GENIX Assembler Reference Manual

September, 1983

1. Introduction

The purpose of this document is to describe the assembler provided on the GENIX™ development system. The assembler is also part of the GENIX cross-support package. This manual should be read with the *NS16000™ Programmer's Reference Manual* as a reference to the instruction set.

This manual assumes an understanding of the NS16000 family architecture and some familiarity with assembly language programming. If greater detail is needed on assembly language programming the NS16032 see the NS16000 Cross-Assembler User's Manual, which describes a related assembler.

1.1. Assembler Overview

1.1.1. General Description

The assembler functions as a component of the GENIX environment or as a part of the GENIX cross-support package available under Berkeley 4.1bsd UNIX[1] on a VAX[2]. It is invoked explicitly when a user issues the *as* command under GENIX or *nasm* command under UNIX or implicitly when used by compilers to generate object code from compiler generated assembler source.

The assembler accepts a source file as input and produces an object code file which includes a text (code) segment, an initialized data segment, and a symbol table, in a form acceptable to the GENIX linker *ld*(1). Each object code file produces one NS16000 software module when loaded.

The source file is composed of statements of two kinds: directives and instructions. The syntax and semantics of all the instructions except LXPD are described in the *NS16000 Programmer's Reference Manual*, while the syntax and semantics of all the directives and the LXPD instruction are contained in this manual.

When the assembler finds an error it provides the relative character offset and an error message through standard output, to the invoking process. Depending on when the error was discovered, the character offset provided is either the beginning of the line or what the assembler considers the offending field or subfield.

Most errors will inhibit the assembler from generating an object file.

1.1.2. Features

The GENIX assembler provides more features than a basic assembler:

- extensions to common features, e.g., unary operators for one's complement and complement of the least significant bit.
- features supporting the NS16000 family modular software.
- features supporting the use of procedures in assembly language.

The assembler minimizes the size of any displacement, including those with references and forward references to labels or expression with symbols.

The use of symbols is extended to allow a symbol to represent not only an address, but any legal operand such as an addressing mode (a way to calculate an address). The assembler also has directives which can be used to separate instructions and data interspersed in the source file into separate text (code) and data segments. The directives which are used to separate the text and code segments are part of a group of "scoped" directives. The scoped directives are also the directives which define the addressing mode for symbols. Each of the scoped directives has a variable "storage class" assoociated with it. It is this storage

---

GENIX, and NS16000 are trademarks of National Semiconductor Corporation.

[1] Unix is a trademark of Bell Laboratories.

[2] VAX is a trademark of Digital Research Corporation.

GENIX was derived from 4.1bsd which was developed at the University of California at Berkeley.

The information in this document is for reference only and is subject to change without notice.

class which controls the addressing mode associated with a symbol defined in some particular scope. An example of a scoped directive is *.static*. Any symbol declared between a *.static* directive and another scoped directive (or the *.endseg* directive) will refer to a location which is SB-relative. Any data associated with that location will be stored in the data segment of the module. The scoped directives can be nested up to sixteen levels. The *.endseg* directive ends the scope of the directive currently in force.

Certain directives support the modular software features of NS16000 family. For example, *.import* causes a link table entry to be generated for a symbol that is external to the assembly. All references to the external symbol will reference the link table entry. The actual symbol value will be resolved at link time.

The assembler provides directives to support the use of procedures in an assembly language program. These directives define the start and name of a procedure, the inputs to the procedure, the return values, the local variables, and the body of the procedure.

2. Metasyntactic Conventions

The following notational conventions are used throughout this document.

item1 | item2 | item3	Vertical bars indicate that one of item1, item2, and item3 must be present.
[text]	Brackets indicate that 'text' is optional.
text...	Elipses indicate that 'text' may occur multiple times.
text	Boldface text in an example or command line indicates a literal. Elsewhere, boldface indicates the defining occurrence of a term.
text	Italics (or underlining) is used when a literal is referenced in explanatory text.

3. Symbol Construction

A **symbol** is a name that has a value. A **local symbol** only exists for use within the current source file (software module). A **global symbol** is placed in the object file for use with other module files. A **symbolic address** has a value that is a location, referenced by an addressing mode which may be relocatable (see Chapter 5).

A symbol is composed of an alphabetic character or underscore or percent sign or dot (a-z A-Z _ % .) followed by as many as 31 alphanumeric characters, underscores or dots (a-z A-Z _ . 0-9). Symbols may be predefined, such as instruction mnemonics, or user-defined as labels. Predefined symbols are not case sensitive, that is predefined symbols may be typed in either upper- or lower-case. However, user symbols are differentiated based on case unless the assembler is invoked with the *-i* (ignore case) option. If *-i* is used, all upper-case characters in symbol names are converted to lower-case. A label should not have the name of any valid command, operator, or addressing mode. A label may be referenced before it is defined, and it may be defined only once.

4. Constants

A **constant** is a fixed value. A constant's value may be numeric, either integer or floating-point, or an ASCII character or string.

4.1. Numeric Constants

Integer values may be decimal (decimal is the default radix for integer constants, a decimal value is also indicated by *d'*digits), binary (*b'*digits), octal (*o'*digits or *q'*digits), or hex (*h'*digits or *x'*digits).

Floating-point values are indicated by the presence of a decimal point or the E or e exponent flag. *f'* followed by eight hex digits will be interpreted as an encoded short floating-point value, and *l'* followed by sixteen hex digits as an encoded long floating-point value.

Examples:

decimal constants:	1
	d'12345678
binary constants:	b'1
	b'11110000111100001111000
octal constants:	o'32570
	q'7700
hexadecimal constants:	h'e0
	x'f012ff89
floating constants:	-5.0705
	9.0e+34
	f'e01267ac
	l'12a945bd4266ecf0

Decimal constants are sign-extended to double-words. Hex, octal and binary constants are zero-extended, and floating constants are encoded in either long or short floating-point format, depending on what instruction or directive it is used with. Short floating-point numbers range from a most positive value of 3.40282346E38 to a least positive value of 1.17549430E-38, and the corresponding negative values. Long floating-point point numbers range from a most positive value of 1.79769313486231305E308 to a least postive value of 2.22507385850720100E-308, and the corresponding negative range. Integer constants may have the range from -2147483648 to 2147483647 for double-word constant.

4.2. ASCII Character or String Constants

Character or string values are specified with single or double quotation marks. If a string contains a quote character it may be quoted by the use of an additional quotation mark, as 'it''s' or "it""s".
Examples:

ASCII constants:	"@"
	'Wow!'

If the string requires less space than allocated for its storage, the string is zero-extended to fill the storage.

5. Addressing Modes

An **addressing mode** is one of the NS16000 general addressing modes (see Figure 1).

Addressing Mode	Syntax
Register	Rn or Fn
Immediate	constant or an expression evaluating to a constant
Absolute	@address
Register Relative	offset(Rn)
	offset(FP)
	offset(SP)
	offset(SB)
	offset(PC)
Memory Relative	offset_ 2(offset_ 1(FP))
	offset_ 2(offset_ 1(SP))
	offset_ 2(offset_ 1(SB))
Top of Stack	TOS
External	EXT(index)+offset
	or EXT(index) ;where offset=0
Scaled Indexed	addr. mode[Rn:B,W,D,or Q]

Figure 1. Addressing Mode Syntax

The address, index, or offsets required by an addressing mode are specified with a constant.

6. Expressions

Expressions are constructed by combining constants and addressing modes with operators.
Examples:

```
24
R6
* - 10
10 + 40(FP)
(4+4)(SP)
(0(FP)-3(FP))+((h'a0(4(SB))-(0(4(SB))+2
((EXT(3)+(12(FP)-8(FP)))-(EXT(3)+b'10000))+*
```

6.1. Expression Evaluation

A operand is evaluated in order of the precedence of its operators. Evaluation is done from left to right between operators of equal precedence. The precedence of the operators may be overriden with the use of parentheses to group operators.

In general, sensible combinations of constants, relocation modes and operators are allowed. For example, 10(FP)+15 is allowed and results in a value 25(FP), while 10(FP)+15(SB) is not allowed. The following paragraphs give the rules for allowed combinations of constants, addressing modes and operators. Each descriptive paragraph is introduced with a line indicating the terms and operators discussed in the paragraph.

(constant) (any operator) (constant) produces (constant)

Any operand can combine any two constants to produce a constant.

(FP,SP,SB,EXT,register relative mode) (+,OR,XOR,AND,*) (constant) produces (addressing mode)

The +, OR, XOR, AND, and * operators can combine a constant and a frame pointer relative, stack pointer relative, static base register relative, external addressing or register relative mode. The operation is performed to the offset of the addressing mode.

Either operand may be the first operand.

(FP,SP,SB,EXT,register relative mode) (/,MOD,SHL,SHR) (constant) produces (addressing mode)

A constant and frame pointer relative, stack pointer relative, static base register relative, external addressing or register relative mode can be combined with the /, MOD, SHL, or SHR operators.

The operation is performed to the offset of the addressing mode.

The constant must be the second operand.

(FP,SP,SB,EXT mode) (-) (FP,SP,SB,EXT mode) produces (constant)

A frame pointer relative, stack pointer relative, static base register relative, or external (using the same link table index) addressing mode may be subtracted from an addressing mode using the same addressing mode.

The operation is performed between the offsets of the addressing modes to produce a constant.

(memory relative mode) (+,OR,XOR,AND,*,/,MOD,SHL,SHR) (constant) produces (same addressing mode)

The operation is performed between the outer offset and constant.

(memory relative mode) (-) (memory relative mode) produces (constant)

The dedicated register and the inner offset must be the same in both memory relative references. The outer offsets are subtracted.

6.2. Current Location Counter

An asterisk "*" or the reserved token PC is interpreted as the value of the location counter for the current instruction.

6.3. Operators

An **operator** specifies an operation to be performed upon one or more operands. There are three levels of precedence among operators. The higher an operator's level of precedence, the more tightly it binds.

(1) highest level (unary operators)

+	Identity.
-	Negation.
com	One's complement of operand.
not	Complement least significant bit.
ext	The constant operand is the index into the link table.
@	The constant expr becomes the absolute memory address specifying absolute addressing mode.

(2) middle level

*	Multiply the first operand by the second.
/	Divide the first operand by the second. Follows the details of the NS16000 DIVD instruction for integers.
mod	Modulus of the first operand with base of the second operand. Follows the detail of the NS16000 MODD instruction for integers.
and	Bitwise logical AND of the first operand with the second.
shl	Shift left the first operand by second operand positions with zero extension.
shr	Shift right the first operand by second operand positions with zero extension.

(3) lowest level

+	Add first operand to second operand.
-	Subtract second operand from first operand.
or	Bitwise logical OR of first operand with second operand.
xor	Bitwise logical XOR of first operand with second operand.
:b|:w|:d	Set length of expression to specified length: byte, word or double-word. Produces an error if expression is too long.

6.4. Displacements and Expression Lengths

Most assembly statements which take operands allow the operand to be an expression. Depending upon the particular statement being constructed, forward references of symbols utilized within the expression are acceptable. The assembler cannot determine the length of an operand constructed from expressions with forward references until the forward reference is defined. Instead the assembler attempts to determine how many bytes to allocate for each displacement it assembles. It uses the following rules:

(1) If the length of the expression is defined, then that length is used.

(2) If the expression is composed of one undefined symbol plus or minus a constant and the length is undefined, then one byte is allocated, an entry is made in the span dependent instruction link-list, and the actual size required to hold the displacement is determined at the completion of the initial pass.

(3) If the expression can be evaluated and the length is not defined, then the minimum number of bytes needed to store the displacement is used.

7. Assembly Language Statements

A **statement** consists of zero to four **fields**. The fields are label, command, operands, and comment.
Format:

[*LABEL***:**] [*COMMAND* [*OPERANDS*]] [**;**COMMENT]

The occurrence of a field may be required or forbidden within a particular assembly language construct, or it may be left as an option to the programmer.

7.1. The Label Field

A **label** is used to assign a value to a symbol.
Format:

LABEL: | *LABEL*:: | *LABEL*:-

A single colon following the label name indicates a local symbol definition, a double colon indicates a global symbol, and a colon minus indicates that the symbol is both global (exported) and external (imported) at the same time. The colon minus form is useful for globally defined procedures which are often called from within the same module thus necessitating an external call.

7.2. The Command Field

A **command** is used to specify an NS16000 instruction or to control the assembler.
Example:

```
WAIT    ;A command without label or operands.
```

A command to specify an NS16000 instruction is called an **instruction mnemonic** (see Chapter 8). A command to control the assembler is called an **assembler directive** (see Chapter 9). A command may be typed in any combination of upper- and lower-case characters.

7.3. The Operand Field

Operands may be required by a symbol definition or a command.
Example:

```
movd    8(fp),r0    ;A command with operands.
```

An operand may be a constant or addressing mode specification or expression. A command imposes its own restrictions on the number of operands; for example, a symbol definition has a single constant operand.

7.3.1. Constant Operands

A **constant operand** is a constant or an expression that evaluates to a constant value.

7.3.2. Addressing Mode Operands

An **addressing mode operand** is a NS16000 addressing mode or an expression that evaluates to an addressing mode.

7.4. The Comment Field

In any assembly language statement a **comment** field is always optional. A semicolon begins a comment unless the semicolon is within an ASCII constant; the comment continues until the end of the line.

8. Instruction Mnemonics

This section lists the NS16000 instruction set, describes the assembler-defined instruction LXPD and discusses restrictions on operands.

8.1. NS16000 Instruction Mnemonics

The following legend is used to describe the options available for each instruction described in this section:

i	B,W,D (byte,word,double-word)
f	F,L (floating,long)
op,op1,op2	general addressing mode
quick	sign extended integer (4 bits)
disp	a displacement
cons4	unsigned constant (4 bits)
cons3	unsigned constant (3 bits)
cons5	unsigned constant (5 bits)
reg	general register (3 bits)
reglist	a list of general purpose registers enclosed in brackets "[]"
cfglist	a list of configuration register bits enclosed in brackets "[]"
procreg	processor register: SP, SB, MOD, INT, PSR, FP, SP, UPSR
mmureg	memory management register: BPR0, BPR1, BPR2, BPR3, PF0,PF1, PF2, PF3, SCA, SCB, BC, PTB0, PTB1, MSR, EIA
cond	condition code test: EQ, NE, LT, LE, GT, GE, LO, HI, LS, HS, FS, FC, CS, CC
label	a PC relative addressing mode
xproc	external procedure name

INTEGER INSTRUCTIONS

```
MOVi    op1,op2    ;move
ADDi    op1,op2    ;add
ADDCi   op1,op2    ;add with carry
SUBi    op1,op2    ;subtract
SUBCi   op1,op2    ;subtract with carry
CMPi    op1,op2    ;compare
NEGi    op1,op2    ;negate
ABSi    op1,op2    ;absolute value
MULi    op1,op2    ;multiply
DIVi    op1,op2    ;divide
MODi    op1,op2    ;modulus
QUOi    op1,op2    ;divide w/zero trunc.
REMi    op1,op2    ;remainder w/zero tr.
ANDi    op1,op2    ;logical AND'ing
ORi     op1,op2    ;logical OR'ing
BICi    op1,op2    ;bit clear
XORi    op1,op2    ;exclusive OR'ing
COMi    op1,op2    ;complement
ASHi    op1,op2    ;arithmetic shift
LSHi    op1,op2    ;logical shift
ROTi    op1,op2    ;rotate
MOVXii  op1,op2    ;sign extend op1 to op2
MOVZii  op1,op2    ;zero extend op1 to op2
```

QUICK INTEGER INSTRUCTIONS

```
MOVQi   quick,op   ;move quick integer
ADDQi   quick,op   ;add quick integer
CMPQi   quick,op   ;compare quick integer
```

EXTENDED INTEGER INSTRUCTIONS

```
MEIi    op1,op2    ;multiply extended integer
DEIi    op1,op2    ;divided extended integer
```

BOOLEAN INSTRUCTIONS

```
NOTi    op1,op2    ;not
Scondi  quick,op   ;conditional set
```

BIT INSTRUCTIONS

```
TBITi     op1,op2        ;test bit
SBITi     op1,op2        ;set bit
CBITi     op1,op2        ;clear bit
SBITIi    op1,op2        ;set bit interlocked
CBITIi    op1,op2        ;clear bit interlocked
IBITi     op1,op2        ;invert bit
CVTP      reg,op1,op2    ;convert bit pointer
FFSi      op1,op2        ;find first set bit
```

FIELD INSTRUCTIONS

```
EXTi      reg,op1,op2,disp          ;extract field
INSi      reg,op1,op2,disp          ;insert field
EXTSi     op1,op2,cons3,cons5       ;extract field short
INSSi     op1,op2,cons3,cons5       ;insert field short
```

STRING INSTRUCTIONS

```
MOVSi     [B[,]][U|W]     ;move string
MOVST     [B[,]][U|W]     ;move string with translate
CMPSi     [B[,]][U|W]     ;compare string
CMPST     [B[,]][U|W]     ;compare string with translate
SKPSi     [B[,]][U|W]     ;skip string
SKPST     [B[,]][U|W]     ;skip string with translate
```

PACKED DECIMAL INSTRUCTIONS

```
ADDPi     op1,op2     ;add packed decimal
SUBPi     op1,op2     ;subtract packed decimal
```

ARRAY INSTRUCTIONS

```
INDEXi     op1,op2         ;calculate array index
CHECKi     reg,op1,op2     ;check array index
```

BLOCK INSTRUCTIONS

```
MOVMi     op1,op2,cons4     ;move multiple
CMPMi     op1,op2,cons4     ;compare multiple
```

PROGRAM CONTROL INSTRUCTIONS

```
JUMP    op              ;jump
BR      label           ;unconditional branch
Bcond   label           ;conditional branch
CASEi   op              ;computed branch
ACBi    quick,op,label  ;add, compare, and branch
JSR     op              ;jump to subroutine
BSR     label           ;branch to subroutine
RET     disp            ;return from subroutine
CXP     xproc           ;call external procedure
CXPD    op              ;CXP w/descriptor
RXP     disp            ;return from external proc.
RETI                    ;return from interrupt
RETT    disp            ;return from trap
```

PROCESSOR SERVICE INSTRUCTIONS

```
SAVE        reglist         ;save processor registers
RESTORE     reglist         ;restore processor registers
ENTER       reglist,disp    ;enter procedure
EXIT        reglist         ;exit procedure
LPRi        procreg,op      ;load processor register
SPRi        procreg,op      ;store processor register
ADJSPi      op              ;adjust stack pointer
BISPSRi     op              ;bit set in PSR
BICPSRi     op              ;bit clear in PSR
SETCFG      cfglist         ;set configuration
```

MISCELLANEOUS INSTRUCTIONS

```
NOP                 ;no operation
WAIT                ;wait
ADDR    op1,op2     ;calculate address
ADDRB   op1,op2     ;calculate address, truncate to byte
ADDRW   op1,op2     ;calculate address, truncate to word
LXPD    xproc,op    ;load external procedure descriptor
SVC                 ;supervisor call
FLAG                ;flag trap
BPT                 ;breakpoint trap
DIA                 ;diagnostic
```

FLOATING-POINT UNIT INSTRUCTIONS

```
MOVf      op1,op2    ;move
MOVLF     op1,op2    ;move long to floating
MOVFL     op1,op2    ;move floating to long
MOVif     op1,op2    ;float
CMPf      op1,op2    ;compare
ADDf      op1,op2    ;add
SUBf      op1,op2    ;subtract
MULf      op1,op2    ;multiply
DIVf      op1,op2    ;divide
NEGf      op1,op2    ;negate
ABSf      op1,op2    ;absolute value
ROUNDfi   op1,op2    ;round
TRUNCfi   op1,op2    ;truncate
LFSR      op         ;load FSR
SFSR      op         ;store FSR
```

MEMORY-MANAGEMENT UNIT INSTRUCTIONS

```
LMR       mmureg,op  ;load MMU register
SMR       mmureg,op  ;store MMU register
RDVAL     op         ;read address validate
WRVAL     op         ;write address validate
MOVSUi    op1,op2    ;move supervisor to user
MOVUSi    op1,op2    ;move user to supervisor
```

8.2. LXPD Instruction

LXPD *xproc,index*

LXPD is an instruction defined by the assembler. This instruction is mainly used by the compilers to look at the external procedure descriptor. The instruction generated in the object file is the *addr* instruction. However, the use of *lxpd* imposes certain restrictions. It is legal only if the *xproc* uses external addressing mode with zero as offset. *Index* is the link table index. The operation is:

```
dest := MEMORY[ MEMORY[ MOD+4 ] + index*4 ]
```

8.3. Notes and Restrictions

Certain restrictions apply to the operands of some instructions.

- An operand that is the destination of an instruction may not be immediate.
- The size of immediate operands must be within the limits set by the [B,W,D] choice, (or, as in floating-point instructions, the limits set by the [F,L] choice).

  Note: If the first operand of the *ADDPi* or *SUBPi* is a constant the processor expects BCD encoding. The assembler only generates two's complement encoding. However a valid BCD number preceeded by h' will be correctly encoded, since both hexadecimal and BCD use the same encodings within the BCD range.

The following examples should make these points clear:

```
ADDD    R0,3        ;illegal.  destination is immediate
MOVB    345,R0      ;illegal.  345 cannot fit in a byte
MOVB    145,R0      ;legal.  145 can fit in a byte
MOVF    1.1e56,F0   ;illegal.  1.1e56 is too big for float
MOVL    1.1e56,F0   ;legal.   it is legal for long format
ADDPB   15,R0       ;illegal BCD value written as h'0f
ADDPB   35,R0       ;legal BCD value written as h'23
                    ;adds decimal 23 to R0
ADDPB   h'23,R0     ;legal BCD value generates the same instruction
                    ;as the previous example
ADDPB   h'3A,R0     ;illegal BCD value
```

9. Assembler Directives

There are three classes of assembler directives for controlling the production of the object file:

- Addressing mode directives
- Procedure interface directives
- Storage directives

A fourth class, the program listing directives, controls the production of program listings.

9.1. Addressing Mode Directives

The **addressing mode directives** control the location into which instructions and data are loaded. The addressing mode directives set the location counter, or current address, and the addressing mode used to access symbols that are defined while the directive is in effect. The location counter is assigned an address that is unrelocatable (absolute) or relocatable (relative to the frame pointer, stack pointer, static base register, program segment, or a link table entry). Once an addressing mode directive has set the addressing mode and relocatability of the current address, all instructions and data are entered relative to the same scope until changed by another addressing mode directive.

The .endseg directive ends the scope of the current addressing mode directive. If a new addressing mode directive is given instead of the .endseg, the new directive overrides the effect of the first, without ending its scope. Instead the previous directive is placed on a stack. If this new directive is ended by a .endseg directive, the stack is popped and the previous scope is restored. The stack has a maximum depth of sixteen levels. This maximum includes any implied by the procedure interface directives (see Section 9.2).

Instructions may only appear in program segment.

With an absolute, static base register relative, program counter relative, or link table entry relative address, any labels are assigned to the current address and the current address is then incremented by the size of the allocated instruction or data.

With a frame pointer relative or stack pointer relative address, the address is decremented by the size of the data and any labels are then assigned to the current address. In the case of the stack pointer relative addresses, actual address calculatation is done at the end of the segment so that the last address assigned will be the first available address on the stack.

No label may appear on a .LOC, .DSECT, .STATIC, .PROGRAM, .MODULE, or .ENDSEG directive.

The addressing mode directives are as follows:

9.1.1. .DSECT

.DSECT

The .DSECT directive sets the current address to immediate zero. The storage class is absolute. No instructions or data are generated (it is used for defining symbols). The defined symbols are most useful as offsets in register relative addressing modes.

9.1.2. .ENDSEG

.ENDSEG

The .ENDSEG directive restores the current address to its address and scope before the most recent .LOC, .DSECT, .PROGRAM, or .STATIC directive. A 16 level stack of nested scopes is maintained at all times. The .PROC directive also uses this stack.

9.1.3. .EXPORT

.EXPORT *name,...*

The .EXPORT directive makes all of the listed names into global symbols. These symbols are made available to other modules through the software module mechanism. That is the symbol may be accessed from another module through that module's link table (see *.IMPORT*) via the external adddressing mode.

9.1.4. .EXPORTP

.EXPORTP *name,...*

The .EXPORTP directive makes all of the listed names into global symbols and marks them as being procedure entry points. It is an error for a name that has been exported as a procedure to be imported as data.

9.1.5. .IMPORT

.IMPORT *name,...*

The .IMPORT directive creates a link table entry that will contain a pointer to data for each symbol name. It is an error for these names to be defined within the current module. The linker will make the link table entries point to the appropriate data symbols in other modules. Link table entries are allocated in the order in which the .IMPORT directives appear. The assembler gives the value EXT(i) + 0 to the symbol, where i is the index of the link table entry just created.

9.1.6. .IMPORTP

.IMPORTP *name,...*

The .IMPORTP directive creates a link table entry that will contain an external procedure descriptor for each symbol name. If this name is also defined within the module, then the assembler will make the link table entry point to the local symbol regardless of whether the symbol has been exported or not. This name should be a procedure entry that is to be called with CXP or CXPD. It is an error for one module to export a data symbol that another module imports from outside its module as a procedure.

9.1.7. .LOC

.LOC *address*

The .LOC directive sets the current address, and addressing mode. The address may be immediate, absolute, or relative to the frame pointer, stack pointer, static base register, program counter, or a link table entry (external data area). Instructions may only be used after a .LOC if the directive sets the mode to PC-relative addressing. Initialized data may only be used after a .LOC directive that sets the mode to PC-relative or SB-relative addressing.

9.1.8. .PROGRAM

.PROGRAM

The .PROGRAM directive sets the current address to one byte past the last allocated address in the program storage class. Any symbols defined while the directive is in effect will have PC-relative addresses. Instructions and data will be placed in the text section of the object file.

9.1.9. .STATIC

.STATIC

The .STATIC directive sets the current address to one byte past the last allocated address in the static base storage class. Any symbols defined while the directive is in effect will have SB-relative addresses. Data will be placed in the initialized data section of the object file.

9.2. Procedure Interface Directives

The **procedure interface directives** support the software module mechanism described in *NS16000 Programmer's Reference Manual*. They allow assembly language routines to have a format similar to those used in high-level languages, i.e., designated structures for declaring input parameters, return values and local variables separate from the routine's code. The following example illustrates the organization of a procedure which uses the procedure interface directives.

Example:

```
myproc:   .PROC
param1:   .blkb
param2:   .blkb
          .RETURNS
ret:      .blkb
          .VAR       [r1,r2]
local1:   .blkb
local2:   .blkb
          .BEGIN
            ...               ;code for prodedure myproc
          .ENDPROC
```

Myproc is a procedure with parameters *param*1 and *param*2, return value *ret*, and local variables *local*1 and *local*2. Registers R1 and R2 are saved and restored.

If *.PROC* is used *.VAR*, *.BEGIN*, and *.ENDPROC* are required, *.RETURNS* is optional. No label may appear on a .RETURNS, .VAR, or .BEGIN directive.

9.2.1. .BEGIN

.BEGIN

The .BEGIN directive sets the current address to one greater than the last program counter relative address allocated; that is, .BEGIN implies the equivalent of .PROGRAM directive. Subsequent instructions and data will be in the text section of the object file; data will be addressed relative to the PC-register. The name of the procedure is assigned the address of the first byte of the procedure. .BEGIN generates an enter instruction using the register list specified in the .VAR directive and a displacement the size of the .VAR scope, i.e., the size of the local variable storage.

9.2.2. .ENDPROC

.ENDPROC [*disp*]

The .ENDPROC directive generates an EXIT instruction, which restores the registers specified in the .VAR statement. The .ENDPROC statement also generates a RET or RXP instruction depending on whether the label on the corresponding .PROC directive was local or global. Usually the directive is used without an operand, in which case the operand of the RET or RXP instruction is the size of the corresponding .VAR scope. If an operand is used with the directive, the assembler uses that operand as the operand for the RET or RXP instruction.

9.2.3. .MODULE

.MODULE *name*

The .MODULE directive starts a new software module. Only one module directive is allowed in a source file.

9.2.4. .PROC

.PROC

The .PROC directive declares a procedure, function, or subroutine. The name of the procedure is supplied in the label, which may be global or local (local is illustrated in the example). A local procedure is called with a JSR or BSR instruction. A global procedure is called with a CXP or CXPD instruction. This directive sets the storage class to local relocation, the following parameters, if any, will be at positive offsets from the frame pointer register.

Storage allocated after the .PROC directive is for parameters to the procedure. The frame pointer offsets are assigned so that the last storage allocated is at the lowest available offset. The lowest available offset is eight for local procedures (this reserves space for saving the program counter) and twelve for global procedures (this reserves space for saving the program counter and the MOD register).

9.2.5. .RETURNS

.RETURNS

The .RETURNS directive sets the current address back to the start of the parameter block. The space allocated in this section overlaps the space in the parameter block, which allows the return value location to be accessed by a new name.

9.2.6. .VAR

.VAR [**[***Rn,...***]**]

The .VAR directive includes an optional register list, which is used in an ENTER instruction. The register list is enclosed in brackets. The .VAR directive sets the current address to the address of the byte preceding the saved register block. Any local variables declared between the .VAR and the .BEGIN will be at negative offsets from the frame pointer. The stack adjustment in the ENTER instruction is the number of bytes specified in the local section.

9.3. Storage Directives

The **storage directives** allocate and initialize areas of memory.

9.3.1. .ALIGN

.ALIGN *base,distance*

The .ALIGN directive inserts uninitialized bytes into the module file until the current address modulo *base* equals *distance*. The distance is optional, with a default value of zero. Both operands must be constants. No label may appear on a .ALIGN directive.

9.3.2. .BLKB, .BLKW, .BLKD, .BLKF, .BLKL

```
.BLKB   count   ;element size 1 byte
.BLKW   count   ;element size 2 bytes
.BLKD   count   ;element size 4 bytes
.BLKF   count   ;element size 4 bytes
.BLKL   count   ;element size 8 bytes
```

The .BLKB, .BLKW, .BLKD, .BLKF, and .BLKL directives allocate a block of storage whose length, in elements, is given by an integer constant. The count is optional, with a default size of one. The element length is implicit in the directive.

9.3.3. .BYTE, .SBYTE, .WORD, .SWORD, .DOUBLE, .SDOUBLE, .FLOAT, .LONG

```
.BYTE     [[repetitions]value,...]   ;element size 1 byte
.SBYTE    [[repetitions]value,...]   ;element size 1 byte
.WORD     [[repetitions]value,...]   ;element size 2 bytes
.SWORD    [[repetitions]value,...]   ;element size 2 bytes
.DOUBLE   [[repetitions]value,...]   ;element size 4 bytes
.SDOUBLE  [[repetitions]value,...]   ;element size 4 bytes
.FLOAT    [[repetitions]value,...]   ;element size 4 bytes
.LONG     [[repetitions]value,...]   ;element size 8 bytes
```

The .BYTE, .SBYTE, .WORD, .SWORD, .DOUBLE, .SDOUBLE, .FLOAT, and .LONG directives allocate and initialize storage area. The value must be small enough to fit within the storage available. .BYTE may have either a signed or unsigned value between -128 and 255. .SBYTE may only be used for a value between -128 and 127. Similarly .WORD may use either a signed or unsigned value between -32768 and 65535, while .SWORD may only use a value between -32768 and 32767.

The signed versions of these directives allow the assembler to produce an error message if a generated signed value is too large for storage. .SDOUBLE is the same as .DOUBLE.

Hex, octal, and binary constants are treated by the assembler as 32-bit two's-complement quantities; thus x'ffffffff is -1, while x'ff is 255.

The number of repetitions must be a constant. The value may be an expression that evaluates to a constant. Both operands are optional, with a default for the number of repetitions of one. If no arguments are specified, a block of storage of the size specified by the directive is allocated with the current address. If one or more operands are specified, then each value is stored into one element of the length implicit in the directive.

The .FLOAT and .LONG directives require floating constants.

If a string constant is used for a value, then enough elements are allocated for the value to represent the entire string. Strings are stored with the first (leftmost) byte in the lowest address. The string is padded on the right with null characters to fill an integral number of elements.

9.3.4. .COMM

.COMM *name,expr,...*

The .COMM directive requests the listed pairs from other modules. They have the attributes of external (common) data type and a specified size. The specified *expr* must reduce to a constant.

9.3.5. .FIELD

.FIELD [*length*]*value,...*

The .FIELD directive initializes arbitrary-length bit fields. A field is created for each element. The length specifies the field size in bits, and the value specifies its contents. The total amount of allocated storage is rounded up to the nearest byte at the end of the statement. Both operands must be constants.

9.4. Program Listing Directives

The **program listing directives** control the production of program listings.

9.4.1. .EJECT

.EJECT

The .EJECT directive causes a page eject. The assembler statement with the directive is not printed in the program listing.

9.4.2. .LIST and .NOLIST

.NOLIST
...
.LIST

The .NOLIST directive turns off program listing until the next .LIST directive. Error messages are printed even if the .NOLIST directive is in effect.

9.4.3. .SUBTITLE

.SUBTITLE *string*

The .SUBTITLE directive causes a subtitle to be printed as the second line of every page. The string may be up to eighty characters long. The subtitle may be redefined an arbitrary number of times.

9.4.4. .TITLE

.TITLE *string*

The .TITLE directive causes its string argument to be printed as the top line of every page. The string may be up to eighty characters long. The title may be redefined an arbitrary number of times.

9.4.5. .WIDTH

.WIDTH *columns*

The .WIDTH directive sets the line length of the output device for the program listing. The operand must be an integer constant between 80 and 132.

10. Programming Examples

The following sections provide sample assembly language programs which illustrate assembly language programming techniques.

10.1. Factorial Numbers

```
;A parameter passed in register 0 is returned as the corres-
;ponding factorial number in register 0.
;


        .PROGRAM
fac:    .DOUBLE   0
        .DOUBLE   1
        .DOUBLE   2
        .DOUBLE   6
        .DOUBLE   24
        .DOUBLE   120
        .DOUBLE   720
        .DOUBLE   5040
        .DOUBLE   40320
        .DOUBLE   362880
        .DOUBLE   3628800
        .DOUBLE   39916800
        .DOUBLE   479001600
num:    CMPD      12,R0          ;is parameter in range?
        BHI       error          ;if not, return an error
        MOVD      fac[R0:D],R0   ;otherwise, index into
                                 ;the array for result
        RET       0
error:  BISPSRB   b'00100000     ;set the error flag
        RET       0
```

This procedure returns any of the factorial numbers which can be represented with a double-word integer. The factorial of number *n* is the product 1 X 2 X 3 X ... X *n*. If the procedure is passed a parameter whose factorial cannot be represented as a double-word integer, it returns the integer unchanged and sets the F code of the PSR.

10.2. Square Root Calculation

```
;The closest integer less than, or equal to, the  square  root
;of an integer passed on the stack is returned on the stack.
;


        .PROGRAM
sqrt:   .PROC
        .VAR       [R0,R1,R2]
        .BEGIN
        MOVQD      1,R0         ;start guessing at one
        MOVD       8(FP),R1     ;get the parameter
        CMPQD      0,R1         ;is it a good parameter?
        BLE        error        ;if not, return an error
loop:   MOVD       R1,R2        ;otherwise, make a  copy
        DIVD       R0,R2        ;divide the copy by the guess
        CMPD       R0,R2        ;is the answer ready?
        ADDD       R0,R2        ;sum the result with the guess
        ASHD       -1,R2        ;take their average
        MOVD       R2,R0        ;make that the new guess
        BHI        loop         ;if not ready, continue
        MOVD       R0,8(FP)     ;return the answer
        BR         exit         ;and exit
error:  BISPSRB    b'00100000   ;set the error flag
exit:   .ENDPROC
```

This procedure calculates the square root of a positive integer. It uses a successive approximation algorithm. If it is invoked on a nonpositive integer, the integer is returned unchanged and the F code of the PSR is set.

10.3. Ackerman's Function

```
;procedure ack(a,b)
;       if a = 0 then
;               ack = b + 1
;       else if b = 0 then
;               ack = ack(a - 1,1)
;       else ack = ack(a - 1,ack(a,b - 1))
;
;The parameters 'a' and 'b' are passed on the stack,with 'b'
;pushed first. The result is returned in register 0.
;


            .PROGRAM
ack:        CMPQD     0,4(SP)     ;if a = 0 then
            BEQ       a_ is_ 0    ;  ack = b + 1
            CMPQD     0,8(SP)     ;else if b = 0 then
            BEQ       b_ is_ 0    ;  ack = ack(a - 1,1)
                                  ;else ack =
            MOVD      8(SP),TOS   ;  push b
            ADDQD     -1,TOS      ;  b = b - 1
            MOVD      8(SP),TOS   ;  push a
            BSR       ack         ;  ack(a,b - 1)
            MOVD      R0,TOS      ;  push ack(a,b - 1)
            MOVD      8(SP),TOS   ;  push a
            ADDQD     -1,TOS      ;  a = a - 1
            BSR       ack         ;ack(a - 1,ack(a,b - 1))
            RET       8
            .ALIGN    4
a_ is_ 0:   MOVQD     1,R0        ;case when a = 0
            ADDD      8(SP),R0    ;ack = b + 1
            RET       8
            .ALIGN    4
b_ is_ 0:   MOVQD     1,TOS       ;case when b = 0
                                  ;  push b
            MOVD      8(SP),TOS   ;  push a
            ADDQD     -1,TOS      ;  a = a - 1
            BSR       ack         ;ack = ack(a - 1,1)
            RET       8
```

This procedure implements Ackerman's function. It is a well-known example of a recursive procedure which terminates for all positive integer values of its two parameters.

10.4. String Sorting

```
;procedure string_ sort(array,e_ cnt)
;set flag
;while flag set
;       clear flag
;       for i = 0 to e_ cnt - 1
;               if string(array[i]) > string(array[i+1])
;                       temp = array[i]
;                       array[i] = array[i+1]
;                       array[i+1] = temp
;                       set flag
;
;The maximum length of a string is 'max_ length', an imported
;variable.  The array address and element count, 'array' and
;'e_ cnt',  are passed on the stack,  with  'e_ cnt'  on  top.
;


           .MODULE    example
           .IMPORT    max_ length
sort::     .PROC
array:     .BLKD
e_ cnt:    .BLKD
           .VAR       [R0,R1,R2,R3,R4,R7]
           .BEGIN
           BISPSRB    b'00100000          ;set flag
           MOVD       e_ cnt,R3           ;  get e_ cnt
           ADDQD      -1,R3               ;  e_ cnt = e_ cnt - 1
loop2:     BFC        p_ exit             ;while flag set
           BISPSRB    b'00100000          ;clear flag
           MOVQD      0,R7                ;for i = 0 ...
loop1:     CMPD       R3,R7               ;       ... to e_ cnt - 1
           BEQ        loop2
           MOVD       max_ length,R0      ;  set up cmpsbu limit
           MOVD       array[R7:D],R1      ;  set up array[i]
           ADDQD      1,R7                ;  i = i + 1
           MOVD       array[R7:D],R2      ;  set up array[i+1]
           MOVQD      0,R4                ;  set up  end of string
           CMPSBU                         ;if string(array[i]) ...
           BHS        loop1               ;...> string(array[i+1])
           ADDR       array[R7:D],R0      ;  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
           BISPSRB    b'00100000          ;set flag
           BR         loop1
p_ exit:   .ENDPROC
```

This procedure implements a bubble sorting algorithm for an array of pointers to strings. A bubble sorting algorithm performs successive exchanges of unordered neighbors.

10.5. Bit Scanning

```
;The addressing  for first  and  last bits  of the field are
;passed  on the stack.  The addressing for the first set bit
;is returned on the stack. The search is started by searching
;until the first double-word boundary.  After that the search
;is continued by testing for nonzero double-words.  Individual
;bits are tested only after a non-zero double-word is found.
;If no set bits are found, the F code of the PSR is set.
;


scan:      .PROC
addr_ f:   .BLKD
disp_ f:   .BLKD
addr_ l:   .BLKD
disp_ l:   .BLKD
           .RETURNS
addr_ s:   .BLKD
disp_ s:   .BLKD
           .VAR       [R0,R1]
           .BEGIN
           FFSD       addr_ f,disp_ f   ;search to next double-
                                        ;word boundary
           BFS        p_ count          ;is the bit there?
           BR         exit              ;if so, exit
p_ cont:   MOVD       addr_ f,R0        ;get end address
           MOVD       addr_ l,R1        ;get beginning address
           ADDQD      1,R1              ;move it to next whole
                                        ;double-word address
           SUBD       R1,R0             ;calculate count limit
           MOVQD      0,R4              ;zero is while character
           SKPSDW                       ;search to first nonzero
                                        ;double-word
           FFSD       R1,R4             ;search for a set  bit
                                        ;within the double-word
           CMPD       addr_ l,R1        ;is its address same as
                                        ;for last bit?
           BLO        p_ exit           ;if not, exit
           CMPD       disp_ l,R4        ;is its displacement
                                        ;beyond last bit?
           BLS        p_ exit           ;if not, exit
           BISPSRB    b'00100000        ;if the search stopped
                                        ;beyond the last bit of
                                        ;the field, set the flag
           BR         exit              ;and, exit
p_ exit:   MOVD       R1,addr_ s        ;return address of the
                                        ;set bit
           MOVD       R4,disp_ r        ;return displacement of
                                        ;the set bit
exit:      .ENDPROC
```

This procedure implements a scanning algorithm for the first set bit among an arbitrary length field of contiguous bits in memory.

10.6. String Packing and Unpacking

```
;The addresses of a packed and an unpacked string are passed
;on the stack, with the address of the packed string on top.
;A zero marks the end of either string.
;


          .MODULE    string_ procs
;
pack::    .PROC
str_ u1:  .BLKD
str_ p1:  .BLKD
          .VAR       [R0,R1]
          .BEGIN
          MOVD       str_ u1,R1         ;put the address of the
                                        ;unpacked string in r1
          MOVD       str_ p1,R0         ;put the address of the
                                        ;packed string in r0
          MOVQD      0,R2               ;initialize the pointer
                                        ;to the packed string
loop1:    INSB       R2,0(R1),0(R0),7   ;pack a character
          ADDQD      1,R1               ;get next unpacked char.
          ADDQD      7,R2               ;get next packed char.
          CMPQB      0,-1(R1)           ;at the end of string?
          BNE        loop1              ;if not, continue
          .ENDPROC
;
unpack::  .PROC
str_ u2:  .BLKD
str_ p2:  .BLKD
          .VAR       [R0,R1]
          .BEGIN
          MOVD       str_ u2,R1         ;put the address of the
                                        ;unpacked string in r1
          MOVD       str_ p2,R0         ;put the address of the
                                        ;packed string in r0
          MOVQD      0,R2               ;initialize the pointer
                                        ;to the packed string
loop2:    EXTB       R2,0(R0),0(R1),7   ;unpack a character
          ADDQD      1,R1               ;get next unpacked char.
          ADDQD      7,R2               ;get next packed char.
          CMPQB      0,-1(R1)           ;at the end of string?
          BNE        loop1              ;if not, continue
          .ENDPROC
```

These procedures convert between packed and unpacked ASCII character strings. Because the eighth bit of an ASCII character is not required for the normal character set, it is possible to economize on memory by using seven bit fields to represent ASCII characters.

11. Reserved Tokens

In addition to the mnemonics representing machine instructions and assembler directives there are symbols reserved for register identification and expression operators. For definitions and descriptions of these symbols refer to *The NS16000 Programmer's Reference Manual.*

The *NS16000 Cross-Assembler Reference Manual* may also be useful.

The symbols are valid in either upper- or lower-case, or in combination. Following is a list of the symbols:

Symbol	Token Type
and	operator token
bc	memory register
bpr0	memory register
bpr1	memory register
com	operator token
eia	memory register
ext	operator token
f0	floating register
f1	floating register
f2	floating register
f3	floating register
f4	floating register
f5	floating register
f6	floating register
f7	floating register
fp	program register
intbase	program register
is	program register
mod	operator token
mod	program register
msr	memory register
not	operator token
or	operator token
pc	program register
pf0	memory register
pf1	memory register
psr	program register
ptb0	memory register
ptb1	memory register
r0	general register
r1	general register
r2	general register
r3	general register
r4	general register
r5	general register
r6	general register
r7	general register
sb	program register
sca	memory register
scb	memory register
shl	operator token
shr	operator token
sp	program register
tos	program register
upsr	program register
us	program register
xor	operator token

12. Applicable Documents

(1) *NS16000 Programmer's Reference Manual.*

(2) Szymanski, T., "Assembling Code for Machines with Span-Dependent Instructions", Communications of ACM, April 1978, Volume 21, Number 4.

Table of Contents