Chapter 3

DPEAC Syntax

3.1 Instruction Types

The following sections describe the basic syntactic elements that make up the DPEAC instruction set. The order of discussion is

- Source lines
- Comments
- Numbers
- Expressions
- IU Register Names
- VU Register Names
- Register Strides
- Register Indirection
- Memory Addresses
- Memory Striding
- Memory Indirection
- Vector Length
- Vector Length
- Immediate Operands
- VU Selectors

3.2 Source Lines

DPEAC is at base an assembly language, so individual statements are fully contained on single source lines. Generally, a source line occupies a single line from the source file — that is, the text between newline chanracters. However, it is possible to construct a single source line from multiple lines of the file by ending all but the last file line with a backslash (\) character. The following example shows three source lines constructed from five source file lines:

```
This is line one
This is \
line \
two
This is line three
```

3.3 Comments

DPEAC provides three methods for including comments in the code. The first method passes the comments directly to **as**; the second and third do not. The indicators used are

source code `!` *comment text*
`/*` *comment* `*/`
`#comment` *comment text*

To pass comments to **as**, place the comment text outside of, or following, legal syntax in DPEAC instructions. DPEAC passes such text, including comment text at the end of DPEAC instruction lines, unchanged to the output. Thus, all comments of the form "`!` *comment text* ..." are passed to **as**, which treats them as comments.

Comments of the form "`/*` *comment* `*/`" allow the comment text to span multiple lines. Such comments are not passed to **as**. Instead, the **dpas** preprocessors (internal or **cpp**) replace these comments with space characters.

Finally, the internal lexical preprocessor includes a "`#`*comment*" directive, which is not passed to the **dpas** output. **Note:** The **cpp** preprocessor does not support this directive.

3.4 Numbers

Numbers are 64-bit constants. Parsing of constants follows the conventions of the C language. Any item starting with a digit is assumed to be a constant. Any constant starting with a non-zero digit, and consisting only of digits is assumed to be a decimal (radix 10) number, e.g. ”1234”, ”1985”. Any number starting with zero, and consisting entirely of digits, is assumed to be an octal (radix 8) number. Alternate radices can be specified using radix prefixes, as follows:

`0x1AF`...,	`0X1AF`...	Hexidecimal constants
`0b101`...,	`0B101`...	Binary Constants
`0o123`...,	`0O123`...	Octal constants
`0n987`...,	`0N987`...	Decimal constants

Floating point constants in various forms may also be represented. Floating point numbers are designated as follows:

0*t*[+,-]*nnn*[.[*nnn*]][{E,e}[+,-]*nnn*]

where *t* indicates the type, one of:

`f` (or `F`)	IEEE Single Precision (32 bit)
`r` (or `R`)	IEEE Single Precision (32 bit)
`d` (or `D`)	IEEE Double Precision (64 bit)

For example:

```
0f0.0           0f-1.0        0d6.02E23
0F.31415E+1     0D.000001     0f1
```

Constants prefixed by “`0f`” are IEEE single precision constants having the form:

s	*exponent*	*fraction*
1	8	23

where *s* is the sign (1 is negative), *exponent* is an excess-127 binary exponent, and *fraction* is the 24-bit binary characteristic. (The 24th bit of the fraction is ”hidden” and is assumed to be one, i.e. the constant is normalized.)

DPEAC also provides the alternate “`0r`” single-precision syntax for compatibility with the SPARC assembler. This format does not have special syntax for double precision constants.

Constants prefixed by "`0d`" are IEEE double precision floating point constants, having the form:

s	*exponent*	*fraction*
1	11	52

where fields are as before, except that the *exponent* is excess-1023 and 32 additional bits of significance are added to the *fraction* part.

Constants composed of ASCII characters may also be specified by enclosing them in apostrophes, as follows:

`'A'`	is equivalent to	`0x41`, `0101`, `65` and `0b1000001`
`'ABC'`	is equivalent to	`0x414243`

3.5 Expressions

The DPEAC language uses three types of expressions

- *constant-expression*
- *as-expression*
- *general-expression*

Constant-expressions

Most numerical values required in DPEAC instructions must be given as expressions that can be evaluated at assembly time (as opposed to link time). Such an expression is called a *constant-expression*.

A *constant-expression* contains only assembly-time constants as operands. These may contain symbols only if those symbols can be translated into constants by the preprocessor.

The expression syntax of a *constant-expression* is the same as **`as`** syntax, plus some useful extensions. In particular, the syntax allows use of

- The comparison and logical operators **`<=`** and **`&&`**, which work as in C.
- Floating point constants, such as the single precision **`0r1.0`** (and its equivalent, **`0f1.0`**), and the double precision **`0d1.0`**.

The computation of a *constant-expression* is always performed in 64-bit integer arithmetic. Multiply and divide are done as signed integers; all other computation is done as unsigned integers. Inclusion of floating point numbers, therefore, is equivalent to using their bit-pattern counterparts as integer operands; floating point calculations are not done.

The following table lists the supported operators in a *constant-expression*. Operators are shown from highest to lowest precedence. (However, parentheses may be used to alter the order of evaluation.)

Unary Operators:

`+`	Plus (unary, has no effect)
`-`	Negate (2's complement)
`!`	Logical not
`~`	Bitwise 1's Complement (invert)
`%lo`	Extract low 10 bits (and with `0x3FF`)
`%hi`	Extract high 22 bits (shift right 10)

Bitwise Logical Operators: (Evaluated Left To Right)

`&`	Bitwise AND
`|`	Bitwise OR
`^`	Bitwise XOR

Scaling Operators: (Evaluated From Left To Right)

`*`	Signed multiply
`/`	Signed divide
`<<`	Logical Left shift (as in "C")
`>>`	Logical Right shift (as in "C")

Offsetting Operators: (Evaluated From Left To Right)

`+`	Addition
`-`	Subtraction

Comparison Operator: (Yield Value Of 1=True, Or 0=False)

`<`	Less than (unsigned)
`<=`	Less than or equal to (unsigned)
`==`	Equal to
`!=`	Not equal to (<> is also allowed)
`>`	Greater than (unsigned)
`>=`	Greater than or equal to (unsigned)

Logical Operators: (Yield Values 0 Or 1 Only)

`&&`	Logical And
`||`	Logical OR

As-expressions

In a few places in the DPEAC syntax, an *as-expression* is called for or is allowed as an alternative to the use of a *constant-expression*. An *as-expression* is an expression that is passed verbatim to the **`as`** assembler. Therefore, *as-expressions* use the expression syntax described in the SUN4 Assembly Language Reference Manual. Several distinctions between *as-expressions* and *constant-expressions* should be noted. In particular, *as-expressions*

- are evaluated as 32-bit integers.
- can contain symbols, such as link-time externals, that are processed by **`as`**
- cannot contain the comparison and logical operators **`<=`** and **`&&`**

All numeric constant and symbol syntax in *as-expressions* is that of the SUN4 assembly language. Floating point constants are not allowed.

General-expressions:

A *general-expression* is an expression that can be either a *constant-expression* or an *as-expression*. **`dpas`** first attempts to parse a *general-expression* as if it is a *constant-expression*. If that fails (for example, due to unevaluatable symbols in the expression), **`dpas`** assumes it is an *as-expression*, and passes it directly to **`as`**.

3.6 IU Registers

All registers in the language are those of the SPARC IU processor. The **`as`** register syntax is taken from the SPARC architecture (e.g. **`%g1`**, **`%i3`**, etc.). Consult the **`as`** and SPARC manuals for register naming.

In the syntax descriptions throughout this manual, registers of the IU are represented as "*as-register*".

3.7 VU Registers

The 128 registers in the register file of the accelerator chip are known as *VU registers*. These registers can be referenced by name in several ways. (None of the name specifiers are case-sensitive.)

The general names of the 128 VU registers are **`R0`** through **`R127`**. These names reference each of the 128 registers individually. Leading zeros can precede the decimal register number. Thus, **`R07`** is the same as **`R7`**. Double precision numbers are stored in aligned pairs of registers. Thus, to put a double precision number in **`R6`** requires **`R6`** and **`R7`**.

Registers **`R0`** through **`R15`** can also be specified as **`S0`** through **`S15`** for single-precision operations. For double precision operations, the even-numbered registers **`R0`** through **`R30`** can be specified as **`S0`** through **`S30`**.

For the Curious: The **`Sn`** register names are provided because the scalar short instruction formats limit their *Rs2*, *Rls* and *Rd* operands to the range **`R0`** through **`R15`** for single precision operations and the even-numbered registers **`R0`** through **`R30`** for double precision. This results in the standard practice of using these registers for holding scalar data, hence the **`Sn`** abbreviations

The VU registers can also be accessed as 16 vector registers of 8 single precision (or 4 double precision) values each. These vector registers are named **`V0`** through **`V15`**. (Again, leading zeroes can appear in the register number.)

The vector registers are mapped to the register file as follows:

Registers	Vector	Registers	Vector
R0 – R7	**V0**	**R64 – R71**	**V8**
R8 – R15	**V1**	**R72 – R79**	**V9**
R16 – R23	**V2**	**R80 – R87**	**V10**
R24 – R31	**V3**	**R88 – R95**	**V11**
R32 – R39	**V4**	**R96 – R103**	**V12**
R40 – R47	**V5**	**R104 – R111**	**V13**
R48 – R55	**V6**	**R112 – R119**	**V14**
R56 – R63	**V7**	**R120 – R127**	**V15**

The individual elements of vector registers can be referenced using the syntax

Vn[k**]**

indicating the k'th element of **Vn**. Note that the elements of a vector in which double precision numbers are stored are found in **Vn[0]**, **Vn[2]**, **Vn[4]**, and **Vn[6]**. A vector element reference to a register number greater than 127 (for example, **V5[214]**) is not recognized.

Usage note: For completeness, this "vector" indexing is also allowed in the **Rn** and **Sn** register specifiers. For example, specifying **S2[2]** is the same as **S4** (or **R4**). This allows the two halves of a double stored in **Sn** to be referenced, as **Sn[0]** and **Sn[1]**. Similarly, the syntax **Rn[i]** is always equivalent to **Rn+i**.

When longer vectors are used, vector registers are doubled up. For example, to use vectors holding 16 single precision values, you would use **V0**, **V2**, **V4**, etc., through **V14**. Elements of these vectors are numbered **Vn[0]** through **Vn[15]**.

Registers that are not the base of any of the 16 vector registers (that is, registers other than **R0**, **R8**, **R16**, etc.) require a full 7 bits for their specification; moreover, the specification of these unaligned registers may be disallowed in some instruction forms or may force the use of the slower long format instructions.

Usage Note: In the instruction syntax descriptions below, the **Rn** form is always used to refer to arbitrary single registers, while **Vn** is always used to refer to one of the 16 vector base registers. Also, **R0** is special in some instruction decodings, as noted in the individual instruction descriptions.

3.8 Register Strides

During vector instructions, each register is incremented by a stride between each operation. For any register with which striding sytax is not appended, the default is to stride by the operand precision, i.e. by 1 when operating on single precision data, and by 2 when operating on double precision data. This is referred to as a unit stride. Since the result precision of an operation may be different than the destination precision (as in the type conversion operations), the unit stride for the result register may be different than for the operand registers.

The syntax for specifying a stride within an instruction is shown below. In each case, *register* is any valid VU register name, and *stride* and *set–stride* are integer constants or constant-expressions in the range –128 to +128.

Syntax	Effect
register	Use unit stride
register`:`*stride*	Temporarily use specified stride.
register`:=`*stride*	Set **`dp_stride_rs1`** to *stride*, then use it.
register`:`*stride*`=`*set–stride*	Set **`dp_stride_rs1`** to *set–stride*, but use *stride*.
register`:`*mode*	Stride by contents of **`dp_stride_rs1`**
register=*stride*	Just set **`dp_stride_rs1`** to *stride*. (Used in scalar instructions.)

Note: The latter four syntax forms are valid only for setting the first source register (Rs1) stride.

3.9 Register Indirection

One form of DPEAC instruction allows the use of a *Register Indirect* mode for specifying the first instruction operand. As described in the previous section, register operands are normally strided by a fixed stride value. In the Register Indirect mode, however, the vector of registers used as the first operand is constructed by adding a vector of register number offsets to a base register number. The syntax is:

Syntax	Effect
base–register`(`*offset–register*`)`	Indirect addressing, unit stride.
base–register`(`*offset–register*`:`*offset–stride*`)`	” , constant stride.

The vector of offsets is contained in the *offset–register* and at subsequent registers at the specified stride (which defaults to the unit stride of one). The offsets are packed *four to a register*, one in each of the four bytes of the word. Since offsets cannot exceed 127 (7 bits), the eighth bit of each byte must be zero.

The four offsets are accessed in right-to-left order, as shown below:

0	*offset 3*	**0**	*offset 3*	**0**	*offset 3*	**0**	*offset 3*
31	30 … 24	23	22 … 16	15	14 … 8	7	6 … 0

After accessing four offsets, the *offset–stride* or unit stride of one is added to the *offset–register* number to get the register number containing the next four offsets, and so on. For example, condider this syntax:

```
R16(R54:2)
```

And let us assume that the offsets are stored as follows:

```
R54 = hex 03020301
R56 = hex 04000002
```

Given a vector length of 8, the specified operand will be taken from the following sequence of registers (the offset is shown beneath):

```
R17  R19  R18  R19  R18  R16  R16  R20
(1)  (3)  (2)  (3)  (2)  (0)  (0)  (4)
```

Note: As shown in this example, it *is* legal to use offsets more than once.

3.10 Memory Addresses

Virtual memory addresses are specified using the Sun4 Assembler syntax:

Syntax	Memory Address
[*as–register***]**	Contents of *as–register*
[*as–register1* **+** *as–register2***]**	Sum of *as–register1* and *as–register2*
[*as–register* **+** *offset***]**	Contents of *as–register* + *offset*

where *offset* is limited to the range **–4096** to **+4095**.

Terminology Note: In the syntax descriptions below, memory operand specifiers are given as *as–mem–operand*.

Usage Note: It is illegal to reference double precision memory data at odd word address boundaries, i.e. addresses that are not even multiples of 8.

3.11 Memory Striding

Vectors read or written from or to memory may be strided; that is, the elements making up the vector can be located a constant distance apart. The memory stride of a vector is the difference between the addresses of adjacent vector elements in bytes.

The memory stride used by instructions which read or write memory is stored as a settable mode in the **`dp_stride_memory`** control register. All instructions allow the memory stride to be temporarily overridden to a unit stride or a double-unit stride, that is, 4 or 8 bytes for single-precision memory operations, 8 or 16 bytes for double-precision memory operations. Also, certain instruction formats allow the **`dp_stride_memory`** control register to be overridden with any 32-bit value and then optionally copy that value to the **`dp_stride_memory`** control register. The syntax is:

Syntax	Effect
as–mem–operand`:4`	Use unit stride for single precision loads/stores.
as–mem–operand`:8`	Use unit stride for double precision loads/stores (or double unit stride for single precision).
as–mem–operand`:16`	Use double unit stride for double precision loads/stores.
as–mem–operand`:=`*stride*	Set **`dp_stride_memory`** and use it now.
as–mem–operand`:{4,8,16}=`*stride*	Set and override memory stride.
as–mem–operand`=`*stride*	Set **`dp_stride_memory`** only (used only in Scalar instructions).

where *stride* is a *general–expression* (typically an integer constant) indicating the stride in bytes, or an *as-register* containing the desired stride in bytes. The accelerator chip restricts the stride to a 24-bit signed integer (plus or minus 16 Mbyte). In fact, if bits 30 and 31 of the value are not both zero or both one, the instruction will be incorrectly decoded.

For example,

```
[%i3+100]:=4
```

will access a memory location with an address that is 100 plus the contents of %i3, and will implicitly set and use a memory stride of 4.

3.12 Memory Indirection

One form of DPEAC instruction allows the use of a *Memory Indirect* mode for specifying the memory address in a DPEAC memory operation.

As described in the previous section, the memory address is normally a specified bank address strided by a fixed stride value. Using Memory Indirection causes the sequence of memory addresses to be constructed by adding a vector of byte offsets to a base address. The syntax is:

Syntax	Effect
as–mem–operand`(`*offset–register*`[:`*stride*`])`	Memory Indirection

The *as–mem–operand* is a standard SPARC memory address specifier. The vector of offsets is contained in the *offset–register* and at subsequent registers at the specified *stride* (which defaults to the unit stride of one).

The offset registers are single-precision registers, each containing a byte offset. This offset is added to the base address to produce the memory address. The addition is done in two's-complement, so negative offsets will work correctly.

For example, condider this syntax:

```
[%i2](R54:2)
```

Let us assume that the offsets are stored as follows:

```
R54 = hex 00000100
R56 = hex 00000200
R58 = hex 00000008
R60 = hex 00000200
```

and that **`%i2`** contains the value **`10000`** hex.

Given a vector length of 4, the following sequence of memory addresses will be used in the current Memory Operation (the hex offset is shown in parentheses):

```
10100 hex (100)
10200 hex (200)
10008 hex (8)
10200 hex (200)
```

3.13 Vector Length

The vector length used by instructions is a parameter controlled by a register (**dp_vector_length**). Certain instruction formats allow the overriding and optional setting of the vector length. The syntax involves suffixing the length to the arithmetic opcode as follows:

Syntax	Effect
*opcode*******vlen*	Temporarily use vector length *vlen* (a constant).
*opcode****=***vlen*	Use and set mode to vector length *vlen*.
*opcode*******as–register***<**	Temporarily use vector length from *as–register* (bits **19:22** plus one).
*opcode****=***as–register***<**	Use and set vector length from *as–register* (bits **19:22** plus one).
*opcode*******as–register*	Temporarily use vector length from *as–register.*
*opcode****=***as–register*	Use and set vector length taken from *as–register.*
*opcode***=***vlen*	Set to vector length *vlen* only (Used only in Scalar instructions.)

where *vlen* is a *constant-expression* evaluating to a positive integer in the range **1** to **+16**, or an *as-register.*

The vector length can also be given in an *as-register*, for example:

```
*=%i2
```

Two forms are offered. The first, which is faster (two SPARC instructions), requires that the desired vector length be decremented by one and positioned in bits **19** through **22** of the register. This is specified by appending the "**<**" chara ter to the register name.

The second, assumed if the "**<**" is omitted, takes the vector length from the register in the low bits (undecremented). In either case, all other bits in the register must be zero.

If the "***=**" syntax is used, the value in bits **19** through **22** of the register is also copied to the **dp_vector_length** mode register (in its decremented form). This correctly sets the default vector length to that used in the instruction.

Note: When permanently setting the vector length (by including "**=**"), the **dp_vector_length** register is set to the specified vector length minus one.

For example, note how the following lines of code specify vector lengths for their vector adds:

Code	Vector Length Control
dfaddv	Use value in **dp_vector_length**.
*dfaddv****8**	Use length **8** for this instruction only.
*dfaddv****=8**	Use length **8** and set **dp_vector_length** to **7**.
*floadv****=%i3**	Use value in **%i3** as vector length, and set **dp_vector_length** to that value minus one.
*floadv****%i3<**	Use vector length contained in **%i2** (bits **19** through **22** plus one).
*floadv****=%i3<**	Copy value contained in **%i2**, bits **19** through **22**, to **dp_vector_length** and use that value, plus one, as the vector length.

The following example shows the construction of a vector length in a register, followed by its use as a vector length:

```
#include <cmsys/dp.h>   /* define DP_VL__POS (=19) */

! Assume that %i2 contains the desired vector length

sub %i2,1,%g1           ! decrement vector length
sll %g1,DP_VL__POS,%g1  ! Shift vector length by 19
faddv*=%g1<   v1,v2,v3  ! Use prepared vector length
```

3.14 Immediate Operands

One DPEAC instruction format allows its second operand to be a single, scalar, so-called *immediate* operand, rather than a vector of data. The immediate operand is a 32-bit value. Its syntax accepts either an *as-register* containing the immediate value or any *general-expression* evaluating (at assembly or link-time) to the immediate value.

Note: Older syntax required an immediate value expression to be preceded by a dollar sign ("**$**"). This syntax is still supported.

The accelerator chip internally sign extends immediate values used in double integer typed arithmetic. For double unsigned operations, the value is zero-extended.

For double precision constants, only the upper 32 bits are included in the instruction. Thus, only floating point numbers with 0's in the 32 least significant bits of their mantissas are allowed.

For example:

Syntax	Immediate Value
`0r2.81`	Single precision `2.81` in IEEE bit form
`0d3.0`	Upper 32 bits of `3.0` is used
`foo+2`	"`foo+2`" is passed to the assembler
`%i3`	The contents of SPARC register `%i3`

3.15 VU Selectors

In a few places in the syntax, you must specify a VU Selector. This is an integer number that specifies one or more VUs.

Terminology Note: In the syntax definitions, this is indicated as *VU-selector*.

You may specify a VU Selector in any of the following ways:

Syntax	Immediate Value
constant–expression	Use the specified selector value (see table below).
as–register	Use the selector value in SPARC register.
*as–register***<**	Use the selector value in SPARC register (bits `12:15`).
*****	Use all VUs.
******constant–expression*	Use both VUs on accelerator chip N.
$*constant–expression*	Use the specified physical VU number.

There are two methods of specifying the VU Selector in a SPARC register:

- The unshifted version (the default), in which the Selector value is in bits 0 through 3 of the register
- A shifted version (specified by appending "**<**" to the register name), in which the Selector value is in bits 12 through 15 of the register.

Performance Note: The shifted version is one SPARC instruction faster, and its use is encouraged for optimal performance.

The possible VU Selector values, given as a constant or in a specified as–register, are shown in the table below, along with corresponding symbols defined in the file **dp.h**. Also shown is special syntax for values that select more than one VU.

VU selector	Symbol	Selected VUs	Special Syntax
0	**DP_0**	VU 0	
2	**DP_1**	VU 1	
4	**DP_2**	VU 2	
6	**DP_3**	VU 3	
1	**DP_4**	VU 4	
3	**DP_5**	VU 5	
5	**DP_6**	VU 6	
7	**DP_7**	VU 7	
8	**ALL_DPS**	ALL VUs	*****
10	**DPS_0_AND_1**	VUs 0 and 1	***0**
12	**DPS_2_AND_3**	VUs 2 and 3	***1**
11	**DPS_4_AND_5**	VUs 4 and 5	***2**
13	**DPS_6_AND_7**	VUs 6 and 7	***3**

Normally, the VU selectors are not the same as the physical VU numbers used in the physical addressing of the accelerator chip. It is therefore not correct to code a literal "**3**" to indicate VU number 3.

If you wish to provide the physical VU number as the VU selector, you must specify the VU selector as a constant (or *constant-expression*) preceded by a dollar sign ("**$**"); the value is then taken to be the physical VU number, as defined by the accelerator chip specification. The following table gives the physical VU values:

Physical VU	Symbol	Selected VU's
$0	**DP_PHYS_NUM_0**	VU 0
$1	**DP_PHYS_NUM_1**	VU 1
$2	**DP_PHYS_NUM_2**	VU 2
$3	**DP_PHYS_NUM_3**	VU 3
$4	**DP_PHYS_NUM_4**	VU 4
$5	**DP_PHYS_NUM_5**	VU 5
$6	**DP_PHYS_NUM_6**	VU 6
$7	**DP_PHYS_NUM_7**	VU 7
$8	**ALL_PHYS_NUM_DPS**	ALL VUs
$9	**DPS_PHYS_NUM_0_AND_1**	VUs 0 and 1
$11	**DPS_PHYS_NUM_2_AND_3**	VUs 2 and 3
$13	**DPS_PHYS_NUM_4_AND_5**	VUs 4 and 5
$15	**DPS_PHYS_NUM_6_AND_7**	VUs 6 and 7

For example:

```
dpwrt dp_2,{...}     ! write data register of VU 2

dpwrt %i2,{...}      ! write data register of VU
                     ! selected by %i2 bits 0:3

dpwrt *,{...}        ! write data register in all VUs

dpwrt %i4<,{...}     ! write data register of VU
                     ! selected by %i4 bits 12:15

dpwrt DPS_2_AND_3,{...}  ! write data reg of VUs 2 and 3
dpwrt *1,{...}           ! write data reg of VUs 2 and 3
dpwrt $11,{...}          ! write data reg of VUs 2 and 3
```