Chapter 6

Indirect VU Instruction Execution

The **dpas** assembler normally places VU instructions into tables in the text section of memory. To execute the instructions, the interface between the IU and the accelerator chip retrieves the instructions from these tables and writes them into the VU Instruction Regions of memory.

DPEAC also provides the ability to execute an instruction indirectly from a predetermined memory location. Indirect execution requires three steps:

1. Use the **dpasm**[**d**] instruction to assemble a VU instruction and to place the resulting instruction in a memory location, labeled with a global label.

2. At run-time, load a SPARC register with a pointer to the instruction.

3. Use the **dpexec** instruction to execute the instruction at the pointer.

6.1 The dpasm[d] Instruction

The **dpasm**[**d**] instruction allows you to assemble a VU instruction and to place the resulting instruction object in memory at the current assembly location, optionally labeled with a global label. Two versions of this instruction exist: **dpasm**, which generates instructions in the short (S) format, and **dpasmd**, which generates instructions in any of the long formats.

The syntax of **dpasm** is:

dpasm[**d**]   [*label*], *VU Instruction*

Labels

[*label*] is any label of your choice. **dpasm** uses whatever label you specify and generates **.align** and **.global** statements for it. For example,

```
dpasm mylabel, faddv v1,v2,v3; floadv [],V2
```

generates the following SPARC assembly:

```
!! S: VS-S=2 LS-F=3 rLS=2 ALU-F=00 rS1=8 rS2=2 rD=3
!! dpasm mylabel, faddv v1,v2,v3; floadv [],V2
        .align 4
        .global mylabel
mylabel:
        .word 0x99000823
```

If you omit the *label* statement, **dpas** generates neither *label*, **.global**, nor **.align** statements. This allows you to generate tables of instructions. If you do this, you are responsible for making sure the table is word-aligned for short format instuctions (use **.align 4**) or doubleword aligned for long format instructions (use **.align 8**).

Postponing Specification Of Values

Notice, in the example given above, that the memory address in the **floadv** operation was omitted in the instruction source. The memory address is supplied when the instruction is executed using **dpexec**.

Notice similarly, in the next example, how the specification of vector length in a Mode Set instruction format is postponed until execution time:

```
dpasm mylabel, faddv*=0 v1,v2,v3
```

The value 0, not normally a legal vector length, tells **dpas** to generate a Mode Set format instruction, but to leave the vector length field of the instruction zeroed out. See the description of **dpexec**[**d**] for details on the execution of such an instruction.

Where To Place The Instruction Object

The instruction object created by the **dpasm** instruction is placed immediately in the current assembly location. This is *not* executable code! It is not correct to place this in the instruction stream. Rather, you should place this off in some table area.

Usage Note: Cache behavior can be important when placing the instruction object. It can be placed in the text (code) segment near to the code that will actually execute it, so long as it will not be encountered in the normal flow of control. (Normally assembled VU instructions are located this way by **dpas**.)

6.2 The dpexec[d] Instruction

The **dpexec**[**d**] instruction provides the method for executing instructions which have been assembled to memory via **dpasm**[**d**]. The syntax is:

dpexec[**d**] *instr–obj–address*, [*vu–mem–address*],
[*instr–ext–reg*], [{**align**, **noalign**}]

Note: The format of the instruction, as encoded by **dpasm**, is important. If the instruction was assembled using the short format, you must use **dpexec** to execute it. Otherwise, you must use the **dpexecd** instruction.

The instr–obj–address Argument

The **instr–obj–address** argument is a SPARC memory operand specifying the location of the instruction object. In other words, it is a pointer to the instruction.

The vu–mem–address Argument

The **vu–mem–address** argument is the parallel memory address to be used for the instruction if it included a memory operation (recall that it was omitted when the instruction was assembled). This is an *as–mem–operand*, for example **[%i3+4]**. If the instruction does not include a memory operation, and the default memory address register has not been disabled using the **dpregs** instruction, then this may be left blank.

The instr–ext–reg Argument

The *instr–ext–reg* argument is optional and and specifies an IU register containing a 32-bit value to be used either as the immediate value (for Immediate format instructions), the memory stride (for Memory Stride format instructions), or the vector length (in Mode Set format instructions) if the instruction being executed was assembled in one for those formats.

Note: This argument must not be given if the instruction (previously assembled into memory using **dpasm**) is not in one of the three formats listed above. Since these are all long formats, the third argument is only meaningful in the **dpexecd** instruction. An error is signalled if it is specified in the **dpexec** instruction.

When specifying a register containing the vector length for a Mode Set format instruction, the register name must be preceded by an asterisk, as in:

```
dpexecd [%i0],,*i2
```

which will take the argument from **%i2**. An example of vector length specification can be found below.

Caution: If you omit the asterisk, **dpas** cannot catch this error, and will generate a totally meaningless instruction.

You can improve the performance of the **dpexecd** instruction by preparing the *instr–ext–reg* register in a particular format. Specifically, the vector length can be decremented and shifted into bits 19 through 22 of the register. All other bits of the register must be zero. If you prepare the register like this, you must append the “**<**” character to the register to indicate that it is shifted. The above example, using such a prepared register, becomes:

```
dpexecd [%i0],,*i2<
```

The align/noalign Modifier

The fourth argument is used only for the **dpexecd** instruction. If given as **align**, it guarantees that the memory address is doubleword aligned. This guarantee allows **dpas** to generate more efficient code to execute the instruction (specifically, a **std** instruction). Without the guarantee, it takes an extra SPARC instruction to execute the VU instruction. It is not necessary to guarantee alignment if (1) the instruction is a SHORT format instruction (i.e. **dpexec** is used), (2) no memory operation is performed, or (3) the **–a** switch was given to **dpas** which indicates that alignment is guaranteed by default. Setting the argument to

noalign forces the slower safe code to be used. If the **–a** switch was given, then instructions with memory addresses that may not be doubleword aligned must have **noalign** as their third argument.

6.3 Examples

As an example, assume that in some DPEAC source file, the following was coded:

```
.seg   "data"
dpasmd _vector_add_instr, iaddv v2,0r1.0,V3; iloadv [],v2
dpasmd _vector_sub_instr, isubv v2,0r1.0,V3; iloadv [],v2
```

Notice that, due to the use of the immediate operand, this instruction is a long format instruction, the LI format, requiring use of **dpasmd**.

Now we will write a DPEAC routine that adds or subtracts integer 1 to a vector in memory, putting the result in V3. It will do this by taking as an argument (in **%i0**) a pointer to the correct instruction to execute (**_vector_add_instr** or **_vector_sub_instr**) and executing that instruction. A second argument (in **%i1**) will be the memory address of the vector, which is known to be doubleword aligned. We will assume that the accelerator's control registers have previously been set up with the proper vector length and memory stride values.

```
dpentry _add_or subtract_one,1,0
dpexecd [%i0],[%i1],,align   ! execute instr at %i0,
aligned mem addr in %i1
dpretn
```

The following instructions then call the above routine from DPEAC (or SPARC assembly) code using the memory address stored in **%l3**:

```
sethi   %hi(_vector_add_instruction),%o0
or      %o0,%lo(_vector_add_instruction),%o0
mov     %l3,%o1
call    _add_or_subtract_one
```

Or the equivalent in C (which requires that we have a leading underscore in the label on the instruction object):

```
extern int *vector_add_instruction;
extern int *vector_sub_instruction;
int maddr;
  ...
add_or_subtract_one(&vector_add1_instruction, maddr);
  ...
```

Although not shown, the calling code can easily select between the two instructions (or more) simply by loading the pointer with the correct address.

We can enhance this to add or subtract any integer value (passed to the DPEAC routine) by using the *instr–ext–reg* argument to **dpexecd** instruction to specify the immediate value used by the instruction:

```
dpentry _add_or subtract_N,1,0
dpexecd [%i0],[%i1],%i2,align   ! exec instr at %i0,
immediate value in %i2
dpretn
```

We call the routine as before, but this time we pass it the value to be added (assumed to be in **%l2**):

```
sethi   %hi(_vector_add_instruction),%o0
or      %o0,%lo(_vector_add_instruction),%o0
mov     %l3,%o1
mov     %l2,%o2
call    _add_or subtract_N
```

Or the equivalent in C:

```
extern int *vector_add_instruction;
extern int *vector_sub_instruction;
int maddr, value;
  ...
add_or_subtract_N(&vector_add1_instruction, maddr,
value);
  ...
```

For another example, we show how to assemble an Mode Set format instruction for which the vector length will be given at execution time.

Here is the assembly of the instruction into a memory location:

```
dpasmd  mylabel, floadv*=%i2 [],V2
```

Now we can execute the assembled instruction, and supply the value to be used as the vector length from %i2:

```
!assume %l3 contains address of "mylabel", and %l4 contains
!the doubleword-aligned memory address for the load.
!Register %i2 contains the desired vector length.
dec     %i2                      ! Format %i2 with length-1
sll     %i2,19,%i2               !  ...in bits 19 through 22
dpexecd [%l3],[%l4],*%i2<,align ! execute and indicate
                                 ! shifted register
```