If the program is then executed:

```
%expr
%
```

2.7 The Debugger gdb

Having written a program, which apparently executed, we have no way of knowing what it did, as it produced no output. In fact, we do not even know if the program was correct. The technique of inserting **printf** statements into a program to verify correctness and to find bugs is a rather cumbersome process in assembly language, as arguments have to be placed in registers and `_printf` called. If we are trying to debug a program that has other variables stored into the registers needed to call printf, real problems start to develop. The debugger **gdb** provides a way of printing out values without having to change the program in any way. The debugger **gdb** may also be used to execute a program, to stop execution at any point, and to single-step execution. A detailed description of gdb is to be found in [20], however, the discussion relates mostly to C language debugging.

In order to use **gdb**, it is necessary to run the compiler with the "-g" switch. Having assembled the program, placing the output into **expr** as we did in the above example, **gdb** may be entered by typing:

```
%gdb expr
```

gdb prints a disclaimer and waits for commands:

```
GDB 3.2, Copyright (C) 1988 Free Software Foundation, Inc.
There is ABSOLUTELY NO WARRANTY for GDB; type "info warranty"
for details.  GDB is free software and you are welcome to
distribute copies of it  under certain conditions; type
"info copying" to see the conditions.
Reading symbol data from /home2/lou/expr...done.
Type "help" for a list of commands.
(gdb)
```

To run the program in **gdb**, type "r":

```
(gdb) r
Starting program: /home2/lou/book/ch2/sparc

Program exited with code 0345.
(gdb)
```

Apparently the program executed, but we are not much further ahead than we were when we executed the program within the shell. We need to set a "breakpoint" in the program. A breakpoint may be set at any address and whenever the computer

is about to execute the instruction at which the breakpoint was set, it stops and returns to **gdb**, whereupon the program and its state of execution may be examined. Typing "c" will tell **gdb** to continue execution from the breakpoint. In order to set a breakpoint at a memory address we need to type:

```
(gdb)b *addr
```

where addr is the machine memory address. A good place to break our program would be at the first instruction after the `save` instruction has been executed. To do this in gdb we type:

```
(gdb) b main
Breakpoint 1, 0x2290 in main ()
(gdb)
```

Why did we type only `main` and not `_main`? The C compiler prepends an _ to all identifiers so that the symbol `main` in C becomes `_main` in assembly language. As this happens all the time, **gdb** always tries prepending an _ to any symbol typed in case the _ version is present. The command "b" followed by a label sets a breakpoint at the instruction following the labeled instruction; gdb assumes the labeled instruction to be a `save` instruction.

If we then run the program:

```
(gdb) r
Starting program: /home2/lou/book/ch2/sparc

Breakpoint 1, 0x2290 in main ()
(gdb)
```

gdb tells us that we are at Breakpoint 1, which should be the first instruction in our program. The program counter, %pc, will have the address of the instruction 2294.

We can examine memory by typing "x" followed by an address. In this case we would like to use the contents of the %pc as the address. To do this, we type:

```
(gdb) x/i $pc
0x2294 <main+4>:        mov 9, %l0
(gdb)
```

The examining command "x" has to be followed by a format specified to tell **gdb** how to print out the value stored in the memory location. The "i" format specifier states that the contents of the memory location should be interpreted as a machine instruction. In **gdb** all machine registers are referred to by a $ in place of the % used in as.

By typing a return we repeat the last command but with the address incremented by the size of the last data element typed out:

```
(gdb)
0x2298 <main+8>:        sub %l0, 1, %o0
(gdb)
```

We may print the entire program by typing `disassemble`[1]. This command prints all the instructions of the current function:

```
(gdb) disassemble                                    x/12i main
Dump of assembler code from 0x2290 to 0x22b8:
0x2290 <main>:  save %sp, -64, %sp
0x2294 <main+4>:        mov 9, %l0
0x2298 <main+8>:        sub %l0, 1, %o0
0x229c <main+12>:       sub %l0, 7, %o1
0x22a0 <main+16>:       call 0x409c <_DYNAMIC+156>
0x22a4 <main+20>:       nop
0x22a8 <main+24>:       sub %l0, 0xb, %o1
0x22ac <main+28>:       call 0x4090 <_DYNAMIC+144>
0x22b0 <main+32>:       nop
0x22b4 <main+36>:       mov %o0, %l1
0x22b8 <main+40>:       mov 1, %g1
0x22bc <main+44>:       t 0
End of assembler dump.
(gdb)
```

If we want to see whether the program ran correctly we can set another breakpoint at the `trap` instruction located at main+44. To obtain an address, given a label, we prepend an `&` much as we would do in C. Thus, to set a breakpoint at `_main + 44`, we would type:

```
(gdb) b *& main + 44
Breakpoint 2 at 0x22bc
(gdb)
```

While `*&` is an identity operation in C, it is not in **gdb**.

We would then command **gdb** to continue execution by typing "c" (remember we are currently stopped at the first location in our program):

```
(gdb) c
Continuing.

Breakpoint 2, 0x22bc in main ()
(gdb)
```

The program executes and stops at the last breakpoint we set. At this point the value should be stored in register %l1. To print the contents of a register we use the print command "p:"

[1] In some versions of gdb the `disassemble` command is: `asdump`.


```
(gdb) p $l1
$2 = -8
(gdb)
```

This tells us that the contents of register %l1 is −8, the correct value. The `$2 =` is part of **gdb's** history feature. The value −8 has been saved in a history variable $2 and may be used at any time by typing $2.

What would happen if our program were incorrect and did not compute the correct value? We could single-step the program starting at the beginning by typing "ni" for next machine instruction. To do this at this point we would need to run the program again:

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home2/lou/book/sparc/ch02/a.out

Breakpoint 1, 0x2294 in main ()
1: x/i $pc  0x2294 <main+4>:    mov 9, %l0
(gdb)
```

We are executing the program, but it would be helpful to know what instructions were being executed. We can discover this by examining the memory location the %pc is pointing to:

```
(gdb) x/i $pc
0x2294 <main+4>:        mov 9, %l0
(gdb)
```

and indeed we have just executed the first instruction and are about to execute the second. If we execute the second instruction, then %l0 should contain the value 9:

```
(gdb) p $l0
$2 = 9
(gdb)
```

and indeed it does.

As we single-step our program we would probably like to have the instruction to which the program counter is pointing printed out every time without our having to type p $pc. We can do this with the "display" command, which prints its value every time a command is executed:

```
(gdb) display/i $pc
1: x/i $pc  0x2298 <main+8>:    sub %l0, 1, %o0
(gdb) ni
0x229c in main ()
1: x/i $pc  0x229c <main+12>:   sub %l0, 7, %o1
(gdb)
```

Then when we execute the **next** command, the instruction about to be executed is automatically printed out. We are now about to execute the call to .mul:

```
(gdb) ni
0x22a0 in main ()
1: x/i $pc  0x22a0 <main+16>:   call 0x409c <.mul>
(gdb)
0x22a4 in main ()
1: x/i $pc  0x22a4 <main+20>:   nop
(gdb)
0x22a8 in main ()
1: x/i $pc  0x22a8 <main+24>:   sub %l0, 0xb, %o1
(gdb)
```

Note that the "delay slot" instruction is executed before the call to .mul. We have been typing "ni" for next instruction. We could have typed "si" but this would have stepped us through the .mul routine, a thing we probably don't want to do. Both "ni" and "si" execute single instructions, but "ni" does not single-step through any functions that are called. Note also that after typing "ni" the first time, we then typed only a carriage return; in **gdb** a carriage return repeats the last command.

These commands are not all the commands available to **gdb** but are enough to begin with and will enable you to write and to debug simple programs. One final command you must know is "q," to quit gdb and to return to the operating system:

```
(gdb) q
The program is running.  Quit anyway? (y or n) y
>
```

2.8 Filling Delay Slots

The `call` instruction is called a "delayed control transfer" instruction. A delayed transfer instruction changes the address from which future instructions will be fetched after the instruction following the delayed transfer instruction has been executed. The instruction following the delayed control transfer instruction is called the "delayed instruction" and it is located in the delay "slot." Whenever a branch or call instruction is executed it changes the contents of %npc, not the %pc. The instruction that follows the branching instruction will be executed **before** the branch or call happens. By filling the delay slot with a **nop** instruction we have not accomplished very much; the pipeline machine wastes an instruction execution every time it branches. However, as the delay instruction is executed before the first instruction at the branch address was executed, we may move the instruction prior to the branch instruction into the delay slot.

In the following version of the program we have moved the **sub** instructions, which compute the final argument to **.mul** and **.div** into the delay slots thereby

eliminating the **nop** instructions. The resulting code does not lose any cycles at all.

```
  .global _main
_main:
  save    %sp, -64, %sp
  mov     9, %l0          !initialize x
  sub     %l0, 1, %o0     !(x - 1) into %o0
  call    .mul
  sub     %l0, 7, %o1     !(x - 7) into %o1
  call    .div
  sub     %l0, 11, %o1    !(x - 11) into %o1, the divisor
  mov     %o0, %l1        !store it in y

  mov     1, %g1          !trap dispatch
  ta      0               !trap to system
```

Filling the delay slots in this manner makes reading the program more difficult, but by filling the delay slots the resulting execution is faster and the size of the program smaller. Care must be taken in filling delay slots in order to ensure that the algorithm is not changed. In general, when we write assembly language programs we will be expected to fill all possible delay slots.

2.9 Branching

We can now add, subtract, multiply, divide, and move data around. What we cannot yet do is to test and to branch. Without these capabilities we will not be able to write very interesting programs. Branching is used in conjunction with testing, which we will discuss first.

2.9.1 Testing

In the HP Calculator, the last number computed could be tested. For example, there was an instruction `ifeq`, which would skip the next instruction in line if the result last computed was zero. A similar technique is used in many computers, in which the state of the execution of each instruction may be tested. In order to do this, only information about the result need be kept, not the result itself. The state of execution is saved in terms of four variables:

Z whether the result was zero

N whether the result was negative

V whether execution resulted in a number too large to store in the register