

CSSE3001/7000 2007

Prac 4: Introduction to VIS – SIMD Instructions for SPARC

Last Update: 27/03/2007 1:58 PM

Author: [Simon Leung](#)

Preparations

You will need to have a good understanding of Prac 1, 2 and 3 in order to take advantage of practical VIS skills.

Don't forget you need to use the very useful reference document "[The SPARC Architecture Manual, version 9](#)" (also [available internally](#)) to look up information. Use it smartly and carefully, looking at the relevant sections and use the search function of the PDF reader to help you navigate in this book.

In addition, the VIS Instruction Set User's Manual ([available internally](#)) will be useful. This prac does not cover all VIS instructions, so take a look at the listing in Ch.4 of the manual.

Investigate the Unix command "alias" – it can save you much typing. Also check out the functions of the "up arrow" and "down arrow" on your keyboard in the Unix shell.

Setup

For this prac, we will use compilers from Sun (Sun Workshop 6). You will need to set up your path to the Sun compiler installed on agave, so make sure `PATH` has these entries:

```
/opt/SUNWspro/bin
/usr/ccs/bin
/opt/local/bin
/usr/bin
/sbin
/usr/sbin
/opt/local/sbin
/usr/openwin/bin
```

For example, the command:

- `export PATH=$PATH:/opt/SUNWspro/bin:/usr/ccs/bin:/opt/local/bin\:/usr/bin:/sbin:/usr/sbin:/opt/local/sbin:/usr/openwin/bin`

will achieve the desired result.

The VIS toolkit is installed on `/opt/local/pkg/cad/csse3001/SUNWv SDK`. You will need to add some additional command flags at compile time to integrate VIS instructions into your program.

Procedure

Task 1: Pointers, Addresses, Type Casting and Moving Data between Integer & Floating Point Registers

Recall the relationship between pointer, address and contents in C:

Eg., if `p=&c` then `p` is a pointer to variable `c` which has a value equal to the address of `c`. `*p` is also the content of the variable `c`.

Extract from Kernighan & Ritchie:

```
int x=1, y=2, z[10];
int *ip;    /* ip is declared a pointer to an int type variable */
ip = &x;    /* ip now has the memory address of x, or ip now points to x */
y = *ip;    /* y is now equal to 1 */
*ip = 0;    /* x is now equal to 0 */
ip = &z[0]; /* ip now points to z[0] */
```

- Make sure you understand the code segment above and how it works.

Also recall type casting:

```
int i;
double d;
d=(double)i;    /* d now has the value i, but changed to double precision
floating point format */
```

If you examine the (RAW) bit pattern of `i` and `d`, they are different. Yet they both represent the same value.

So probably it is no surprise to you that pointers can be cast as well:

```
int* i;
double* d;
d=(double*)i;    /* resolve any type issues as i & d are pointers of
different types */
```

- Is the casting necessary? Will all compilers accept the same statement without the cast?

This preliminary work paves the way for us to construct statement in C which allows the content of the integer registers to be transferred to the floating point registers (via memory/stack) and vice versa. This is very important for VIS type instructions as they only work on floating point registers.

If `f` is a `float` type variable and `i` is an unsigned `int` type variable, answer the following questions:

1. What is `&i`?
2. What is `(float*)&i`?
3. What is `*((float*)&i)`?
4. If `i=0x0000FFFF`, what will be the bit pattern for `f` after the statement `f=*((float*)&i)`?
5. How do you copy a bit pattern from a floating point variable into an integer variable?

Task 2: Utilities Inline useful for VIS instructions

Recall in your previous studies, floating point numbers are standardised by IEEE. Most common ones are the single precision (32 bits) and double precision (64 bits) representations.

For example, the 32-bit pattern `0x41100000` represent `+9.0` in single precision format.

Use the following code as a starting point (and modify if necessary), answer the following questions:

```
#include <stdio.h>

int main()
{
    unsigned int i=0x41100000;
    float f_cast, f_transfer;
    f_cast = (float)i;
    f_transfer = *((float*)&i);
    printf("The bit pattern for the unsigned integer i is 0x%x.\n", i);
    printf("The value of the unsigned integer i is %d.\n", i);
    printf("The value of f_cast is %f.\n", f_cast);
    printf("The value of f_transfer is %f.\n", f_transfer);
    return (0);
}
```

1. What is the value in `f_cast`? Run the code to confirm your prediction. Why does `f_cast` have that value?
2. What is the value in `f_transfer`? Why?
3. What is the value of the single precision representation bit pattern `0xc0c40000`?
4. What is the single precision bit pattern for the value 114.25?

In SPARC microprocessor architectures, all VIS instructions operate only on 64 bit floating point registers. Hence, we must put the data into a 64-bit floating point type (double) so that we could use VIS instructions. As you can see, each VIS operation can potentially be performed on two 32-bit floating point numbers and speed up could result. Before we actually investigate the VIS instructions, let us look at some inline utilities to move data into and out of the 64-bit floating point variables.

Use the following code as a starting point (and modify if necessary), and note the extra `#include` statement. In order to compile this code, you need to tell the compiler where to find `vis.h` and other information. Also VIS instructions are only supported by SPARC architecture V8+ and later only. So include the flags `-I/opt/local/pkg/cad/csse3001/SUNWv sdk/include` and `-xarch=v8plusa`, and compile with the inline file `/opt/local/pkg/cad/csse3001/SUNWv sdk/lib/vis_32.il` along with the source file, for example:

```
cc -I/opt/local/pkg/cad/csse3001/SUNWv sdk/include -xarch=v8plusa -xvis \  
-o vis vis.c /opt/local/pkg/cad/csse3001/SUNWv sdk/lib/vis_32.il
```

Refer to Chapter 3.2 of the VIS Instruction Set User's Manual for more compiling information.

```
#include <stdio.h>
#include <vis.h>
#include <stdio.h>

int main()
{
    unsigned int a=0x41100000;
    unsigned int b=0xc0c40000;
    unsigned int x,y;
    float f,g;
    /*** define c as a type of your choice here ... */
    double d;

    d = vis_to_double(a,b);
    f = vis_read_hi(d);
    g = vis_read_lo(d);
    x = *((unsigned int*)&c);
    y = *((unsigned int*)&d);

    printf("The two bit patterns for the unsigned integer a & b are 0x%x and
```

```

0x%x.\n", a,b);
    printf("The value of the 64 bit double is %f.\n", d);
    printf("The pattern in the upper 32-bit of d has a value %f.\n",
(double)f);
    printf("The pattern in the lower 32-bit of d has a value %f.\n",
(double)g);
    printf("Variable x of unsigned int type has a bit pattern of 0x%x.\n",
x);
    printf("Variable y of unsigned int type has a bit pattern of 0x%x.\n",
y);
    return (0);
}

```

Answer the following questions:

1. What does the function `vis_to_double()` do?
2. What does the function `vis_read_hi()` do?
3. What does the function `vis_read_lo()` do?
4. What are the other “Utilities Inlines” made available by SUN to help with VIS instructions?
5. What is the value of the double precision floating point number with bit pattern `0x40934a449ba5e354`?
6. What is the bit pattern in double precision format for the number 2999.8887?
7. What happens if you accidentally declare `a,b,x` and `y` as `int` (i.e., signed `int`) rather than `unsigned int`?
8. What happens if you vary the type of `c`?

Task 3: VIS Instructions

Using your favourite text editor, create a file called `sum_vis.c` with the following code:

```

#include <stdio.h>
#include <vis.h>

signed int i1,i2,j1,j2,k1,k2;
float tmp1,tmp2;
double di,dj,dk;

int main()
{
    i1=4;
    i2=6;
    j1=8;
    j2=10;

    di = vis_to_double(i1,i2);
    dj = vis_to_double(j1,j2);

    dk = vis_fpadd32(di,dj); /* The first VIS instruction */

    tmp1 = vis_read_hi(dk);
    tmp2 = vis_read_lo(dk);
    k1 = *((signed int*)&tmp1);
    k2 = *((signed int*)&tmp2);

    printf("The first vector is [%d; %d].\n", i1, i2);
    printf("The second vector is [%d; %d].\n", j1, j2);
    printf("The addition result is [%d; %d].\n", k1, k2);

    return (0);
}

```

Compile this code to observe the assembly output & the results. Answer the following questions:

1. How is the addition performed, ie. is it i_1+i_2 & j_1+j_2 , or i_1+j_1 & i_2+j_2 ?
2. The `vis_fpadd32()` function adds the numbers together. How many assembler instruction(s) is/are used to perform the add?
3. Why do we need variables `tmp1` and `tmp2`?
4. Would you want to do this if you are really only adding two pairs of signed integers? Why or why not? Perform some measurements of time/instruction count.
5. Using this code as the template, explore at least two other VIS arithmetic instructions as shown on Ch.4.6 of the VIS Instruction Set User's Manual.

Of course, the usefulness of VIS instructions is shown when operating on a set of numbers already in memory, rather than individually.

Using your favourite text editor, create a file called `sum_vis2.c` with the following code:

```
#include <stdio.h>
#include <vis.h>

signed int i[4], j[2], k;
double d1,d2,d3;

int main()
{
    for (k=0;k<4;k++)
        i[k]=k;          /* putting data into memory */

    d1 = vis_ld_d64(&i);    /*loading the first fp register with two 32-bit
int i[0] and i[1]*/
    d2 = vis_ld_d64_i(&i,8); /*loading the second fp register with the next
two 32-bit int i[1] and i[2]*/

    d3 = vis_fpadd32(d1,d2);

    vis_st_d64(d3,&j);      /*storing the result of the vis_add to array j*/

    printf("The first vector is [%d; %d].\n", i[0], i[1]);
    printf("The second vector is [%d; %d].\n", i[2], i[3]);
    printf("The addition result is [%d; %d].\n", j[0], j[1]);

    return (0);
}
```

1. Investigate the functionalities of `vis_ld_d64`, `vis_ld_d64_i` and `vis_st_d64`.
2. Each `int` occupies 4 bytes of memory (32 bits). In the first `vis_ld_d64`, how many bytes are loading from the array `i`?
3. Why is an offset of 8 needed to load the second FP register?
4. When referencing the arrays, why is the operator `&` used instead of a direct reference? Is this really necessary?
5. According to Ch.4.8 of the manual, what are the other loads and stores available for use?
6. Write a C program which will use VIS instructions to print the sum of an array of 20 short integers.

Summary

In this prac, we have covered:

- Pointers in C;

- Type casting in variables and pointers;
- Integers vs Floating Point and their registers;
- Some sample VIS instructions and the way to use them.

Suggested Exercises

You should reinforce your understanding of these concepts by modifying the code and try to make some mistakes and redundant steps. As a suggestion, experiment with the other VIS instructions listed in the manual. In particular, if you are interested in DSP, there are quite many instructions which will allow you to do processing much faster.

Throughout this prac we have been using the C compiler from SUN, designed by SUN and optimised for their SPARC architecture. Most of you would be familiar with gcc, a free GNU compiler. It is also available on agave as long as you don't modify the PATH variable at the start of the prac. Can VIS instructions be handled by gcc?
