICK INTEGER INSTRUCTIONS

	SYNTAX		OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
movq*i*	*src*,	*dest*	Move Quick Integer	—	—
	quick	write.*i*			
	quick	*dest* := *src*			
			(*src* sign-extended to *dest* length)		
cmpq*i*	*src*1,	*src*2	Compare Quick Integer	z	—
	quick	read.*i*	n		
	quick		z := 1 if *src*2=*src*1; z := 0 otherwise	l	
			n := 1 if *src*2<*src*1; n := 0 otherwise		
			(signed operands)		
			l := 1 if *src*2<*src*1; l := 0 otherwise		
			(unsigned operands)		
			(*src*1 sign-extended to *src*2 length)		
addq*i*	*src*,	*dest*	Add Quick Integer	c	—
	quick	rmw.*i*		f	
	quick		*dest* := *dest* + *src*		
			c := 1 on carry; c := 0 on no carry		
			f := 1 on overflow; f := 0 on no overflow		
			(*src* sign-extended to *dest* length)		

5.4 EXTENDED INTEGER INSTRUCTIONS

	SYNTAX		OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
mei*i*	*src*, read.*i*	*dest* rmw.*2i*	Multiply Extended Integer *dest* := *src* * (*dest* mod 2***i*) (unsigned operands) (low-order half of *dest*)	—	—
dei*i*	*src*, read.*i*	*dest* rmw.*2i*	Divide Extended Integer *dest* := (*dest* div *src*) * 2***i* + *dest* mod *src* (unsigned operands)	—	DVZ

5.5 BOOLEAN INSTRUCTIONS

SYNTAX			OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
not*i*	*src*, read.*i*	*dest* write.*i*	NOT *dest* := *src* XOR 1	—	—
s{*cond*}*i*		*dest* write.*i*	Save Condition Code as a Boolean if cond, then *dest* := 1 else, *dest* := 0	—	—

5.6 BIT INSTRUCTIONS

SYNTAX				OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
tbit*i*	*offset*, read.*i*	*base* regaddr		Test Bit f := BIT(*base*, *offset*)	f	—
sbit [i]*i*	*offset*, read.*i*	*base* regaddr		Set Bit f := BIT(*base*, *offset*) BIT(*base*, *offset*) := 1	f	—
cbit [i]*i*	*offset*, read.*i*	*base* regaddr		Clear Bit f := BIT(*base*, *offset*) BIT(*base*, *offset*) := 0	f	—
ibit*i*	*offset*, read.*i*	*base* regaddr		Invert Bit f := BIT(*base*, *offset*) BIT(*base*, *offset*) := NOT[BIT(*base*, *offset*)]	f	—
cvtp	*offset*, reg reg	*base*, addr	*dest* write.d	Convert to Bit Pointer *dest* :=(8*ADDR(*base*) + *offset*) mod 2**32	—	—

ffs*i*	*base*,
	read.*i*
		offset
		rmw.B
				Find First Set Bit
				if (*offset*< 0 or *offset*≥length in bits of *base*),
				then operation is undefined
				else
				j := *offset*
				while (j < length of *base* and BIT (*base*,j) = 0)
				do j := j+1
				if j = length of *base* then
				f := 1 ; *offset* := 0 else
				f := 0 ; *offset* := j
					f	—

5.7 BIT FIELD INSTRUCTIONS

	SYNTAX				OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
ext*i*	*offset,*	*base,*	*dest,*	*length*	Extract Field	—	—
	reg	regaddr	write.*i*	disp			
	reg				*dest* := FIELD(*base*, *offset*, *length*)		
exts*ii*	*base,*	*dest,*	*offset,length*		Extract Field Short	—	—
	regaddr	write.*i*	imm				
			cons3,cons5		*dest* := FIELD(*base*, *offset*, *length*)		
ins*i*	*offset,*	*src,*	*base,*	*length*	Insert Field	—	—
	reg	read.*i*	regaddr	disp			
	reg			disp	FIELD(*base*, *offset*, *length*) := *src*		
inss*i*	*src,*	*base,*	*offset,length*		Insert Field Short	—	—
	read.*i*	regaddr	imm				
			cons3,con5		FIELD(*base*, *offset*, *length*) := *src*		

5.8 STRING INSTRUCTIONS

SYNTAX		OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
movs*i*	[b[,]][u | w]	Move String *string2* := *string1* f:=1 if until/while condition is met; f:=0 otherwise	f	—
movst	[b[,]][u | w]	Move String with Translation *string2* := *translate-string* f:=1 if until/while condition is met; f:=0 otherwise	f	—
cmps*i*	[b[,]][u | w]	Compare String f:=1 if until/while condition is met; f:=0 otherwise z:=1 if *string1=string2* and f=0; else z:=0 n:=1 if *string2<string1* and f=0; else n:=0 l:=1 if *string2<string1* and f=0; else l:=0	z n l f	—
cmpst	[b[,]][u | w]	Compare String with Translation f:=1 if until/while condition is met; f:=0 otherwise z:=0 if translate-string≠*string2* and f=0; z:=0 otherwise n:=1 if translate-string>*string2* and f=0; n:=0 otherwise l:=1 if translate-string>*string2* and f=0; l:=0 otherwise	z n l f	—
skps*i*	[b[,]][u | w]	Skip String f:=1 if until/while condition is met; f:=0 otherwise	f	—
skpst	[b[,]][u | w]	Skip String with Translation f:=1 if until/while condition is met; f:=0 otherwise	f	—

The flags b, w, and u are optional. The u and w flags are mutually exclusive. The comma is required whenever both b and either u or w are specified.

5.9 BLOCK INSTRUCTIONS

SYNTAX				OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
movm*i*	*block1*, addr	*block2*, addr	*length* disp cons4	Move Multiple *block2* := *block1*	—	—
cmpm*i*	*block1*, addr	*block2*, addr	*length* disp cons4	Compare Multiple z:=1 if *block1*=*block2*; else z:=0 n:=1 if *block1*>*block2*; else n:=0 (signed integers) l:=1 if *block1*>*block2*; else l:=0 (unsigned integers)	z n l	—

5.10 PACKED DECIMAL INSTRUCTIONS

SYNTAX			OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
addp*i*	*src*, read.*i*	*dest* rmw.*i*	Add Packed Decimal *dest* := *dest* + *src* + c c := 1 on carry; c := 0 on no carry f := 0	c f	—
subp*i*	*src*, read.*i*	*dest* rmw.*i*	Subtract Packed Decimal *dest* := *dest* - *src* - c c := 1 on borrow; c := 0 on no borrow f := 0	c f	—

5.11 ARRAYINSTRUCTIONS

SYNTAX				OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
index*i*	*accum*, reg reg	*length*, read.*i*	*index* read.*i*	Calculate Array Index $accum := (length + 1) * accum + index$	—	—

check*i*	*dest*,
	reg
	reg
		bounds,
		addr
			src
			read.*i*
				Check Array Index
				if *bounds*(*upper*) > = *src* > = *bounds*(*lower*)
				then,
				  dest := *src* - *bounds*(*lower*)
				  f := 0
				else;
				  dest := undefined
				  f := 1
					f	—

5.12 PROCESSOR CONTROL INSTRUCTIONS

SYNTAX				OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
Direct and Conditional Jumping						
jump	*dest* addr			Jump pc := ADDR(*dest*)	—	—
b {*cond*}	*disp* short	*dest* disp label		Conditional Branch if *cond*, then pc := pc + *disp*	—	—
br	*dest* disp label			Unconditional Branch pc := pc + *disp*	—	—
case*i*	*index* read.*i*			Case Branch pc := pc + *index* (signed *index*)	—	—
acb*i*	*inc*, quick quick	*index*, rmw.*i*	*dest* disp label	Add, Compare, and Branch *index* := *index* + *inc* if *index* <> 0, then pc := pc + *disp*	—	—
Subroutine and Procedures						
jsr	*dest* addr			Jump to Subroutine PUSHD(return address) pc := ADDR(*dest*)	—	—
bsr	*dest* disp label			Branch to Subroutine PUSHD(return address) pc := pc + *disp*	—	—
ret	*constant* *disp*			Return from Subroutine POPD(pc) sp := sp + *constant*	—	—
cxp	*constant* disp external			Call External Procedure sp := sp - 2 PUSHW(mod) PUSHD(return address) temp := DOUBLEWORD (DOUBLEWORD(mod+4) +*constant*) mod := WORD(temp) sb := DOUBLEWORD(mod+0) pc := DOUBLEWORD(mod+8) + WORD(temp+2)	—	—

Subroutine and Procedures

SYNTAX		OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
cxpd	*desc* addr	Call External Procedure with Descriptor	—	—

```
sp := sp - 2
PUSHW(mod)
PUSHD(return address)
mod := WORD(descriptor)
sb  := DOUBLEWORD(mod+0)
pc  := DOUBLEWORD(mod+8)
      + WORD(descriptor+2)
```

SYNTAX		OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
rxp	*constant* *disp*	Return from External Procedure	—	—

```
POPD(pc)
POPW(mod)
sb  := DOUBLEWORD(mod+0)
sp  := sp + constant + 2
```

Service Return

SYNTAX		OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
rett	*constant* *disp*	Return from Trap §	all	ILL

```
if u=1, then TRAP(ILL)
else
  POPD(pc)
  POPW(mod)
  POPW(psr)
  sb := DOUBLEWORD(mod)
  sp := sp + constant
```

SYNTAX		OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
reti		Return from Interrupt §	all	ILL

```
if u=1, then TRAP(ILL)
else,
  POPD(pc)
  POPW(mod)
  POPW(psr)
  sb := DOUBLEWORD(mod)
```

5.13 PROCESSOR SERVICE INSTRUCTIONS

SYNTAX			OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
Register/Stack Manipulation					
adjsp*i*	*src* read.*i*		Adjust Stack Pointer sp := sp - *src* (*src* is signed) S bit specifies current sp	—	—
bicpsr{b | w}	*src* read.{b | w}		Bit Clear in psr § if w *length* if bicpsrw and u=1, then TRAP(ILL) else, psr := psr AND NOT(*src*)	all	ILL
bispsr{b | w}	*src* read.{b | w}		Bit Set in psr § if w *length* if bispsrw and u=1, then TRAP(ILL) else, psr := psr OR *src*	all	ILL
save	*reglist* imm reglist		Save General Purpose Registers for each register *rn* in *reglist*, PUSHD(*rn*) in numerical order.	—	—
restore	*reglist* imm reglist		Restore General Purpose Registers for each register *rn* in *reglist*, POPD(*rn*) in reverse numerical order.	—	—
enter	*reglist*, imm reglist,	*constant* disp disp	Enter New Context PUSHD(fp) fp := sp sp := sp - *constant* for each register *rn* in *reglist*, PUSHD(*rn*) in numerical order.	—	—
exit	*reglist* imm reglist		Exit Context for each register *rn* in *reglist*, POPD(*rn*) in reverse numerical order. sp := fp POPD(fp)	—	—
lpr*i*	*procreg*, short procreg	*src* read.*i*	Load Processor Register § if psr or intbase if u=1 and ((*procreg*=psr) or (*procreg*=intbase)) then TRAP(ILL) else *procreg* := *src* (S bit specifies current sp) (all flags affected if *procreg* = psr or upsr)	all	ILL

SYNTAX			OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
Register/Stack Manipulation					
spri	*procreg*, short procreg	*dest* write.*i*	Store Processor Register § if psr or intbase if u=1 and ((*procreg*=psr) or (*procreg*=intbase)) then TRAP(ILL) else *dest* := *procreg* (s bit specifies current sp)	—	ILL
setcfg	*cfglist* short cfglist		Set Configuration Register § if u=1, then TRAP(ILL) else cfg := short	—	—
Exceptions					
bpt			Breakpoint Trap TRAP(BPT)	—	BPT
svc			Supervisor Call Trap TRAP(SVC)	—	SVC
flag			Flag Trap if f=1, then TRAP(FLG)	—	FLG
Miscellaneous					
nop			No Operation pc := pc + 1	—	—
wait			Wait for Interrupt pc := pc + 1 Wait until next interrupt	—	—
dia			pc := pc cycle infinite loop until an interrupt occurs	—	—

5.14 MEMORY MANAGEMENT INSTRUCTIONS

	SYNTAX		OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
lmr	*mmureg*, short mmureg	*src* read.d	Load MMU Register *mmureg* := *src*	—	UND* ILL**
smr	*mmureg*, short mmureg	*dest* write.d	Store MMU Register *dest* := *mmureg*	—	UND* ILL**
rdval	*src* addr		Validate Address for Reading § if ADDRESS(*src*) in User mode may be read, f := 0 else f := 1	f	UND* ILL** ABT***
wrval	*dest* addr		Validate Address for Writing § if ADDRESS(*dest*) in User mode may be written to, then f := 0 else f := 1	f	UND* ILL** ABT***
movsu*i*	*src*, addr	*dest* addr	Move Value from Supervisor to User Space § *dest* := *src* (*src* is in supervisor space; *dest* in user space)	—	UND* ILL**
movus*i*	*src*, addr	*dest* addr	Move Value from User to Supervisor Space § *dest* := *src* (*src* is in user space; *dest* in supervisor space)	—	UND* ILL**

* TRAP(UND) if m bit in *cfg* is 0.

** TRAP(ILL) if u flag in *psr* is 1.

*** TRAP(ABT) if level 1 page table address invalid.

5.15 NS32081 FLOATING-POINT INSTRUCTIONS

SYNTAX			OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
mov*f*	*src*, read.*f*	*dest* write.*f*	Move Floating-Point *dest* := *src*	—	UND*
movlf	*src*, read.l	*dest* write.f	Move Long Floating to Floating *dest*.f := *src*.l	— fsr:tt uf if	UND* FPU
movfl	*src*, read.f	*dest* write.l	Move Floating to Long Floating *dest*.l := *src*.f	— fsr:tt	UND* FPU
mov*if*	*src*, read.*i*	*dest* write.*f*	Move Integer to Floating-Point *dest.f* := *src.i*	— fsr:tt if	UND* FPU
round*fi*	*src*, read.*f*	*dest* write.*i*	Round Floating-Point to Integer (round to even) *dest.i* := *src.f* If overflow, then TRAP(FPU) (*src.f* rounded to nearest integer, or to nearest even integer if a tie)	— fsr:tt if	UND* FPU
trunc*fi*	*src*, read.*f*	*dest* write.*i*	Truncate Floating-Point to Integer *dest.i* := *src.f* if overflow, then TRAP(FPU) (*src.f* rounded toward zero)	— fsr:tt if	UND* FPU
floor*fi*	*src*, read.*f*	*dest* write.*i*	Floor Floating-Point to Integer *dest.i* := *src.f* if overflow, then TRAP(FPU) (round *src.f* toward negative infinity)	— fsr:tt if	UND* FPU
add*f*	*src*, read.*f*	*dest* rmw.*f*	Add Floating-Point *dest* := *dest* + *src*	— fsr:tt uf if	UND* FPU
sub*f*	*src*, read.*f*	*dest* rmw.*f*	Subtract Floating-Point *dest* := *dest* - *src*	— fsr:tt uf if	UND* FPU

NS32081 FLOATING-POINT INSTRUCTIONS (Cont)

	SYNTAX		**OPERATIONS**	**FLAGS AFFECTED**	**TRAPS TAKEN**
mul*f*	*src*, read.*f*	*dest* rmw.*f*	Multiply Floating-Point *dest* := *dest* * *src*	— fsr:tt uf if	UND* FPU
div*f*	*src*, read.*f*	*dest* rmw.*f*	Divide Floating-Point if *src*=0, then TRAP(FPU) else *dest* := *dest* / *src*	— fsr:tt uf if	UND* FPU
cmp*f*	*src1*, read.*f*	*src2* read.*f*	Compare Floating-Point z := 1 if *src2*=*src1*; else z := 0 n := 1 if *src2*<*src1*; else n := 0 l := 0 (always)	z n l fsr:tt	UND* FPU
neg*f*	*src*, read.*f*	*dest* write.*f*	Negate Floating-Point *dest* := 0 - *src* (*src* sign bit complemented)	— fsr:tt	UND* FPU
abs*f*	*src*, read.*f*	*dest* write.*f*	Absolute Value of Floating-Point if *src*< 0, *dest* := 0 - *src* if *src*>=0, *dest* := *src*	— fsr:tt	UND* FPU
lfsr	*src* read.d		Load fsr fsr := *src*	— fsr:all	UND*
sfsr	*dest* write.d		Store fsr *dest* := fsr	—	UND*

* TRAP(UND) if f bit in *cfg* is 0.

5.16 NS32181and NS32381 FLOATING-POINT INSTRUCTIONS

	SYNTAX		OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
mov*f*	*src*, read.*f*	*dest* write.*f*	Move Floating-Point *dest* := *src*	—	UND*
movlf	*src*, read.l	*dest* write.f	Move Long Floating to Floating *dest*.f := *src*.l	— fsr:tt uf if	UND* FPU
movfl	*src*, read.f	*dest* write.l	Move Floating to Long Floating *dest*.l := *src*.f	— fsr:tt	UND* FPU
mov*if*	*src*, read.*i*	*dest* write.*f*	Move Integer to Floating-Point *dest.f* := *src.i*	— fsr:tt if	UND* FPU
round*fi*	*src*, read.*f*	*dest* write.*i*	Round Floating-Point to Integer (round to even) *dest.i* := *src.f* If overflow, then TRAP(FPU) (*src.f* rounded to nearest integer, or to nearest even integer if a tie)	— fsr:tt if	UND* FPU
trunc*fi*	*src*, read.*f*	*dest* write.*i*	Truncate Floating-Point to Integer *dest.i* := *src.f* if overflow, then TRAP(FPU) (*src.f* rounded toward zero)	— fsr:tt if	UND* FPU
floor*fi*	*src*, read.*f*	*dest* write.*i*	Floor Floating-Point to Integer *dest.i* := *src.f* if overflow, then TRAP(FPU) (round *src.f* toward negative infinity)	— fsr:tt if	UND* FPU

NS32181 and NS32381 FLOATING-POINT INSTRUCTIONS(Cont)

	SYNTAX		OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
add*f*	*src*, read.*f*	*dest* rmw.*f*	Add Floating-Point *dest* := *dest* + *src*	— fsr:tt uf if	UND* FPU
sub*f*	*src*, read.*f*	*dest* rmw.*f*	Subtract Floating-Point *dest* := *dest* - *src*	— fsr:tt uf if	UND* FPU
mul*f*	*src*, read.*f*	*dest* rmw.*f*	Multiply Floating-Point *dest* := *dest* * *src*	— fsr:tt uf if	UND* FPU
div*f*	*src*, read.*f*	*dest* rmw.*f*	Divide Floating-Point if *src*=0, then TRAP(FPU) else *dest* := *dest* / *src*	— fsr:tt uf if	UND* FPU
cmp*f*	*src1*, read.*f*	*src2* read.*f*	Compare Floating-Point z := 1 if *src2*=*src1*; else z := 0 n := 1 if *src2*<*src1*; else n := 0 l := 0 (always)	z n l fsr:tt	UND* FPU
neg*f*	*src*, read.*f*	*dest* write.*f*	Negate Floating-Point *dest* := 0 - *src* (*src* sign bit complemented)	— fsr:tt	UND* FPU
abs*f*	*src*, read.*f*	*dest* write.*f*	Absolute Value of Floating-Point if *src*<0, *dest* := 0 - *src* if *src*>=0, *dest* := *src*	— fsr:tt	UND* FPU
lfsr	*src* read.d		Load fsr fsr := *src*	— fsr:all	UND*
sfsr	*dest* write.d		Store fsr *dest* := fsr	—	UND*

SYNTAX			**OPERATIONS**	**FLAGS AFFECTED**	**TRAPS TAKEN**
scalb*f*	*src,*	*dest*		—	UND
	read.*f*	rmw.*f*	*dest* := *dest* * 2^{src} *src* = *integer*	fsr:all	FPU
logb*f*	*scr,*	*dest*		—	UND
	read.*f*	write.*f*	*dest* := unbiased exponent of *src*	fsr:all	FPU
dot*f*	*src,*	dest	Scalar Product	—	UND
	read.*f*	read.*f*	f0 := (*src* * *dest*) + f0	fsr:all	FPU
poly*f*	*src,*	dest	Polynomial Step	—	UND
	read.*f*	read.*f*	f0 := (f0 * *src*) + *dest*	fsr:all	FPU

* TRAP(UND) if f bit in *cfg* is 0.

5.17 NS32580 FLOATING-POINT INSTRUCTIONS

	SYNTAX		OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
mov*f*	*src*, read.*f*	*dest* write.*f*	Move Floating-Point *dest* := *src*	—	UND*
movlf	*src*, read.l	*dest* write.f	Move Long Floating to Floating *dest*.f := *src*.l	— fsr:tt uf if	UND* FPU
movfl	*src*, read.f	*dest* write.l	Move Floating to Long Floating *dest*.l := *src*.f	— fsr:tt	UND* FPU
mov*if*	*src*, read.*i*	*dest* write.*f*	Move Integer to Floating-Point *dest.f* := *src.i*	— fsr:tt if	UND* FPU
round*fi*	*src*, read.*f*	*dest* write.*i*	Round Floating-Point to Integer (round to even) *dest.i* := *src.f* If overflow, then TRAP(FPU) (*src.f* rounded to nearest integer, or to nearest even integer if a tie)	— fsr:tt if	UND* FPU
trunc*fi*	*src*, read.*f*	*dest* write.*i*	Truncate Floating-Point to Integer *dest.i* := *src.f* if overflow, then TRAP(FPU) (*src.f* rounded toward zero)	— fsr:tt if	UND* FPU
floor*fi*	*src*, read.*f*	*dest* write.*i*	Floor Floating-Point to Integer *dest.i* := *src.f* if overflow, then TRAP(FPU) (round *src.f* toward negative infinity)	— fsr:tt if	UND* FPU

	SYNTAX		OPERATIONS	FLAGS AFFECTED	TRAPS TAKEN
add*f*	*src*, read.*f*	*dest* rmw.*f*	Add Floating-Point *dest* := *dest* + *src*	— fsr:tt uf if	UND* FPU
sub*f*	*src*, read.*f*	*dest* rmw.*f*	Subtract Floating-Point *dest* := *dest* - *src*	— fsr:tt uf if	UND* FPU
mul*f*	*src*, read.*f*	*dest* rmw.*f*	Multiply Floating-Point *dest* := *dest* * *src*	— fsr:tt uf if	UND* FPU
div*f*	*src*, read.*f*	*dest* rmw.*f*	Divide Floating-Point if *src*=0, then TRAP(FPU) else *dest* := *dest* / *src*	— fsr:tt uf if	UND* FPU
cmp*f*	*src1*, read.*f*	*src2* read.*f*	Compare Floating-Point z := 1 if *src2*=*src1*; else z := 0 n := 1 if *src2*<*src1*; else n := 0 l := 0 (always)	z n l fsr:tt	UND* FPU
neg*f*	*src*, read.*f*	*dest* write.*f*	Negate Floating-Point *dest* := 0 - *src* (*src* sign bit complemented)	— fsr:tt	UND* FPU
abs*f*	*src*, read.*f*	*dest* write.*f*	Absolute Value of Floating-Point if *src*<0, *dest* := 0 - *src* if *src*>=0, *dest* := *src*	— fsr:tt	UND* FPU
lfsr	*src* read.d		Load fsr fsr := *src*	— fsr:all	UND*
sfsr	*dest* write.d		Store fsr *dest* := fsr	—	UND*
mac*f*	*src*, read.*f*	*dest* read.*f*	move(gen1*gen2)+l1/f1 to l1/f1 with two rounding errors		UND FPU
sqrt*f*	*scr*, read.*f*	*dest* write.*f*	move the square root if float to long Float		UND FPU

* TRAP(UND) if f bit in *cfg* is 0.

5.18 NS32532 INSTRUCTIONS

The NS32532 supports the full *Series 32000* instruction set, as described in Sections 5.2 through 5.13.

The NS32532 also contains four registers dedicated for debugging functions:

Debug Condition Register (dcr)
Debug Status Register (dsr)
Compare Address Register (car)
Breakpoint Program Counter (bpc)

These registers are accessed using privileged forms of the lpr and spr instructions.

The NS32532 supports a privileged Cache Invalidation (cinv instruction and privileged access to the following dedicated registers using the lpr and spr instructions: cfg, usp(sp1), dcr, dsr, car, and bpc.

	SYNTAX		OPERATION	FLAGS AFFECTED	TRAPS TAKEN
cinv	[*options*],*	*src* *gen* read.d	Cache Invalidated	—	ILL
lpr*i*	*procreg*,** *short*	*src* *gen* read.i	Load Processor Register	—	ILL
spr*i*	*procreg*,** *short*	*dest* *gen* write.i	Store Processor Register	—	ILL

* *options* are specified by listing the letters `a`, `i` or `d` separated by comma(s) within brackets.

** *procreg* can be the user stack pointer (usp or sp1), configuration register (cfg), and the debug registers in addition to the processor registers supported by the NS320*xx* and NS32332.

5.19 NS32CG16 and NS32CG160 INSTRUCTIONS

In addition to supporting the full *Series 32000* instruction set (as described in Sections 5.2 through 5.13), the NS32CG16 and NS32CG160 support the following instructions:

SYNTAX	OPERATIONS	FLAGS AFFECTED	TRAPS
bb{ and | or | xor | stod} [da | ia [,]] [s | -s]	Bit-aligned block transfer	—	—
bbfor	Bit-aligned block transfer	—	—
bitwit	Bit-aligned word transfer	—	—
extblt	External bit-aligned block transfer	—	—
movmp*i*	Move multiple pattern	—	—
tbits { 0 | 1 }	Test bit string	f,l,n,z	—
sbits	Set bit string	f	—
sbitps	Set bit string perpendicular	—	—

Note that the following instructions are not available in the NS32CG16 and NS32CG160:

Instruction	**Purpose**
lmr	Load MMU register
smr	Store MMU register
rdval	Validate address for reading
wrval	Validate address for writing
movsu*i*	Move value from supervisor to user space
movus*i*	Move value from user to supervisor space

For the NS32CG160, the setcfg instruction accepts the `de` (direct exception) configuration operand.

5.20 NS32GX32 and NS32GX320 INSTRUCTIONS

The NS32GX32 and the NS32GX320 support the same instruction set as the NS32532 (Section 5.18), except for the following instructions which are not supported:

Instruction	**Purpose**
lmr	Load MMU register
smr	Store MMU register
rdval	Validate address for reading
wrval	Validate address for writing
movsu*i*	Move value from supervisor to user space
movus*i*	Move value from user to supervisor space

In addition, the m configuration option of the setcfg instruction is not supported.

For the NS32GX320, four new DSP instructions are supported:

SYNTAX			**OPERATIONS**	**FLAGS** AFFECTED	**TRAPS** TAKEN
MULWD	*src*, read.*w*	*dest* rmw.*D*	Multiply word to double	—	—
MACTD	*src1*, read.*D*	*src2* read.*D*	Multiply and accumulate twice double if *src1* = (A2,A1), *src2* = (B2,B1) then R0 := R0 + A1*B1 + A2*B2	—	OVF
CMACD	*src1*, read.*D*	*src2* read.*D*	Complex multiply double and accumulate twice if *src1* = (A2,A1), *src2* = (B2,B1) then R0 := R0 + A1*B1 - A2*B2 and R1 := R1 + A1*B2 + A2*B1	—	OVF
CMULD	*src1*, read.*D*	*src2* read.*D*	Complex multiply double if *src1* = (A2,A1), *src2* = (B2,B1) then R0 := A1*B1 - A2*B2 and R1 := A1*B2 + A2*B1	—	OVF

5.21 NS32FX16 INSTRUCTIONS

In addition to supporting the full *Series 32000* instruction set (as described in Sections 5.2 through 5.14), the NS32FX16 supports the following instructions:

SYNTAX	OPERATIONS	FLAGS AFFECTED	TRAPS
bb{ and | or | xor | stod} [da | ia [,]] [s | -s]	Bit-aligned block transfer	—	—
bbfor	Bit-aligned block transfer	—	—
bitwit	Bit-aligned word transfer	—	—
extblt	External bit-aligned block transfer	—	—
movmp*i*	Move multiple pattern	—	—
tbits { 0 | 1 }	Test bit string	f,l,n,z	—
sbits	Set bit string	f	—
sbitps	Set bit string perpendicular	—	—

Note that the following instructions are not available in the NS32FX16:

Instruction	**Purpose**
lmr	Load MMU register
smr	Store MMU register
rdval	Validate address for reading
wrval	Validate address for writing
movsu*i*	Move value from supervisor to user space
movus*i*	Move value from user to supervisor space

Chapter 6

GNX ASSEMBLER DIRECTIVES

6.1 INTRODUCTION

Directives are commands to the assembler which allow the programmer to control the assembler in its generation of object code and production of listings.

The GNX Assembler directives are divided into functional groups as follows:

Directive	Function	Section
Symbol Creation	Assigns a name, type, and value to a symbol.	Section 6.2
Data Generation	Initializes a block of memory with constant values.	Section 6.3
Storage Allocation	Reserves a block of memory for data storage.	Section 6.4
Listing Control	Controls format of program listings.	Section 6.5
Linkage Control	Exports and imports data and procedures.	Section 6.6
Segment Control	Defines physical or logical image segments.	Section 6.7
Module Table	Manages the task of building a module table.	Section 6.8
Filename	Names the source file.	Section 6.9
Symbol Table	Specifies symbol table entry data.	Section 6.10
Line Number Table	Specifies a line number table entry.	Section 6.11
Macro Support	Provides macro and conditional assembly support.	Section 6.12
Procedure Support	Provides an easy method of writing assembly procedures.	Section 6.13

The remainder of this chapter will discuss these directives in detail.

6.2 SYMBOL CREATION DIRECTIVE

The symbol creation directive causes the assembler to compute the value of an expression and assign that value to a symbol name.

Directive	**Function**
`.set`	creates a symbol name

6.2.1 .set

Syntax:	`.set` *symbol, expression*	
where:	`.set`	is the directive name.
	symbol	is a symbol name. It consists of a series of characters. The characters may be letters, numbers, period (.), or underscore (_). The first character must not be a number.
	expression	is a constant or an expression. It may evaluate to any type.

Description: The `.set` directive causes the GNX Assembler to compute the value of the *expression* and assign this value to the symbol name. The *expression* may evaluate to any type except undefined, refer to Section 2.7. The *expression* may not be of type external (undefined), not forward reference.

For each symbol defined with the `.set` directive, the GNX Assembler enters the symbol name and value in its internal symbol table. The symbol may then be used in expressions in subsequent portions of the assembly.

Example:

```
1  .set  SYMBA,     5
2  .set  SYMBB,     LABELA + SYMBA
3  .set  SYMBC,     'A'
```

Line 1 defines the symbol SYMBA and assigns it the value 5.

Line 2 defines the symbol SYMBB and assigns it the value of LABELA+SYMBA. If SYMBA has the value 5, then SYMBB is assigned the value of LABELA+5 and the type of LABELA.

Line 3 defines the symbol SYMBC and assigns it the value of the 'A' expression. Note that only single character constants may be used in expressions (refer to Section 2.7.2).

6.3 DATA GENERATION DIRECTIVES

The data generation directives place constant data in the instruction stream during assembly-time. These are the following data generation directives:

Directive	**Function**
`.ascii`	assigns ASCII encoded textual data
`.byte`	assigns byte-long data
`.word`	assigns word-long data
`.double`	assigns double word-long data
`.float`	assigns single-precision floating-point number
`.long`	assigns double-precision floating-point number
`.field`	assigns bit field
`.xpd`	assigns external procedure descriptor
`.xdd`	assigns external data descriptor

Each of the above directives places one or more bytes of data in the object code of the program currently assembling. Data generation directives may be specified only in Program Code segments where data is written to the object file (*i.e.*, when the location counter is in the text segment, the data segment, the static segment, or the link segment).

All the numeric data generation directives, *i.e.*, all directives listed except `.field`, `.ascii`, `.xpd`, and `.xdd`, have the following form:

[*label*]*directive*({ [[*repetition-factor*]]*expression* | *string*}),,,

The *directive* stores the *expression* value in the instruction stream. If a *repetition-factor* is specified, the *directive* stores the *expression* value in consecutive locations as specified by the *repetition-factor*. A *label* is optional.

The `.byte`, `.word`, `.double`, `.float`, and `.long` directives may specify one or more *expressions*. Multiple *expressions* must be separated by commas. Each *expression* is evaluated and stored in the number of bytes specified by the directive. An *expression* must evaluate to an absolute value within the range specified by the directive, but *expressions* for the `.long` and `.float` directives should evaluate to a long value. (The assembler evaluates all floating-point expressions as long floating-point numbers. If necessary, the result is then converted to a single-precision floating-point value.) If no expression is specified, the GNX Assembler issues an error message and terminates code generation.

A *repetition-factor* may be any expression which evaluates to a positive absolute value. The *repetition-factor* expression may use symbolic values, but no forward symbol references are allowed.

Packed Decimal numbers may be generated using the `.byte`, `.word`, and `.double` directives. A Packed Decimal is created by specifying it as a hexadecimal constant. For example, **`.word H'1289`** creates the Packed Decimal number 1289.

The `.byte`, `.word`, and `.double` directives may be used for both signed and unsigned numbers.

6.3.1 .ascii

Syntax: [*label*] **`.ascii`** "*string*"

where:

label	is an optional label.
`.ascii`	is the directive name.
"string"	specifies a string constant. The string must not contain an embedded new-line. The user may use the escape sequence "\n" to enter a new-line into a string constant.

Description: The `.ascii` directive generates textual data. The GNX Assembler places the text in the instruction stream at the current address specified by the location counter. The assembler stores the ASCII value of each character in the *string* in one byte, placing the first character of the string at the lowest byte address and the last character of the string at the highest byte address. Unprintable ASCII characters may be included via the escapes defined in Section 2.4.3. No special string terminator is implied or inserted by the assembler.

Example:

```
1                           .data
2   D00000000   4572726f    .ascii "Error:  unknown command.\n"
                723a2075
                6e6b6e6f
                776e2063
                6f6d6d61
                6e642e0a
3   D00000018   55736167    .ascii "Usage:  list [-tdrx]"
                653a206c
                69737420
                5b2d7464
                72785d20
```

Line 2 places the ASCII character string "Error: unknown command." followed by a new-line character (\n) in consecutive bytes beginning at address 0 of the data segment.

Line 3 places the character string "Usage: list [–tdrx]" in consecutive bytes starting at address 00000018 in the data segment.

6.3.2 .byte

Syntax: [*label*] `.byte` ({ [[*repetition-factor*]] *expression* | *string*}),,,

where:

label	is an optional label.
`.byte`	is the directive name.
[*repetition-factor*]	(optional) specifies the number of occurrences of the specified data byte. It must be an expression which evaluates to a positive absolute value. If the *repetition-factor* is specified, it must be enclosed in "[ ]" brackets.
expression	specifies the data byte value. This value must be in the range of −128 to 255.
string	specifies a string constant. The assembler issues a warning if the string contains an embedded new-line. Therefore, it is preferable to use the "\n" escape sequence.

Description: The `.byte` directive generates one or more byte constants. The GNX Assembler places the constants in the instruction stream at the current address specified by the location counter. If multiple constants are specified (*e.g.*, *repetition-factor* is greater than one or more than one *expression* is given), the constants are stored in consecutive bytes beginning at the current address.

If a *string* is specified, the assembler places the *string*, starting with the first character in the string, in one or more bytes beginning at the current address. The assembler stores the ASCII value of each character in the *string* in one byte. Character constants appearing as terms in the *expression* are converted to integers (see Section 2.7.2, Rule 5).

Example:

```
1   T00000000  81        .byte 129
2   T00000001  03030303  .byte [5] 3
               03
3   T00000006  414243    .byte "ABC"
4   T00000009  034142    .byte 3,"AB"
5   T0000000c  2202      .byte ´f´/3,´f´/´3´
6   T0000000e  81        .byte −127
```

Line 1 places 81 in a byte at address 000000 of the text segment.

Line 2 places 3 (repeated 5 times) in five consecutive bytes starting at address 000001 in the text segment.

Line 3 places the ASCII values of “ABC” in three consecutive bytes starting at address 000006 in the text segment.

Line 4 places 3 in the byte at address 000009 in the text segment followed by the ASCII values of “AB” in two consecutive bytes.

Line 5 places the value of the expressions ´f´/3 and ´f´/´3´ in consecutive bytes beginning at address 00000C in the text segment. The value of ´f´/3 (0x22) is first, followed by the value of ´f´/´3´ (0x02).

Line 6 places 81 in a byte at address 00000E in the text segment.

6.3.3 .word

Syntax: [*label*] **`.word`** ({ [[*repetition-factor*]] *expression* | *string*}),,,

where:

label	is an optional label.
`.word`	is the directive name.
[*repetition-factor*]	(optional) specifies the number of occurrences of the specified data word. It must be an expression which evaluates to a positive absolute value. If the *repetition-factor* is specified, it must be enclosed in "[ ]" brackets.
expression	specifies the data word value. It must evaluate to an absolute value within the range of –32768 to 65535.
string	specifies a string constant. If the string is not composed of an even multiple of two characters, it is null padded by the appropriate amount.

Description: The `.word` directive generates one or more word length constants. The assembler places the constants in the instruction stream at the current address specified by the location counter. The assembler stores the least-significant byte at the lower address and the most-significant byte at the higher address.

If multiple constants are specified (*e.g.*, *repetition-factor* is greater than one, or more than one *expression* is given), the constants are stored in consecutive words beginning at the current address.

When a string is specified as an operand of the `.word` directive, it is output as a byte string beginning at the lowest address and padded at the high address to an even multiple of two bytes if necessary.

Example:

```
1  T0000000  0180      .word 32769
2  T0000002  34123412  .word [2] 0x1234
3  T0000006  41004142  .word ´A´, "AB"
4  T000000a  0100      .word 0x41424344/0x41424344
5  T000000c  0180      .word -32767
```

Line 1 places the constant 32769 in a word at the address 000000 in the text segment.

Line 2 places the constant 0x1234 (repeated twice) in two consecutive words.

Line 3 places the word values of the character constant ´A´ and the string “AB” (evaluated as integers) in two consecutive words.

Line 4 places the value of the expression 0x41424344/0x41424344 in a word at the address 00000A in the text segment.

Line 5 places 0x8001 (–32767) in a word at address 00000C in the text segment.

6.3.4 .double

Syntax: [*label*] **`.double`** ({[[*repetition-factor*]] *expression* | *string*}),,,

where:

label	is an optional label.
`.double`	is the directive name.
[*repetition-factor*]	(optional) specifies the number of occurrences of the specified double-word. It must be an expression which evaluates to a positive absolute value. If the *repetition-factor* is specified, it must be enclosed in "[ ]" brackets.
expression	specifies the double-word value. It must evaluate to an absolute value within the range of -2^{31} to $2^{31}-1$.
string	specifies a string constant. If the string is not composed of an even multiple of four characters, it is null padded by the appropriate amount.

Description: The `.double` directive generates one or more double-word constants. The assembler places the constants in the instruction stream at the current address specified by the location counter. The assembler places the bytes in ascending order, beginning with the least-significant byte at the lowest address.

If multiple constants are specified (*e.g.*, *repetition-factor* is greater than one, or more than one *expression* is given), the constants are stored in consecutive double-words, beginning at the current address. When a string is specified as an operand of the `.double` directive, it is output as a byte string, beginning at the lowest address and padded at the high address to an even multiple of four bytes if necessary.

Example:

```
1  T0000000  ffff0000  .double 0x0000FFFF, 0xFFFF0000
             0000ffff
2  T0000008  03000000  .double [2] 3
             03000000
3  T0000010  41424300  .double ''ABC''
4  T0000014  01000000  .double 0x41424344/0x41424344
5  T0000018  70ffffff  .double -144, 257
             01010000
```

Line 1 places the constants 0x0000ffff and 0xffff0000 in two consecutive double-words.

Line 2 places the constant 3 (repeated twice) in two consecutive double-words.

Line 3 places the value of the string “ABC” in a double-word.

Line 4 places the value of the expression 0x41424344/0x41424344 in a double-word at address 00000014 in the text segment.

Line 5 places the value of the signed constants –144 and 257 in consecutive double-words.

6.3.5 .float

Syntax: [*label*] `.float` ([[*repetition-factor*]] *expression*),,,

where:

label	is an optional label.
`.float`	is the directive name.
[*repetition-factor*]	(optional) specifies the number of occurrences of the specified floating-point number. It must be an expression which evaluates to a positive absolute value. If the *repetition-factor* is specified, it must be enclosed in brackets.
expression	specifies a single-precision floating-point constant (refer to Section 2.4.2). Strings are not permitted.

Description: The `.float` directive generates one or more single-precision floating-point constants. The assembler places the constants in the instruction stream at the current address specified by the location counter. The assembler stores a single-precision floating-point constant in a double-word (32 bits).

If multiple constants are specified (*e.g.*, *repetition-factor* is greater than one, or more than one *expression* is given), the constants are stored in consecutive double-words beginning at the current address.

Example:

```
1  T0000000  4cdc3654  .float 3.14152E+12
2  T0000004  1ff47c3f  .float [2]0.9881
             1ff47c3f
```

Line 1 places the floating-point constant 3.14152E+12 in a double-word at the current address.

Line 2 places the floating-point constant 0.9881 (repeated twice) into two consecutive double-words.

6.3.6 .long

Syntax: [*label*] `.long` ([[*repetition-factor*]]*expression*),,,

where:

label	is an optional label.
`.long`	is the directive name.
[*repetition-factor*]	(optional) specifies the number of occurrences of the specified floating-point number. It must be an expression which evaluates to a positive absolute value. If the *repetition-factor* is specified, it must be enclosed in "[ ]" brackets.
expression	specifies a double-precision floating-point constant (refer to Section 2.4.2). Strings are not permitted.

Description: The `.long` directive generates one, or more double-precision floating-point constants. The assembler places the constants in the instruction stream at the current address specified by the location counter. The assembler stores a double-precision floating-point constant in a quad-word (64 bits).

If multiple constants are specified (*e.g.*, *repetition-factor* is greater than one, or more than one *expression* is given), the constants are stored in consecutive quad-words beginning at the current address.

Example:

```
1  T0000000  00002078  .long 3.14152E+12
             89db8642
2  T0000008  3695efe2  .long 6.12E-23, [3] 0.9881
             1e7f523b
             e6ae25e4
             839eef3f
             e6ae25e4
             839eef3f
             e6ae25e4
             839eef3f
```

Line 1 places the floating-point constant 3.14152E+12 in a quad-word at the current address.

Line 2 places the floating-point constants 6.12E–23 and 0.9881 (repeated three times) in four consecutive quad-words.

6.3.7 .field

Syntax: [*label*] **`.field`** ([*subfield-size*]*subfield-value*),,,

where:

label	is an optional label.
`.field`	is the directive name.
[*subfield-size*]	(required) specifies the length in bits of the field being generated. It may be any expression which evaluates to a positive absolute value. No forward referencing of symbols is permitted. The *subfield-size* must be enclosed in "[ ]" brackets.
subfield-value	(required) specifies a field value. It may be any expression which evaluates to a non-negative absolute value. It must be within the range specified by the field size (*e.g.*, 0 to 15 for a 4-bit field, 0 to 31 for a 5-bit field).

Description: The `.field` directive generates one or more bit fields. The assembler places the field(s) in the instruction stream at the current address specified by the location counter. The directive provides no default values; thus, both *subfield-size* and *subfield-value* must be specified.

If the directive specifies more than one *subfield-size*/*subfield-value* pair, the values are placed in contiguous fields. If a field or a combination of fields do not extend to a byte boundary, the assembler zero-fills the remaining bits.

If multiple constants are specified, the *subfield-size*/*subfield-value* pairs must be separated by commas. See lines 2 and 3 in the following example.

Example:

```
1  T0000000  08     .field [4] 8
2  T0000001  3f     .field [4] 15, [4] 3
3  T0000002  2143   .field [4] 1, [4] 2, [4] 3, [4] 4
```

Line 1 places 8 in a 4-bit field at address 0000000 in the text segment and zero-fills the four high-order bits.

Line 2 places 15 and 3 in two consecutive 4-bit fields at address 0000001 in the text segment.

Line 3 places 1, 2, 3, and 4 in four consecutive 4-bit fields. The fields occupy two bytes beginning at address 0000002 in the text segment.

6.3.8 .xpd

Syntax: [*label*] `.xpd` *expression*

where:

label	is an optional label.
`.xpd`	is the directive name.
expression	specifies the entry point of a function using the `cxp` `rxp` calling discipline.

Description: The `.xpd` directive generates an external procedure descriptor for the specified entry point. A procedure descriptor is a double-word of data. The low-order two bytes specify the module table entry for the module that contains the function. The high-order two bytes contain the offset of the function entry point from the module's program code base. The module entry and offset values are updated by the linker at link time.

The assembler generates the procedure descriptor at the current location. Normally, the current location will be in the link table (link segment). However, the `.xpd` directive may also be used to put a procedure descriptor for a function at a known memory location for use with the cxpd instruction.

The definition of an xpd symbol through the `.xpd` directive should precede any reference to it.

Example:

```
1                          .link
2  L0000000   00000000   .xpd _main
3  L0000004   00000000   .xpd _fun1
4  L0000008   00000000   .xpd _fun2
```

Line 2 generates a procedure descriptor link table entry for the function _main at address 00000000 in the link segment.

Line 3 generates a procedure descriptor for the function _fun1 at address 00000004 in the link segment.

Line 4 generates a procedure descriptor for the function _fun2 at address 00000008 in the link segment.

Both sample functions are external to the assembly. The actual module table entry and code offset will be filled in by the linker.

6.3.9 .xdd

Syntax:	**.xdd** *expression*	
where:	`.xdd`	is the directive name.
	expression	specifies a double-word value.

Description: The `.xdd` directive or external data descriptor, defines link table entries for external data variables.

If *expression* is specified by a single symbol of non-absolute type, references to this symbol can be addressed via the external addressing mode. If the external data variable is defined by:

offset: `.xdd` *value*

the syntax for addressing the symbol *value* via the external addressing mode is either by *value*, which is equivalent to *offset*(ext), or by *disp* (*value*) and *value+disp*, which is equivalent to *disp*(*offset*(ext)), where *disp* is an expression of absolute type.

If *expression* is specified by a combination of symbols (*e.g.*, sym+var), or a combination of symbols and constants (*e.g.*, value+10), references to these symbols (*i.e.*, sym, var, value) will be addressed via their appropriate default addressing mode, (*i.e.*, absolute addressing mode if symbol in link segment).

The assembler generates the data descriptor at the current location. Normally, the current location will be in the link table, that is, the link segment. However, the `.xdd` directive may also be used to put a data descriptor at any other segment.

The definition of an xdd symbol through the `.xdd` directive should precede any reference to it.

Example:

```
1                                    .data
2  D0000000   00000000   foo1:      .blkd
3
4                                    .link
5  L0000000   00000000   ext_var: .xdd foo1
```

Line 5 generates an external data variable link table entry for the variable foo at address 00000000 in the link segment.

6.4 STORAGE ALLOCATION DIRECTIVES

There are six storage allocation directives:

Directive	Function
`.blkb`	allocates byte storage
`.blkw`	allocates word storage
`.blkd`	allocates double-word storage
`.blkf`	allocates double-word(s) for floating-point storage
`.blkl`	allocates quad-word(s)for long floating-point storage
`.space`	allocates a block of storage

All storage allocation directives except `.space` have the following form:

[*label*] *directive* [*expression*]

The optional *expression* specifies the number of bytes, words, double-words, or quad-words to be allocated. It must evaluate to a non-negative absolute value. If the *expression* evaluates to zero, no storage is allocated. If no *expression* is specified, the default value is one. The *expression* may use symbolic values, but no forward symbol references are allowed.

When storage allocation directives occur in the text segment, the allocated bytes, words, double-words, or quad-words allocated are initialized to the nop instruction and appear in the program listing as generated code. When storage allocation directives occur in the data, static, or link segments, the allocated bytes, words, double-words, or quad-words are initialized to zero and appear in the program listing as generated code. For all other segment types, the allocated space is uninitialized.

Sections 6.4.1 through 6.4.6 define the syntax of these directives.

6.4.1 .blkb

Syntax: [*label*] **`.blkb`** [*expression*]

where:

label	is an optional label.
`.blkb`	is the directive name.
expression	specifies the number of bytes to be allocated. It must be an unsigned integer constant or an expression which evaluates to a non-negative absolute value. The default value is one.

Description: The `.blkb` directive allocates zero or more consecutive bytes of memory for data storage. The bytes begin at the current location counter address.

Example:

```
1                                  .static
2  S00000000  00                   .blkb 1
3  S00000001  00000000  AA:        .blkb 15
              00000000
              00000000
              000000
4  S00000010  00000000             .blkb (.-AA)/3
              00
5  S00000015  00                   .blkb
```

Line 2 allocates a single byte for data storage. The byte is located at address 00000000 in the static segment.

Line 3 allocates 15 consecutive bytes for data storage, beginning at address 00000001 in the static segment. The label AA is assigned the address of the first byte.

Line 4 allocates the number of bytes specified by the "(.–AA)/3" expression. The expression evaluates to 5, *i.e.*, (16 (static relative) – 1 (static relative)) = 15 (absolute), 15/3 = 5. Therefore, 5 bytes are allocated, beginning at address 00000010 static segment relative.

Line 5 allocates a single byte for storage.

6.4.2 .blkw

Syntax: [*label*] `.blkw` [*expression*]

where:

label	is an optional label.
`.blkw`	is the directive name.
expression	specifies the number of words to be allocated. It must be an unsigned integer constant or an expression which evaluates to a non-negative absolute value.

Description: The `.blkw` directive allocates zero or more consecutive words of memory for data storage. The words begin at the current location counter address.

Example:

```
1                                   .text
2  T00000000  a2a2                  .blkw 1
3  T00000002  a2a2a2a2    AA:       .blkw 15
              a2a2a2a2
              a2a2a2a2
              a2a2a2a2
              a2a2a2a2
              a2a2a2a2
              a2a2a2a2
              a2a2
4  T00000020  a2a2a2a2              .blkw (.-AA)/3
              a2a2a2a2
              a2a2a2a2
              a2a2a2a2
              a2a2a2a2
5  T00000034  a2a2                  .blkw
```

Line 2 allocates one word for data storage at address 00000000 in the text segment.

Line 3 allocates 15 consecutive words for data storage, beginning at address 00000002 in the text segment. The label AA is assigned the address of the first word.

Line 4 allocates the number of words specified by the "(.–AA)/3" expression. The expression evaluates to 10, *i.e.*, (32 (text relative) – 2 (text segment relative)) = 30 (absolute), 30/3 = 10. Therefore, 10 words are allocated, beginning at address 00000020 in the text segment.

Line 5 allocates one word for storage.

6.4.3 .blkd

Syntax: [*label*] **`.blkd`** [*expression*]

where:

label	is an optional label.
`.blkd`	is the directive name.
expression	specifies the number of double-words to be allocated. It must be an unsigned integer constant or an expression which evaluates to a non-negative absolute value.

Description: The `.blkd` directive allocates zero or more consecutive double-words of memory for data storage. The double-words begin at the current location counter address.

Example:

```
1               .text
2   text_start:
3               .dsect lo_text, text_start
4               .blkd 1
5   AA:         .blkd 15
6               .blkd (.-AA)/3
7               .blkd
```

Line 4 allocates one double-word for data storage, overlaid onto address 000000 of the text segment.

Line 5 allocates 15 consecutive double-words for data storage, overlaid onto address 000004 of the text segment. The label AA is assigned the address of the first double-word.

Line 6 allocates the number of double-words specified by the "(.–AA)/3" expression. The expression evaluates to 20, *i.e.*, (64 (text relative) – 4 (text relative)) = 60 (absolute), 60/3 = 20. Therefore, 20 double-words are allocated and overlaid onto address 000040 of the text segment.

Line 7 allocates a single double-word for storage.

6.4.4 .blkf

Syntax: [*label*] **.blkf** [*expression*]

where:	*label*	is an optional label.
	`.blkf`	is the directive name.
	expression	specifies the number of double-words to be allocated. It must be an unsigned integer constant or an expression which evaluates to a non-negative absolute value.

Description: The `.blkf` directive allocates zero or more consecutive double-words of memory for storage of single-precision floating-point (32-bit) numbers. The double-words begin at the current location counter address.

Example:

```
1              .udata
2              .blkf 1
3      AA:     .blkf 15
4              .blkf
```

Line 2 allocates one double-word for data storage at the address of the bss segment.

Line 3 allocates 15 consecutive double-words for data storage, beginning at the current address of the bss segment. The label AA is assigned the address of the first double-word.

Line 4 allocates one double-word for storage at the address of the bss segment.

6.4.5 .blkl

Syntax: [*label*] **`.blkl`** [*expression*]

where:

label	is an optional label.
`.blkl`	is the directive name.
expression	specifies the number of quad-words to be allocated. It must be an unsigned integer constant or an expression which evaluates to a non-negative absolute value.

Description: The `.blkl` directive allocates zero or more consecutive quad-words of memory for storage of double-precision floating-point (64-bit) numbers. The quad-words begin at the current location counter address.

Example:

```
1           .udata
2           .blkl 1
3     AA:   .blkl 15
4           .blkl
```

Line 2 allocates one quad-word for data storage at address 00000000 of the bss segment.

Line 3 allocates 15 consecutive quad-words for data storage, beginning at address 00000008 of the bss segment. The label AA is assigned the address of the first quad-word.

Line 4 allocates a single quad-word for storage at 00000128 of the bss segment.

6.4.6 .space

Syntax: [*label*] **`.space`** *expression*

where:

label	is an optional label.
`.space`	is the directive name.
expression	specifies the number of bytes to be allocated. It must be an unsigned integer constant or an expression which evaluates to a non-negative absolute value.

Description: The `.space` directive allocates a consecutive block of memory for data storage. The block begins at the current location counter address. The size in bytes of the storage block is specified by *expression.*

Example:

```
1                                 .static
2  S00000000  00                  .space 1
3  S00000001  00000000  AA:       .space 15
              00000000
              00000000
              000000
4  S00000010  00000000            .space (.-AA)/3
              00
5  S00000015  00                  .space 1
```

Line 2 allocates one byte for data storage. The byte is located at address 00000000 in the static segment.

Line 3 allocates 15 consecutive bytes for data storage, beginning at address 00000001 in the static segment. The label AA is assigned the address of the first byte.

Line 4 allocates the number of bytes specified by the "(.–AA)/3" expression. The expression evaluates to 5, *i.e.*, (16 (static relative) – 1 (static relative)) = 15 (absolute), 15/3 = 5. Therefore, five bytes are allocated, beginning at address 00000010 static segment relative.

Line 5 allocates a single byte for storage.

6.5 LISTING CONTROL DIRECTIVES

The listing control directives control the format of the GNX Assembler's program listing:

Directive	**Function**
`.title`	prints title at top of program listing
`.subtitle`	prints subtitle at top of program listing
`.nolist`	suppresses the printing of lines of source program to listing
`.list`	restores printing of lines of source program to listing
`.eject`	continues listing at top of next page
`.width`	sets width of listing page

Sections 6.5.1 through 6.5.6 describe the listing control directives in detail.

6.5.1 .title

Syntax: [*label*] **`.title`** "*string*"

where:	*label*	is an optional label.
	`.title`	is the directive name.
	string	specifies the character string to be printed at the top of the listing page. The string (required) may consist of any combination of up to 126 letters, numbers, and text characters and must be enclosed in double-quotes.

Description: The `.title` directive causes the assembler to print the specified *string* at the top of each new page of the program listing.

The first `.title` directive affects the current listing page as well as all previous pages.

If a program contains more than one `.title` directive, the last `.title` directive to be specified before the page break affects subsequent pages. If a page other than the first page has no `.title` directive, it receives the title of the previous page.

If a program contains no `.title` directive, no title is printed.

No title is printed on the cross-reference page.

Example:

```
.title  "John´s Program"
```

The preceding example causes the string "John's Program" to be printed at the top of the current page of the program listing. If it is the only `.title` directive in the program, all pages will have the same title.

6.5.2 .subtitle

Syntax: [*label*] **`.subtitle`** "*string*"

where:		
	label	is an optional label.
	`.subtitle`	is the directive name.
	string	specifies the character string to be printed at the top of listing page. The string (required) may consist of any combination of up to 126 letters, numbers, and text characters and must be enclosed in double-quotes.

Description: The `.subtitle` directive causes the assembler to print the specified *string* at the top of each new page of the program listing. If a `.title` directive is also specified, the subtitle *string* appears below the title *string*.

The first `.subtitle` directive affects the current listing page as well as all previous pages.

If a program contains more than one `.subtitle` directive, the last `.subtitle` directive to be specified before page break affects the subsequent page. If a page has no `.subtitle` directive, it receives the subtitle of the previous page.

If a program contains no `.subtitle` directive, no subtitle is printed.

No subtitle is printed on the cross-reference listing page.

Example:

```
.subtitle  "Written 7/7/81"
```

The preceding example causes the string "Written 7/7/81" to be printed at the top of the current page of the program listing. If it is the only `.subtitle` directive in the program, all pages will have the same subtitle.

6.5.3 .nolist

Syntax:	[*label*] **`.nolist`** [*qualifier_list*]	
where:	*label*	is an optional label.
	`.nolist`	is the directive name.
	qualifier_list	macro listing qualifiers to be set off. Can be any combination of the qualifiers: `mac_source`, `mac_expansions` and `mac_directives`, as described in Section 8.14.

Description: The `.nolist` directive suppresses the printing of source program lines. All lines following the `.nolist` directive are assembled but are not printed to the program listing.

The `.nolist` directive does not affect the printing of error messages.

The `.nolist` directive may be disabled by specifying a `.list` directive (see Section 6.5.4).

Example:

```
.nolist
movd    r0, r1
addb    TEMP, r1
subb    r1, r0
.list
```

In the preceding example, the `.nolist` directive suppresses printing of the statement containing the `.nolist` directive and the following three lines of source. Printing is restored by the `.list` directive. Only the statement containing the `.list` directive is printed.

6.5.4 .list

Syntax:	[*label*] **`.list`** [*qualifier_list*]	
where:	*label*	is an optional label.
	`.list`	is the directive name.
	qualifier_list	macro listing qualifiers to be set on. Can be any combination of the qualifiers: `mac_source`, `mac_expansions` and `mac_directives`, as described in Section 8.14.

Description: The `.list` directive restores the printing of lines of the source program after suppression by a `.nolist` directive. All lines following the `.list` directive are printed to the program listing. The statement containing the `.list` directive is also printed to the program listing.

Example:

```
        .nolist
        movb    r0, r1
        addb    TEMP, r1
        subb    r1, r0
        .list
NXT:    cmpb    r1,r0
```

In this example, the `.list` directive restores printing after the previous `.nolist` directive. Only the statement labelled NXT: and the .list statements are printed.

6.5.5 .eject

Syntax: [*label*] **`.eject`**

where:		
	label	is an optional label.
	`.eject`	is the directive name.

Description: The `.eject` directive causes the program listing to continue at the top of the next page. The statement containing the `.eject` directive is printed in the program listing.

Example:

```
.eject
```

This example causes the program listing to continue at the top of the next page. The statement containing the `.eject` directive is printed.

6.5.6 .width

Syntax:	[*label*] **`.width`** *expression*	
where:	*label*	is an optional label.
	`.width`	is the directive name.
	expression	specifies page width in characters. It must be an unsigned integer constant or expression which evaluates to an absolute value within the range of 80 to 132.

Description: The `.width` directive sets the width (in characters) of the program listing lines which follow the directive. (The first `.width` directive effects all preceding pages as well.) More than one `.width` directive is allowed, with each directive effective until the next or until the end of the file. If there is no `.width` directive, the width is 132 characters by default. The new-line character is included in the maximum width.

If the *expression* value is outside the specified range, an error message is generated.

Example:

```
.width   MYPAGEWIDTH - 12
```

The preceding example sets the page width to the value of the expression MYPAGEWIDTH-12. The expression must evaluate to a number within the range 80 to 132.

6.6 LINKAGE CONTROL DIRECTIVES

The linkage control directives provide support for modular programming by allowing symbols and procedures to be exported from, or imported to, separately assembled modules. These directives are:

Directive	Function
`.globl`	declares external data symbols
`.comm`	declares external undefined data symbols

The `.globl` directive declares a symbol external, either for import or export, but does not define the symbol. The `.comm` directive is similar, except an associated size is specified. At link time, symbols declared with `.comm` are resolved and allocated in the bss segment.

Sections 6.6.1 through 6.6.2 describe the linkage control directives.

6.6.1 .globl

Syntax: **`.globl`** *symbol* ,,,

where:

`.globl`	is the directive name.
symbol	is the name of a symbol. If more than one *symbol* is specified, the symbols must be separated by commas.

Description: The `.globl` directive declares a symbol to be external, that is, a symbol intended to be used by multiple, separately assembled pieces of the same program. The `.globl` directive guarantees that a symbol table entry will be generated in the object file, marked external. The linker uses these entries to resolve external symbol references at link time. Symbols declared with the `.globl` directive may or may not be defined within the current assembly. Defined symbols that are not declared to be external are assumed to be local symbols and may not be used to resolve undefined external references at link time. Undefined symbols are assumed to be external, with or without declaration, but it is good practice to declare all external symbols.

An alternate way to declare external symbols is to replace the colon of the label definition with a double colon (::).

Example:

```
           .globl FIRST,  SECOND
FIRST:
SECOND:
THIRD::
```

This example defines and exports three symbols: FIRST, SECOND, THIRD.

NOTE: Because .globl symbols are used by the linker to resolve external symbol references at link time, the user is advised to declare all .globl symbols as either external procedure descriptors (through the `.xpd` directive) or external data descriptors (through the `.xdd` directive), when assembling the program module with the modularity flag (–X on UNIX/MS-DOS, or /MODULAR on VMS).

6.6.2 .comm

Syntax: `.comm` *symbol, expression*

where:

`.comm`	is the directive name.
symbol	is the name of a data symbol referenced, but not defined, in the current module.
expression	specifies the number of bytes allocated for the symbol. It may be any expression which evaluates to a positive absolute value.

Description: The `.comm` directive imports the specified symbol and assigns it an external undefined type. When the module is linked, the symbol will be placed in the .bss section.

Example:

```
1                          .comm   SYM1,16
2                          .comm   SYM2,4
3  T00000000   14a8c000   movb    SYM1,r0
               0000
4  T00000006   57a8c000   movd    SYM2,r1
               0000
```

NOTE: Because .comm symbols are used by the linker to resolve external symbol references at link time, the user is advised to declare all .comm symbols as either external procedure descriptors (through the `.xpd` directive) or external data descriptors (through the `.xdd` directive), when assembling the program module with the modularity flag (–X on UNIX/MS-DOS, or /MODULAR on VMS).

6.7 SEGMENT CONTROL DIRECTIVES

The segment control directives control the current segment type and the value of the assembler's location counter. These directives are:

Directive	Function
`.dsect`	sets the location counter to a user-defined segment
`.text`	sets the location counter to the text segment
`.data`	sets the location counter to the date segment
`.bss`	assigns space in the bss segment, updates the location counter
`.udata`	sets the location counter to the bss segment
`.static`	sets the location counter to the static segment
`.link`	sets the location counter to the link segment
`.section`	defines a section with attributes
`.org`	sets the location counter to specified value
`.align`	sets the location counter to specified offset
`.ident`	places the string argument in the .comment section of the object file

The segment control directives permit definition of program segments. A segment is a group of sequential statements whose addresses are all relative to the same base. Segments permit data or instructions to be processed as a unit and to be stored in a contiguous block within memory at run-time.

Sections 6.7.1 through 6.7.11 describe the syntax and operation of the segment control directives.

6.7.1 .dsect

Syntax: `.dsect` *symbol expression* [, *specifier*]

where:

`.dsect`	is the directive name.
symbol	specifies the name of the dummy section.
expression	specifies the value and type of the location counter for the segment. The expression is required the first time a named *dsect* is invoked. Subsequent `.dsect` directives using the same name may omit the expression.
specifier	is a plus sign (+) or a minus sign (–). *Specifier* indicates whether the location counter should be incremented or decremented.

Description: The `.dsect` directive defines a named, user-defined (or dummy) segment. A dummy segment is used to define symbols which may be used in expressions or as instruction operands to access data. No code or initialized data may be generated in a *dsect.*

The assembler assigns a location counter to the segment with the value and type specified by the expression. If the type of the expression is relative, for example text or data, the dummy segment may be thought of as an overlay of an existing memory segment. For example, a dummy segment might be used to define differing logical data structures that occupy the same storage space, as in a C union or a Pascal variant record.

An optional specifier may be used to indicate whether the location counter for the dummy segment will increment or decrement. If the optional specifier is omitted, the value of *expression* determines whether the location counter increments or decrements. If the value of the expression is negative, the assembler decrements the location counter. If the value of the expression is positive or zero, the assembler increments the location counter. In either case, labels are assigned the lowest byte address of the following statement. That is, the location counter is post-incremented and pre-decremented.

Example:

```
            .dsect DATE_REC, 0
MONTH:      .blkb
DAY:        .blkb
YEAR:       .blkw
```

This example defines three absolute symbols in a dummy segment named DATE_REC. The symbols have the absolute values of 0, 1, and 2.

The symbols can be used as offsets into any block of memory. In the example below, r0 contains the address of a block of memory for storing the data. The instructions in the example zero-fill the month, day, and year fields.

```
        .udata
DATE:   .blkb    4
        .text
        movd     DATE, r0
        movqb    0, MONTH(r0)
        movqb    0, DAY(r0)
        movqw    0, YEAR(r0)
```

6.7.2 .text

Syntax: **`.text`**

where: `.text` is the directive name.

Description: The `.text` directive indicates the beginning of a program text segment or code segment. The assembler assigns the current location counter the next available text segment address. Subsequent storage allocation, data generation, or program statements generate code and constant data that will be placed in the .text section of the object file. Storage allocated in the text segment is filled with nop instructions. The location counter is incremented after every assignment, storage allocation, or code generation.

Symbols defined in the text segment are of type text. The assembler uses the Program Counter (PC) Relative addressing mode for all symbols or expressions of type text. When the text segment is loaded into memory, it contains a module's instructions and constant data and is, therefore, protected for read-only access.

Example:

```
.text
```

In the preceding example, the location counter is set to text segment type. The offset is set to the next available offset. Instructions and data directives that follow the `.text` directive generate code in the .text section of the object file.

6.7.3 .data

Syntax: **`.data`**

where: `.data` is the directive name.

Description: The `.data` directive indicates the beginning of an initialized data segment. An initialized data segment contains writable data or program code and will be placed in the .data section of the object file. When the data segment is loaded into memory, it is protected for read-write access.

The `.data` directive sets the location counter to the next available data segment address. The location counter is incremented after every data assignment or code generation. Symbols defined in the data segment are of type data. The assembler uses the Absolute addressing mode for all symbols or expressions of type data.

Example:

```
.data
```

In the preceding example, the location counter is set to the data segment. The offset is set to the next available data segment address. Subsequent data directives, or instructions, are output to the .data section of the object file.

6.7.4 .bss

Syntax:	**.bss** *symbol, expression1, expression2*	
where:	`.bss`	is the directive name.
	symbol	is a symbol name.
	expression1	specifies the symbol size.
	expression2	specifies an alignment value for the bss location counter. The alignment value may not be zero.

Description: The `.bss` directive defines a symbol in the bss or the uninitialized data segment. There is no code or data in the object file associated with the bss segment. The `.bss` directive is a shorthand way to align the location counter associated with the bss segment, define a symbol, and allocate the appropriate number of bytes of storage space. It does not change the current location counter to the bss segment. To change the current location counter, use the `.udata` directive, see Section 6.7.5.

The `.bss` directive performs the following actions:

1. aligns the bss location counter to a multiple of *expression2*. The value of the location counter is incremented if necessary.
2. defines the specified symbol. The symbol is assigned the current value of the bss segment location counter and type bss. The assembler uses the Absolute addressing mode to reference symbols or expressions of type bss.
3. adds the number of bytes specified by *expression1* to the bss location counter.

Example:

```
.bss name_str, 25, 4
```

In the preceding example, the bss segment location counter is aligned to the next multiple of four bytes, incrementing if necessary. The symbol name_str is defined and assigned the value of the bss location counter. The bss segment location counter is incremented by 25.

6.7.5 .udata

Syntax: **`.udata`**

where: `.udata` is the directive name.

Description: The `.udata` directive indicates the beginning of a bss or uninitialized data segment. It is used to define symbols and allocate storage space. As with dummy sections, no code or data is generated in the object file. However, storage space is accumulated. The total accumulated size of the segment is recorded in the a.out header and the .bss section header of the object file. Memory is allocated for the total size of the bss segment at load time.

The directive sets the location counter to the next available bss segment address. Symbols defined in the bss (udata) segment are of type bss. The assembler uses the Absolute addressing mode for all symbols or expressions of type bss.

Example:

```
.udata
```

In the preceding example, the location counter is set to type bss. The offset is set to the next available offset.

6.7.6 .static

Syntax: **`.static`**

where: `.static` is the directive name.

Description: The `.static` directive defines a static base segment. The assembler assigns the current location counter the next available static segment address. Subsequent storage allocation, data generation, or program statements generate code in the static segment. The location counter is incremented after every assignment or code generation. The data will be placed in the .static section of the object file. Storage allocated in the static segment is zero-filled. All symbols defined in the static segment are of type static. The assembler addresses all symbols or expressions of type static with the SB Relative addressing mode.

The `.static` directive is useful when building *Series 32000* modules. The assembler generates all SB Register Relative and SB Memory Relative addresses as offsets from zero unless a `.module` directive or a `.modentry` directive is issued to set the static base address directly.

Example:

```
1                                     .static
2  S00000000  00        ALPHA:        .blkb
3  S00000001  00000000  BETA:         .blkw  2
4  S00000005  00000000  GAMMA:        .blkd
```

The preceding example defines three symbols in a static base segment. The `.static` directive (Line 1) sets the location counter to static segment type and to the next available static segment address. Lines 2, 3, and 4 allocate a total of 9 bytes of storage, zero-filled.

6.7.7 .link

Syntax: **`.link`**

where: `.link` is the directive name.

Description: The `.link` directive defines a link table segment. The link segment contains a *Series 32000* module's link table. A *Series 32000* module requires a link table entry for each variable the module imports from another module and for each procedure the module imports or exports that uses the cxp, rxp calling discipline. The link table offset should be used with the External addressing mode to access all external variables from a *Series 32000* module. The `.xdd` directive should be used to generate link table entries for data variables. The `.xpd` directive should be used to generate procedure descriptor link table entries for functions. Care should be taken that link table entries remain aligned on 4-byte boundaries if any other directives or instructions are used in the link segment.

The `.link` directive assigns the current location counter the next available link segment address. The location counter is incremented after every `.xdd` or `.xpd` directive. The link table will be placed in the .link section of the object file. Any storage allocated in the link segment is zero-filled. All symbols defined in the link segment are of type link. The assembler addresses all symbols or expressions of type link with the absolute addressing mode.

The definition of a link table entry in the link table segment should precede any reference to it.

```
            .link
LINK_ENTRY: .xpd    scan_arg
            .text
            addr    LINK_ENTRY, r0
            cxpd    r0
```

In this example, the addr instruction uses the absolute addressing mode to access LINK_ENTRY.

Example:

```
 1                                       .globl   _main
 2                                       .globl  _printf
 3
 4                                       .link
 5  L00000000  00000000  main:           .xpd     _main
 6  L00000004  00000000  printf:         .xpd     _printf
 7
 8                                       .text
 9                       _main:
10  T00000000  820000                    enter    [],0
11  T00000003  e7d5c000                  addr     msg,tos
               0020
12  T00000009  22c00000                  cxp      printf
               07
13  T0000000e  7ca5fc                    adjspb   $-4
14  T00000011  9200                      exit     []
15  T00000013  3200                      rxp      0
16
17                                       .static
18                       msg:
19  S00000000  48656c6c                  .ascii   "Hello, World"
               6f2c2057
               6f726c64
               210a00
```

Lines 1 and 2 declare the variables _main and _printf to be external, *i.e.*, available for export or necessary to import.

Line 4 begins the link table segment. The current location counter is set to link segment type, offset 0.

Line 5 generates a procedure descriptor link table entry for _main, beginning at link table segment address L00000000. Each link table entry is four bytes long. The current location counter is address L00000004 of the link segment at the end of line 5.

Line 6 generates a procedure descriptor link table entry for _printf. The current location counter is link segment address L00000008 at the end of line 6.

Line 8 begins a text segment. The current location counter is set to the next available text segment address, which is T00000000.

Lines 9 to 15 generate program code. At the end of line 15, the current location counter is text segment address T00000015, hexadecimal. This is the next available text address.

Line 17 begins a static segment. The current location counter is set to the next available static segment address, S00000000.

Line 19 generates the ASCII character string "Hello, World!\12\0" in the static segment.

6.7.8 .section

Syntax: `.section` *section_name , string* or
`.section` *section_name*

where:		
	section_name	is any legal identifier, only eight significant characters
	string	is a quoted string consisting of any combination of the following letters:

b -> STYP_BSS
c -> STYP_COPY
i -> STYP_INFO
d -> STYP_DSECT
x -> STYP_TEXT
n -> STYP_NOLOAD
o -> STYP_OVER
l -> STYP_LIB
w -> STYP_DATA

Description: The `.section` directive allows the assembly programmer to define a section with attributes, refer to the *Series 32000 GNX — Version 3 COFF Programmer's Guide* for a description of section attributes. *Section_name* is the name of the section, and each character in *string* represents an attribute. Symbols declared within a section belong to the particular section. A section is active until the next `.section`, `.text`, `.data`, `.udata`, `.link`, or `.static` directive. In the default case, reference to symbols of a user-defined section are referenced via the absolute addressing mode. Only 10 sections are allowed including .text, .data, .bss, .link, .static, .mod, and .comment. The .mod and .comment sections are optional; therefore, there can only be 3, 4 or 5 user-defined sections.

Example:

```
 1                                      .globl   start
 2                                      .globl   istart
 3                                      .globl   mcount
 4
 5
 6                            start:
 7 T00000000   7ca508                   adjspb   $8
 8 T00000003   57ce0800                 movd     8(sp),0(sp)
 9 T00000007   27c80c                   addr     12(sp),r0
10 T0000000a   570604                   movd     r0,4(sp)
11 T0000000d   02c00000                 bsr      istart
               13
12 T00000012   02ffffff                 bsr      _main
               ee
13 T00000017   7ca5f4                   adjspb   $-12
14                            mcount:
15 T0000001a   1200                     ret      0
16
17                                      .data
18                                      .align   4
19                            environ:
20 D00000000   00000000                 .double  0
21
22                                      .section .init,"x"
23                            istart:
24 00000000    820700                   enter    [r0,r1,r2],0
```

In this program, line 22 is the declaration of a section called .init, whose section attribute is STYP_TEXT. The label “istart” and the “enter” instruction both belong to the .init section.

6.7.9 .org

Syntax: **`.org`** *expression*

where:

`.org`	is the directive name.
expression	specifies the new value of the location counter. The expression must evaluate to type absolute or the type of the current location counter.

Description: The `.org` directive changes the value of the current location counter within a segment. It sets the location counter to the value specified by *expression*. The type of the expression must be compatible with that of the current location counter, or it must be an absolute address. If the expression evaluates to an absolute address, the assembler generates a warning message, and sets the location counter to the value specified by *expression* + the starting location of the current segment.

If the current segment is an object file segment, that is, one of text, data, static, or link, then the value of the expression must be greater than, or equal to, the current location counter (*i.e.*, backstepping is not permitted). Furthermore, for object file segments, the GNX Assembler fills the bytes between the current and the new location with alignment values as filled for the `.align` directive. The added bytes are included in the program listing.

Example:

```
1                       .set    NUM_CHNKS, 10
2                       .set    CHNK_SIZE, 4096
3
4                       .udata
5  B00000000  c_ptr:    .blkd   10
6  B00000028  pool:     .org    pool + (NUM_CHNKS * CHNK_SIZE)
7  B0000a028  mark:     .blkd
```

This example uses the `.org` directive to leave a large area of memory available in the bss segment.

6.7.10 .align

Syntax: `.align` *expression1* [,*expression2*]

where:

`.align`	is the directive name.
expression1	specifies the basis of a new location counter value. It must evaluate to a positive absolute value. No forward symbol references are permitted.
expression2	specifies the offset of the new location counter value. It must evaluate to a non-negative absolute value and must be less than the value of *expression1*. Default value is zero. No forward symbol references are permitted.

Description: The `.align` directive sets the location counter to a new value without changing the current type. The new value is the sum of a multiple of the basis, *expression1*, and the offset, *expression2*. The new value is always equal to, or greater than, the current location counter and satisfies the following equation:

new value MOD *expression1* = *expression2*

The new value is the multiple of the basis that is greater than, or equal to, the current location counter. For example, if *expression1* is 6 and the current location counter is 20, then the new value is 24 (*i.e.*, 4*6). The default value of *expression2* is zero.

If both *expression1* and *expression2* are specified, the new value is the sum of the multiple of the basis and the offset. For example, if *expression1* is 4, *expression2* is 3, and the current location counter is 22, then the new value is 27 (*i.e.*, 6*4+3).

The assembler will optimize the fill pattern if the current section is .text. The optimized filler can be viewed as a fancy nop. The assembler will use “movb r7,r7” for 2-bytes fillers , “orb $0,r7” for 3-bytes fillers , “orw $0,r7” for 4-bytes fillers , “orw $0,r7” and “nop“ for 5-bytes fillers , “ord $0,r7” for 6-bytes fillers. All other alignments are filled with combinations of the above.

If the `.align` directive is used in the data, static, or link segment, then the assembler zero-fills all bytes between the current location and the specified address and includes up to 128 bytes of the zero-filled bytes in the program listing.

Example:

```
 1                                          .static
 2  S00000000  00          FIRST:           .blkb
 3  S00000001  000000                       .align 4
 4  S00000004  00          SECOND:          .blkb
 5  S00000005  00000000                     .align 4, 2
               00
 6  S0000000a  00          THIRD:           .blkb
```

The preceding example contains two `.align` directives (lines 3 and 5). In line 3, the directive sets the location counter to a multiple of 4. The current location counter is S00000001 (static segment), so the new location counter will be S00000004 (*i.e.*, 1*4). In line 5, the directive sets the location counter to a multiple of 4 plus 2. If the current location counter is 5, then the new location counter is 10 (0xa) (*i.e.*, 2*4 + 2).

Example:

```
 1                            _mail:
 2  T00000000   a2                      nop
 3                            LABEL:
 4  T00000001   d439                    .align 3
 5  T00000003   0a00                    .word 10
 6  T00000005   d8a100                  .align 4
 7  T00000008   01                      .byte 1
 8  T00000009   0a00                    .word 10
 9  T0000000b   a2                      .align 2
10  T0000000c   1200                    ret 0
```

The preceding example contains three `.align` directives (lines 4, 6, and 9). In line 4, the directive sets the location counter to a multiple of 3. The current location counter is T00000001 (text segment), so the new location counter is T00000003 (*i.e.*, 1*3). The 2-byte filler "movb r7,r7" denoted by the opcode d439 (low bytes first) is used. In line 6, the directive sets the location counter to a multiple of 4. The current location counter is T00000005, so the new location counter will be T00000008 (*i.e.*, 2*4). The 3-byte filler "orb $0,r7" denoted by the opcode d8a100 is used. In line 9, the directive sets the location counter to a multiple of 2. The current location counter is T0000000b, so the new location counter will be T0000000c (*i.e.*, 6*2). The single byte filler "nop" denoted by the opcode a2 is used in this case.

6.7.11 .ident

Syntax: **`.ident`** *string*

where: *string* is a quoted string.

Description: The `.ident` directive takes its string argument and places it in the .comment section of the object file. This directive may be used more than once. The .comment section is given the section attribute of STYP_INFO. The Linker will combine all .comment sections at link time.

Example:

```
1                          .text
2                          .ident "This is .ident"
3  T00000000  a2a2a2a2   .space 10
              a2a2a2a2
              a2a2
4                          .ident "Another .ident"
```

In this program, the strings "This is .ident" and "Another .ident" are placed in the .comment section of the object file.

6.8 MODULE TABLE DIRECTIVES

The module table directives manage the task of building the module table. Defined module table entries are placed by the GNX Assembler into the .mod section of the COFF output file. The following are the module table directives discussed in this section:

Directive	**Function**
`.module`	names a module, associates the assembled code and static local data with the module, and defines a module table entry for the module.
Warning	The `.text` and `.static` sections of the file where `.module` is used will be treated as modular sections by the GNX linker even when the assembler was not invoked with the " -X" ("/MODULAR" on VMS) invocation option. In particular, during the link, input section rules of the form *(`.text`) or *(`.static`), will not apply to such files. Please see the GNX Linker Programmer's Reference Manual for further information on the linking process.
`.modentry`	defines a module table entry for a named module.

A module table entry consists of four 32-bit entries corresponding to each component of a module:

- The *Static Base* (sb) entry contains the base address for the module's static local data.
- The *Link Base* (lb) entry contains the base address for the module's link table.
- The *Program Base* (pb) entry contains the base address for the module's program code.
- A fourth entry is currently unused but reserved.

Each base address is a standard *Series 32000* address. The relocation information for the sb, lb, and pb entries depends on how these base addresses have been specified. The following discusses the module table entries and their relocation information.

Static Base (sb) Entry

If the sb entry is specified by an expression of:

1. absolute type (*e.g.*, sb=200), the module table entry's static base address is the specified absolute value. No relocation information will be generated.
2. non-absolute type (*e.g.*, sb=sb_sym+10 where sb_sym is non-absolute), the module table entry's static base address is the value of the expression, and a relocation entry will be generated as follows:

```
R_ADDRTYPE    = R_ADDRESS
R_RELTO       = R_ABS
R_FORMAT      = R_NUMBER
R_SIZESP      = R_S_32
```

with symbol table index pointing to the symbol table entry that is being relocated (*e.g.*, sb_sym). Refer to the *Series 32000 COFF Programmer's Guide* for a definition of these symbols.

If the sb entry is not specified, the assembler will use location zero as the module table entry's static base address. At link time, the linker will set the module's static base to the lowest address of the output section that contains the input .static sections for the module.

The relocation entry, generated by the assembler when the sb entry is not specified, is as follows:

```
R_ADDRTYPE    = R_STATIC_SEC
R_RELTO       = R_ABS
R_FORMAT      = R_NUMBER
R_SIZESP      = R_S_32
```

with symbol table index pointing to the module name. Refer to the *Series 32000 COFF Programmer's Guide*, for a definition of these symbols.

Link Base (lb) Entry

If the lb entry is specified by an expression of:

1. absolute type (*e.g.*, lb=200), the module table entry's link table base address is the specified absolute value. No relocation information will be generated.
2. non-absolute type (*e.g.*, lb=lb_sym+10 where lb_sym is non-absolute), the module table entry's link table base address is the value of the expression, and a relocation entry will be generated as follows:

```
R_ADDRTYPE    = R_ADDRESS
R_RELTO       = R_ABS
R_FORMAT      = R_NUMBER
R_SIZESP      = R_S_32
```

with symbol table index pointing to the symbol table entry that is being relocated (*e.g.*, lb_sym). Refer to the *Series 32000 COFF Programmer's Guide*, for a definition of these symbols.

If the lb entry is not specified, the assembler will use location zero as the module table entry's link table base address. At link time, the linker will set the module's link table base to the lowest address of the output section that contains the input .link sections for the module.

The relocation entry generated by the assembler when the lb entry is not specified is as follows:

```
R_ADDRTYPE    = R_LINK_SEC
R_RELTO       = R_ABS
R_FORMAT      = R_NUMBER
R_SIZESP      = R_S_32
```

with symbol table index pointing to the module name. Refer to the *Series 32000 COFF Programmer's Guide*, for a definition of these symbols.

Program Base (pb) Entry

If the pb entry is specified by an expression of:

1. absolute type (*e.g.*, pb=200), the module table entry's program code base address is the specified absolute value. No relocation information will be generated.
2. non-absolute type (*e.g.*, pb=pb_sym+10 where pb_sym is non-absolute), the module table entry's program code base address is the value of the expression, and a relocation entry will be generated as follows:

   ```
   R_ADDRTYPE    = R_ADDRESS
   R_RELTO       = R_ABS
   R_FORMAT      = R_NUMBER
   R_SIZESP      = R_S_32
   ```

   with symbol table index pointing to the symbol table entry that is being relocated (*e.g.*, pb_sym). Refer to the *Series 32000 COFF Programmer's Guide*, for a definition of these symbols.

If the pb entry is not specified, the assembler will use location zero as the module table entry's program code base address. At link time, the linker will set the module's program base to the lowest address of the output section that contains the input .text sections for the module.

The relocation entry generated by the assembler when the pb entry is not specified is as follows:

```
R_ADDRTYPE    = R_TEXT_SEC
R_RELTO       = R_ABS
R_FORMAT      = R_NUMBER
R_SIZESP      = R_S_32
```

with symbol table index pointing to the module name. Refer to the *Series 32000 COFF Programmer's Guide*, for a definition of these symbols.

6.8.1 .module

Syntax: **.module** *symbol* [**,sb=***static base*] [**,lb=***link base*]
[**,pb=***program base*]

where:

`.module`	is the directive name.
symbol	is the name of the module. This symbol will define the module's module table entry.
`sb=`*static base*	explicitly sets the static base address for the module.
`lb=`*link base*	explicitly sets the link table base address for the module.
`pb=`*program base*	explicitly sets the program code base address for the module.

Description: The `.module` directive declares a module name, associates the text, link, and static local data segments generated by this assembly to the module table entry name and optionally defines a module table entry for the module.

If none of the optional arguments are specified, *symbol* is a global, undefined symbol unless *name* is previously defined by the `.modentry` directive. See Section 6.8.2 for the description on the `.modentry` directive.

If any of the optional arguments are specified, the assembler generates a 16-byte module table entry that contains the module's static base address, link table base address, program code base address, and a reserved double-word set to zero in the .mod section of the output COFF file. The value for the module table entry's static base address, link table base address, and program code base address will be as specified by the sb, lb, pb contents, or by default, the lowest address of .static, .link, .text section for the named module as output by the linker, if there is one, otherwise zero, and their corresponding relocation entries. Refer to Section 6.8 for the description on module table entries and their relocation information. *Symbol*, in this case, is global and is defined in the .mod section with the value of the address of the module table entry.

There may be no more than one `.module` directive per assembly.

Example:

```
.module hello, sb=.static, lb=.link
```

6.8.2 .modentry

Syntax: **`.modentry`** *symbol* [**`,sb=`***static base*] [**`,lb=`***link base*] [**`,pb=`***program base*]

where:

`.modentry`	is the directive name.
symbol	is the module name. This symbol defines the module's module table entry.
`sb=`*static base*	explicitly sets the static base address for the module. If not specified, the assembler will use the default value of zero and at link time, the linker will set it to the lowest address of the output .static section.
`lb=`*link base*	explicitly sets the link table base address for the module. If not specified, the assembler will use the default value of zero and at link time, the linker will set it to the lowest address of the output .link section.
`pb=`*program base*	explicitly sets the program code base address for the module. If not specified, the assembler will use the default value of zero and at link time, the linker will set it to the lowest address of the output .text section.

Description: The `.modentry` directive defines a module table entry for a named module by generating a 16-byte module table entry that contains the module's static base address, link table base address, program code base address, and a reserved double-word set to zero in the .mod section of the output COFF file. The value for the module table entry's static base address, link table base address, and program code base address will be as specified by the sb, lb, pb contents, or by default, the lowest address of the .static, .link, .text section for the named module as output by the linker, if there is one, otherwise zero, and their corresponding relocation entries. Refer to Section 6.8 for the description on module table entries and their relocation information.

Symbol identifies the module.

Example:

```
devices.s:
    .file "devices.s"
    .modentry devA  # Define mod table entry for device A
    .modentry devB  # Define mod table entry for device B
    .modentry devC  # Define mod table entry for device C
devA.s:
    .file "devA.s"
    .module devA    # Must not use optional args
devB.s:
    .file "devB.s"
    .module devB    # Must not use optional args
devC.s:
    .file "devC.s"
    .module devC    # Must not use optional args
```

6.9 FILENAME DIRECTIVE

The filename symbol directive specifies the name of the source file:

Directive	Function
`.file`	specifies the source filename

6.9.1 .file

Syntax: **`.file`** "*symbol* "

where:

`.file`	is the directive name.
"symbol"	specifies source filename for the current assembly. Must be enclosed in double-quotes.

Description: The `.file` directive specifies the name of the source file currently being assembled. The GNX Assembler records the filename in the object file as an auxiliary symbol table entry of the special symbol .file. Only one `.file` directive per source file is allowed. It may appear anywhere in the file. If no `.file` directive is specified, the filename is the input source filename.

If more than one `.file` directive is specified, the first specified filename is taken, and a warning message is issued for the rest of them.

The `.file` directive is used by compilers to associate the name of a high-level language source file with the object file produced by the GNX Assembler.

Example:

```
.file "stress.c"
```

This example defines the symbol stress.c as the name of the source file associated with the current assembly.

NOTE: When using the debugger, the *symbol* must be the same as the filename since the debugger uses this as the name of the source file.

6.10 SYMBOL TABLE ENTRY DEFINITION DIRECTIVES

The symbol table entry definition directives specify symbolic information which the GNX Assembler records in the object file. The directives provide a means to record a variety of information useful to symbolic debuggers. Symbol table entry directives do not affect the execution of an assembly language program.

The basic symbol table entry directives are `.def` and `.endef`. They mark the start and the end of a symbol definition. Between these, various directives may be used to assign attributes to the symbol, for example, its size, value, and type or its location in the source file.

Each `.def` begins to define a new symbol table entry. Therefore, all information to be recorded about a single symbol must be included between the `.def` directive and the matching `.endef` directive.

Symbol table entry definitions may not be nested.

The symbol table entry definition directives are as follows:

Directive	**Function**
`.def`	begins symbol table entry definition
`.dim`	defines the dimensions of an array
`.line`	specifies a source line number
`.scl`	specifies the symbol's storage classification
`.size`	specifies the symbol's storage size
`.tag`	specifies the tag name associated with a type
`.type`	specifies the symbol's type
`.val`	specifies the symbol's value
`.endef`	terminates the symbol table entry definition

Sections 6.10.1 through 6.10.9 describe the symbol table entry directives in detail.

NOTE: It is important to fully understand the Common Object File Format (COFF) symbol table requirements before attempting to use these directives. For complete specification of COFF requirements refer to the *Series 32000 GNX — Version 3 COFF Programmer's Guide.* For useful constant definitions see the include files:

File	**Contents**
`syms.h`	Symbol table entry definition, auxiliary entry definition, type and derived type values
`storclass.h`	Storage class values

6.10.1 .def

Syntax:	**`.def`** *symbol*	
where:	`.def`	is the directive name.
	symbol	is a symbol name. It consists of a series of characters which may be letters, numbers, period (.), or underscore (_). The first character must not be a number.

Description: The `.def` directive causes the GNX Assembler to begin the definition of a Common Object File Format (COFF) symbol table entry for the specified symbol. The GNX Assembler creates the new symbol table entry and enters the symbol name. The Assembler does not check the COFF validity of the given values for symbol table entries definiton.

Example:

```
.def    _n_ptr
        .val    _n_ptr
        .scl    2
        .type   2 | (1 << 4)
.endef

.globl  _n_ptr
.comm   _n_ptr,4
```

This example is a symbolic definition associated with the C declaration:

```
char    *n_ptr;
```

The `.def` directive starts the definition. The symbol table entry is assigned the value _n_ptr, a storage class of external (C_EXT) represented by the value 2, a base type of character (T_CHAR) represented by the value 2, and a derived type of pointer (DT_PTR) represented by the value 1. The `.endef` directive ends the definition. For more information about the structure of a COFF symbol table entry, the meaning of various fields, and the values each may contain, refer to the *Series 32000 GNX — Version 3 COFF Programmer's Guide.*

6.10.2 .dim

Syntax: **`.dim`** *expression,,,*

where:	`.dim`	is the directive name.
	expression	specifies the size of one dimension of an array.

Description: The `.dim` directive defines the dimensions of an array. Each argument *expression* specifies the number of elements in one array dimension. The symbol table entry format allows the specification of up to four array dimensions.

The GNX Assembler enters the specified expressions into the array dimension field of the auxiliary symbol table entry for the symbol that is being defined. If no auxiliary entry exists, the GNX Assembler creates one.

Example:

```
.dim  5,10
```

This example is a portion of the symbolic definition for a two-dimensional array. Dimension one is 5, dimension two is 10.

6.10.3 .line

Syntax: **`.line`** *expression*

where:

	`.line`	is the directive name.
	expression	is the source file line number of the symbol declaration.

Description: The `.line` directive specifies the source file line number on which a symbol has been declared. The GNX Assembler enters the specified value, *expression*, into the line number field of the auxiliary symbol table entry for the symbol that is being defined. The Assembler generates an auxiliary entry if one does not exist.

The `.line` directive should be used when the symbol being defined is a block symbol. Block symbols include the special symbols .bf and .ef which define the beginning and ending of functions, the special symbols .bb and .eb which define the beginning and ending of blocks, and all symbols defined within a block.

NOTE: The `.line` directive should be used only where the Common Object File Format symbol table entry specification requires and accepts a line number. For additional information, refer to the *Series 32000 GNX — Version 3 COFF Programmer's Guide.*

Example:

```
.line 25
```

This example is part of the definition of a block symbol declared on source line number 25.

6.10.4 .scl

Syntax: `.scl` *expression*

where:	`.scl`	is the directive name.
	expression	is the value of a storage classification as defined in the *Series 32000 GNX — Version 3 COFF Programmer's Guide.*

Description: The `.scl` directive assigns a storage class value to the symbol definition. The storage class of a symbol affects the interpretation of the "value" field of the entry. Storage classes are as follows:

C_AUTO — automatic variable, whose value is a stack offset.

C_EXT — external symbol, whose value is a relocatable address.

C_STAT — C style static or local variable, whose value is a relocatable address.

C_REG — register variable, whose value is the number of the register. For example, if the register is r0 the register number is 0.

C_LABEL — an assembly language label, whose value is a relocatable address.

C_MOS — member of a structure, whose value is the offset of the field from the start of the structure.

C_ARG — function argument, whose value is a stack offset.

C_STRTAG — structure tag (name), whose value is 0.

C_MOU — member of a union, whose value is the offset of the field from the start of the union.

C_UNTAG — union tag (name), whose value is 0.

C_TPDEF — type definition, whose value is 0.

C_ENTAG — enumeration tag (name), whose value is 0.

C_MOE — member of an enumeration, whose value is the enumeration number.

C_REGPARM — register parameter, whose value is the number of the register.

C_FIELD — bit field, whose value is the bit displacement.

C_BLOCK — beginning or end of block, whose value is a relocatable address.

C_FCN — beginning or end of a function, whose value is a relocatable address.

C_EOS — end of a structure, whose value is the structure size.

C_FILE — filename entry, whose value is the symbol table index of the next .file symbol or the beginning of the global symbols if there are no more .file symbols.

C_ALIAS — duplicate tag, whose value is the symbol table index of the tag definition.

For more complete information about storage classes and their values, refer to the *Series 32000 GNX — Version 3 COFF Programmer's Guide.*

Example: `.scl 2`

This example specifies a storage classification of C_EXT (external), represented by the value 2.

6.10.5 .size

Syntax: `.size` *expression*

where:

`.size`	is the directive name.
expression	specifies the size of a structured variable.

Description: The `.size` directive specifies the total size of a structured type, an array, or an enumerated type. The GNX Assembler enters the specified value into the size field of the auxiliary symbol table entry for the symbol that is being defined. If no auxiliary entry exists, the Assembler generates one. For example, the C declaration:

char name_list[20] [200];

generates the following symbol specification:

```
 1  .def     _name_list
 2           .val       _name_list
 3           .scl       2
 4           .type      0362
 5           .dim       20,200
 6           .size      4000
 7  .endef
 8           .globl     _name_list
10           .comm      _name_list,4000
```

The storage size specified by the `.size` directive in line 6 is 4000 bytes (20*200*sizeof(char)), where the size of a character is one byte.

Example: **.size 200**

This example specifies a symbol's storage size as 200 bytes. The Assembler enters the value 200 into the size field of the auxiliary symbol table entry for the symbol that is being defined.

6.10.6 .tag

Syntax:	**`.tag`** *symbol*	
where:	`.tag`	is the directive name.
	symbol	is a symbol. The symbol is the tag name of a data structure definition, for example, a C struct or union.

Description: The `.tag` directive associates the tag name of a data structure with a symbol. The GNX Assembler enters the symbol table index of the tag name into the tag index field of the auxiliary entry for the symbol that is being defined. If no auxiliary entry exists, the GNX Assembler generates one.

Example:

```
.def  _coord
      .scl 10; .type 010; .size 12; .endef
.def  _a
      .val 0; .scl 8; .type 04; .endef
.def  _b
      .val 4; .scl 8; .type 04; .endef
.def  _c
      .val 8; .scl 8; .type 04; .endef
.def  .eos
      .val 12; .scl 102; .tag _coord; .size 12; .endef

.def  _bar
      .val _bar; .scl 2; .type 010; .tag _coord; .size 12; .endef

      .globl _bar
      .comm _bar,12
```

This example defines the symbols associated with the C declarations:

```
struct   coord {
         int      a;
         int      b;
         int      c;
};
struct   coord    bar;
```

The special symbol .eos (end of structure) uses the `.tag` directive to point back to the definition of the structure coord.

The bar symbol, which is of type *struct coord,* also uses the `.tag` directive to point to the entry for coord.

6.10.7 .type

Syntax: **`.type`** *expression*

where:

`.type`	is the directive name.
expression	specifies the type of a symbol.

Description: The `.type` directive specifies type information associated with the symbol that is being defined. The GNX Assembler enters the *expression* into the type field of the main symbol table entry for the symbol that is being defined.

The type field consists of sixteen bits, of which the low-order four contain the base type. The remaining bits contain derived types, each of which is specified in a two-bit field. For definition of types and derived types see the *Series 32000 GNX — Version 3 COFF Programmer's Guide.*

Examples:

```
1.   .type   (2 | (2 << 4)) | 1 << 6
2.   .type   4
```

The first example is a symbolic definition associated with the C declaration:

```
char     *fn();
```

The base type is T_CHAR (type character) represented by the value 2. The first derived type is DT_FCN (function) represented by the value 2. The second derived type is DT_PTR (pointer) represented by the value 1. The entire type field is interpreted as a pointer to a function that returns a character.

The second example is associated with the C declaration:

```
int      flag;
```

The `.type` directive specifies the type T_INT (integer) represented by the value 4.

6.10.8 .val

Syntax:	`.val` *expression*	
where:	`.val`	is the directive name.
	expression	specifies the value of the symbol.

Description: The `.val` directive specifies the value field of the main symbol table entry for the symbol that is being defined.

Example:

```
.val  _flag
```

This example sets the value field of the symbol table entry to the address of the symbol _flag.

6.10.9 .endef

Syntax: **`.endef`**

where: `.endef` is the directive name.

Description: The `.endef` directive causes the GNX Assembler to end the definition of a Common Object File Format (COFF) symbol table entry for the specified symbol. The GNX Assembler adds the new symbol table entry to the symbol table. The GNX Assembler generates an auxiliary entry if the symbol specifications require one and fills in any symbol table index fields as necessary.

Example:

```
.def    _flag
        .val   _flag
        .scl   2
        .type  4
.endef
```

This example is a symbolic definition associated with the C declaration:

```
int     flag;
```

The `.endef` directive ends the definition.

6.11 LINE NUMBER TABLE CONTROL DIRECTIVE

Each section in the object file may have an associated line-number table, for the purpose of source-level debugging support. The line-number table maps source file line numbers to addresses within the section. Each line number table entry is either a function entry or a line number entry. Function entries record the symbol table index for the function. Line number entries record a line number offset from the start of the function and an associated physical address.

Function entries are generated automatically by the assembler when a function is defined, refer to Section 6.10. Line number table entries are created with the `.ln` directive.

Directive	**Function**
`.ln`	specifies a line number entry

6.11.1 .ln

Syntax: **`.ln`** *expression1* [*,expression2*]

where:

`.ln`	is the directive name.
expression1	specifies the source file line offset from the beginning of a function.
expression2	specifies an associated memory address. This value defaults to the current location.

Description: This directive is used to equate higher level source code line numbers to assembly code, normally generated by compilers. *Expression1* must yield a value of absolute type that gives a line number in the source code. *Expression2* if present, must have a value of type TEXT, DATA, or BSS that gives the address within the section where the line number occurs. If the second operand is missing, the value of the current location counter will be used as the address of the line number.

Example:

```
.ln 1
```

This example defines a line number entry for the first line of a function. The associated memory address is the value of the current location counter.

6.12 MACRO-ASSEMBLER DIRECTIVES

The macro-assembler directives provide the macro and conditional assembly support. They enable the definition and usage of macros, and allow for the inclusion or deletion of optional assembly statements. Other macro-assembler directives help minimize programming errors and speed the development process. For more details see Chapter 8.

The macro-assembler directives are as follows:

Directive	**Function**
`.macro`	begins a macro-procedure definition
`.endm`	ends a macro-procedure definition
`.if`	begins a conditional macro-assembler statement
`.elsif`	begins an elsif close for the conditional macro-assembler statement
`.else`	begins an else close for the conditional macro-assembler statement
`.endif`	ends a conditional macro-assembler statement
`.repeat/.irp`	begins a macro repetitive block
`.endr`	ends a macro repetitive block
`.exit`	terminates processing of the current repetitive block
`.macro_on`	enables macro-procedure expansions
`.macro_off`	disables macro-procedure expansions
`.include`	includes another file
`.mwarning`	generates an assembler warning message
`.merror`	generates an assembler error message

6.12.1 .macro

Syntax: **`.macro`** *macro-name* [*formal-arg* [, *formal-arg*] . . .]

where:

macro-name	is the macro-procedure name. It may be any legal assembler symbol.
formal-arg	is a macro-variable defining a formal argument.

Description: The `.macro` directive begins the macro-procedure definition. The macro-procedure associates a macro name with a sequence of statements which follow the `.macro` directive, up to the `.endif` directive.

Example:

```
.macro   clear_array      size, base_reg
```

Defines a macro procedure named `clear_array` with two formal arguments `size` and `base_reg`.

6.12.2 .endm

Syntax: *.endm* [*macro_name*]

where: `.endm` **ends the macro-procedure definition.**

Description: **The `.endm` directive marks the end of the macro-procedure definition.** ***macro_name*** **is an optional specification for the name of the macro to be ended.**

Example:

```
.macro   clear-array   size, base-reg   # defines clear-array
    .
    .                                     # macro statements
    .
.endm    clear-array                    # ends the definition of clear_
```

6.12.3 .if

Syntax: **`.if`** *if_condition*

where: *if_condition* is an arithmetic macro-expression.

Description: The `.if` directive begins a conditional macro assembler statement. *if_condition* is a condition to be tested during macro processing phase. If found to be true, the statements following it (until a corresponding `.elseif,` `.else` or `.endif` directive) are processed by the macro processor.

Example:

```
.if {reg_num} > 5
    movqd   5, r{reg_num}
.elsif {reg_num} > 3
    movqd   3, r{reg_num}
.else
    movqd   1, r{reg_num}
.endif
```

`If` `reg_num` holds the value `6` this is expanded to

movqd 5, r6

if `reg_num` holds the value `4` this is expanded to

movqd 3, r4

and if `reg_num` holds the value `0` this is expanded to

movqd 1, r0

6.12.4 .elsif

Syntax: `.elsif` *elsif_condition*

where: *elsif_condition* is an arithmetic macro-expression.

Description: In a conditional block, if the *if_condition* is found to be false, the *elsif_condition* arguments are evaluated until one is found to be true. If the *elsif_condition* is found to be true, the corresponding *elsif_conditional_body* statements following the *elsif_condition* are processed.

6.12.5 .else

Syntax: `.else` *else_conditional_body*

where: *else_conditional_body*
consists of valid assembly language statements, directives, macro-procedure calls and macro-assembler directives, repetitive blocks and macro-procedure definitions.

Description: In a conditional block, if the previously specified *if_condition* or *elsif_condition* is found to be false, then the *else_conditional_body* statements (following the *elsif_condition*) are processed.

6.12.6 .endif

Syntax: `.endif`

Description: Ends an `.if` conditional macro-assembler statement.

6.12.7 .repeat

Syntax: **`.repeat`** [*iteration_count* [, *iteration_var*]]

where:

iteration_count	specifies the number of iterations.
iteration_var	is a macro-variable name used as an iteration index.

Description: The `.repeat` directive begins a macro repetitive block, which ends with a `.endr` directive. The number of repetitions is determined by the *iteration_count* argument. Repetitive blocks may appear inside a macro-procedure definition, in conditional blocks, and may be nested without limit.

If given, the *iteration_var* argument holds a string representing the current iteration number for each iteration. After the repetitive block has been processed, it holds the *iteration_count* value. If the *iteration_count* argument is evaluated as a negative or zero value, the statements in the block are read textually without being processed until an `.endr` directive is reached. If the *iteration_count* argument is not given, then the repetitive block is processed repeatedly until an `.exit` directive is processed (see section 8.9.3).

Example:

```
.repeat      8, i
  movqd      0, r{{i} - 1}
.endr
```

generates code that clears `r0` through `r7`.

6.12.8 .irp

Syntax:	`.irp` *iteration_var, iteration_list*	
where:	*iteration_var*	is a macro-variable name to be used as an iteration variable.
	iteration_list	is a macro-list.

Description: The `.irp` directive begins a special macro repetitive block, which ends with a `endr` directive. For each element in the *iteration_list* argument, the macro-processor assigns its string value to *iteration_var*, and process the code between the `.irp` statement and the corresponding `.endr` statement. If the *iteration_list* argument is an empty macro-list, the statements in the block are read textually without being processed. After the repetitive block has been processed, *iteration_var* contains the last element of *iteration_list*.

Example:

```
.irp   reg, [r0,r1,r2,r3,r4,r5,r6,r7]
   movqd        0,{reg}
.endr
```

generates code that clears registers `r0` through `r7`.

6.12.9 .endr

Syntax: **`.endr`**

Description: The `.endr` directive ends a macro repetitive block.

6.12.10 .exit

Syntax: **`.exit`**

Description: Terminates the processing of the current repetitive block that begins with either a `.repeat` or `.irp` directive. Statements following this directive are read textually without being processed, until an `.endr` statement is encountered.

Example:

```
x:=1
.repeat
      .if   {x} > 30
            .exit
      .endif
      .byte {x}
      x:={{x}*2}
.endr
```

will generate the code

```
.byte  1
.byte  2
.byte  4
.byte  8
.byte  16
```

6.12.11 .macro_on and .macro_off

Description: The **`.macro_on`** and **`.macro_off`** directives enable and disable macro-procedure expansions, respectively, in selective parts of the source text.

Example:

```
.macro       addd   op1,op2
  bsr    count_additions
 .macro_off
  addd       {op1},{op2}
 .macro_on
 .endm
```

the following macro-procedure call:

```
addd   r1,r2
```

will generate:

```
bsr    count_additions
addd   r1,r2
```

6.12.12 .include

Syntax: **`.include`** *included_file*

where: *included_file* is an existing file name

Description: The `.include` directive allows for the inclusion of text from another file as part of the file being assembled.

Example: `.include filehdr.h`

NOTE: If the *included_file* does not contain the full directory path of the file to be included, the assembler will search for it in either the current directory, or in a directory specified with the -MI invocation option (macro include directory).

6.12.13 .mwarning

Syntax: **`.mwarning`** *warning_message*

Description: The `.mwarning` directive generates an assembler warning message.

Example:

```
xx:= 222
.mwarning  current value of "xx" is : {xx}.
```

`.mwarning` is used to write the current value of macro-variable `xx` to the listing output. The assembler will issue the following warning message:

```
Assembler (Macro-Processor): "filename.s" , line 2 ,
WARNING: current value of "xx" is : 222
```

6.12.14 .merror

Syntax: `.merror` *error_message*

Description: The directive `.merror` generates an assembler error message.

Example:

```
.merror
Wrong value used for addr "address"
```

The assembler will issue the following error message:

```
Assembler (Macro-Processor) Error: "filename.s", line 1, statement is ==> .merror "err"
Wrong value used for addr "address"<== ERROR: Wrong value used for addr "address"
```

6.13 PROCEDURE SUPPORT DIRECTIVES

The procedure support directives enable the definition of assembly procedures and an easy interface with other assembler or HLL procedures. The assembler procedure handling conforms to the GNX standard calling convention.

The procedure support directives are as follows:

Directive	Function
`.proc`	defines an ordinary procedure
`.proct`	defines a trap procedure
`.proci`	defines an interrupt procedure
`.var`	starts definition of local variables for procedure
`.begin`	starts the body of the procedure
`.endproc`	ends the procedure definition
`.call`	calls an assembler or an HLL procedure

6.13.1 .proc

Syntax: `.proc`

Description: The `.proc` directive starts an ordinary procedure definition. It marks both the definition point of the procedure and the beginning of the parameter block definition.

Example:

```
p1:      .proc            # define procedure, p1
par1:    .blkd            # double-word parameter, par1
par2:    .blkw            # two-byte parameter, par2
         .begin           # starts procedure body
           .                  .
           .              # procedure body
           .                  .
         .endproc         # ends procedure body
```

6.13.2 .proct

Syntax: `.proct`

Description: The `.proct` directive starts the definition of a trap procedure. The body of a trap procedure is exited via the `rett` instruction.

Example:

```
trap_proc:      .proct          # define trap procedure
                .begin          # starts body
                   .                .
                   .            # trap procedure body
                   .                .
                .endproc        # exit via rett
```

6.13.3 .proci

Syntax: `.proci`

Description: The `.proci` directive starts the definition of an interrupt procedure. The body of an interrupt procedure is exited via the `reti` instruction.

Example:

```
int_proc:       .proci          # defines interrupt procedure
                .begin          # starts body
                   .                 .
                   .            # interrupt procedure body
                   .                 .
                .endproc        # exit from int_proc here is t
                                # the reti instruction
```

6.13.4 .var

Syntax: `.var` [*reglist*]

where: *reglist* is a list of the registers to be saved upon procedure entrance and restored upon procedure exit. The registers are specified within brackets, and seperated by commas.

Description: The `.var` directive starts the local variable block definition. It also ends the parameter block definition started with the .proc, .proct or .proci directives.

Example:

```
p1:     .proc           # starts definition of procedure, p1
par1:   .blkd           # a double-word parameter
        .var [r4,r5]    # starts local variable block description
var1:   .blkd           # a double-word variable
var2:   .blkw           # a two-byte variable
        .begin          # starts procedure body
           .            # implied saving r4 and r5
           .                 .
           .            # procedure body
           .                 .
        .endproc        # ends procedure body
                        # restoring r4 and r5 implied
```

6.13.5 .begin

Syntax: `.begin`

Description: The `.begin` directive begins the procedure body. It also ends the variable block definition.

The `.begin` directive develops into an enter sequence for entering the procedure body, saving registers specified with the `.var` directive; and allocates stack area for the local variables.

Example:

```
p1:     .proc
        .var [r2]
var1:   .blkd           # a double-word variable
var2:   .blkd           # a double-word variable
        .begin          # starts procedure body
           .            # saves r2
           .            # allocates 8 byte stack area for var1 and var2
           .                 .
           .                 .
           .            # procedure body
           .                 .
           .                 .
        .endproc        # ends procedure body
```

6.13.6 .endproc

Syntax:	`.endproc` [*return_value* [: *return_size*]]	
where:	*return_value*	is an optional value to be returned from the procedure.
	return_size	is an optional size specification for the *return_value*. It can be either of the specifications: **b**, **w**, **d**, **f** or **l**.

Description: The `.endproc` directive ends the procedure definition. It also marks the end of the procedure body and develops into an exit sequence from it.

The exit sequence may prepare a return value; it releases stack area allocated for local variables; restores saved registers; and returns from the procedure using `cxp`, `rett` or `reti`, depending on the procedure type.

Example:

```
p1:     .proc
        .var          [r3,r4,r5]
var1:   .blkw
        .begin                     # starts procedure body
           .
           .                       # procedure body
           .
        .endproc      var1:d       # ends procedure body
                                   # prepares return value var1 in r0
                                   # releases stack area for var1
                                   # restores r3, r4, r5
                                   # returns to caller through the
                                   #  ret instruction
```

6.13.7 .call

Syntax: `.call` *proc_name* [*param_1:x:y, ... param_n:x:y*]

where:	*proc_name*	is the name of the procedure to be called.
	param	is an actual parameter for the called procedure.
	x,y	are size specifications for the actual and formal parameters, respectively. They can be any of the following specifications: **b**, **w**, **d**, **f**, or **l**.

Description: The `.call` directive calls the procedure *.proc_name* with the specified parameters *param_1* through *param_n*, adhering to the GNX standard calling convention.

Example:

```
.call      cproc, r3:d, $50:d
```

Calls procedure `cproc` with two double-word parameters: r3 and the immediate value : 50.

Chapter 7

PROCEDURE SUPPORT

7.1 INTRODUCTION

A procedure is a sequence of instructions that can be called from several different places in a program. After a called procedure has finished executing, it returns control to the caller.

Assembly procedures can be defined and called within assembly code. Symbolic parameters and local variables may be defined and used within each assembly procedure, enabling easy, maintainable, and well structured assembly programming. The GNX Assembler conforms to the standard GNX calling convention, thus making the interface with high-level-language written code easy.

7.1.1 Procedure Operation

The following steps are performed in the call and execution of an assembly procedure:

1. The caller pushes parameters onto the stack.
2. The caller passes execution control to the first instruction of the procedure.
3. The procedure saves the contents of the specified general purpose registers and allocates storage for the local variables on the stack.
4. The procedure's code is executed.
5. The procedure stores a possible return value in `r0` or `f0`.
6. The procedure releases the storage allocated for local variables, restores the contents of the saved general purpose registers by popping them off the stack, and returns control to the caller.

7.2 PROCEDURE DEFINITION

Syntax:

```
procedure: procedure_head
         [parameter_block]
     [ [.var            [reglist]
         [local_var_block]  ]
         .begin
         [procedure_body]
         .endproc          [return_value:[return_size]]
```

where:	*procedure*	is an assembly label defining the procedure name. Should appear within a text section. If a double colon (::) is used instead of a single colon (:), the procedure is defined as global.
	procedure_head	begins the procedure definition. The directives `.proc`, `.proct` or `.proci` should be used for defining either an ordinary procedure, a trap procedure, or an interrupt procedure, respectively.
	parameter_block	is a definition of the procedure's formal parameters. It consists of storage allocation statements of the form: [*param_name* :] *.blkx* [*block_size*] *.blkx* can be any storage allocation directive (`.blkb`, `.blkw`, `.blkd`, `.blkf`, or `.blkl`).
	`.var`	is a directive that specifies the beginning of the local variable block.
	reglist	is an optionally specified list of general purpose registers to be saved upon entering the procedure and restored upon exiting.
	local_var_block	is a definition of the procedure's local variables. It consists of storage allocation statements of the form: [*var_name* :] *.blkx* [*block_size*] *.blkx* can be any storage allocation directive (`.blkb`, `.blkw`, `.blkd`, `.blkf`, or `.blkl`).

`.begin`	starts the procedure body. It generates an enter sequence for register saving and local variables storage allocation.
procedure_body	assembly statements that constitute the actual procedure code to be executed.
`.endproc`	ends the procedure body. It generates an exit sequence for releasing local variables, restoring saved registers, and exiting the procedure.
return_value	is an optional return value to the procedure.
return_size	is an optionally specified size for *return_value*. It can be one of the following specifications: **b**, **w**, **d**, **f**, or **l**.

7.3 PROCEDURE TYPES

Three types of procedures are supported by the GNX assembler:

- Ordinary procedures and functions
- Trap handler procedures
- Interrupt handler procedures

For each procedure type, a different exit instruction is generated when the `.endproc` directive is encountered.

Ordinary procedures or functions are specified by the `.proc` directive. They should be called with the `bsr` instruction. The exit sequence uses the `ret` instruction. When the assembler is invoked using the modularity option, the procedures should be called using the `cxp` instruction, and the `rxp` instruction is used instead of the `ret` instruction.

The trap handler procedure is specified by the `.proct` directive. The exit sequence uses the `rett` instruction.

The interrupt handler procedure is specified by the `.proci` directive. The exit sequence uses the `reti` instruction.

The GNX calling convention defines standards for using registers within different procedure types. These standards are discussed in detail in Section 7.7.

7.4 CALLING A PROCEDURE

A procedure can be called using the `.call` directive.

Syntax:

.call *proc_name* [,*actual_param* [:*x*[:*y*]],...]

where:		
	proc_name	is the name of the procedure to be called.
	actual_param	is an actual value that is passed to the called procedure. It is any legal assembly expression. Each parameter can be either an integer or a floating-point.
	x	is a size specification for an actual parameter. It can be either **b**, **w** or **d** for integer values; and either **f** or **l** for floating-point values.
	y	is a size specification for the formal parameter as appears in the procedure definition. It can be either **b**, **w** or **d** for signed integer parameters; either **ub**, **uw** or **ud** for unsigned integer parameters; and either **f** or **l** for floating-point parameters. For both integer and floating-point parameters, the size of the actual parameter must not be greater than the size of the formal parameter.

Description: The `.call` directive develops into a calling sequence that prepares parameters on the stack and calls the procedure. A more detailed description follows below.

7.4.1 The Calling Sequence

In the normal calling sequence, parameters are pushed on top of stack (`tos`) using the `mov` instructions, the procedure is called using a call instruction, and on the return from the call the parameters are released from the stack using the `adjsp` instruction.

This sequence is generated when either optimization is off, or when optimization is on but the `.call` directive appears outside a procedure definition.

Example:

The call

```
.call cmul, opd1:b:d, opd2:f:l
```

develops into

```
movfl  opd2, tos          # prepare opd2
movxbd opd1, tos          # prepare opd1
bsr    cmul               # call cmul
adjspb $-12               # release parameter storage from stack
```

7.4.2 Optimizing the Calling Sequence

The generated calling sequence can be optimized using the assembler optimization option (-O on UNIX, /OPTIMIZE on VMS). Optimized code is generated based on the location of the `.call` directive in your program code.

When the `.call` directive appears inside a procedure definition, a special scratch area is allocated on the top of stack upon entrance to the procedure containing the call. The `.call` directive moves parameters into the special scratch area using `mov` instructions and calls the procedure. The scratch area is released only upon exit from the procedure containing the call. This reduces the number of `adjsp` instructions normally generated for each procedure call to a minimum, thereby improving performance.

Example:

The call

```
.call cmul, opd1:d, opd2:d
```

develops into

```
                    # at this point, scratch area of at least 8 bytes
                    # should be allocated on top of stack.
movd opd2, 4(sp)    # prepare opd2
movd opd1, 0(sp)    # prepare opd1
bsr  cmul           # call cmul
                    # scratch area should be released from the top
                    # of stack later on.
```

Note: Use of the debug invocation option (-g on UNIX, /DEBUG on VMS) suppresses procedure optimization since there is no frame when optimization is on.

7.4.3 Passing Parameters

Parameters are prepared on top of stack, from right to left (i.e., last specified actual parameter is innermost from top of stack). Parameters are prepared on the stack according to the parameters' sizes as specified with the `.call` directive. No implicit stack alignment is done by the assembler. If the formal parameter size (*y*) is not specified, it is assumed to always be the same as the actual parameter size (*x*). When no size is specified for a parameter, the default **l** is used for immediate floating-point values; the default **d** is used for any other value.

Example:

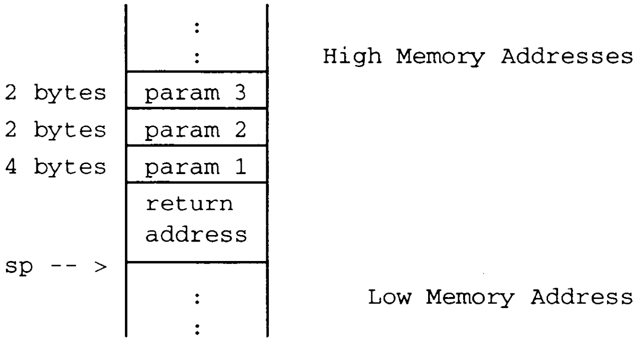
The call

```
.call cproc, var1:d, var2:w, var3:b:w
```

develops into

```
movxbw  var3, tos  # push var3 on top of stack
movw    var2, tos  # push var2 on top of stack
movd    var1, tos  # push var1 on top of stack
bsr     cproc
```

After `cproc` is called using the `bsr` instruction, the stack layout is



You must ensure type and size consistency between actual parameters specified for the `.call` directive and the corresponding formal parameters in the procedure definition. Further, you must verify that when interfacing HLL written procedures, integer parameters are double-word aligned, and float parameters are 8-byte aligned, as defined by the standard GNX calling convention (see Appendix E).

Example:

Calling the HLL `printf` procedure:

```
print_byte:    .proc
byte_param:    .blkb          # a one byte parameter
               .begin
                              # printf expects aligned parameters
               .call _printf, $format_string: d, byte:b:ud
               .endproc
               .data
format_string: .ascii "%d\n\0"
```

7.4.4 The Call Instruction

The call instruction generated for the `.call` directive is dependent on the assembler invocation line. If the modularity option is specified (-X on UNIX, /MODULAR on VMS), the `cxp` instruction is generated. Otherwise, the `bsr` instruction is generated.

Example:

The call

```
        .call cproc
```

normally uses the `bsr` instruction

```
        bsr cproc
```

When using 32000 modularity, the `.call` directive uses the `cxp` instruction

```
        cxp cproc
```

7.5 THE PARAMETER BLOCK

The parameter block defines formal parameters for the procedure. It consists of storage allocation statements. Each statement either defines a parameter or is an alignment statement.

Syntax for parameter definition:

param_name : *.blkx* [*expression*]

Syntax for an alignment statement:

{*.blkx*, .align, .space} [*expression*]

where:		
	param_name	is an assembly label defining a parameter name.
	.blkx	can be any storage allocation directive (`.blkb`, `.blkw`, `.blkd`, `.blkf`, or `.blkl`).

Description: Parameter definitions and alignment statements constitute the parameter block. All parameter definitions and alignment statements must be specified as one contiguous block between the `.proc` and the following procedure directive (the `.var` or `.begin` directives). This block should not be broken by any other segment.

Each parameter has a size and type associated with it, based on the storage allocation directive specified. The following storage allocation directives are used:

`.blkb` specifies a one byte integer.
`.blkw` specifies a two byte integer.
`.blkd` specifies a four byte integer.
`.blkf` specifies a four byte (single-precision) floating-point.
`.blkl` specifies an eight byte (double-precision) floating-point.

Example:

```
par1:   .blkw           # 2-byte integer parameter named par1.
        .align   4      # align par1 to double-word.
par2:   .blkf           # 4-byte floating-point parameter named par2.
        .space   4      # align par2 to quad-word.
par3:   .blkd           # 4-byte integer parameter named par3.
ERROR in line number 441 incorrect number of fields
line is: .if
```

7.5.1 Parameter Allocation

Parameters are allocated on the stack by the caller; the right parameter is located intermost from the top of stack. The assembler addresses the parameters as either sp-relative or fp-relative addresses.

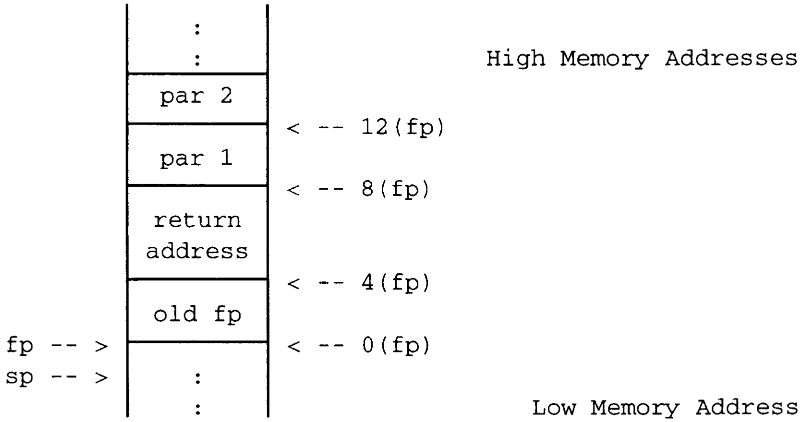
Normally, the assembler uses the fp-relative addressing mode. When invoked with the optimization option (-O on UNIX, /OPTIMIZE on VMS), the assembler uses the sp-relative addressing mode since the frame is not used. When the debug option (-g on UNIX, /DEBUG on VMS) is used together with the optimization option, procedure optimization is suppressed and the fp-relative addressing mode is used.

Example:

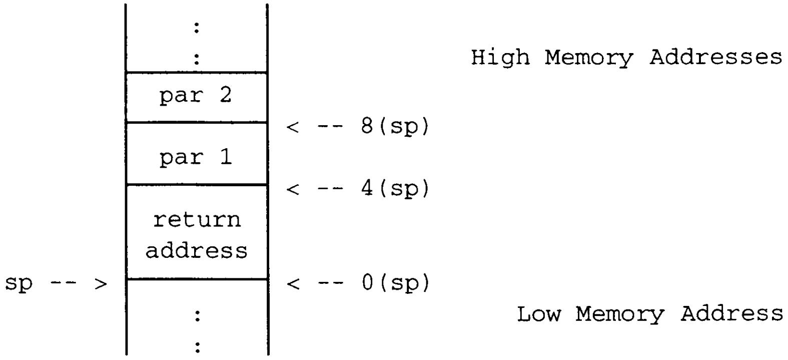
For the procedure definition

```
p1:      .proc
par1:    .blkd
par2:    .blkd
         .begin
         addr         par1,r1
         .endproc
```

`par1` is normally addressed as 8(fp). The stack layout is



When procedure optimization is on, `par1` is addressed as 4(sp). The stack layout is



Refer to Section 7.9 for more details on stack usage.

7.5.2 Parameter Alignment

You must ensure type and size consistency between formal parameter definitions and the corresponding actual parameters as specified in the procedure call.

The assembler does not implicitly align each of the procedure parameters. When interfacing HLL code, it is your responsibility to align integer parameters to double-word and floating-point parameters to eight byte addresses on the stack, as defined by the GNX standard calling convention (see Appendix E).

Example:

Calling an assembly procedure from a C module

```
char   c = '\1' ;
short   s = 2 ;
int     i = 3 ;
float   f = 4.0 ;
double  d = 5.0 '

c_func ( )
{
         asm_proc (c,s,i,f,d);
}
```

The corresponding assembly procedure definition is

```
_asm_proc::      .proc
par_c:           .blkd          # aligned to double-word
par_s:           .blkd          # aligned to double-word
par_i:           .blkd          # aligned to double-word
par_f:           .blkl          # aligned to long float
par_l:           .blkl          # aligned to long float
                 .var
                 . . .
                 .begin
                 . . .
                 any user code
                 . . .
                 .endproc
```

7.5.3 Parameter Block Size

After a parameter block has been defined, the parameter block size value is available through the predefined variable `param_size`. This value can be used throughout the procedure's definition, starting with the `.var` directive through the `.endproc` directive.

7.5.4 Parameter Scope

Parameters can only be referenced within the body of the procedure in which they are defined. Procedure parameters need not be uniquely named among different procedures.

7.6 THE VARIABLE BLOCK

The variable block defines the local variables for the procedure. It consists of storage allocation statements. Each statement defines a parameter or is an alignment statement.

Syntax for variable definition:

var_name : *.blkx* [*expression*]

Syntax for an alignment statement:

{*.blkx*, .align, .space} [*expression*]

where:	*var_name*	is an assembly label that defines the local variable name.
	.blkx	can be any storage allocation directive (`.blkb`, `.blkw`, `.blkd`, `.blkf`, or `.blkl`).

Description: Variable definitions and alignment statements constitute the variable block. All variable definitions and alignment statements must be specified as one contiguous block between the `.var` and the `.begin` directives. This block should not be broken by any other segment. If the procedure has no variables, the `.var` directive can be omitted.

Each variable has a size and type associated with it, based on the storage allocation directive specified. The following storage allocation directives are used:

`.blkb` specifies a one byte integer.
`.blkw` specifies a two byte integer.
`.blkd` specifies a four byte integer.
`.blkf` specifies a four byte (single-precision) floating point.
`.blkl` specifies an eight byte (double-precision) floating point.

Example:

```
var1:    .blkw          # 2-byte integer variable named var1.
         .align  4      # align to double-word.
var2:    .blkb          # 1-byte integer variable named var2.
         .align  4      # align to double-word.
var3:    .blkf          # 4-byte floating-point variable named var3.
```

7.6.1 Variable Allocation

Local variables are allocated on the stack upon entering the procedure body; the first variable is located intermost from the top of stack. The assembler addresses them as either sp-relative or fp-relative addresses.

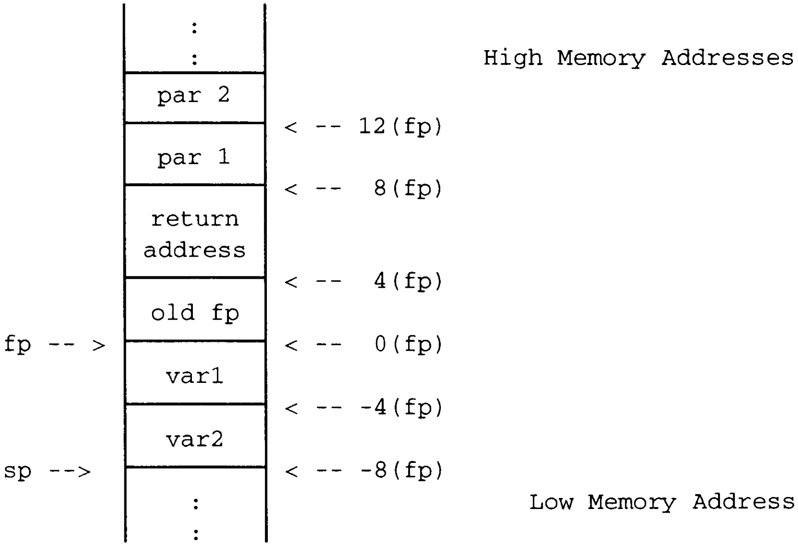
Normally, the assembler uses the fp-relative addressing mode. When invoked with the optimization option (-O on UNIX, /OPTIMIZE on VMS), the assembler uses the sp-relative addressing mode since the frame is not used. When the debug option (-g on UNIX, /DEBUG on VMS) is used together with the optimization option, the procedure optimization is suppressed and the fp-relative addressing mode is used.

Example:

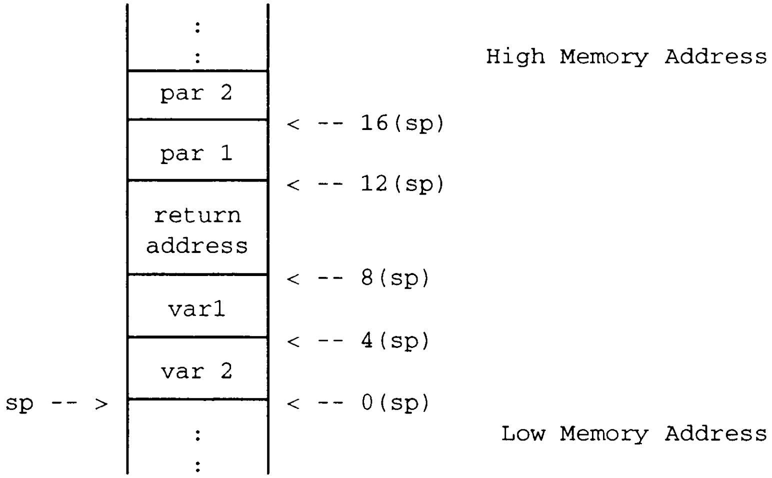
For the procedure

```
p1:       .proc
par1:     .blkd
par2:     .blkd
          .var
var1:     .blkd
var2:     .blkd
          .begin
          .endproc
```

`var1` will normally be addressed as -4(fp). The stack layout will be



When the procedure optimization is on, `var1` will be addressed as 4(sp). The stack layout will be



Refer to Section 7.9 for more details.

7.6.2 Variable Alignment

The assembler does not implicitly align each of the procedure variables. It is your responsibility to align local variables according to your target bus width in order to achieve better performance at run time. However, the whole variable block is always double-word aligned.

Note that the assembler assumes the stack is properly aligned upon entering a procedure. Therefore, you must ensure the proper alignment of parameters before entering the procedure.

Example:

```
        .var
var1    .blkw       # allocates 2 bytes for var1
var2:   .blkb       # allocates 1 byte for var2
        .begin
```

The complete variable block will occupy 4 bytes since it is double-word aligned.

7.6.3 Variable Block Size

After a variable block has been defined, the variable block size value is available through the predefined variable `var_size`. This value can be used throughout the procedure's body, starting with the `.begin` directive through the `.endproc` directive.

7.6.4 Variable Scope

Variables can only be referenced within the body of the procedure in which they are defined. They need not be uniquely named among different procedures.

7.7 REGISTER USAGE

There is no special support for the usage of registers within a procedure. Registers may be used throughout the procedure, provided they are used consistently, and saved and restored when necessary.

When using the GNX standard calling convention, certain rules apply for the use of registers. Volatile registers (r0-r2, f0-f3) can be freely modified within an ordinary assembly procedure; non-volatile registers (r3-r7, f4-f7) should be saved when a procedure is entered if they are to be modified. This also means that before calling a procedure that conforms to the GNX calling covention:

1. You should save volatile registers whose values you wish to keep, since they may be changed in the called procedure.
2. You do not need to save non-volatile registers, since they are guaranteed to be saved in the called procedure.

Both volatile and non-volatile registers should be saved when entering a trap or an interrupt procedure if they will be modified within the procedure (see Appendix E for a complete description of the GNX calling convention).

You should save registers when entering a procedure, and restore them when exiting a procedure, by using the *reglist* option of the `.var` directive.

Syntax:

.var [*reglist*]

where:	*reglist*	is a list of registers to be saved upon procedure entrance (`.begin`) and to be restored upon procedure exit (`.endproc`). The registers are specified within brackets, and separated by commas.

Description: The assembler uses the `enter` or `save` instruction for saving registers and the `exit` or `restore` instruction for restoring registers, depending on the assembler invocation options (for details see Sections 7.8.1 and 7.8.3).

Example:

For the following assembly procedure

```
Preg:   .proc
        .var     [r4,r5]
        .begin
        addd     r0,r1
        subd     r1,r4
        muld     r0,r5
        .endproc
```

r4 and r5 are saved when the `.begin` directive is encountered, and restored when the `.endproc` directive is encountered. The actual code is

```
enter   [r4,r5], $0   # save r4,r5
addd     r0,r1
subd     r1,r4
muld     r0,r5
exit    [r4,r5]       #restore r4,r5
```

7.8 THE PROCEDURE BODY

The procedure body is constructed from assembly statements between the `.begin` and `.endproc` directives:

.begin

[*procedure_body*]

.endproc [*return_value*[:*return_size*]]

7.8.1 Entering a Procedure Body

The `.begin` directive starts the procedure body. It develops into an enter sequence that saves the registers specified with the `.var` directive and allocates space for local variables on the stack. In the case of procedure optimization, the enter sequence allocates an additional scratch area on the stack for optimization purposes (as described below).

The assembler generates different enter sequences depending on the specified invocation options. Normally, the assembler generates an enter sequence that enables using the frame within the procedure body. Local variables are allocated on the frame and the *reglist* registers are saved on the stack using the `enter` instruction.

Example:

In the assembly procedure `p1`

```
p1:     .proc
par1:   .blkd
        .var      [r4]
var1:   .blkd               # local variable var1
        .begin
        addr      var1,r4
        .endproc  r4:d
```

the `.begin` directive develops into

```
enter    [r4],$4        # allocate stack frame area for var1
                        # and save register r4
```

When invoked with the optimization option, the assembler does not use the frame. Rather, the *reglist* registers are saved using the `save` instruction, and local variables are allocated using the `adjsp` instruction. In addition, if the procedure contains calls to other procedures using the `.call` directive, the same `adjsp` instruction allocates an additional scratch area for passing parameters to the subsequent calls.

Example:

In the assembly procedure p2

```
p2:     .proc
par1:   blkd
        .var        [r4]
var1:   .blkd
        .begin
        addr        var1, r4
        .call       p1, par1:d, r4:d
        .endproc    var1:d
```

The `begin` directive develops into

```
save    [r4]    # save register r4
adjspw  $12     # allocate stack area for var1 (4 bytes) and scratch
                  area for passing the 2 parameters to p1 (8 bytes)
```

7.8.2 Within a Procedure Body

The procedure body should contain assembly statements for execution. Such statements can symbolically reference the procedure's parameters and local variables. They can also reference global symbols. No symbolic reference is allowed to local parameters or variables of a different procedure.

Both symbolic references to parameters and local variables are interpreted as references to their addresses on the stack. These addresses may be fp-relative or sp-relative (see Sections 7.5.1 and 7.6.1).

When procedure optimization is on, the assembler assumes a fixed stack-pointer value throughout the procedure body. This value is used for referencing parameters and local variables as sp-relative addresses. Therefore, when using the optimization option, you should not alter the stack-pointer value within the procedure body (by using either the `adjsp, save, restore, enter, exit` instructions or the `tos` adressing mode).

Registers can be used throughout the procedure body as described in Section 7.7.

Transferring control from one procedure to another should be done using the `.call` directive. Nested procedure definition is illegal. The procedure body may be broken up by other segments.

7.8.3 Exiting a Procedure Body

The `.endproc` directive is provided for exiting a procedure.

Syntax:

.endproc [*return_value* [:*x*[:*y*]]

where:	`.endproc`	marks the end of procedure body and exits from it.
	return_value	is an optional value to be returned from the procedure.
	x	is a size specification for the source of the return value. It can be either **b**, **w** or **d** for integer values; and either **f** or **l** for floating-point values.
	y	is a size specification for the destination of the return value. It can be either **b**, **w** or **d** for signed integer values; either **ub**, **uw** or **ud** for unsigned integer values; and either **f** or **l** for floating-point values. For both integer and floating-point values, the source size must not be greater than the destination size.

Description: The `.endproc` directive ends the procedure body. It develops into an exit sequence for releasing the stack storage that was allocated upon procedure entrance. The `.endproc` directive also restores saved registers and returns from the procedure.

If a *return_value* is specified, a code preparing the return value precedes the exit sequence. The return value is prepared either in `r0`, `f0`, or `l0` according to the standard calling convention. Integer values are returned in `r0`. Floating-point values are returned either in `f0` or `l0`.

Note that registers are restored after the *return_value* is prepared. Therefore, the return value will be lost if `r0` is one of the restored registers (e.g. specified in *reglist* with the `.var` directive). Also note that when a floating point value, beeing a part of a HLL expression, is returned as a single-precision value in `f0`, it is expanded to a double-precision value by the HLL code after returnning from the assembly procedure.

The assembler generates different exit sequences depending on the specified invocation option. Normally, the assembler prepares *return_value* in `r0`, `f0`, or `l0` restores saved registers and releases the frame using the `exit` instruction, and returns from the procedure using a return instruction.

The return instruction generated for the exit sequence is dependent on the procedure type and the assembler invocation options. The `ret` instruction is generated for ordinary non-modular procedures. The `rxp` instruction is generated for ordinary modular procedures. The `rett` instruction is generated for trap procedures. The `reti` instruction is generated for interrupt procedures.

Example:

In the assembly procedure `p1`

```
p1:      .proc
par1:    .blkd
         .var     [r4]
var1:    .blkd               # local variable var1
         .begin
         addr     var1,r4
         .endproc r4:d
```

the `.endproc` directive develops into

```
movd     r4, r0            # prepare return value r4 in r0
exit     [r4]              # restores r4 and releases frame
ret      0                 # returns to caller
```

When invoked with the optimization option, the assembler prepares *return_value* in `r0` or `f0`, releases the allocated stack storage (including both the local variables and the scratch area storage) using the `adjsp` instruction. The assembler restores saved registers using the `restore` instruction, and returns from the procedure using the return instruction.

Example:

In the assembly procedure `p2`

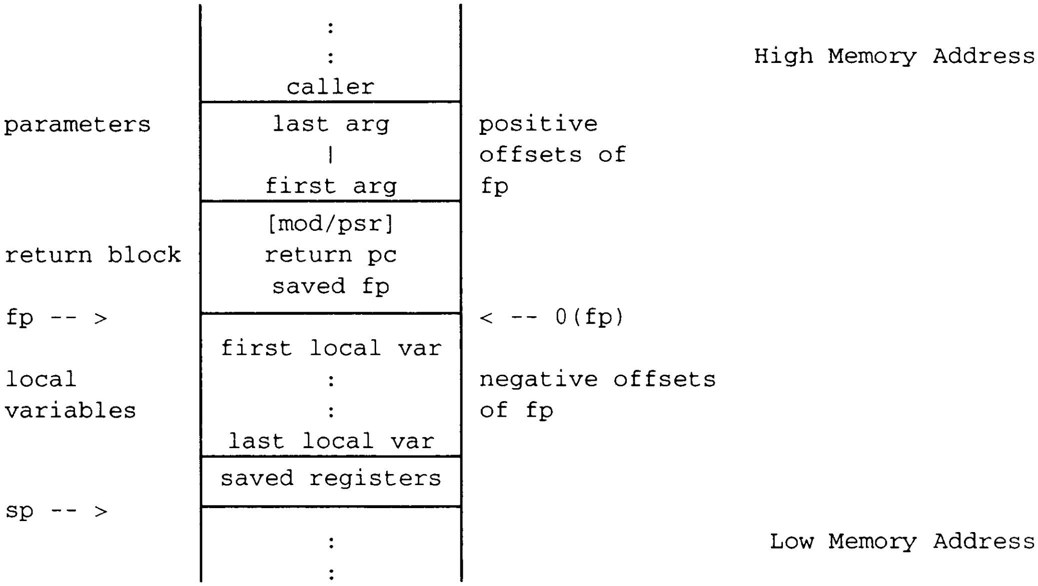
```
p2:     .proc
par1:   .blkd
        .var        [r4]
var1:   .blkd
        .begin
        addr        var1, r4
        .call       p1, par1:d, r4:d
        .endproc    var1:d
```

The `.endproc` directive develops into

```
movd    8(sp), r0    # prepare return value var1 in r0
adjspw  $-12         # release stack area for local variable
                     # and scratch area
restore [r4]         # restore r4
ret     $0           # return to caller
```

7.9 STACK USAGE

Normally within a procedure body the stack layout will be

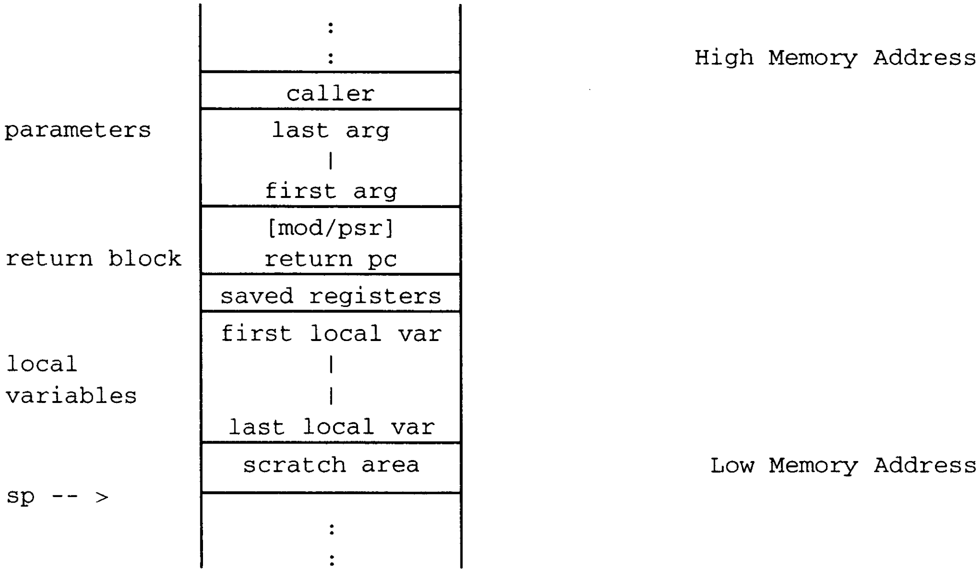


Parameters are prepared before calling the procedure, and are referenced within the procedure body as fp-relative addresses (*offset* (fp), *offset* being positive).

The return block is created by the `call`, `save` and `adjsp` instructions. When *Series 32000* modularity is used or in cases of trap or interrupt procedures, the `mod` and `psr` values (which are a total of four bytes), are pushed on the stack.

The procedure enter sequence save registers on stack. Local variables are allocated on the stack by the enter sequence, and are referenced within the procedure body as fp-relative addresses (*offset* (fp), offset being negative).

When procedure optimization is on, the stack layout will be



The frame is not used in this layout. Parameters are prepared before calling the procedure and are referenced within the procedure body as sp-relative addresses.

The return block is created by the `call`, `save` and `adjsp` instructions. When *Series 32000* modularity is used or in cases of trap or interrupt procedures, the `mod` and `psr` values (which are a total of four bytes), are pushed on the stack.

The procedure enter sequence saves registers on stack. Local variables are allocated on the stack by the enter sequence, and are referenced within the procedure body as sp-relative addresses.

The scratch area is a special stack storage. It is allocated once upon entering the procedure body, and released when the procedure body is exited. The scratch area is used for passing parameters to other procedures called with the `.call` directive. Therefore an area does not have to be allocated and released each time a subsequent procedure is called.

7.9.1 Sample Assembly Procedure

The procedure

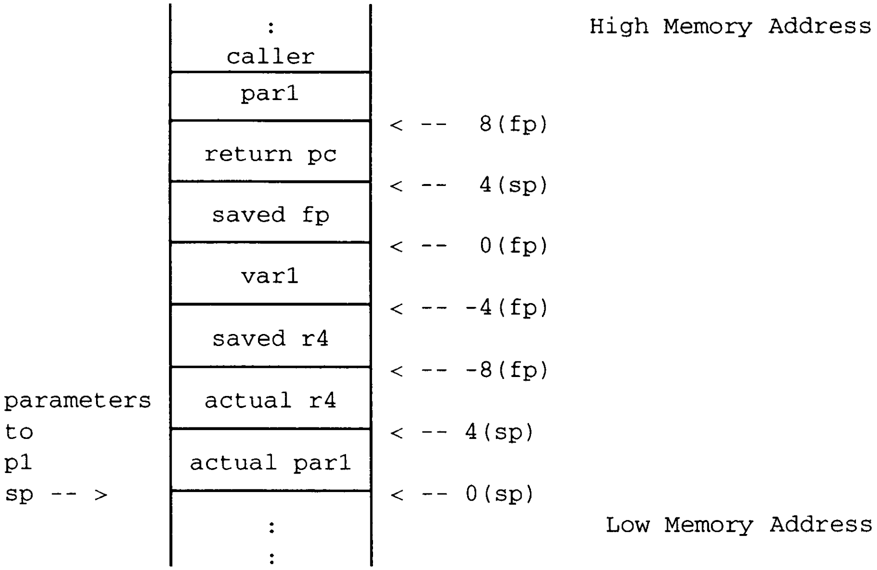
```
xproc:   .proc
par1:    .blkd
         .var [r4]
var1:    .blkd
         .begin
          addr var1, r4
         .call p1, par1:d, r4:d
         .call p2, $100:d
         .call p3, $20:d,  r0
         .endproc var1:d
```

will normally expand to

```
enter    [r4], $4        # save r4, allocate frame
addr     -4(fp), r4
movd     r4, tos         # push r4
movd     8(fp), tos      # push par1
bsr      p1              # call p1
adjspb   $-8             # releases parameter area of par1
movd     $100, tos       # push immediate value 100
bsr      p2              # call p2
adjspb   $-4             # release parameter area of p2
movd     r0, tos         # push r0
movd     $20, tos        # push immediate value 20
bsr      p3              # call p3
adjspb   $-8             # release parameter area of p3
addd     r0, -4(fp)      # update var1
movd     -4(fp), r0      # prepare return value of r1
exit     [r4]            # restore r4, release frame
ret      $0              # return
```

In this example, an `adjsp` instruction is generated for every procedure called with parameters within the `xproc` procedure.

Just before the bsr to `p1` the stack layout will be

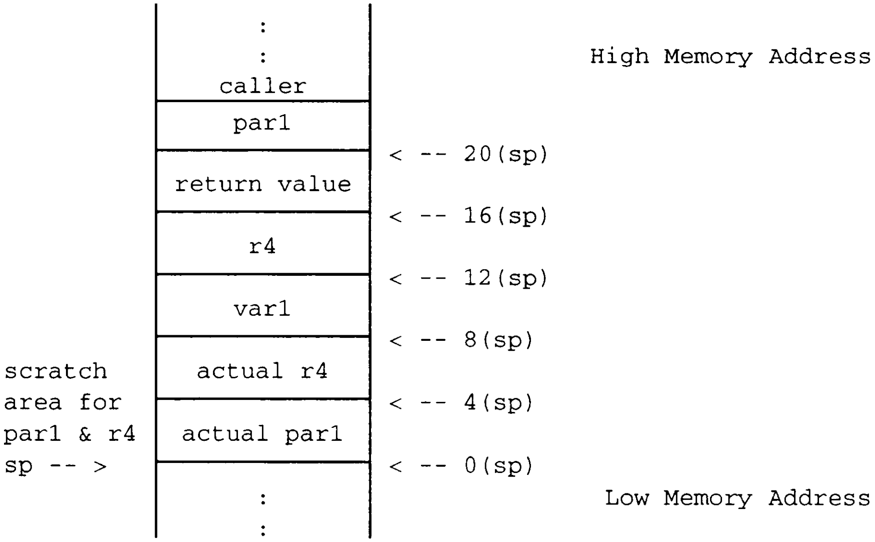


When procedure optimization is on, the procedure will expand to

```
xproc:  .proc
par1:   .blkd
        .var [r4]
var1:   .blkd
        save    [r4]            # save r4
        adjspw  $12             # allocate var1 and scratch area on stack
        addr    8(sp), r4
        movd    r4, 4(sp)       # pass r4 to p1
        movd    20(sp), 0(sp)   # pass par1 to p1
        bsr     p1              # call p1
        movd    $100, 0(sp)     # prepare immediate value 100
        bsr     p2              # call p2
        movd    r0, 4(sp)       # prepare r0
        movd    $20, 0(sp)      # prepare immediate value 20
        bsr     p3              # call p3
        addd    r0, 8(sp)       # update var1
        movd    8(sp), r0       # prepare return value of var1
        adjspw  $-12            # restore stack area
        restore [r4]            # restore r4
        ret     $0              # return
```

In this example, `adjsp` instructions are generated only for the entrance to and exit from the `xproc` procedure.

Just before the bsr to `p1` the stack layout will be



Chapter 8

MACRO AND CONDITIONAL ASSEMBLER

8.1 INTRODUCTION

The GNX macro-assembler makes writing assembly programs easier. It eliminates the need to rewrite similar assembly source code repeatedly, and simplifies program documentation. The conditional assembler feature allows for the inclusion or deletion of optional assembly statements. Other macro-assembler features help minimize programming errors and speed the development process.

The macro-assembler is automatically invoked by the assembler.

The macro-assembler described in this chapter is a completely new element for the GNX Version 4 Assembler. It is characterized by powerful and flexible features. This new macro-assembler is **not** compatible with the previously supplied macro-assembler (from Version 2.0 and up).

For compatibility purposes, the Version 2 macro-assembler is still supported in this release (but will be obsolete in Version 5). However it must now be invoked by the -MC invocation option for the UNIX environment, and by the /MCOMPATIBILITY invocation option for the VMS environment. See Appendix F for details.

8.1.1 Overviewof the Major Macro-Assembler Features

The GNX macro-assembler supports the following features:

- **Macro-procedures ("macros")**

  A macro-procedure is equivalent to the common term "macro". For example, for the following macro-procedure:

```
.macro move_bytes  source,dest,length

save    [r0,r1,r2]
movd    ${length},r0
addr    {source},r1
addr    {dest},r2
movsb
restore [r0,r1,r2]

.endm
```

the macro-procedure calls:

```
move_bytes   aa,12(-8(fp)),1024

move_bytes   0(fp)[r7:b],0(r5),512
```

will generate the code:

```
save     [r0,r1,r2]
movd     $1024,r0
addr     aa,r1
addr     12(-8(fp)),r2
movsb
restore [r0,r1,r2]

save     [r0,r1,r2]
movd     $512,r0
addr     0(fp)[r7:b],r1
addr     0(r5),r2
movsb
restore [r0,r1,r2]
```

This feature is fully described in Section 8.10.

- **Conditional Code Generation**

  Code may be generated according to conditions tested in the macro-assembly phase. For example the sequence:

```
.if    {STR_EQ [{GNX_FPU}, ns32381]}
    logbf    f1, f2
.else
    movf     f2, tos
    movf     f1, tos
    bsr      _sqrtf
    adjspb   $-8
    movf     f0, f2
.endif
```

  will generate code using the `logbf` opcode if the predefined variable `GNX_FPU` holds the value `32381`. Otherwise a calling sequence to the subroutine `_sqrtf` will be generated.

  This feature is fully described in Section 8.8.

- **Macro Variables**

  String values may be assigned to macro-variables. These variables may be later utilized in place of the string value. For example:

```
base_reg:= r0

movqd   0, 0({base_reg})
```

is equivalent to :

```
movqd   0, 0(r0)
```

This feature is fully described in section 8.4.

- **Repetitive Code Generation**

  This feature allows for the easy repetition of sequences of statements, by specifying either:

  - the number of repetitions, as for example

```
.repeat  3,index
.align   4
.word    {index}
.endr
```

  which is equivalent to the code

```
.align   4
.word    1
.align   4
.word    2
.align   4
.word    3
```

  - or a repetition list

```
.irp     val, [25,3,1989]
.align   4
.word    {val}
.endr
```

  which is equivalent to the code

```
.align   4
.word    25
.align   4
.word    3
.align   4
.word    1989
```

  This feature is fully described in section 8.9.

- **Text Inclusion**

  Text from another file may be included as part of the file being assembled. For example:

```
.include    useful.definitions
```

  will place code that is the contents of file `useful.definitions`.

  This feature is fully described in section 8.12.

- **Listing of Expanded Code**

  A listing output may be produced to display all expanded code. The listing output can be generated after either the macro-processing phase or after the second phase of the assembly process.

  This feature is fully described in section 8.3.

- **User Error and Warning Messages**

  Error and warning messages can be issued by the user. For example, given the following macro:

```
.macro  check_reg_number    reg_number

.if   {reg_number} > 7
    .merror    invalid register number specified.
.endif

.endm
```

  the call:

```
check_reg_number   10
```

  will result in the error message:

```
Assembler (Macro-Processor) Error :
"filename.s", line 4, statement is ==>  .merror   invalid register number specified<==
. . . from line 9 : while calling "check_reg_number" with "ARG_LIST"=[10]

ERROR :   invalid register number specified
```

  These features are fully described in section 8.13.

- **Arithmetic Operations and Expressions**

  Arithmetic operations and expressions (including arithmetic comparisons) can be performed on constants and variables.

  For example, assuming the macro-variable `x` holds the string value `100`, the statement:

```
result:= { {x} * ({x} - 1) }
```

  is processed by the macro-assembler so that the macro-variable `result` will hold the string value `9900`.

  Arithmetic operations and expressions are fully described in section 8.5.

- **Built-in Macro Functions**

  Built-in macro-functions provide the following capabilities:

  1. Manipulation of strings. For example

```
{SUB_STR[abcde, 2, 3]}
```

is equivalent to the string `bcd`.

2. Manipulation of *macro-lists* (special strings used for implementing complex data structures). For example:

```
reg_list:={sublist[[R1,R5,R7],2,2]}
```

sets `reg_list` to hold R5 and R7.

3. Integer and floating-point conversions.

4. Manipulation of NS32000 instruction operand strings. For example

```
{OP_INDEX_REG[xx+9(sp)[r3:b]]}
```

is evaluated as a string specifying the index register `r3`.

8.2 THE MACRO-PROCESSING PHASE

Assembly source text is processed by the assembler in two distinct phases: the macro-processing phase and the assembly phase.

The macro-processing phase involves the reading and processing of source text statement by statement. Strings between braces (`{}`) are handled and replaced with the appropriate value. If the resulting statement is a macro-directive statement or a macro-procedure call it is acted upon. All other statements are not processed by the macro-processor and are passed directly to the assembly phase.

The assembly phase is performed in two passes and generates the appropriate output files.

A more detailed explanation of the various stages of the macro-processing follows below.

A string between braces is handled as follows:

- A macro-variable name is replaced with the current value of the variable. For example, if the variable `a` holds the value `xxy 100`, then `{a}` is replaced by `xxy 100`.
- An arithmetic macro-expression is evaluated and replaced with the result. For example, `{100*(10+10)}` is replaced by `2000`.
- A built-in macro-function call is evaluated and replaced by the result. For example, `{STR_LEN[abcde]}` is replaced by `5`.
- All other braced strings cause an error message to be issued.

Pairs of braces may be nested, in which case the string contained by the inner pair of braces is evaluated and replaced first.

For example, assuming the macro-variable `op` holds the value `r2` ,then the following statement

```
movd  $0,r{ {SUB_STR[ {op}, 2, 1]} + 4}
```

will be replaced by

```
movd  $0,r6
```

– First, `{op}` is replaced by `r2`.
– Then, the function call `{SUB_STR[r2, 2, 1]}` is replaced by `2`.
– Finally, `{2 + 4}` is arithmetically evaluated and replaced by the resulting string `6`.

Braces are not processed in the macro-processing phase if they appear within either an ascii constant (such as `"{1+1}"`) or a character constant (such as `'{'`).

A statement followed by a backslash (`\`) before a carriage-return (<CR>) is concatenated with its previous statement. The statements are then treated as a single statement (any number of lines may be concatenated in this way). This does not apply to comments. Comments will be terminated by a carriage return (<CR>) even if preceded by a backslash (`\`). This feature may be used in macros for breaking complex expressions into several lines. All error messages refer to the first concatenated statement.

Macro-directives and macro-procedure calls are handled as follows:

- When the opcode field is the name of a previously defined macro-procedure, the statement is considered a macro-procedure call.
  The statements in the macro-procedure body are processed, as if they were encountered at this point, after matching the actual arguments with the formal arguments of the macro-procedure.
- When the statement is of the form "*symbol* := *value*", the statement is considered a macro-variable assignment.
  The variable statement on the left side of the := is assigned the value specified on the right side.
- When the opcode field is a `.macro` directive, the statement is considered a macro-procedure definition.
  The statements following the `.macro` directive, but before the `.endm` directive, are read textually without being processed and are stored internally.
- When the opcode field is an `.if` directive, a conditional block is began.
  Statements following a true clause are processed; statements following an untrue clause are read textually without being processed and are discarded.
- When the opcode field is a `.repeat` or a `.irp` directive, a repetitive block is begun. The statements of the block are repetitively processed, according to the

operands of the `.repeat` or `.irp` directive.

- When the opcode field is a `.include` directive, the specified file is read and processed.

8.3 INVOCATION

Several aspects of macro-processing can be controlled by assembler invocation options. The following table presents these specific options:

FLAG (VMS)	FLAG (UNIX)	DEFINITION
/MCOMPATIBILITY	–MC	Invokes the Version 3 macro-assembler.
/MDEFINE= (*name*[=*def*][,...])	–MD*name* or –MD*name=def*	Defines name to macro assembler as if by macro assignment statement.
/MLIBRARY= (*libname*[,...])	–ML*filename*	Includes macro library file.
/MINCLUDE= (*directory*[,...])	–MI*dir*	Specifies an include search directory.
/MONLY	–MO	Invoke only macro-processing phase.
/MPRINT=*filename*	–MP*filename*	Prints the macro-processing output.

When the -MC option (/MCOMPATIBILITY on VMS) is used, version 4 macro-processing is suppressed. The old, version 3, macro-assembler is used instead. If this option is specified, it may not be combined with the other macro options described here.

The -MD option (/MDEFINE on VMS) assigns an initial value to a macro variable.

The -ML option (/MLIBRARY on VMS) includes an already existing macro-library. A macro-libarary is any valid assembly file using the Version 4 macro-assembler features; as described in Section 8.12 for the *included_file* of the `.include` directive.

The -MI option (/MINCLUDE on VMS) sets a search directory for included files. The assembler searches for .include files which do not begin with a slash (/) (or an open bracket ([) on VMS), in the directory of the specified input file first, then in the directory named in this option.

The -MO option (/MONLY on VMS) invokes only the macro-processing phase of the assembler.

The -MP option (/MPRINT on VMS) causes the assembler to print the macro-processor's output to *filename*. If *filename* is not given, the output is written to standard output on UNIX, or a .mac file on VMS.

8.4 MACRO VARIABLES

Macro-variables are variables that are active only during the macro-processing phase.

The name of a macro-variable may be any assembly symbol as defined in section 2.5, *i.e.* a sequence of letters, digits, underscores (_) and periods (.). The first character may not be a digit. A period (.) should not be used as the first character of the variable name since it may be confused with a directive name.

The initial value of any macro-variable is the empty string, unless it has been assigned a value on the invocation line (see section 8.3) through the -MD macro invocation option (/MDEFINE on VMS). Generally a macro-variable is assigned a value by the user through a macro-variable assignment statement.

The value of an undefined variable is the empty string.

Syntax:	*macro_var* := [*value*]
Description:	This assigns the value of the macro variable *value* to *macro_var*, after stripping leading and trailing blanks. If *value* is omitted, an empty string is assigned to *macro_var*.

A macro-variable is substituted with its current value when its name is enclosed within braces.

Syntax: { *macro_var* }

Examples:

1. `AAA := 5+5`

   assigns the string value `5+5` to the macro-variable `AAA`.

2.
   ```
   XXX := 7
   XXX := {XXX}+1
   ```

   assigns the string value `7+1` to the macro-variable `XXX`.

3.
   ```
   XXX := 7
   XXX := {{XXX} + 1}
   ```

   assigns the value `8` to the macro-variable `XXX`.

4.
   ```
   VAR_NAME:= XXX
   {VAR_NAME} := 7
   ```

   assigns the value `7` to the macro-variable `XXX` .

5. `EEE := eee`
   `FFF := fff`
   `LLL := [ ddd, {EEE}, {FFF}, ggg]`

   assigns the value of the macro-list `[ ddd, eee, fff, ggg]` to the macro-variable `LLL`.

8.5 ARITHMETIC MACRO-EXPRESSIONS

An arithmetic macro-expression is a string whose contents are a legal combination of integer constants, arithmetic operators, comparison operators and parentheses. This string can be evaluated as an integer value.

Examples of various arithmetic macro-expressions are:

1. `1000`
2. `20+8*(3/2)`
3. assuming that the value of `a` is `50` and that the value of `b` is `+`, then:

   `{a} {b} 27`

   is also a legal arithmetic macro-expression (equivalent to 50 + 27).

When an arithmetic macro-expression is enclosed between braces or used in an arithmetic context (for example, the clause of a `.if` / `.elsif` directive or as the first operand of a `.repeat` directive), it is evaluated by the macro-processor and substituted with a string representing its value. This string contains an integer constant in signed decimal notation with no leading blanks. Arithmetic macro-expressions are evaluated and converted by the macro-assembler to a 32-bit signed integer representation. All arithmetic operations are performed on 32-bit signed integer operands, and also return a 32-bit integer value.

Each arithmetic macro-operator in a macro-expression has a level of precedence. This determines the macro-expression's order of evaluation. Table 8-1 lists all the macro-operators and their precedence for evaluation.

The user must follow these rules when writing arithmetic macro-expressions:

1. All unary operators must precede a single term and cannot be used to separate two terms.
2. All binary operators must separate two terms. For example, the macro-expression **`8*4`** is legal, but **`8**4`** is illegal.

Table 8-1. Macro Operator Precedence

PRECEDENCE	OPERATOR	NAME	DESCRIPTION OF OPERATION
Unary Operator			
1	-	Unary minus	Two's complement (= negation).
1	~	Unary complement	One's complement.
Binary Operator			
2	*	Multiply	Multiply 1st term by 2nd term.
2	/	Divide	Divide 1st term by 2nd term.*
2	%	Modulus	Remainder from 1st term divided by 2nd term.**
2	<<	Shift left	Shift 1st term by 2nd term; emptied bits are zero-filled.
2	>>	Shift right	Shift 1st term by 2nd term; emptied bits are zero-filled.
2	~	Logical OR / complement	Bit-wise OR of 1st term and one's complement of 2nd term.
3	&	Logical AND	Bit-wise AND of 1st and 2nd terms.
3	|	Logical OR	Bit-wise OR of 1st and 2nd terms.
3	^	Logical XOR	Bit-wise XOR of 1st and 2nd terms.
4	+	Add	Add 1st and 2nd terms.
4	-	Subtract	Subtract 2nd term from 1st term.
5	=	Equal	1 if 1st and 2nd terms are equal, 0 otherwise
5	<>	Not Equal	1 if 1st and 2nd terms are not equal, 0 otherwise
5	>	Greater Than	1 if 1st term is greater than 2nd term, 0 otherwise
5	<	Less Than	1 if 1st term is less than 2nd term, 0 otherwise
5	>=	Greater or Equal	1 if 1st term is greater than or equal to 2nd term, 0 otherwise
5	<=	Less or Equal	1 if 1st term is less than or equal to 2nd term, 0 otherwise

* Rounds toward 0, *e.g.*, -7/3 = -2 and 7/3 = 2

** *e.g.*, -7%3 = -1 and 7%3 = 1.

3. Compound macro-expressions are valid. A macro-expression may be constructed from other macro-expressions using unary and binary operators. For example, the two individual macro-expressions `{A}+1` and `{B}+2` may be combined with a multiply operator and parentheses to form the single macro-expression `({A}+1)*({B}+2)`. Note that the parentheses override the default precedence rules.
4. Evaluation of a macro-expression is governed by three factors:
   - *Parentheses* – macro-expressions enclosed in parentheses are evaluated first. For example, `{8/4/2}` is evaluated as `1`, but `{8/(4/2)}` is evaluated as `4`.
   - *Precedence Groups* – an operation of a higher precedence group is evaluated before an operation of a lower precedence group whenever parentheses do not otherwise determine the evaluation order. For example, `{8+4/2}` is evaluated as `10`, but `{8/4+2}` is evaluated as `4`.
   - *Left to Right Evaluation* – macro-expressions are evaluated from left to right whenever parentheses and precedence groups do not determine evaluation order. For example, `{8*4/2}` is evaluated as `16`, and `{8/4*2}` is evaluated as`4`.

8.6 MACRO LISTS

A macro-list is a sequence of strings separated by commas and enclosed between brackets. Each string in the macro-list is called an element. An element of a macro-list may itself be a macro-list, allowing for multilevel macro-lists.

Macro-lists are useful for implementing macro data-structures (such as arrays, records, stacks) in conjunction with built-in functions that perform macro-list manipulations, such as search, insertion and deletion of elements (see Section 8.16). Some examples of various types of macro-lists are:

Examples:

1. `[]`

   a macro-list with no elements
2. `[xx,yy]`

   a macro-list with two elements: `xx` and `yy`.
3. `[a,,]`

   a macro-list with three elements: `a` and two empty strings.

4. `[[r1,r2],100]`

   a macro-list with two elements, a macro-list with two elements and the string `100`.

5. `[12[r2:w],@xx]`

   a macro-list with two elements.

8.7 BUILT-IN MACRO FUNCTIONS

The macro-assembler provides built-in functions to manipulate macro-strings, arithmetic constants, macro-lists and assembly operands.

The general syntax for calling a macro-function is:

Syntax: {*macro_func param_list*}

where:	*macro_func*	is the name of the function
	param_list	is a macro-list in which each element is a parameter to the function.

Leading and trailing blanks of parameters are stripped before processing the macro-function. The macro-function call is then evaluated and replaced by the result of the function call.

Example: The macro-function call

```
{SUB_STR[ abcde , 3 , 2]}
```

is replaced by the string `cd`.

Below is a list of available built-in functions. The macro-list and operand functions are advanced features of the macro-assembler and therefore may not be necessary for all users. For a detailed description of these functions see Sections 8.15 through 8.18.

String Functions:

- **{STR_LEN[** *string* **]}**
- **{STR_EQ[** *string1*, *string2* **]}**
- **{SUB_STR[** *string*, *start* [, *length*] **]}**
- **{STR_FIND[** *string*, *substring* **]}**

Macro-List Functions:

- `{LIST_GET[` *list, element_number* `]}`
- `{SUB_LIST[` *list, start* [*, length*] `]}`
- `{LIST_FIND[` *list, string* `]}`
- `{LIST_REPL[` *list, element_number, string* `]}`
- `{LIST_INS[` *list, string, element_number* `]}`
- `{LIST_DEL[` *list, element_number* `]}`
- `{LIST_LEN[` *list* `]}`

Data Conversion Functions:

- `{CNV_HEX[` *integer_constant* `]}`
- `{CNV_HEXF[` *constant* `]}`
- `{CNV_HEXL[` *constant* `]}`

Instruction Operand Functions:

- `{OP_TYPE[` *operand* `]}`
- `{OP_REG[` *operand* `]}`
- `{OP_DISP1[` *operand* `]}`
- `{OP_DISPSIZE1[` *operand* `]}`
- `{OP_DISP2[` *operand* `]}`
- `{OP_DISPSIZE2[` *operand* `]}`
- `{OP_VAL[` *operand* `]}`
- `{OP_VALSIZE[` *operand* `]}`
- `{OP_LIST[` *operand* `]}`
- `{OP_IS_INDEXED[` *operand* `]}`
- `{OP_INDEX[` *operand* `]}`
- `{OP_INDEX_BASE[` *operand* `]}`
- `{OP_INDEX_REG[` *operand* `]}`
- `{OP_INDEX_SCALE[` *operand* `]}`

8.8 CONDITIONAL ASSEMBLY

Sequences of statements may be generated according to conditions tested during the macro-processing phase.

8.8.1 Conditional Block

Syntax:

`.if` *if_condition*
 if_conditional_body

[**`.elsif`** *elsif_condition*

 elsif_conditional_body]...

[**`.else`**

 else_conditional_body]

`.endif`

where: *if_condition* and *elsif_condition*(s)are arithmetic macro-expressions.

Description: A condition evaluated by the macro-assembler as a non-zero value is considered to be true. See Section 8.5 for details on macro-expression evaluation.

In a conditional block the *if_condition* argument is evaluated first, and only if found to be true the statements in *if_conditional_body* are processed. If the *if_condition* is found to be false, the *elsif_condition*(s) arguments are evaluated until one of them is found to be true, in which case the corresponding *elsif_conditional_body* statements are processed. Otherwise, if an `.else` statement has been specified, the *else_conditional_body* statements are processed.

The types of statements that are allowed in *conditional_bodies* are valid assembly language statements, directives, macro-procedure call and macro-assembly directives, with all the conditional blocks, repetitive blocks and macro-procedure definitions being complete.

Example:

```
.if {reg_num} > 5
    movqd   5, r{reg_num}
.elsif {reg_num} > 3
    movqd   3, r{reg_num}
.else
    movqd   1, r{reg_num}
.endif
```

If `reg_num` holds the value `6` this is expanded to

```
movqd   5, r6
```

if `reg_num` holds the value `4` this is expanded to

```
movqd   3, r4
```

and if `reg_num` holds the value `0` this is expanded to

```
movqd   1, r0
```

8.9 REPETITIVE DIRECTIVES

The basic constructs of a repetitive block are:

.repeat [*iteration_count* [*, iteration_var*]]

repetitive_body

.endr

and

.irp *iteration_var, iteration_list*

repetitive_body

.endr

Repetitive blocks may appear inside a macro-procedure definition, in conditional blocks, and may be nested without limit.

The types of statements allowed in a repetitive block are valid assembly language statements, directives, macro-procedure calls, macro-assembly directives (except the `.macro` and the `.endm` directives) with all conditional blocks and repetitive blocks being complete.

8.9.1 .repeat Directive

Syntax: **.repeat** [*iteration_count* [*, iteration_var*]]

where:

iteration_count	specifies the number of iterations.
iteration_var	is a macro-variable name used as an iteration index.

Description: The *iteration_count* argument is evaluated by the macro-processor. If its value is positive, the code following the `.repeat` statement through the corresponding `.endr` statement, is processed *iteration_count* amount of times.
If given, the *iteration_var* argument holds a string representing the current iteration number for each iteration. It receives values from 1 to *iteration_count*. After the the processing of the repetitive block has been completed, it holds the *iteration_count* value.
If the *iteration_count* argument is evaluated as a negative or zero value, the statements in the block are read textually without being processed until an `.endr` directive is reached.
If the *iteration_count* argument is not given, then the repetitive block is

processed repeatedly until an `.exit` directive is processed (see Section 8.9.3).

Examples:

1.
```
.repeat        8, i
  movqd        0, r{(i} - 1}
.endr
```

generates code that clears `r0` through `r7`.

2.
```
.repeat 4
  nop
.endr
```

generates 4 consecutive `nop` instructions.

8.9.2 `.irp` Directive

Syntax: **`.irp`** *iteration_var*, *iteration_list*

where:	*iteration_var*	is a macro-variable name to be used as an iteration variable.
	iteration_list	is a macro-list.

Description: For each element in the *iteration_list* argument, the macro-processor assigns its string value to *iteration_var*, and process the code between the `.irp` statement and the corresponding `.endr` statement.
If the *iteration_list* argument is an empty macro-list, the statements in the block are read textually without being processed.
After the processing of the repetitive block has been completed, *iteration_var* contains the last element of *iteration_list*.

Example:

```
.irp   reg, [r0,r1,r2,r3,r4,r5,r6,r7]
   movqd        0,{reg}
.endr
```

generates code that clears registers `r0` through `r7`.

8.9.3 `.exit` Directive

Syntax: **`.exit`**

Description: Terminates the processing of the current repetitive block. Statements following this directive are read textually without being processed, until an `.endr` statement is encountered.

Example:

```
x:=1
.repeat
    .if   {x} > 30
         .exit
    .endif
    .byte {x}
    x:={{x}*2}
.endr
```

will generate the code

```
.byte  1
.byte  2
.byte  4
.byte  8
.byte  16
```

8.10 MACRO PROCEDURES (*MACROS*)

Use of a macro-procedure makes it possible to associate a macro name with a sequence of statements. This sequence can be generated by specifying the macro name in the opcode field, optionally with arguments.

The macro-procedure directives (`.macro` and `.endm`) in this version are not compatible with the GNX-assembler version 3.0 However, old code can be assembled using the -MC invocation option (/MCOMPATIBILITY on VMS) (see Section 8.3 for more details).

8.10.1 MacroProcedure Definition

Syntax: **`.macro`** *macro-name* [*formal-arg* [, *formal-arg*] . . .]

macro-procedure-body

`.endm` [*macro-name*]

where:	*macro-name*	is the macro-procedure name. It may be any legal assembler symbol.
	formal-arg	is a macro-variable defining a formal argument.
	macro-procedure-body	are the statements to be inserted into the assembler code when the macro-procedure is called.

Description: The statements of the macro-procedure body are read textually without being processed and are stored internally.

Within a macro-procedure body, other macro-procedure definitions are not allowed and all conditional and repetitive blocks must be complete. If *macro-name* is given in the `.endm` directive, it must be the same *macro-name* as given in the corresponding `.macro` directive.

A macro-procedure can only be defined once in an assembly file and its definition must precede any call to it.

The formal arguments in the `.macro` directive specify the names of the macro-variables to be assigned values according to the actual arguments, when the macro-procedure is called and expanded. The specification of formal arguments in the definition of a macro-procedure is optional.

Example:

```
.macro   clear_array      size, base_reg
   # clears an array of 'size' double-words whose
   # base address is in 'base_reg'

.repeat {size}, elem_num
   clear_elem    {elem_num}, {base_reg}
.endr

.endm

.macro   clear_elem      elem_num, base_reg
   # clears element number 'elem_num' of
   # an array whose address is in 'base_reg'

movqd    0, {4 * ({elem_num} - 1)}({base_reg})

.endm

clear_array  3, r4
```

will expand to

```
movqd    0, 0(r4)
movqd    0, 4(r4)
movqd    0, 8(r4)
```

8.10.2 Macro Procedure Call and Expansion

Syntax: *macro-name* [*actual-arg* [, *actual-arg*] . . .]

Description: A macro-procedure is called by specifying its name in the opcode field of the statement, provided it has already been defined. The name of the invoked macro-procedure may be followed by a sequence of actual arguments separated by commas.

When a macro-procedure call is processed, the current value of each macro-variable specified as formal argument is *saved*, and the macro-variable is assigned the value of its corresponding actual argument instead.

The body of the called macro-procedure is read from storage and processed as if it were inserted instead of the macro-procedure call statement. This is called macro-procedure expansion.

A macro-variable specified as a formal argument for the macro-procedure may be used in the macro-procedure body as any other macro-variable.

The number of actual arguments and the number of formal arguments do not have to correspond. If there are more formal arguments than actual arguments, the unmatched formal arguments will be assigned the value of an empty string. If there are more actual than formal arguments, the unmatched actual arguments can be accessed by using the predefined macro-procedure `ARG_LIST`. See the following section for more details on `ARG_LIST`.

8.10.3 Predefined Macro Procedure Variables

Two macro-variables, `ARG_COUNT` and `ARG_LIST`, are predefined macro-procedure variables. When a macro-procedure is called and expanded, their current values are saved, and they are assigned new values according to:

1. `ARG_COUNT` - is assigned the number of arguments actually passed to the macro-procedure.
2. `ARG_LIST` - is assigned the value of a macro-list, whose elements are the actual arguments to the macro-procedure. The first element of `ARG_LIST` will always be the first actual argument.
3. `ARG_LABEL` - is assigned the value of the label of the macro-procedure invocation. It is assigned a value only if a label appears on the same line as a macro invocation.

These predefined variables cannot be specified as formal arguments.

Example: "print_i_call" creates a calling sequence for the subroutine "print_integers" by pushing its parameters, and the number of parameters on the stack.

```
.macro    print_i_call

.irp       arg,{ARG_LIST}
movd     {arg},tos
.endr
movd       ${ARG_COUNT}, tos
bsr        print_integers
adjspd     ${-4*({ARG_COUNT}+1)}

.endm
```

The following call:

```
print_i_call   $100, xx, 0(r3)
```

will generate

```
movd    $100, tos
movd    xx, tos
movd    0(r3), tos
movd    $3, tos
bsr     print_integers
adjspd  $-16
```

8.11 `.macro_on` and `.macro_off` Directives

The **`.macro_on`** and **`.macro_off`** directives enable and disable macro-procedure expansions, respectively, in selective parts of the source text. This is useful when macro-procedure names contradict opcode mnemonic or assembler directives. Thus, if for example opcode `addd` is redefined as a macro-procedure without the using the **`.macro_off`** directive (as shown below), it would develop into an infinite sequence of recursive macro-procedure calls. However, the **`.macro_off`** directive allows disabling of macro-procedure expansions. As can be seen for:

```
.macro       addd   op1,op2
  bsr    count_additions
.macro_off
  addd      {op1},{op2}
.macro_on
.endm
```

the following macro-procedure call:

```
addd   r1,r2
```

will generate:

```
bsr    count_additions
addd   r1,r2
```

8.12 TEXT INCLUSION

This feature allows for the inclusion of text from another file as part of the file being assembled. The inclusion of text can also be specified from the invocation line by use of the -ML macro-library option (/MLIBRARY on VMS).

Syntax: **`.include`** *included_file*

where: *included_file* is an existing file name

Description: An `.include` directive causes the macro-processor to process statements from the file named *included_file* before processing the statements following the `.include` directive in the original file.
By default, if the *included_file* argument does not start with a `/`, only the directory in which the source file resides is searched. Additional directories for the *included_file* argument can be searched as specified on the invocation line using the macro Include Search Directory option (-MI on UNIX, /MINCLUDE on VMS).

Included files may contain any valid assembly directives and statements, macro-procedure call or macro-assembly directives (in particular `.include` directives), macro-procedure calls or macro-assembly directives, with all conditional blocks, repetitive blocks and macro-procedure definitions being complete.

Example: `.include filehdr.h`

8.13 MACRO WARNING AND ERROR MESSAGES

The directives `.mwarning` and `.merror` generate assembler warning and error messages.

8.13.1 `.mwarning` Directive

Syntax: **`.mwarning`**

Description: When a statement with a `.mwarning` directive is processed by the macro-processor, a warning message with the source file name, the current line number and *warning_message* is displayed on the assembler listing output (or written to the standard error file, if no listing output has been requested in the invocation line).

Example:

```
xx:= 222
.mwarning  current value of "xx" is : {xx}.
```

In this example the `.mwarning` directive may be used to write the current value of macro-variables on the listing output. The assembler will issue the following warning message:

```
Assembler (Macro-Processor): "filename.s", line 2 , WARNING : current
value of "xx" is : 222
```

8.13.2 `.merror` Directive

Syntax: **`.merror`** *error_message*

Description: When a statement with a `.merror` directive is processed by the macro-processor, an error message with the source file name, the current line number and *error_message* is displayed on the listing output (or written to the standard error file, if no listing has been requested in invocation line). The assembly process that follows is terminated after the macro-processing phase is completed, and the second phase, the assembly phase, is suppressed.

Example: `.merror      Wrong value used for addr "address"`

The assembler will issue the following error message:

```
Assembler (Macro-Processor) Error:
"f.s", line 1, statement is ==> .merror  Wrong value used for addr "address" <==
ERROR:          Wrong value used for addr "address"
```

8.14 LISTING CONTROL

Macro processor expansions can be output in two ways. After the macro processing phase, expansions can be output to the assembler. After the full assembly process is completed, a complete assembly listing file can be produced.

To display macro processor expansions after the macro processing phase, invoke the assembler with the -MP option (/MPRINT on VMS). The display will contain the expansions of the macro processor as assembly statements, with other non-macro assembly statements. See 8.3 for full details on the -MP option.

To list macro processor expansions after the full assembly process, invoke the assembler with the -L option (/LIST on VMS). This option will produce a complete listing. When the -L option is used, the `.list` and `.nolist` directives can be used to select parts of the assembly source file to be listed. In addition, qualifiers can be used with these directives to include or exclude certain levels of macro expansions. `.list` turns the qualifiers ON and `.nolist` turns them OFF.

The qualifiers are:

mac_source - When *mac_source* is ON, the assembler lists user source lines, before any macro expansions or macro substitutions have been done. The default setting is ON.

mac_expansions - When *mac_expansions* is ON, the assembler lists user source lines, after all macro substitutions have been performed on them. The default setting is OFF.

mac_directives - When *mac_directives* is ON, the macro directives also appear in the source listing. The default setting is ON.

It is not necessary to include the `.list` directive to use the default settings of the qualifiers. The -L option automatically produces a list and assumes the default qualifier settings.

For source level debugging, use the default settings of the qualifiers to produce a listing in which the displayed lines correspond to the line numbering recognized by the debugger.

For assembly level debugging, set *mac_source* and *mac_directives* OFF and *mac_expansions* ON to produce a listing in which the displayed lines correspond to the actual generated code.

When both *mac_source* and *mac_expansions* are OFF, no listing is produced. This combination is equivalent to `.nolist` with no parameters.

It is not advisable to use the *mac_source* option when both *mac_directives* and *mac_expansions* are OFF. This combination will produce output which is difficult to read.

In the default setup, the expansions of macro procedure calls, `.repeat`, and `.irp` blocks, are not listed.

Example : (Default)

mac_source= ON
mac_expansions= OFF
mac_directives= ON

This source file:

```
                .macro    zero_reg regno
                movqd     0, r{regno}
                .endm

        lab1:
                zero_reg  0
                .repeat   7, i
                zero_reg  {i}
                .endr
```

Produces this listing:

```
GNX Assembler Version X.XX    date    Page: 1

##### File "list1.s" #####

     1                                            .macro   zero_reg regno
     1                                            movqd    0, r{regno}
     1                                            .endm
     4
     5  T00000000                       lab1:
     6  T00000000  5f00                           zero_reg  0
     7                                            .repeat   7, i
     7                                            zero_reg  {i}
     7  T00000002  5f085f10                       .endr
                   5f185f20
                   5f285f30
                   5f38
```

When mac_expansions is ON, and *mac_source* and *mac_directives* are both OFF, only the output of the macro processing phase, as passed on to phase-1 of the assembler, is listed.

Example:

mac_source= OFF
mac_expansions= ON
mac_directives= OFF

This source file:

```
                      .list     mac_expansions
                      .nolist   mac_source mac_directives
                      .macro    zero_reg regno
                      movqd     0, r{regno}
                      .endm

            lab1:
                      zero_reg  0
                      .repeat   7, i
                      zero_reg  {i}
                      .endr
```

Produces this listing:

```
GNX Assembler Version X.XX    date    Page: 1


##### File "list2.s" #####

     1                                .list    mac_expansions
     2                                     .nolist       mac_source
mac_directives
     6
     7  T00000000            lab1:
     8  T00000000  5f00               movqd    0, r0
     9  T00000002  5f08               movqd    0, r1
     9  T00000004  5f10               movqd    0, r2
     9  T00000006  5f18               movqd    0, r3
     9  T00000008  5f20               movqd    0, r4
     9  T0000000a  5f28               movqd    0, r5
     9  T0000000c  5f30               movqd    0, r6
     9  T0000000e  5f38               movqd    0, r7
```

When both mac_source and *mac_expansions* are ON, each source line expanded by the macro assembler is printed twice: first as it appears in the source, and then as it appears after the expansion.

Example:

mac_source= ON
mac_expansions= ON
mac_directives= OFF

This source file:

```
                    .list     mac_expansions
                    .macro    zero_reg regno
                    movqd     0, r{regno}
                    .endm

          lab1:
                    zero_reg  0
                    .repeat   7, i
                    zero_reg  {i}
                    .endr
```

Produces this listing:

```
GNX Assembler Version X.XX    date    Page: 1


##### File "list3.s" #####

     1                                              .list     mac_expansions
     2                                              .macro    zero_reg regno
     2                                              movqd     0, r{regno}
     2                                              .endm
     5
     6  T00000000               lab1:
     7                                              zero_reg 0
     7                                              movqd     0, r{regno}
     7  T00000000  5f00                             movqd     0, r0
     8                                              .repeat  7, i
     8                                              zero_reg {i}
     8                                              zero_reg 1
     8                                              movqd     0, r{regno}
     8  T00000002  5f08                             movqd     0, r1
     8                                              zero_reg {i}
     8                                              zero_reg 2
     8                                              movqd     0, r{regno}
     8  T00000004  5f10                             movqd     0, r2
     8                                              zero_reg {i}
     8                                              zero_reg 3
```

```
8                                         movqd     0, r{regno}
8  T00000006  5f18                        movqd     0, r3
8                                         zero_reg {i}
8                                         zero_reg 4
8                                         movqd     0, r{regno}
8  T00000008  5f20                        movqd     0, r4
8                                         zero_reg {i}
8                                         zero_reg 5
8                                         movqd     0, r{regno}
8  T0000000a  5f28                        movqd     0, r5
8                                         zero_reg {i}
8                                         zero_reg 6
8                                         movqd     0, r{regno}
8  T0000000c  5f30                        movqd     0, r6
8                                         zero_reg {i}
8                                         zero_reg 7
8                                         movqd     0, r{regno}
8  T0000000e  5f38                        movqd     0, r7
8                                        .endr
```

After expansion of a macro or a `.repeat/.irp` **block has started, it cannot be reversed. However, it is possible to expand only one level by starting a macro or** `.repeat/.irp` **block with** ***mac_expansion*** **ON, and switch it OFF inside a block. Only the outer level will be expanded.**

Example:

This source file:

```
                .list      mac_expansions
                .macro     zero_reg regno
                movqd      0, r{regno}
                .endm

        labl:
                zero_reg   0
                .repeat    7, i
                .if        {i} = 2
                   .nolist  mac_expansions
                .endif
                zero_reg {i}
                .endr
```

Produces this listing:

```
GNX Assembler Version X.XX    date    Page: 1

##### File "list4.s" #####

     1                                  .list     mac_expansions
     2                                  .macro    zero_reg regno
     2                                  movqd     0, r{regno}
     2                                  .endm
     5
     6  T00000000               lab1:
     7                                  zero_reg  0
     7                                  movqd     0, r{regno}
     7  T00000000  5f00                 movqd     0, r0
     8                                  .repeat   7, i
     8                                  .if       {i} = 2
     8                                  .if       1 = 2
     8                                  .endif
     8                                  zero_reg  {i}
     8                                  zero_reg  1
     8                                  movqd     0, r{regno}
     8  T00000002  5f08                 movqd     0, r1
     8                                  .if       {i} = 2
     8                                  .if       2 = 2
     8                                     .nolist  mac_expansions
     8                                  .endif
     8  T00000004  5f10                 zero_reg  {i}
     8                                  .if       {i} = 2
     8                                  .endif
     8  T00000006  5f18                 zero_reg  {i}
     8                                  .if       {i} = 2
     8                                  .endif
     8  T00000008  5f20                 zero_reg  {i}
     8                                  .if       {i} = 2
     8                                  .endif
     8  T0000000a  5f28                 zero_reg  {i}
     8                                  .if       {i} = 2
     8                                  .endif
     8  T0000000c  5f30                 zero_reg  {i}
     8                                  .if       {i} = 2
     8                                  .endif
     8  T0000000e  5f38                 zero_reg  {i}
     8                                  .endr
```

8.15 STRING FUNCTIONS

The macro-assembler provides a set of built-in functions to manipulate strings: string length, string comparison, substring extraction, and substring search.

Characters in strings are counted starting number 1. For example, in the string `abcde`, `a` is character number 1, `b` is character number 2, and so on.

8.15.1 StringLength

Syntax: **{STR_LEN[** *string* **]}**

Description: Evaluates as the number of characters in *string*.

Examples:

1. `{STR_LEN[abcd]}` is evaluated as `4`.
2. `{STR_LEN[ab cd]}` is evaluated as `5`.
3. `{STR_LEN[]}` is evaluated as `0`.

8.15.2 String Comparison

Syntax: **{STR_EQ[** *string1*, *string2* **]}**

Description: Evaluates as 1 if *string1* and *string2* are the same, and as 0 if they are different.

Example:

```
.macro         addd3   src1, src2, dest
.if   {ARG_COUNT} = 2
addd   {src1}, {src2}
.elsif {STR_EQ[{src2}, {dest}]}
addd   {src1}, {src2}
.elsif {STR_EQ[{src1}, {dest}]}
addd   {src2}, {src1}
.else
movd    {src1}, {dest}
addd    {src2}, {dest}
.endif
.endm
```

8.15.3 Substring Extraction

Syntax: **{SUB_STR[** *string*, *start* [, *length*] **]}**

Description: Extracts a substring of the *string* argument from position *start*. Generally, length is taken to be substring size. If the *length* argument is omitted or is greater than the remaining length of the *string* argument, then the length of the substring is the remaining length of the string.

The function call will be evaluated as an empty string when:

- *start* is less than or equal to zero.
- *start* is greater than the length of the string.
- *length* is less than or equal to zero.

Examples:

1. `{SUB_STR[abcdefgh,2,3]}` evaluates to `bcd`.
2. `{SUB_STR[abcdefgh,3]}` evaluates to `cdefgh`.
3. `{SUB_STR[abcdefgh,1000,3]}` evaluates to an empty string.

8.15.4 Substring Search

Syntax: **{STR_FIND[** *string*, *substring* **]}**

Description: Evaluates as the position of the first character of *substring* in its first occurrence in *string*. If *substring* is not found, the value of the function is `0`.

Examples:

1. `{STR_FIND[abcdefgh,cde]}`

   evaluates to `3`.
2. `{STR_FIND[abcabc,c]}`

   evaluates to `3`.
3. `{STR_FIND[abcdefgh,zz]}`

   evaluates to `0`.

8.16 MACRO-LIST FUNCTIONS

A macro-list is a string that contains substrings separated by commas and that is enclosed between brackets. Each of these substrings is called an *element* of the macro-list. See Section 8.6 for more details about macro-lists.

The macro-assembler includes a set of built-in functions to process macro-lists that allow creation and manipulation of array-like and other complex structures (stacks, queues, etc ...). The built-in functions are : sub-list extraction, retrieval, search, insertion and deletion of elements into/from lists. Another built-in function returns the number of elements in a macro-list.

Elements in macro-lists are counted starting from the left with number 1. For example, in the macro-list `[aa,bb,cc,dd]`, element number 1 is `aa`, element number 2 is `bb`, and so on.

8.16.1 GetElement From List

Syntax: **{LIST_GET[** *list, element_number* **]}**

Description: Evaluates as the element whose number is specified by *element_number*.

Example: `{LIST_GET[[a,b,c,d],2]}` is evaluated as the string `b`.

8.16.2 Sublist Extraction

Syntax: **{SUB_LIST[** *list, start* [, *length*] **]}**

Description: Evaluates as a macro-list of *length* elements from the *list*, starting at element number *start*. If *length* is omitted or is greater than the number of remaining elements, all remaining elements are included in the sublist.

In the following cases, the function call is evaluated as an empty macro-list `[]`:

- *start* is less than or equal to zero.
- *start* is greater than the number of elements in *list*.
- *length* is less than or equal to zero.

Examples:

1. `{SUB_LIST[[a,b,c,d,e,f,g,h],2,3]}` is evaluated as `[b,c,d]`.
2. `{SUB_LIST[[a,b,c,d,e,f,g,h],3]]}`
   is evaluated as
   `[c,d,e,f,g,h]`.
3. `{SUB_LIST[[a,b,c,d,e,f,g,h],1000,3]}` is evaluated as `[]`.

8.16.3 Find An Element In List

Syntax: **{LIST_FIND[** *list, string* **]}**

Description: Evaluates as the position (element number) of the first occurrence of *string* as an element of *list*. If *string* is not an element of *list*, the function call is evaluated as `0`.

Example: After the assignment:

```
dummy_list:=[hhh,r1,ii,x,hh,x]
```

then:

- `{LIST_FIND[{dummy_list},r1]}` is evaluated as `2`.
- `{LIST_FIND[{dummy_list},yyy]}` is evaluated as `0`.
- `{LIST_FIND[{dummy_list},x]}` is evaluated as `4`.

8.16.4 Replace An Element In A List

Syntax: **{LIST_REPL[** *list, element_number, string* **]}**

Description: Evaluates as *list* after replacing the element, whose number is specified by *element_number*, with the given *string*. This macro-function is useful, when a macro-list is handled as an array, for assigning a value to a specified element in a macro-list.

Example:

```
dum_list:=[xx,yy,zz]

dum_list:={LIST_REPL[{dum_list},2,aa]}
```

The second element of `dum_list` has been "assigned" (replaced with) the string `aa`, and `dum_list` now holds the value `[xx,aa,zz]`.

8.16.5 Insert An Element Into A List

Syntax: **{LIST_INS[** *list, string, element_number* **]}**

Description: Evaluates as *list* after inserting *string* as an element before the element specified by *element_number*.

Example:

```
list1:=[aa,bb,cc]

list2:={LIST_INS[{list1},dd,3]}
```

`list2` holds the value `[aa,bb,dd,cc]`.

8.16.6 Delete An Element From A List

Syntax: **{LIST_DEL[** *list, element_number* **]}**

Description: Evaluates as *list* after removing the element whose number is specified by *element_number*.

Example:

```
list1:=[aa,bb,cc]

list2:={LIST_DEL[{list1},2]}
```

`list2` holds the value `[aa,cc]`.
`list1` remains to hold the original value `[aa,bb,cc]`.

8.16.7 Number Of Elements In A List

Syntax: **`{LIST_LEN[`** *list* **`]}`**

Description: Evaluates as the number of elements in *list*.

Example: `vars_list:=[-12(fp),-16(fp),-20(fp),r0,r1[r4:b],r2]`

`{LIST_LEN[{vars_list}]}` evaluates to `6`.

8.16.8 Example of Macro-List Function Usage

Included here is an example showing the capability of the different macro-list functions. A stack-list is implemented using the macro-list functions. We define a set of macro-procedures : `PUSH,POP,TOP,RESET`.

```
.macro   PUSH      list_name,element
                 # pushes an element into a stack list

    {list_name}:={LIST_INS[{{list_name}},{element},1]}
                 # {list_name} evaluates to the NAME of the list.
                 # {{list_name}} evaluates to its VALUE.
.endm

.macro   POP       list_name,el_var_name
                 # returns the first element of a list, and remove that
                 # first element from it.

    {el_var_name}:={LIST_GET[{{list_name}},1]}

    {list_name}:={LIST_DEL[{{list_name}},1]}

.endm

.macro   TOP       list_name,el_var_name
                 # returns the last element of a list.

    {el_var_name}:={LIST_GET[{{list_name}},1]}

.endm

.macro   RESET       list_name
                 # assign a empty list value [] to the list.

    {list_name}:=[]

.endm
```

In the following sequence of macro-procedure calls, the values of the variables after each call are specified in the comments.

```
var:=
RESET          stack1
RESET          stack2
                            # value of :
                            # stack1         | stack2         | var
                            # ======         | ======         | ===
                            # []             | []             | empty string

PUSH           stack1,aa    # [aa]           | []             | empty string
PUSH           stack1,bb    # [bb,aa]        | []             | empty string
PUSH           stack1,cc    # [cc,bb,aa]     | []             | empty string
POP            stack1,var   # [bb,aa]        | []             | cc
PUSH           stack2,{var} # [bb,aa]        | [cc]           | cc
TOP            stack1,var   # [bb,aa]        | [cc]           | bb
RESET          stack1       # []             | [cc]           | bb
```

8.17 DATA CONVERSION FUNCTIONS

The macro-assembler provides a set of built-in functions to convert strings representing assembly numerical constants (as defined in Section 2.4) into hexadecimal digit strings. These are integer hexadecimal, float hexadecimal or long float hexadecimal.

8.17.1 ConvertTo Integer Hexadecimal

Syntax: **{CNV_HEX[** *integer_constant* **]}**

Description: Evaluates as a string of 8 hexadecimal digits representing the constant in hexadecimal integer format. The *integer_constant* may be specified in any of the integer notations.

Example:

Given the definition

```
const := 1024
```

then `{CNV_HEX[{const}]}` is evaluated as `X'00000400`.

8.17.2 Convert To Float Hexadecimal

Syntax: `{CNV_HEXF[` *constant* `]}`

Description: Evaluates as a string of 8 hexadecimal digits representing the constant in float-hexadecimal format. If *constant* is not a single precision floating point constant, it is first converted to this representation.

Examples:

1.
```
{CNV_HEXF[{5-4}]}
```

is evaluated as `f'3f800000.`

2.
```
{CNV_HEXF[1.0e0]}
```

is evaluated as `f'3f800000.`

3.
```
{CNV_HEXF[l'3ff0000000000000]}
# long representation of 1.
```

is evaluated as `f'3f800000.`

8.17.3 Convert To Long Float Hexadecimal

Syntax: `{CNV_HEXL[` *constant* `]}`

Description: Evaluates as a string of the 16 hexadecimal digits representing the *constant* in long hexadecimal-decimal format. If *constant* is not a long floating point constant, it is first converted to this representation.

Examples:

1. `{CNV_HEXL[{5-4}]}`

   evaluates to `e'3ff0000000000000.`

2. `{CNV_HEXL[1.0e0]}`

   evaluates to `e'3ff0000000000000.`

3.
   ```
   {CNV_HEXL[f'3f800000]}
    # single precision float representation of 1.
   ```

   evaluates to `e'3ff0000000000000.`

8.18 INSTRUCTION OPERAND FUNCTIONS

The macro-assembler includes a set of built-in functions for processing instruction operands, including recognition of operand type and extraction of subfields from operands strings. These functions provide for ease in using the diversity of operands types and addressing modes provided by the NS32000 architecture and the GNX assembler.

For example, given an operand string specifying a memory location, another operand string can be created which points to the double word next to that location (*i.e* "location+4"). If the operand is a symbol, a leading `4+` string can be concatenated to the operand string. If the operand has been specified with a leading `@` (absolute addressing mode), `4+` can be inserted after the `@` . However with many other operand notations adding such an offset to the location is not as simple. Therefore some convenient built-in functions are provided which recognize the notation (type) in which the operand has been specified, and extract subfields in operand strings.

8.18.1 RecognizeThe Type Of An Operand

Syntax: **{OP_TYPE[** *operand* **]}**

Description: Evaluates as a string describing the NS32000 type of `operand`, or as an empty string if the string is not a legal NS32000 operand.

A list of possible operands types are:

EXPR : any legal combination of symbols, constants and arithmetic operators optionally followed by a displacement size specification (:b , :w , :d).
examples: xx:b
12
ss+3+(kk-9):d
GREG : r0,r1 ...
FREG : f0,f1,...
LREG : l0,l1,...
PREG : processor register : upsr,cfg,sp ..
MREG : mmu register : tear,mcr ...
REG_REL : expression1(register)
MEM_SPACE : expression1(fp),
or expression1(sp), or expression1(sb)
EXPL_PC_REL : %expression1
EXPL_SB_REL : ^expression1
MEM_REL : expression2(expression1(fp)),
or expression2(expression1(sp)),
or expression2(expression1(sb))
ABS : @expression1
IMM : $expression1
EXT_1 : expression2(expression1(ext))
EXT_2 : expression1(ext)
DREF_SYM : expression2(expression1)
TOS : tos
REG_LIST : register list [r0,r1,..]
OPT_LIST : options list [cc,f,..]

1. `{OP_TYPE[12(sp)]}`

   is evaluated as `MEM_SPACE`.

2. `{OP_TYPE[@xx+121]}`

   is evaluated as `ABS`.

3. `{OP_TYPE[12(param+12)]}`

   is evaluated as `DREF_SYM`.

NOTE: The OP_TYPE built-in macro-function can not always provide the definite addressing mode that will be used for the operand. Information returned by this function is just the most accurate conclusion that can be drawn about the nature of the operand, through scanning the operand string and without any knowledge of the context in which the operand appears or of the type of the user-symbols (*e.g.* labels) involved in the operand. Since this knowledge is mandatory for determining the exact addressing mode in which the operand will be encoded, and since the macro-processing phase is done prior to the assembly phase, this information is unavailable during the macro-processing phase.

8.18.2 Operand Subfields

The following are the various operand subfield functions

- Syntax: **`{OP_REG[`** *operand* **`]}`**

  Description: If the operand is a register, it is evaluated as that register. If the operand has a base register, it is evaluated as the base register. No register is returned if the operand is a register list.

  Example: `{OP_REG[xx:w(yy+8(sp))]}`

  is evaluated as `sp`.

- Syntax: **`{OP_DISP1[`** *operand* **`]}`**

  Description: If the operand contains at least one displacement field, with or without a displacement size specification, the function call is evaluated as the innermost displacement string without the displacement size specification. Otherwise the empty string is returned.

  Note that when the operand type is `DREF_SYM`, the intermost displacement string is returned.

  Example: `{OP_DISP1[xx:w(yy+8(sp))]}`

  is evaluated as `yy+8`.

- Syntax: **{OP_DISPSIZE1[** *operand* **]}**

  Description: If the innermost displacement string has a size specified, that size specification is returned. Otherwise, the empty string is returned.

  Example: `{OP_DISPSIZE1[xx:w(yy+8(sp))]}`

  evaluates to an empty string.

- Syntax: **{OP_DISP2[** *operand* **]}**

  Description: If the operand contains two displacement fields (in MEMORY RELATIVE addressing mode and in some EXTERNAL addressing mode notations), it is evaluated as the string of the outermost displacement without the displacement size specification.

  Note that when the operand type is `EXT_3`, the outermost displacement string is returned.

  Example: `{OP_DISP2[xx:w(yy+8(sp))]}`

  is evaluated as `xx`.

- Syntax: **{OP_DISPSIZE2[** *operand* **]}**

  Description: If the operand contains two displacement fields (in MEMORY RELATIVE addressing mode and in some EXTERNAL addressing mode notations), it is evaluated as the the displacement size specification of the outermost displacement field.

  Example: `{OP_DISPSIZE2[xx:w(yy+8(sp))]}`

  is evaluated as `:w`.

- Syntax: **{OP_VAL[** *operand* **]}**

  Description: If the operand is `EXPR`, `ABS`, `IMM`, `EXPL_PC_REL`, or `EXPL_SB_REL`, it is evaluated as the expression without the size or any preceding literals.

  Example: `{OP_VAL[$yy+8]}`

  is evaluated as `yy+8`.

- Syntax: **{OP_VALSIZE[** *operand* **]}**

  Description: If the operand is `EXPR`, `ABS`, `IMM`, `EXPL_PC_REL` or `EXPL_SB_REL`, it evaluates to its specified size.

  Example: `{OP_VALSIZE[$yy+8:b]}`

  is evaluated as `:b`

- Syntax: **{OP_LIST[** *operand* **]}**

  Description: If the operand is a register list (general purpose registers between []) or an option list (either cfg or cinv option list between []), it is evaluated as the list after sorting of its elements.

  Examples:

  1. `{OP_LIST[[r4,r6,r2,r0]]}`

     is evaluated as a macro-list with the registers which appear in the list after sorting : [r0,r2,r4,r6]

  2. `{OP_LIST[[i,f,c]]}`

     is evaluated as `[c,f,i]`.

- Syntax: **{OP_IS_INDEXED[** *operand* **]}**

  Description: If the operand has a scaling index it evaluates to the boolean value 1; otherwise it evaluates to the boolean value 0.

  Example: `{OP_IS_INDEXED[r0[r2:d]]}`

  is evaluated as the boolean value: 1.

- Syntax: **{OP_INDEX[** *operand* **]}**

  Description: If the operand is a scaled indexed operand, it is evaluated as the scaling index string, including the index register, the index scale specification (:b, or :w, or :d, or :q, or an empty string) and the enclosing brackets.

  Example: `{OP_INDEX[xx+9(sp)[r1:b]]}`

  is evaluated as the index specification string : `[r1:b]`.

- Syntax: **{OP_INDEX_BASE[** *operand* **]}**

  Description: If the operand is a scaled indexed operand, it is evaluated as the operand string without the index mode specification string.

  Example: `{OP_INDEX_BASE[xx+9(sp)[r1:b]]}`

  is evaluates as the "base" of the operand, that is, the operand string without index specification string : `xx+9(sp)`.

- Syntax: **{OP_INDEX_REG[** *operand* **]}**

  Description: If the operand is a scaled indexed operand, it is evaluated as a string specifying the index register.

  Example: `{OP_INDEX_REG[xx+9(sp)[r1:b]]}`

  it is evaluated as the index register `r1`.

- Syntax: **{OP_INDEX_SCALE[** *operand* **]}**

  Description: If the operand is a scaled indexed operand, it is evaluated as a string specifying the index mode scale specification (:b, or :w, or :d, or :q, or an empty string).

  Example: `{OP_INDEX_SCALE[xx+9(sp)[r1:b]]}`

  is evaluated as the scale specification `:b`.

The following table defines the subfields that are relevant to various operand types.

Table 8-2. Relevant Operand Subfields

TYPE	REG	DISP1	SIZE1	DISP2	SIZE2	VAL	VALSIZE	LIST	INDEX
EXPR						+	+		+
GREG	+								+
FREG	+								
LREG	+								
PREG	+								
MREG	+								
REG_REL	+	+	+						+
MEM_SPACE	+	+	+						+
EXPL_PC_REL						+	+		+
EXPL_SB_REL						+	+		+
MEM_REL	+	+	+	+	+				+
ABS						+	+		+
IMM						+	+		
EXT_1		+	+	+	+				+
EXT_2		+	+						+
EXT_3		+	+	+	+				+
TOS									+
REG_LIST								+	
OPT_LIST								+	

The following example illustrates use of the OP_TYPE and of the subfield functions.

The macro-procedure `warn_same_reg` receives an operand string as an argument, and issues a warning message when the operand has scaled indexing and the index register is the same as the base register.

```
.macro   warn_same_reg          operand

     .if   {OP_IS_INDEXED[{operand}]}
              reg : = {OP_REG[{operand}]}
              scaling_reg : = {OP_INDEX_REG[{operand}]}
               .if   {STR_EQ[{reg},{scaling_reg}]}
                   .mwarning   base register and index register are
                               the same in {operand}
              .endif

       .endif

.endm
```

NOTE: It is not necessary to check types of operands. Registers will be empty if operands are irrelevant.

The following example shows a possible usage of the "warn_same_reg" macro-procedure. The macro-procedure "warn_same_reg" is invoked from another macro-procedure, "movd", which first performs a check on both its operands and then actually issues a "movd" instruction.

```
.macro      movd    source,dest

   warn_same_reg  {source}

   warn_same_reg  {dest}

   .macro_off
       # cancel definition of "movd"  as a macro-procedure,
       # to avoid infinite recursive calls
       # "movd" is now considered to be an instruction,
       # and not a macro-procedure.

   movd  {source},{dest}

   .macro_on
       # restore macro-procedure "movd" definition.

.endm


movd     12(r1)[r1:b],r2[r2:q]
       # 2 warning messages are issued, one for each operand.
       #
       # ".... WARNING .... base register and index register are
       #  the same in 12(r1)[r1:b]" .
       #
       # ".... WARNING .... base register and index register are
       #  the same in r2[r2:q]" .
       #
       # the instruction "movd     12(r1)[r1:b],r2[r2:q]" is also
       # generated.

movd     0(r3)[r3:q],r2
       # a warning message is issued for the first operand.
       #
       # ".... WARNING .... base register and index register are
       #  the same in 0(r3)[r3:q]" .
       #
       # the instruction  movd 0(r3)[r3:q],r2 is also generated.

movd     r1[r3:w],@aaa
       # no warning messages are issued.
       # the instruction "movd  r1[r3:w],@aaa" is generated.
```

8.19 PREDEFINED MACRO VARIABLES

Several variables are predefined by the GNX macro-assembler to hold the values of several target specification parameters. These parameters are either set in the .gnxrc file or as invocation switches to the assembler.

The predefined variables are:

Variable	Target Specification
GNX_OS	os
GNX_CPU	cpu
GNX_MMU	mmu
GNX_FPU	fpu
GNX_COMMTYPE	commtype
GNX_BYTESEX	bytesex
GNX_BUSWIDTH	buswidth

The MAC_DEBUG predefined variable is set to "1" if the assembler is invoked with the "-g" ("/DEBUG") option. It is set to "0" otherwise. This can be used to add special test-code during the debugging phase.

The MAC_COMMENT predefined variable is set to the "#" character. It allows user macros to add comments to the output when the macro assembler is used as a macro preprocessor only. (i.e. when the assembler is invoked with the "-MO" and "-MP" flags on UNIX, or "/MONLY" and "/MPRINT" on VMS.)

Chapter 9

INVOCATION AND OPERATION

9.1 INTRODUCTION

The GNX Assembler generates object code from *Series 32000* assembly language source files and optionally produces a listing file and debugging information. Each assembly source file produces one *Series 32000* software module, consisting of a text (code) section and an initialized data section. The module is suitable for execution on *Series 32000*-based systems after the appropriate linking process.

This chapter describes the input and output files used by the GNX Assembler, the GNX Assembler invocation, the assembler listing file, symbol table listing, the cross-reference table, assembly errors, and the GNX Assembler limitations.

9.2 INPUT AND OUTPUT FILES USED/GENERATED BY THE GNX ASSEMBLER

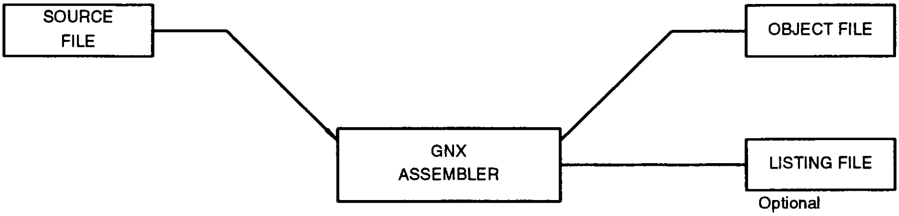
The files used as input and those generated as output by the assembler are shown in Figure 9-1 and described below.

Source file — Input. The source file is a text file containing the source program to be assembled.

Object file — Output. The object file contains the relocatable object code and data produced by the assembler, as well as optional debugging information. When no filename for the object file is given, the default name is the name of the source file with the `.s` suffix, if any, stripped off and a `.o` suffix appended. For example, if the source file is named `build.s`, the name of the object file will be `build.o`. The object file is suitable for use as input to the linker, `ld` (native), or `nmeld` (cross-support).

Listing file — Output. The listing file, created with the -L option (/LIST on VMS), contains the program listing produced by the assembler. On UNIX, the default listing file is the standard output (`stdout`); on VMS it is `filename.lis`. If a filename parameter is specified with the listing option, the filename is the listing output.

Macro-processor output — Output. Contains the macro-processor output. For details see Section 8.14.



GF-01-0-U

Figure 9-1. Input and Output Files for the GNX Assembler

Temporary files used by the GNX Assembler during the assembly process are as follows:

DOS:	`al`*xxxxxx*	temporary files for listing
	`am`*xxxxxx*	temporary source file for listing
	`at`*xxxxxx*	temporary file
	`as`*xxxxxx*	temporary files for pre-processing
	`ax`*xxxxxx*	temporary files for cross-referencing
UNIX:	`/tmp/aslst`*xxxxxx*	temporary files for listing
	`/tmp/amlst`*xxxxxx*	temporary source file for listing
	`/tmp/as`*xxxxxx*	temporary file
	`/tmp/ast`*xxxxxx*	temporary files for pre-processing
	`/tmp/asmd`*xxxxxx*	temporary files for macro definitions
	`/tmp/asma`*xxxxxx*	temporary files for macro arguments
	`/tmp/asx`*xxxxxx*	temporary files for cross-referencing
VMS:	`aslst`*xxxxxx*`.tmp`	temporary files for listing
	`amlst`*xxxxxx*`.tmp`	temporary source file for listing
	`as`*xxxxxx*`.tmp`	temporary file
	`ast`*xxxxxx*`.tmp`	temporary files for pre-processing
	`asmd`*xxxxxx*`.tmp`	temporary files for macro definitions
	`asma`*xxxxxx*`.tmp`	temporary files for macro arguments
	`asx`*xxxxxx*`.tmp`	temporary files for cross-referencing

Where *xxxxxx* is replaced by the current process ID.

The creation of temporary files can cause a file I/O operation failure if there is limited space in the directory. The default location for temporary files can be changed on UNIX/MS-DOS by using the `TMPDIR` eironment variable.

9.3 GNX ASSEMBLER INVOCATION

The GNX Assembler is invoked from the shell by entering the `as` command or the `nasm` command (under X-support), optionally followed by flags and an output filename, followed by a source filename. The following is the assembler syntax:

{**as** | **nasm**} [*options*] *sourcefile*

The source filename and the flags may appear in any order with the exception of the –c option which must come before the –D, –U, or –I options. Only one source filename is permitted. See Table 9-2 for a list of the optional flags, their syntax, and their definition.

Examples:

```
1.  nasm
2.  nasm myfile.s
3.  nasm -L myfile.s
4.  nasm -L -o myfiledebug.o myfile.s
5.  nasm -L myfile.s > myfile.lis
6.  nasm -Lmyfile.lis myfile.s
```

Example 1 does not specify any filename or switch. The assembler will wait for input from `stdin`.

Example 2 will assemble the source file `myfile.s` and generate an object file with the default name `myfile.o`. No listing file will be produced.

Example 3 will generate both an object file, `myfile.o`, and a listing file from the source file `myfile.s`. The listing file will be output to `stdout`.

Example 4 will generate the object file `myfiledebug.o` from the source file `myfile.s`. Because the –L option is specified, a listing will be produced on `stdout`.

Examples 5 and 6 will generate a listing file from the source file `myfile.s` and output it to `myfile.lis`.

9.3.1 Target Machine Specification

The assembler provides a way for the user to tune the code for a specific target system by specifying its CPU, FPU and MMU. This tuning is performed by setting permanent defaults using the GNX Target Setup (GTS) facility, or by specifying `/TARGET` (`-K`) on the command line.

Table 9-1 lists the possible target selection parameters. The values for the CPU, FPU and MMU can either be the complete device name *e.g.*, `NS32GX320` or `NS32381`, or the last characters of the device name, *e.g.* `GX320` or `381`. The absence of an FPU on the target system can be indicated by specifying the parameters `emulation` (emu) or `nofpu`. See the *Series 32000 Support Libraries Manual* for details on the floating-point emulation library (libHfp). The absence of an `mmu` is indicated by specifying the parameter `nommu`. The existence of `mmu` on a the specified CPU is indicated by using the parameter `onchip` (or `mmu_onchip`).

Table 9-1. Target Selection Parameters

CPU (C)	FPU (F)	MMU (M)
[NS32]CG16	nofpu	nommu
[NS32]CG160	emulation	onchip
[NS32]AM160		
[NS32]FX164		
[NS32]FX16	[NS32]381	[NS32]382
[NS32]GX32	[NS32]181	[NS32]082
[NS32]GX320	[NS32]081	
[NS32]532	[NS32]580	
[NS32]332		
[NS32]032		
[NS32]016		
[NS32]008		

Example: The following example specifies an NS32GX320 CPU, an NS32381 FPU, and a buswidth of 4 bytes (the default).

UNIX

`as -KCGX320 -KF381 temp.s` (native-support)
or **`nasm -KCGX320 -KF381 temp.s`** (cross-support)

VMS

```
NASM /TARGET=(CPU=GX320,FPU=381) TEMP.S
```

Table 9-2. Optional Flag Syntax
Sheet 1 of 2

FLAG VMS	**FLAG** UNIX MS-DOS	**DEFINITION**
/DISPLACEMENT= {BYTE | WORD | DOUBLE}	–d1 | –d2 | –d4	Sets the default displacement size to byte (d1), word (d2), or double-word (d4). The default is double-word (d4).
/M4	–m	Runs the `m4` macro pre-processor on the input to the assembler.
/CPP	–c	Runs the C compiler pre-processor (cpp) on the input to the assembler.
*	–R	Deletes (unlinks) input file after assembly. Off by default.
/NODATA	–r	Incorporates the data segment into the text segment. Off by default.
/SAVESYM	–s	Saves compiler-generated labels in the symbol table of the object file.
/VERSION	–V	Writes the version number of the assembler to stderr (UNIX) or SYS$OUTPUT (VMS).
/VMEM	–v	Uses virtual memory for intermediate storage rather than a temporary disk file.
/AMODE = SB	–A s	Overrides default addressing modes. The *s* or *SB* causes all references to symbols of type data to use the Static Base Register Relative addressing mode.
/LIST [=*filename*]	–L[*filename*]	If *filename* is given, produces the listing in that file. If *filename* is not given, then on UNIX the listing is produced in the stdout file; on VMS in the .lis file.
/NOSDI	–n	Disables displacement size optimization.
/OBJECT=*object*	–o *objfile*	Leaves the output of the assembly on the file *objfile*. On UNIX, by default, the output filename is formed by removing the .s suffix, if any, from the input filename and adding a .o suffix. On VMS, by default, the output filename is formed by replacing the extension with a .obj extension.
*	–t	Causes the assembler to show all the utilities it calls. This option is useful for tracing all processes executed by the assembler.
	@ *filename*	Reads options from file *filename* . (MS-DOS only)

Table 9-2. Optional Flag Syntax
Sheet 2 of 2

FLAG VMS	**FLAG** UNIX MS-DOS	**DEFINITION**
/NOWARNING	–w	Supresses assembly warning messages.
/MAP [=*filename*]	–y[*filename*]	Produces a symbol table listing entitled "Symbol Table Dump." On VMS, if *filename* is not given, the output filename is formed by replacing the extension with a .map extension.
/XREF [=*filename*]	–x[*filename*]	Produces a cross-reference listing entitled "Cross-Reference Table". On VMS, if *filename* is not given, the output filename is formed by replacing the extension with a .xrf extension.
/[NO]OPTIMIZE	–O	[Do not] perform procedure optimizations.
/DEFINE= ("*name*[=def]",...)	–D*name* or –D*name=def*	Defines *name* to *cpp*, as if by "#define." If no definition is given, *name* is defined as 1. The –c (or /CPP for VMS) option must precede this option.
/UNDEFINE= ("*name*"[,...])	–U*name*	Removes initial definition of a predefined *name*. *Cpp* supplies initial definition of 1 for predefined names (*e.g.*, NS32000, VMS, UNIX). The –c (or /CPP for VMS) option must precede this option.
/INCLUDEDIR= (*directory*[,...])	–I*dir*	Searches "#include" files that do no begin with / (or [for VMS) in the directory of the *filename* argument first, then the directory named in this option, then the directories on a standard list. The –c (or /CPP for VMS) option must precede this option.
/DEBUG	–g	Produces additional line number information for symbolic debugging.
/MODULAR	–X	Sets the 32000 modularity.
/TARGET= (*parameter*[,...])	–K*parameter*	Allows the user to specify the CPU, FPU, and MMU of the user's target system. The *parameter* is in the form C*cpu*, F*fpu*, M*mmu*, or B*buswidth* on UNIX and in the form CPU=*cpu*, FPU=*fpu*, MMU=*mmu*, and BUSWIDTH=*buswidth* on VMS.
/M*option*	–M*option*	Macro specific option. These options are /MCOMPATIBILITY (-MC), /MDEFINE (-MD), /MLIBRARY (-ML), /MINCLUDE (-MI), /MONLY (-MO), and /MPRINT (-MP).**

* This flag is not available for VMS.
** Refer to Section 8.3 for a detailed description.

9.3.2 Assembler Symbolic Debugging

When invoked with the –g option (/DEBUG option on VMS), the assembler generates a line number entry in the object file for every source line of the input assembly file where a breakpoint can be inserted. The information from the line number entries allows the user to reference the line numbers when using a software debugger, such as DBUG.

Each assembly procedure defined using the GNX Assembler Procedure Support causes the generation of appropriate symbolic information for the debugger. This symbolic information includes the same information generated by the GNX compiler for HLL procedures. Therefore, it is possible to stop in the procedure, reference its variables by name, and receive information on variable types.

Code segments, which are not part of such procedures are grouped by the assembler to form dummy procedures. Dummy procedures start at the first non procedural statement and end at the last non procedural statement of the assembly source file. The name of the dummy procedure is of the form .X*basename_number*, where *basename* is the source file name without the `.s` or `.asm` suffix; and *number* is the file segment number.

Every assembler label with a storage allocation directive (e.g. `.double,` `.blkd`) is given a type based on the storage allocation. The types are assigned as follows:

Storage Allocation Directives	Corresponding Type
.byte, .blkb	unsigned char
.word, .blkw	short int
.double, .blkd	int
.float, .blkf	float
.long, .blkl	double
.ascii	char

When the `.ascii` directive is used or when a repetitive factor is specified for any other storage allocation directive, the associated label is considered an array of the corresponding type.

Each procedure defined using the GNX Assembler Procedure Support will also be given a type based on the return value specified by the `.endproc` directive. If the return value is omitted, a default `int` or `float` will be assigned. The types are assigned as follows:

Return Value Modifer	Procedure Type
b	char
w	short int
d	int
f	float
l	long
ub	unsigned char
uw	unsigned short
ud	unsigned int

If the input source file contains `.ln` directives, no symbolic debugging information will be prepared by the assembler; instead, information from the `.ln` directive will be used to generate the line number entry.

9.4 ASSEMBLER OUTPUT LISTINGS

Figure 9-2 shows a sample assembly language program. The listing produced when the program is assembled is shown in Figure 9-3. Figure 9-4 is an annotated version of Figure 9-3.

```
        .set    p_start, 8
        .dsect  param_list, p_start
vname:  .blkd
num:    .blkd
        .text
indirect_add:
        enter   [r4], 0
        movd    0(vname(fp)), r4
        addd    num(fp), r4
        movd    r4, 0(vname(fp))
       exit    [r4]
       ret     0
```

Figure 9-2. Sample Assembly Program

```
GNX Assembler Version X.XX      date          Page: 1

##### File "exampl.s" #####

     1  A********  00000008            .set    p_start, 8
     2                                 .dsect  param_list, p_start
     3  A00000008            vname:    .blkd
     4  A0000000c            num:      .blkd
     5                                 .text
     6  T00000000            indirect_add:
     7  T00000000  821000              enter   [r4], 0
     8  T00000003  17810800            movd    0(vname(fp)), r4
     9  T00000007  03c10c              addd    num(fp), r4
    10  T0000000a  17240800            movd    r4, 0(vname(fp))
    11  T0000000e  9208                exit    [r4]
    12  T00000010  1200                ret     0
```

Figure 9-3. GNX Assembler Listing File

```
                 1                                  2

GNX Assembler Version X.XX    date          Page: 1

             3

##### File "exampl.s" #####

      4     5           6                      7

      1  A********  00000008              .set    p_start, 8
      2                                   .dsect  param_list, p_start
      3  A00000008               vname:   .blkd
      4  A0000000c               num:     .blkd
      5                                   .text
      6  T00000000               indirect_add:
      7  T00000000  821000                enter   [r4], 0
      8  T00000003  17810800              movd    0(vname(fp)), r4
      9  T00000007  03c10c                addd    num(fp), r4
     10  T0000000a  17240800              movd    r4, 0(vname(fp))
     11  T0000000e  9208                  exit    [r4]
     12  T00000010  1200                  ret     0
```

Callouts 1 to 7:

1	Version number of a tool
2	Listed file page number
3	Source file name. Will reflect included files.
4	Source file line number.
5	Address of the current line. Preceeded by letter representing the section of address.
6	Code or value of source line.
7	User source line itself.

Figure 9-4. GNX Assembler Listing File (Annotated Version)

Figure 9-5 shows a sample assembly language program containing floating-point instructions. The listing produced when the program is assembled with a request for libHfp emulation is shown in Figure 9-6. For a detailed description of the libHfp interface, Refer to Chapter 6 of the *Series 32000 GNX–Version 4 Support Libraries Reference Manual.*

```
        .data
fp_var: .blkf
        .text
lab1:
        enter   [],0
addf    f0, f2
addl    14, 16
movf    f2, fp_var
exit    []
ret     0
```

Figure 9-5. Sample Assembly Program With Floating Point Instructions

```
GNX Assembler Version X.XX    date          Page: 1


##### File "examp2.s" #####

   1                                    .data
   2 D00000000  00000000       fp_var: .blkf
   3                                    .text
   4 T00000000                 lab1:
   5 T00000000  820000                  enter   [],0
   6 T********                          addf    f0, f2
     T00000003  e7adc000 ***            addr    F2__,tos
                0000
     T00000009  d7adc000 ***            movd    F0__,tos
                0000
     T0000000f  02ffffff ***            bsr     addf__
                f1
   7 T********                          addl    14, 16
     T00000014  e7adc000 ***            addr    F6__,tos
                0000
     T0000001a  d7adc000 ***            movd    F4__+4,tos
                0004
     T00000020  d7adc000 ***            movd    F4__,tos
                0000
     T00000026  02ffffff ***            bsr     addl__
                da
   8 T********                          movf    f2, fp_var
     T0000002b  57adc000 ***            movd    F2__,fp_var
                0000c000
                003c
   9 T00000035  9200                    exit    []
  10 T00000037  1200                    ret     0
```

Figure 9-6. GNX Assembler Listing File With libHfp Interface

Note that emulated instructions are marked with `***`.

A sample program with one error is shown in Figure 9-7. When the program is assembled, the error is flagged as shown in Figure 9-8. Assembly errors are discussed in Section 9.5.

```
_main::
        enter   []
        addr    msg, tos
        jsr     _printf
        adjspb  $-4
        exit    []
        ret     0

        .data
msg:    .ascii  "Hello, world\n\0"
```

Figure 9-7. A Sample Program Containing Errors

```
GNX Assembler Version X.XX    date         Page: 1

##### File "examp3.s" #####

     1                          _main::
     2                                  enter   []
"examp3.s", line 2: Too few operands specified, 2 operands expected.

     3                                  addr    msg, tos
     4                                  jsr     _printf
     5                                  adjspb  $-4
     6                                  exit    []
     7                                  ret     0
     9                                  .data
    10                          msg:    .ascii  "Hello, world\n\0"


ERRORS DETECTED : 1.
```

Figure 9-8. GNX Assembler Listing File With Error Mesage

9.4.1 Assembler Symbol Table Listing

The symbol table listing will be entitled "Symbol Table Dump." It will be preceded by a formfeed, and will be output to the specified file. If no output file is specified for it, the symbol table will be output either to `stdout` (On UNIX/MS-DOS systems), or to the `.MAP` file (On VMS systems).

Figure 9-9 shows a sample symbol table source file, and Figure 9-10 shows a sample symbol table listing.

```
          .set x, 10
          bsr foo
          movd foo, r0
foo:
          .globl blap
          movd blap, r0
```

GF-09-0-U

Figure 9-9. Sample GNX Assembler Symbol Table Source File

```
GNX Assembler Version X.XX date           Page: 1
Symbol Table Dump

Symbol  Value  Section
blap    0X0    undefined, external
foo     0X8    .text
```

GF-10-0-U

Figure 9-10. Sample GNX Assembler Symbol Table Listing

The symbols are listed in the order in which they are encountered. The first column of the output is the name of the symbol, the second column is the value (in hexadecimal) of the symbol, and the last column is the name of the section to which it belongs.

9.4.2 Cross-Reference Table Listing

The cross-reference listing will be entitled "Cross-Reference Table". It will be preceded by a formfeed, and will be output to the specified file. If no output file is specified for it, the cross reference will be output either to `stdout` (On UNIX/MS-DOS systems), or to the `.XRF` file (On VMS systems).
Figure 9-11 shows a sample cross-reference source file, and Figure 9-12 shows a sample cross-reference table listing.

```
          .set x, 10
          bsr foo
          movd foo, r0
foo:
          .globl blap
          movd blap, r0
```

GF-11-0-U

Figure 9-11. Sample GNX Assembler Cross-Reference Source File

```
GNX Assembler Version X.XX  date              Page: 1
Cross Reference Table

blap  5+      6
foo   2       3      4^
x     1-
```

GF-012-0-U

Figure 9-12. Sample GNX Assembler Cross-Reference Table Listing

Symbols will be listed in alphabetical order. The numbers listed beside the line numbers are the source lines where the symbol appears. A ^ beside a line number indicates that the symbol is declared on that line. A + beside a line number indicates that the symbol is imported/exported (declared with a `.globl` directive) on that line. A – beside a line number indicates that the symbol is set (or reset with a `.set` directive) on that line.

9.5 GNX ASSEMBLER ERRORS

When the assembler finds an error, it provides an error message through standard error. If the –L option flag on UNIX/MS-DOS systems (or /LIST on VMS) has been selected, the assembler includes the error message in the listing file following the line containing the error. Most errors will inhibit the assembler from generating any further object code (refer to Figure 9-8).

9.6 GNX ASSEMBLER LIMITATIONS

This section contains a list of limitations of the GNX Assembler.

Expression:

Expressions are calculated as 4-byte integers. High order bytes/bits are filled with zero.

Line:

The length of the input line is limited to 64K characters.

Range of values:

The range of values for displacements is:

byte displacement: –64 to 63
word displacement: –8192 to 8191
double-word displacement: –536870912 to 536870911
ie: –(2**29) to (2**29 – 1)

The range of values for floating-point constants is:

single precision: 1.17549436 x 10**–38 to 3.40282346 x 10**38 and
–1.17549436 x 10**–38 to –3.40282346 x 10**38

double precision: 2.2250738585072014 x 10**–308 to
1.7976931348623157 x 10**308 and
–2.2250738585072014 x 10**–308 to
–1.7976931348623157 x 10**308

The range of values for integer constants is:

byte constants: –128 to 255
word constants: –32768 to 65535
double-word constants: –2147483648 to 2147483647
ie: –2**31 to (2**31–1)

Section:

The length of a section name as specified with the `.section` directive must be up to 8 characters.

The number of sections are limited to 10 sections.

- By default, the first 5 sections are: .text, .data, .bss, .link, and .static.
- If there are module table entries there is a .mod section.
- If there are `.ident` directives there is a .comment section.
- Therefore, there are only 3 to 5 sections that can be defined by the user in the assembly source level.

String:

The string length is limited to 256 characters.

Symbol name:

The length of a symbol name in the Cross-reference Table (–x flag on UNIX/MS-DOS or /XREF qualifier on VMS) is truncated to 14 characters.

The length of a symbol name in the Symbol Table Dump (–y flag on UNIX/MS-DOS or /MAP qualifier on VMS) is truncated to 14 characters.

Appendix A

DIRECTIVE SUMMARY

The following is a comprehensive summary of the GNX Assembler Directives.

SYMBOL GENERATION

`.set` *symbol, expression*	sets *symbol* to the value and type specified by *expression*. Scope is local.

DATA GENERATION

[*label*] `.ascii` *string*	generates a string constant. *string* specifies constant value.
[*label*] `.byte`([[*repetition-factor*]]{*expression* | *string*}),,,	generates byte constant or string. *expression* or *string* specifies constant value. *repetition-factor* specifies number of occurrences of value.
[*label*] `.word`([[*repetition-factor*]]{*expression* | *string*}),,,	generates word constants. *expression* specifies constant value.
[*label*] `.double`([[*repetition-factor*]]{*expression* | *string*}),,,	generates double-word constants.
[*label*] `.float`([[*repetition-factor*]]*expression*),,,	generates single-precision floating-point constants.
[*label*] `.long`([[*repetition-factor*]]*expression*),,,	generates double-precision floating-point constants.
[*label*] `.field`([*subfield-length*]*subfield-value*),,,	generates bit fields. *subfield-length* specifies length of field. *subfield-value* specifies field value.

[*label*] `.xpd` *expression*	generates external procedure descriptor. *expression* specifies an external function name.
[*label*] `.xdd` *expression*	generates external data descriptor. *expression* specifies a double-word value.

STORAGE ALLOCATION

[*label*] `.blkb` [*expression*]	allocates consecutive bytes of memory for storage. *expression* specifies the number of bytes.
[*label*] `.blkw` [*expression*]	allocates consecutive words of memory for storage. *expression* specifies the number of words.
[*label*] `.blkd` [*expression*]	allocates consecutive double-words of memory for storage. *expression* specifies the number of double-words.
[*label*] `.blkf` [*expression*]	allocates consecutive double-words for single-precision floating-point storage. *expression* specifies the number of double-words.
[*label*] `.blkl` [*expression*]	allocates consecutive quad-words for double-precision floating-point storage. *expression* specifies the number of quad-words.
[*label*] `.space` *expression*	allocates consecutive bytes for storage. *expression* specifies the number of bytes.

LISTING CONTROL

[*label*] `.title` *string*	prints the specified *string* at the top of each page of the listing file.
[*label*] `.subtitle` *string*	prints the specified *string* at the top of each page of the listing file and below the title string (if any).

MODULE TABLE DIRECTIVES

`.module` *name* [,sb=*static base*][,lb=*link base*] [,pb=*program base*]	declares a module name, associates the text, link, and static local data segments generated by the assembly with the module name and optionally defines a module table entry. *Name* is the module name.
`.modentry` *name* [,sb=*static base*][,lb=*link base*] [,pb=*program base*]	defines a module table entry for a named module. *Name* is the module name.

FILE NAME DIRECTIVE

`.file` "*symbol*"	assigns the source filename *symbol* to the current assembly.

SYMBOL TABLE ENTRY DEFINITION DIRECTIVES

`.def` *symbol*	specifies the start of the definition of a symbol table entry for *symbol*.
`.dim` *expression,,,*	specifies the dimensions of an array variable. Up to four dimensions may be specified.
`.line` *expression*	specifies the source file line number, *expression*, on which a symbol is defined.
`.scl` *expression*	specifies the storage classification, *expression*, of a symbol.
`.size` *expression*	*expression* specifies the size in bytes of a symbol.
`.tag` *symbol*	*symbol* specifies the tag name of a structured data type.
`.type` *expression*	specifies the type, *expression*, of a symbol.
`.val` *expression*	*expression* specifies the value of the symbol that is being defined.

`.endef`	terminates the definition of a symbol table entry.

LINE NUMBER TABLE CONTROL DIRECTIVE

`.ln` *expression1* [, *expression2*]	specifies the source file line number offset from the start of a function and an optional, associated memory address.

MACRO DEFINITION DIRECTIVES

`.macro` *macro-name formal-argument-list*	begins the definition of the macro-procedure.
`.endm`	end the macro-procedure definition.
`.if` *if_condition*	begins a conditional macro assembler statement.
`.elsif` *elsif_condition*	specifies an **elsif** clause for the conditional macro assembler statement.
`.else` *else_conditional_body*	specifies an **else** clause for the conditional macro assembler statement.
`.endif`	ends the conditional macro assembler statement.
`.repeat` [*iteration_count* [,*iteration_var*]]	begins a macro repetitive block.
`.irp` *iteration_var,iteration_list*	begins a special macro repetitive block.
`.endr`	ends a macro repetitive block.
`.exit`	ends the processing of the current repetitive block.
`.macro_on`	enables macro-procedure expansions.
`.macro_off`	disables macro-procedure expansions.
`.include` *included_file*	allows for the inclusion of text from another file.
`.mwarning` *warning_message*	generates an assembler warning message.
`.merror` *error_message*	generates an assembler error message.

PROCEDURE SUPPORT DIRECTIVES

`.proc`	marks the definition point of an ordinary procedure.
`.proct`	marks the definition point of a trap procedure.
`.proci`	marks the definition point of an interrupt procedure.
`.var`	starts the variable block definition.
`.begin`	begins the procedure body.
`.endproc`	ends the procedure body.
`.call`	issues a procedure call.

Appendix B

GNX ASSEMBLER RESERVED SYMBOLS

B.1 INTRODUCTION

This appendix contains lists of the GNX–Version 4 Assembler reserved symbols (*i.e.*, instructions, registers, directives, addressing mode indicators, flags, qualifiers, and temporary labels).

NOTE: The following instructions must be lower-case.

B.2 STANDARD INSTRUCTIONS

absb
absw
absd
acbb
acbw
acbd
addb
addw
addd
addcb
addcw
addcd
addpb
addpw
addpd
addqb
addqw
addqd
addr
adjspb
adjspw
adjspd
andb
andw
andd
ashb
ashw
ashd
bcc
bcs
beq
bfc
bfs
bge
bgt
bhi
bhs
ble
blo
bls
blt
bne
br
bicb
bicw
bicd
bicpsrb
bicpsrw
bispsrb
bispsrw
bpt
bsr
caseb
casew
cased
cbitb
cbitw
cbitd
cbitib
cbitiw
cbitid
checkb
checkw
checkd
cmpb
cmpw
cmpd
cmpmb
cmpmw
cmpmd
cmpqb
cmpqw
cmpqd
cmpsb
cmpsw
cmpsd
cmpst
comb
comw
comd
cvtp
cxp
cxpd
deib
deiw
deid
dia
divb
divw
divd
enter
exit
extb
extw
extd
extsb
extsw
extsd
ffsb
ffsw
ffsd
flag
ibitb
ibitw
ibitd
indexb
indexw
indexd
insb
insw
insd
inssb
inssw
inssd
jsr
jump
lmr
lprb
lprw
lprd
lshb
lshw
lshd
meib
meiw
meid
modb
modw
modd
movb
movw
movd
movmb
movmw
movmd
movqb
movqw
movqd
movsb
movsw
movsd
movst
movsub
movsuw
movsud
movusb
movusw
movusd
movxbd
movxwd
movxbw
movzbd
movzwd
movzbw
mulb
mulw
muld
negb
negw
negd
nop
notb
notw
notd
orb
orw
ord
quob
quow
quod

STANDARD INSTRUCTIONS (CONT)

rdval
remb
remw
remd
restore
ret
reti
rett
rotb
rotw
rotd
rxp
save
sbitb
sbitw
sbitd
sbitib
sbitiw
sbitid
sccb
sccw
sccd
scsb
scsw
scsd
seqb
seqw
seqd
sfcb
sfcw
sfcd
sfsb
sfsw
sfsd
sgeb
sgew
sged
sgtb
sgtw
sgtd
shib
shiw
shid
shsb
shsw
shsd
sleb
slew
sled
slob
slow
slod
slsb
slsw
slsd
sltb
sltw
sltd
sneb
snew
sned
setcfg
skpsb
skpsw
skpsd
skpst
smr
sprb
sprw
sprd
subb
subw
subd
subcb
subcw
subcd
subpb
subpw
subpd
svc
tbitb
tbitw
tbitd
wait
wrval
xorb
xorw
xord

B.3 NS32081 FLOATING-POINT INSTRUCTIONS

absf
absl
addf
addl
cmpf
cmpl
divf
divl
floorfb
floorfw
floorfd
floorlb
floorlw
floorld
lfsr
movbf
movwf
movdf
movbl
movwl
movdl
movf
movl
movfl
movlf
mulf
mull
negf
negl
roundfb
roundfw
roundfd
roundlb
roundlw
roundld
sfsr
subf
subl
truncfb
truncfw
truncfd
trunclb
trunclw
truncld

B.4 NS32181AND NS32381 FLOATING-POINT INSTRUCTIONS

absf
absl
addf
addl
cmpf
cmpl
divf
divl
dotf
dotl
floorfb
floorfd
floorfw
floorlb
floorld
floorlw
lfsr
logbf
logbl
movbf
movbl
movdf
movdl
movf
movfl
movl
movlf
movwf
movwl
mulf
mull
negf
negl
polyf
polyl
roundfb
roundfd
roundfw
roundlb
roundld
roundlw
scalbf
scalbl
sfsr
subf
subl
truncfb
truncfd
truncfw
trunclb
truncld
trunclw

B.5 NS32580 FLOATING-POINT INSTRUCTIONS

absf
absl
addf
addl
cmpf
cmpl
divf
divl
floorfb
floorfd
floorfw
floorlb
floorld
floorlw
lfsr
macf
macl
movbf
movbl
movdf
movdl
movf
movfl
movl
movlf
movwf
movwl
mulf
mull
negf
negl
roundfb
roundfd
roundfw
roundlb
roundld
roundlw
sqrtf
sqrtl
sfsr
subf
subl
truncfb
truncfd
trucnfw
trunclb
truncld
trunclw

B.6 NS32CG16, NS32CG160 AND NS32FX16 HIGH PERFORMANCE GRAPHIC INSTRUCTION

bband
bbfor
bbor
bbstod
bbxor
bitwit
extbl
movmpb
movmpd
movmpw
sbitps
sbits
tbits

B.7 NS32GX320 HIGH PERFORMANCE DSP INSTRUCTION

mulwd
mactd
cmacd
cmuld

B.8 NS32532 CPU INSTRUCTION

cinv

B.9 STANDARD REGISTERS

fp
intbase
mod
psr
r0
r1
r2
r3
r4
r5
r6
r7
sp
upsr
sb

B.10 NS32082 MMU REGISTERS

bcnt
bpr0
bpr1
eia
msr
pf0
pf1
ptb0
ptb1
sc0
sc1

B.11 NS32382MMU REGISTERS

msr
bmr
ptb0
bar
mcr
ptb1
bdr
ivar0
tear
bear
ivar1

B.12 NS32081 FLOATING-POINT REGISTERS

f0
f1
f2
f3
f4
f5
f6
f7

B.13 NS32181,NS32381 AND NS32580 FLOATING-POINT REGISTERS

f0
f1
f2
f3
f4
f5
f6
f7

l0
l1
l2
l3
l4
l5
l6
l7

B.14 NS32532 CPU REGISTERS

bpc
car
cfg
dcr
dsr
ivar0
ivar1
mcr
msr
ptb0
ptb1
tear
usp

B.15 STANDARD DIRECTIVES

.align
.ascii
.blkb
.blkw
.blkd
.bss
.byte
.comm
.data
.def
.dim
.double
.dsect
.eject
.endef
.field
.file
.globl
.ident
.line
.link
.list
.ln
.nolist
.org
.sb
.scl
.section
.set
.size
.space
.static
.subtitle
.tag
.text
.title
.type
.udata
.val
.width
.word
.xdd
.xpd

B.16 FLOATING-POINT DIRECTIVES

.blkf
.blkl
.float
.long

B.17 MACRO DEFINITION DIRECTIVES

.macro
.endm
.if
.else
.endif
.elsif
.repeat
.irp
.endr
.exit
.macro_on
.macro_off
.include
.mwarning
.merror

B.18 PROCEDURE SUPPORT DIRECTIVES

.proc
.proct
.proci
.var
.begin
.endproc
.call

B.19 PROCEDURE SUPPORT PREDEFINED SYMBOLS

param_size
var_size

B.20 MODULARITY DIRECTIVES

.module
.modentry

B.21 ADDRESSING MODE INDICATORS

ext
tos

B.22 FLAGS

b
f
m
u
w
c
i

B.23 NS32332 SETCFG FLAGS

fc
ff
fm
p

B.24 NS32CG160 SETCFG FLAGS

de

B.25 MODULARITY OPTION FLAGS

lb
pb

B.26 NS32CG16 OPTION FLAGS

da
ia
-s
s

B.27 SCALED INDEX QUALIFIERS

b
w
q
d

B.28 NS32532 OPTION FLAGS

a
d
i

B.29 TEMPORARY LABELS

1f
2f
3f
4f
5f
6f
7f
8f
9f

1b
2b
3b
4b
5b
6b
7b
8b
9b

Appendix C

PROGRAM EXAMPLES

C.1 INTRODUCTION

This appendix provides sample assembly programs that illustrate various features of the GNX Assembler. The programs are written in the GNX C compiler style of code generation.

C.2 FACTORIAL NUMBERS

This example illustrates procedure calls between two separately assembled software modules and an object language library module. The assembly language modules implement a factorial number algorithm; the procedure in the library prints out the result.

The two assembly language modules are `main.s`, which contains the procedure `_main`, and `fac.s`, which contains the procedure `_fac`. The procedure `_main` calls the external procedure `_fac`; an argument is passed, and a value is returned in r0.

The procedure `_fac` returns any factorial number that can be represented by a double-word integer. (The factorial of a number *n* is the product 1 x 2 .... *x n*.) If `_fac` is passed as an argument whose factorial cannot be represented as a double-word integer, it returns the integer unchanged and sets the `f` flag of the *psr*. This condition is checked by the flag instruction on return to `_main`.

The `_main` makes three calls to `_fac`, and then calls the C Library routine `_printf` to print the result on standard output. The `_printf` is contained in the object file `/lib/crt0.o`.

The two assembly language modules are assembled separately and then linked with the library object module as follows:

```
as main.s

as fac.s

ld /lib/crt0.o main.o fac.o -lc
```

The module, `main.s`:

```
        .file   "main.s"
        .text
        .globl _main
        .globl _printf          # Import external C Library procedure.
        .globl _fac             # Import external procedure _fac.

        .data
print:  .byte   "%d %d %d\12\0"   # Formatting input for _printf.

        .text
_main::
        enter   [r0],0          # Push previous contents of r0 on stack.

        movqd   1,r0            # Pass input-value 1 to external
        bsr     _fac            # procedure _fac in r0.
        flag                    # Check for out-of-bounds error.
        movd    r0,tos          # Push returned answer in r0 on stack.
        movqd   6,r0            # Prepare to pass input-value 6 to _fac.
        bsr     _fac
        flag
        movd    r0,tos
        addr    @12,r0          # Prepare to pass input-value 12 to _fac.
        bsr     _fac
        flag
        movd    r0,tos

        addr    print,tos       # Push formatting arguments for _printf on
        bsr     _printf         # stack and print answers.
        adjspb  $-16            # Adjust stack pointer to allow for
                                # arguments passed to _printf.
        exit    [r0]            # Restore previous contents of r0.

        ret     0               # Return from _main.
```

The module, `fac.s`:

```
        .file   "fac.s"
        .text
        .globl  _fac

        .data
g:      .double 1
        .double 1
        .double 2
        .double 6
        .double 24
        .double 120
        .double 720
        .double 5040
        .double 40320
        .double 362880
        .double 3628800
        .double 39916800
        .double 479001600

        .text
_fac::
num:    cmpd    r0,$12        # Check for in-bounds (must be less than 13).
        bhi     error         # If not, branch to error handler.
        movd    g[r0:d],r0    # Otherwise, scale-index into the array for
        ret     0             # corresponding factorial result and
                              # move to r0.

error:  bispsrb b'00100000    # Set the psr f bit to "1" for detection by
        ret     0             # flag in calling program.
```

C.3 SQUARE ROOT CALCULATION

This example illustrates local procedure calls, *i.e.*, calls to procedures in the same assembly file. The procedure _main calls the local procedure _sqrt and passes it to an input-value on the stack. The _sqrt calculates the square root of a positive integer using a successive approximation algorithm. If _sqrt is passed a nonpositive integer, the integer is returned unchanged on the stack, and the `f` flag of the *psr* is set. Otherwise, the answer (*i.e.*, the closest integer less than or equal to the square root of the input-value) is returned on the stack and printed out using _printf.

The module sqrt.s is assembled and linked as follows:

```
as sqrt.s

ld /lib/crt0.o sqrt.o -lc

        .file       "sqrt.s"
        .text
        .globl      _main
        .globl      _printf     # Import external C Library procedure.
        .data
print:  .byte "%d %d",0xa,0x0   # Formatting input for _printf.

        .text
_main::
        enter    [],4
        movqd    $1,tos         # Pass input-value 1 to local procedure _sqrt
        bsr      _sqrt          # via stack.
        flag                    # Check for illegal negative input-value.
        movqd    $4,tos         # Pass input-value 4 to local procedure _sqrt
        bsr      _sqrt
        flag
        addr     print,tos      # Push formatting input for _printf on stack.
        bsr      _printf        # Print formatted answers.
        adjspb   $-12           # Adjust stack pointer in allowance for input
        exit     [ ]            # to _printf.
        ret      0              # Return from _main.

_sqrt:
        enter [r0, r1, r2],0    # Save contents of registers to be used
                                # on stack.
        movqd    $1,r0          # Start guessing square-root as 1.
        movd     8(fp),r1       # Get the passed parameter on the stack.
        cmpd     r1,$0          # Check for illegal negative input.
        ble      error          # If so, branch to error-handler.

loop:   movd     r1,r2          # Otherwise, make a copy and divide the copy
        divd     r0,r2          # by the initial guess.
        cmpd     r0,r2          # Is the answer ready yet?
        addd     r0,r2          # Sum the result and the guess.
        ashd     $-1,r2         # Take their average.
        movd     r2,r0          # Make the next guess.
        bhi      loop           # If answer not ready yet, continue.
        movd     r0,8(fp)       # Otherwise, return the answer
        br       exits          # and exit.

error:  bispsrb  b'00100000     # Set the PSR F bit to "1" for detection by
                                # flag after return from procedure.
exits:  exit     [r0,r1,r2]
```

C.4 ACKERMAN'S FUNCTION

This example contains an assembly language program produced by the GNX C compiler. The C program implements Ackerman's function, a well-known example of a recursive procedure that terminates for all positive integer values of its two parameters. Following the C program is the optimized assembler output from the GNX C compiler.

The program is compiled as follows:

```
cc -O -S ack.c
```

The C program:

```
main()
{       int i=3,j=3;
        printf("%d\n",ack(i,j));
}
ack(a,b)
int a,b;
{
        if (a==0)
                return(b+1);
        else if (b==0)
                return(ack(a-1,1));
        else
                return(ack(a-1,ack(a,b-1)));
}
```

The optimized assembly code for the above C program:

```
.text
.data
.text
.globl  _main
                .file "ack.c"
        .align  4
.data
.text
.globl  _ack
.data
.globl  _printf                 # Import external C library procedure.
.L17:
.ascii  "%d\12\0"               # Formatting input for _printf.
.text
_main:
        enter   [],8            # Allow for the amount of data to be
        movqd   3,-4(fp)        # pushed on stack.
        movqd   3,-8(fp)        # Allocate input-values to data storage
        movd    -8(fp),tos      # area of procedure _main and push on
        movd    -4(fp),tos      # stack in preparation to call external
```

```
        bsr     _ack            # procedure _ack.
        adjspb  $-8             # Adjust stack pointer in allowance for
        movd    r0,tos          # input to _ack and push returned answer
        addr    .L17,tos        # in r0 on stack
        bsr     _printf         # Print answer.
        adjspb  $-8             # Adjust stack pointer in allowance for
        exit    []              # input to _printf and exit _main,
        ret     0               # adjusting stack for initial data input.
        .align  4

_ack:
        enter   [],0
        cmpqd   0,8(fp)         # If a is equal to 0 then,
        bne     .L23            #
        movqd   1,r0            # add 1 to b,
        addd    12(fp),r0       # using r0, and
        br      .L20            # branch to exit _ack.

L2000001:
        movqd   1,tos           # Push 1 on stack as 2nd argument
                                # to _ack.
L2000005:
        movd    8(fp),r0        # Move a to r0,
        addr    -1(r0),tos      # subtract 1 and push on stack as 1st
        bsr     _ack            # argument to _ack, then call _ack.
        adjspb  $-8             # Adjust stack pointer in allowance for
                                # arguments to _ack pushed on stack.
.L20:
        exit    []              # Exit _ack.
        ret     0
.L23:
        cmpqd   0,12(fp)        # If a is not equal to 0 and b is equal
        beq     L2000001        # to 0 then branch to L2000001.
        movd    12(fp),r0       # Else move b to r0, subtract 1 and
        addr    -1(r0),tos      # push on stack as 2nd argument to _ack.
        movd    8(fp),tos       # Then push a on stack as 1st argument
        bsr     _ack            # to _ack and call _ack.
        adjspb  $-8             # Adjust stack pointer for arguments.
        movd    r0,tos          # Push result in r0 on stack as 2nd
        br      L2000005        # argument to _ack in recursive call.
                                # Branch to L2000005 for 1st argument.
```

C.5 STRING SORTING

This example implements a bubble-sorting algorithm for an array of pointers to strings. A bubble-sorting algorithm performs successive exchanges of unordered neighbors. The algorithm may be represented in C as follows:

```
string_sort(e_cnt,array)
char *array[];
int e_cnt;
{
   char *temp;
   int f, i;
   f = e_cnt;
   while ( f-- > 0 ) {
                    for (i = 0; i < f ; i++) {
                              if (strcmp(array[i],array[i+1]) > 0) {
                                        temp = array[i];
                                        array[i] = array[i+1];
                                        array[i+1] = temp;
                              }
                    }
   }
}
```

An assembly language module, sort.s, implementing the bubble sort algorithm is given below. The external procedure _sort performs a bubble sort on a passed string array. The maximum allowed length of a string is "max_length," an imported variable. The array address ("array") and element count ("e_cnt") are passed on the stack, with "e_cnt" on top.

```
         .file   "sort.s"
         .globl  max_length
         .dsect  args, 8                  # Set argument offsets from fp.
array:   .blkd
e_cnt:   .blkd
         .text
_sort::
         enter   [r0,r1,r2,r3,r4,r7],0
         movd    e_cnt(fp),r3             # Set e_cnt to range-limit for loop1.
loop2:   addqd   -1,r3                    # Set range-limit to range-limit - 1.
         cmpd    0,r3                     # Branch to p_exit if range-limit
         beq     p_exit                   # is equal to 0.
         movqd   0,r7                     # For i = 0 ...
loop1:   cmpd    r3,r7                    # ... to range-limit.
         beq     loop2                    # Branch to loop2 on reaching range-limit.
         movd    max_length,r0            # Set string-length limit for cmpsb.
         movd    0(array(fp))[r7:d],r1    # Set up array[i].
         addqd   1,r7                     # i = i + 1.
         movd    0(array(fp))[r7:d],r2    # Set up array[i+1].
         movqd   0,r4                     # Set up end of string.
         cmpsb                            # If string(array[i]) > string(array[i+1])
         bls     loop1                    # continue, otherwise branch to next pair.
         addr    0(array(fp))[r7:d],r0    # Get address of array[i+1].
         movd    -4(r0),r1                # temp = array[i].
         movd    0(r0),-4(r0)             # array[i] = array[i+1].
         movd    r1,0(r0)                 # array[i+1] = temp.
         br      loop1
p_exit:  exit    [r0,r1,r2,r3,r4,r7]
```

C.6 MODULAR CODE EXAMPLE

This example shows a *Series 32000* module built from a single source file.

```
1     # Declare the module
2     # We specify the static and link base, the program base defaults to .text.
3
4                                       .file     "hello.s"
5     M00000000   1c000000              .module   hello, sb = .static, lb = .link
                  18000000
                  00000000
                  00000000
6
7                                       .link                # Begin link segment,
8                                                            # lb will point here
9
10    L00000000   00000000   printf:    .xpd      _printf    # Local link table entry for _printf
11
12                                      .text                # Begin code segment,
13                                                           # pb will point here
14
15                           _main::
16    T00000000   820000                enter     [], 0
17    T00000003   e7d5c000              addr      msg, tos
                  0000
18    T00000009   22c00000              cxp       printf
                  00
19    T0000000e   7ca5fc                adjspb    $-4
20    T00000011   9200                  exit      []
21    T00000013   3200                  rxp       0
22
23                                      .static              # Begin static segment,
24                                                           # sb will point here
25                           msg:
26    S00000000   48656c6c              .byte     "Hello, World!\012\0"
                  6f2c2057
                  6f726c64
                  210a00
```

Appendix D

INITIALIZATION OF INTERRUPTS

The following skeleton program illustrates a method of initializing the necessary registers and tables to process the first 10 standard interrupts in a *Series 32000* system with a single ICU. This same technique can be used to initialize the vectored interrupts when needed.

In order to load interrupt vectors at run-time, the appropriate procedure descriptors may be stored in a link table and moved into place during program initialization.

Because the offset portion of the procedure descriptor is 16 bits, the interrupt routines must be within the first 64 Kbytes (65536 bytes) of address space. Since the linker loads files in command line order, this can be accomplished by specifying the object file that contains the interrupt routines early in the load command.

```
         .text
start::
#
#        Initialize Intbase Register to point to the Interrupt
#        Vector Table:
#
         addr   intvec,r0    # Get address of Interrupt Vector Table.
         lprd   intbase,r0   # Save it in Intbase Register.
#
#        Code for process:
#
         ...
         ...                 # Code for program.
         ...
#
#        Code for interrupts:
#
nvi:                         # Code for non-vectored interrupt.
         ...
         ...                 # Code to process interrupt.
         ...
         reti                # Return to interrupted routine.
nmi:                         # Non-maskable interrupt.
         ...
         ...                 # Code to process interrupt.
         ...
         reti                # Return to interrupted routine.
abt:                         # Abort interrupt.
```

```
        ...
        ...                   # Code to process interrupt.
        ...
        reti                  # Return to interrupted routine.
ignore:                       # Ignore state.
        reti                  # Return to interrupted routine.
        .data
#
#
#       Build Interrupt Vector Table entry by entry.
#       Each .xpd directive initializes one entry in the table
#       with a procedure descriptor for the appropriate entry
#       point.  Note that this program chooses not to use
#       interrupts 3 through 10, they are set up to be
#       ignored.
#
#
#
intvec: .xpd   nvi            # non-vectored interrupt (always 0)
        .xpd   nmi            # non-maskable interrupt (always 1)
        .xpd   abt            # abort interrupt (always 2)
        .xpd   ignore         # FPU (always 3)
        .xpd   ignore         # illegal operation (always 4)
        .xpd   ignore         # supervisor call (always 5)
        .xpd   ignore         # divide by zero (always 6)
        .xpd   ignore         # flag (always 7)
        .xpd   ignore         # breakpoint (always 8)
        .xpd   ignore         # trace (always 9)
        .xpd   ignore         # undefined instruction (always 10)
#       11-15 reserved
#       16-31 vectored interrupts
```

Appendix E

SERIES 32000 STANDARD CALLING CONVENTIONS

E.1 INTRODUCTION

The main goal of standard calling conventions is to enable the communication between the routines of one program consisting of different modules, even when written in multiple programming languages. The *Series 32000* standard calling conventions support various special language features (such as the ability to pass a variable number of arguments, which is allowed in C) by using the different calling mechanisms of the *Series 32000* architecture. This convention is employed only to call "externally visible" routines. Calls to internal routines may employ even faster calling sequences, by passing arguments in registers, for instance.

Basically, the calling sequence pushes arguments on top of the stack, executes a call instruction, and then pops the stack, using the fewest possible instructions to execute at the maximum speed. The following sections discuss the various aspects of the *Series 32000* standard calling conventions.

E.2 CALLING CONVENTION ELEMENTS

Elements of the standard calling sequence are:

- **The Argument Stack**
  Arguments are pushed on the run-time stack from right to left. Therefore, the first (left-most) argument is always at a constant offset from the frame pointer (fp) regardless of how many arguments have been passed. This is important because C allows a variable number of arguments. This does not mean that the actual parameters are always evaluated from right to left. Programs cannot rely on the order of parameter evaluation.

  For reasons of efficiency, the run-time stack is required to be aligned to a full double-word boundary. Argument lists always use a whole number of double-words; integer and pointer values use a double-word (by extension, if necessary), floating-point values use eight bytes and are represented as *long* values, structures use a multiple of double-words.

  The above conventions allow writing functions which take a variable number of arguments of unknown types, such as the printf function.

  Note that the stack alignment is maintained by all of National Semiconductor's optimizing compilers through aligned allocation and de-allocation of local variables. Interrupt routines and other assembly-written interface routines are expected to maintain this double-word alignment.

The caller routine must pop the arguments off the stack, upon return from the called routine.

- **Saving registers**
  General registers R0, R1, and R2 and floating registers F0, F1, F2, and F3 are `volatile` or `unsafe` registers whose value may be changed by a called routine. These registers need not be saved upon procedure entry, nor restored before exit. If the other registers (R3 through R7, F4 through F7) are used, their value must be saved (onto the stack) by the called routine immediately upon procedure entry and restored immediately before executing the return instruction.

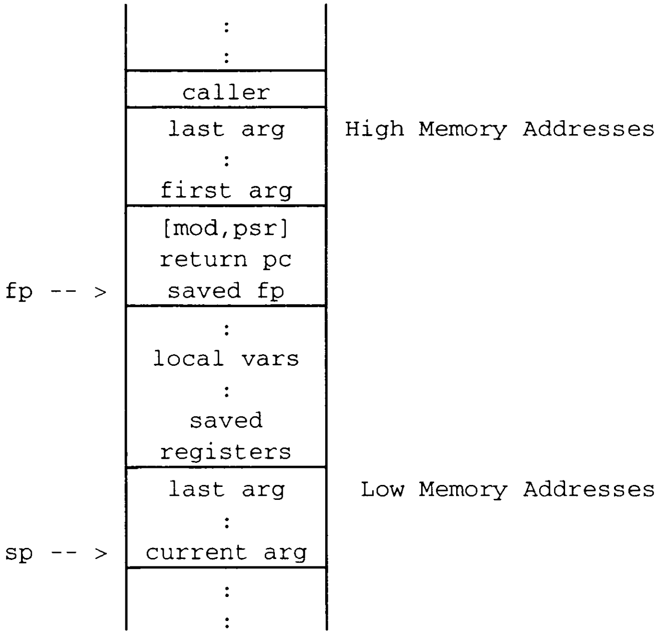
NOTE: Interrupt and trap service routines are required to save/restore all registers they use.

- **Returned Value**
  An integer or a pointer value that is returned from a function is returned in (part of) register R0.

  A long floating-point value that is returned from a function is returned in register pair F0-F1. A float-returning function returns the value in register F0.

  If a function is defined to return a structure, the calling function will pass an additional argument at the beginning of the argument list. This argument points to where the called function will return the structure. The called function copies the structure into the specified location during execution of the return statement. Note that functions which return structures must be correctly declared as such even if the return value is ignored.

When the C optimizing compiler builds an argument list, the layout of the stack looks like this:



Example: given a call to a function **func**(*first, second, third, last*) :

```
i = func(1, x, *cp, j);
```

with these variable declarations:

float x;
register char *cp;
int j;

the compiler might generate the following code:

```
movd      -8(fp),tos     #last argument
movzbd    0(r7),tos      #byte argument
movfl     -4(fp),tos     #float argument
movqd     1,tos          #first argument
bsr       _func
adjspb    -20            #pop args off stack
movd      r0,r6          #save return value
```

func might look as follows:

```
enter   [r7,r6],12   #save regs, alloc vars
movl    f6,tos       #save floating regs f6 and f7
        :
movd    8(fp),r1     #put first arg in temp reg
        :
movd    r7,r0        #return value
movl    tos,f6       #restore floating regs
exit    [r7,r6]      #restore regs
ret     0            #do not pop off args
```

The standard calling sequence decreases the chance of an error which could destroy the stack. Maintaining the stack is crucial since the debugger cannot trace a destroyed stack, and the user must know what functions in a program are currently active.

Appendix F

COMPATIBLY-SUPPORTED MACROS

F.1 INTRODUCTION

This appendix describes the Version 2 and 3 macro-assembler. For compatibility purposes this macro-assembler is still supported in this release, but will be obsolete in Version 5. It is not compatible with the new Version 4 macro-assembler, which is described in Chapter 8.

This old macro-assembler version must be invoked by the -MC invocation option for the UNIX environment, and by the /MCOMPATIBILITY invocation option for the VMS environment.

The following sections explain the process of defining and using old macros.

F.2 DEFINITION OF TERMS

Macro Definition	A method of giving a name to a sequence of instructions. After the macro has been defined, the programmer can write the macro name instead of the sequence of instructions.
Macro Usage	The use of the macro name as an opcode, operand, directive, expression or partial expression.
Macro Expansion	The process of replacing the line containing the macro name with the sequence of instructions from the macro definition. Every formal argument in the macro definition is replaced by the actual argument specified in the macro name.
Formal Arguments	Arguments that are used throughout the sequence of instructions in the macro definition.
Actual Arguments	Arguments specified during the actual use of the macro name. These arguments will replace the formal arguments during the macro expansion.

F.3 DEFINING A MACRO

A macro definition consists of three parts: a header, a body, and a terminator. The macro definition is written only once but can be used any number of times. A macro may not be redefined in a single assembly session.

F.3.1 The Macro Header

The header of a macro definition gives the name of the macro being defined. The header consists of the `.macro` directive, the name of the macro, and a list of arguments to be used in the definition. The `.macro` directive begins the definition of the macro. The macro definition header is followed by the body of the macro definition. The format of the macro definition header is as follows:

`.macro` *macro-name formal-arguments-list*

where *formal-arguments-list* is a list of formal arguments, denoted by ?*n*, and separated by any of the following delimiters:

```
,   &   [    ]    (    )
```

The argument number *n* ranges from 1 to 9. Some examples of *formal-arguments-list* are

```
?1, ?2, ?3       #formal arguments are ''?1'',  ''?2'', ''?3''
                 #delimiter is '','' 
[?1], ?2, &?3&   #formal arguments are ''?1'', ''?2'', ''?3''
                 #delimiters are ''['', '']'', '','' ''&''
```

The argument number *n* does not need to follow a consecutive order, so the following *formal-arguments-list* is also allowed:

```
?2, ?7, ?4
```

Macro names may not be the same as standard assembler mnemonics, directives, or user-defined symbols.

F.3.2 The Macro Body

The body of a macro definition consists of the statements to be inserted into the assembler source code when the macro name is used. The types of statements that are allowed in the body of the macro are current assembly language statements, directives, usage of other macros, and expressions or partial expressions. The macros being used within a macro definition body may be undefined during the time of macro definition, but they must be defined before actual expansion take place.

The body of a macro definition can be empty, *i.e.*, it can contain no statements.

The body of a macro definition cannot contain the definition of another macro name; macro definitions may not be nested.

The formal arguments used in the body of the macro definition are of the form ?*n*, where *n* ranges from 1 to 9 as specified in the macro header. During the use of the macro name, the macro body statements are inserted in the source code at the place where the macro is used with every formal parameter being replaced by the corresponding actual parameter using a "string" substitution.

F.3.3 The Macro Terminator

A macro definition is terminated by the .endm pseudo-instruction. During a macro definition, an `.endm` directive must be found before another `.macro` directive may be used.

The format of the `.endm` directive is as follows:

```
.endm
```

The following are three examples of a macro definition:

Example 1:

```
.macro  store ?1, ?2, ?3    # macro name is "store"
                            # there are 3 formal arguments
.long   ?1                  # ?1 is first argument
.word  ?3                   # ?3 is third argument
.byte   ?2                  # ?2 is second argument
.endm
```

Example 2:

```
.macro loc                  # macro name is "loc"
4(fp)                       # definition body is an expression
.endm
```

Example 3:

```
.macro fun [?1], ?2, &?3&  #macro name is ''fun''
                            #there are 3 formal arguments
                            #first argument is delimited by ''[ ]''
                            #second argument is delimited by '', ,''
                            #third argument is delimited by ''& &''
enter   [?1], ?2
.byte  ?3
exit    [?1]
.endm
```

F.4 USING A MACRO

Macros can be used as directives, operands, opcodes, expressions and even partial expressions. A macro is invoked by using the macro name, followed by a list of the actual arguments to be substituted into the macro definition body if the macro has formal arguments. The format of macro usage is as follows:

macro-name actual-arguments-list

where *macro-name* is the name of a macro which has already been defined, and *actual-arguments-list* is a list of arguments and delimiters following the prototype as specified by the *formal-arguments-list* of the macro definition header. The actual arguments obtained then replace the formal arguments in the macro definition body.

F.4.1 Arguments In Macros

The arguments in the *actual-arguments-list* must be separated by delimiters as specified in the *formal-arguments-list* of the macro definition header. These actual arguments will replace the formal arguments in the order in which they are written using a "string" substitution. Actual arguments not supplied will result in missing arguments during macro expansion. The number of actual arguments associated with the use of the macro must equal the number of formal arguments specified in the macro definition.

All actual arguments will be evaluated at argument usage time, that is, during the time of the expansion of the actual arguments, as opposed to during the time the macro name is used. The actual arguments should be thought of as delimited by leading and trailing spaces so that text concatenations are not possible.

Using the macros defined in Section 8.2, the following are examples of macro usage:

Example 1:

```
store  3.3, 2, 1
```

Example 2:

```
movd   6, loc
```

Example 3:

```
fun    [r0,r1,r2],20, &1,2,3,4,5&
```

Expansion of the macro calls are as follows:

Example 1:

```
.long   3.3               # 3.3 is the first argument
.word   1                 # 1 is the third argument
.byte   2                 # 2 is the second argument
```

Example 2:

```
movd    6, 4(fp)
```

Example 3:

```
enter   [r0,r1,r2], 20  # r0,r1,r2 is the first argument
                        # 20 is the second argument
.byte  1,2,3,4,5        # 1,2,3,4,5 is the third argument
exit    [r0,r1,r2]
```

Appendix G

GLOSSARY

.gnxrc (gnx.ini on VMS) A GNX target specification file that is used by GNX tools to obtain the CPU, FPU, MMU, system bus-width, and OS target specifications.

Assembly Procedure A procedure defined in the assembly source. It provides for easy programming and interface with HLL written code.

Assembly Program segment Part of an assembly program that resides in a contiguous area. Every GNX assembly program produces at least three program segments in the output object file: text, data, and bss. These segments correspond to the .text, .data, and .bss sections of the COFF file. Other *Series 32000* segments or user-defined sections may be included in the assembly source file.

Assembly directive Provides the assembler with control information. Directives define labels, generate data, define procedures, control program listings, control macro-assembly, allocate storage, control linkage, define module table entries, control line number tables, control program segments, define symbol table entries, and define file names.

Assembly expression A combination of terms and operators which evaluate to a single value and type. Valid expressions include addresses and integer expressions, but not floating-point expressions.

Assembly label A user-defined symbol specified at the beginning of an assembly statement, followed by a colon (:) or a double colon (::).

Assembly statement Composed of an optional label, which is a user-defined symbol; followed by an optional instruction or directive mnemonic that is an assembler-reserved symbol; followed by optional operands that are composed of symbols, constant values, and delimiters.

Built-in Macro Functions A set of macro-assembler functions used to manipulate strings, lists, type conversions and *Series 32000* operands.

COFF Acronym for the Common Object File Format. This is the standard object file format for the Unix System V operating system, and for the GNX software tools. A COFF file contains machine code and data and additional information for relocation and debugging purposes.

Calling convention A standard GNX convention for calling procedures from either an assembly or a HLL written code. It defines the way parameters are passed, register usage and how a value should be returned.

Compound Assembly expression An expression constructed from other assembly expressions using unary and binary operators.

Conditional Macro Statement Sequences of statements specified between the .if and the .endif directives. They are generated according to a condition specified with the .if directive.

Cpp An acronym for the C preprocessor.

Cross configuration When the compilation and execution of the compiled program are done on different machines (the host and target machines are different).

DBUG GNX symbolic debugger. DBUG provides a window-oriented user interface for both X-windows and ASCII terminals. It is used for the symbolic debugging of high level and assembly language programs.

Development board The 32000 based system used for developing/running programs and user applications.

Displacement An integer constant that is specified as part of an instruction operand. Its value is an offset added to a specified base address for operand address calculation. It may be encoded as either a byte, word, or double-word.

Displacement operand A displacement size specification that determines a displacement encoding as either byte, word, or double-word.

Dummy segment Defines a symbolic offset for each of its defined labels. It does not contain generated code or data and does not allocate space. It is useful for overlaying portions of specific segments.

External symbol A symbol which is defined outside the assembled module. It can be defined either in another assembly module or in a HLL module.

Floating-point constant An immediate *Series 32000* floating-point value. Can be either a four byte single precision value or an eight byte double precision value.

Global symbols Global symbols are symbols to be used by multiple software modules, either assembly or HLL modules.

Host machine The machine on which the compiler runs.

Initialized data segment Contains initialized data, follows the .data directive, and corresponds to the .data section of the COFF file. The initialized data segment has the same functionality as initialized data in the C language. This functionality enables the start-up of a target system with an automatically initialized data area.

Instruction operand The *Series 32000* instruction operand is defined by the microprocessor architecture as one of nine possible addressing modes: register, immediate, absolute, register-relative, memory space, memory-relative, external, top-of-stack, or scaled index.

Integer constant An immediate integer value. Can be specified either in decimal, hexadecimal or octal format. Integers can be used within assembly expressions that are part of either an instruction or directive operand.

Link segment A special segment of the assembly program that corresponds to the .link section of the COFF file. The link segment defines a module's link table, thereby supporting *Series 32000* modularity. The actual link table entries are specified following the assembly .link directive.

Location counter A relocatable memory address of the current statement within the currently assembled segment.

Macro Procedure Known by the more common name: macro. Consists of legal assembly statements to be expanded on macro call, according to given parameters.

Native configuration When the compilation and execution of the compiled program are done on the same machine (the host and target machines are the same).

Object file A file that is the output of either the assembler or the compiler. It contains compiled code, data and additional control information such as relocation or symbolic information. The assembler's object file conforms to the COFF Common Object File Format.

Option The UNIX term for a parameter, specified on the command line, that is used to control the utility.

Output listing An optional assembler output of the assembled source file. It displays the original assembly source, along with additional useful information. Each source line has an annotated line number, segment type information, and the generated code or data. Macro expansions are also displayed where applicable.

Procedure Body A part of the assembly procedure support, defining the procedure code to be executed. Proper entrance and exit is ensured by beginning and ending the procedure body using the .begin and the .endproc directives, respectively.

Procedure Call A part of the assembly procedure support that calls either an assembly or a HLL procedure from an assembly code. Calling is done using the .call directive, with the operands being the procedure name and actual parameters.

Procedure Definition A part of the assembly procedure support, defining an assembly procedure. Assembly procedure is specified between .proc and .endproc directives. It consists of a procedure body and optional formal parameters, local variables, and registers to be saved.

Procedure Parameters A part of the assembly procedure defining formal parameters. Procedure parameters are defined after the .proc directive.

Procedure Variables A part of the assembly procedure defining local variables. Procedure variables are defined after the .var directive.

Qualifier The VMS term for a parameter, specified on the command line, that is used to control the utility.

Relative value A symbol or expression that specifies an address within one of the COFF sections or the corresponding assembly program segment. Because such addresses are not bound to actual memory locations until link-time, their value is relative to the base or starting address of the segment. Relative values are relocatable, and have a relocatable entry in the generated COFF object file. They are resolved later at link-time.

Relocatable object files Output of the assembly process. Relocatable object files may be linked to create executable files for a *Series 32000* target.

Repetitive Macro Statement Sequences of statements specified between the .repeat or .irp directives and the .endr directive. They are generated repeatedly according to an iteration index specified with the .repeat directive.

Return value An integer or floating-point value that is returned by a function through register R0 or F0, respectively, according to the GNX standard calling convention.

Series 32000 instruction A *Series 32000* instruction mnemonic. Should appear within a text section in order to be executed.

Source file Assembler input. The source file is a text file containing the source program to be assembled.

Static segment A special segment of the assembly program. It follows the .static directive, and corresponds to the .static COFF section. The static segment is used for defining a static base area for each module in the *Series 32000* modularity mode. The static base area maintains specific data for each module, which is considered to be part of the module's environment (i.e. it is saved when switching to another module and restored on returning to the module). The static segment is especially useful in real-time embedded applications.

Target machine The machine on which the program being compiled will run.

Text segment Contains code for execution, follows the .text directive and corresponds to the .text section of the COFF file.

Uninitialized data segment Contains uninitialized data, follows the .bss or the .udata directive. Corresponds to the .bss section of the COFF file.

INDEX