

---

# *Introduction to SPIM Programming*

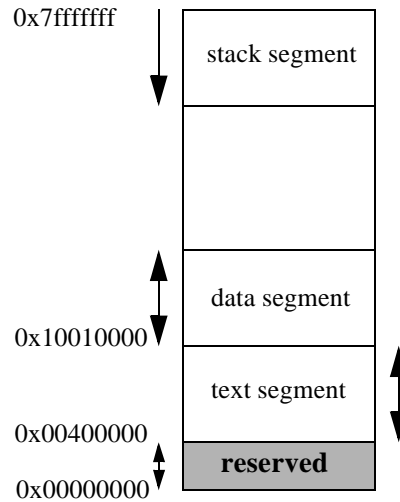
---

It is important to note that the GT SPIM datapath is intended to serve as an example of a datapath programmed at the assembly language level as well as serve to convey basic computer architecture concepts. Only a small set of instructions are supported by the GT SPIM hardware. The complete SPIM instruction set is much more extensive encompassing many more instructions. Commercially available processors that implement this complete instruction set are correspondingly more complex. This chapter discusses key features of the SPIM instruction set which in conjunction with the SPIM documentation and an instruction set simulator enables users to get started on writing and testing non-trivial programs.

---

## *The SPIM Memory Map*

The first step in SPIM programming is knowledge of how SPIM programs use memory. The GT SPIM datapath provides two physically separate memory modules for instructions and data. However the SPIM programming model presents to the programmer a single unified memory address space with addresses ranging from 0x00000000 to 0x7fffffff. In addition a convention is defined regarding the areas of memory that are used for the storage of instructions and data. For example, all instructions in a program are stored in memory starting at address 0x00400000. Memory addresses are byte addresses. Note that instructions are 32-bit words and are constrained to be stored starting on word boundaries. This area of memory is referred to as the *text segment*. The first instruction of a program is stored in memory location 0x00400000, the second instruction is stored in location 0x00400004, and so on. Data stored in memory using data directives utilizes the area of memory starting at address 0x10010000. This



**FIGURE 1. SPIM memory map**

area of memory is referred to as the *data segment*. Memory locations starting at address 0x00000000 and extending upto 0x00400000 are reserved and are not available to user programs. For our purposes we can think of these locations as being reserved and used by the operating system. Finally, there is a fourth area of memory referred to as the stack. We will revisit the stack segment and its functionality later during the discussion of procedure calls. A *memory map* is such an allocation of memory addresses to specific types of data and activities. The SPIM memory map is shown in Figure 1. This is the logical view of memory maintained by the SPIM programmer and implemented by the SPIM assembler. Finally, note that memory is byte addressed. Thus every fourth address is a word address or word boundary. Thus, the two least significant bits of every address corresponding to a word boundary are 00. Equivalently, the least significant digit of a word address in hexadecimal notation will be 0x0, 0x4, 0x8, or 0xc.

The task of the assembler is now easy to describe. A text file containing a SPIM program is processed by the assembler. The data directives are interpreted and cause data and ASCII values to be stored at locations in the data segment. The instructions are processed, encoded according to the appropriate format, and stored at successive locations in the text segment. Execution of the program can now begin. However an understanding of how these programs can be assembled in this

fashion merits a discussion of several key features of the SPIM ISA discussed in the following subsections.

---

### *Anatomy of a SPIM Program*

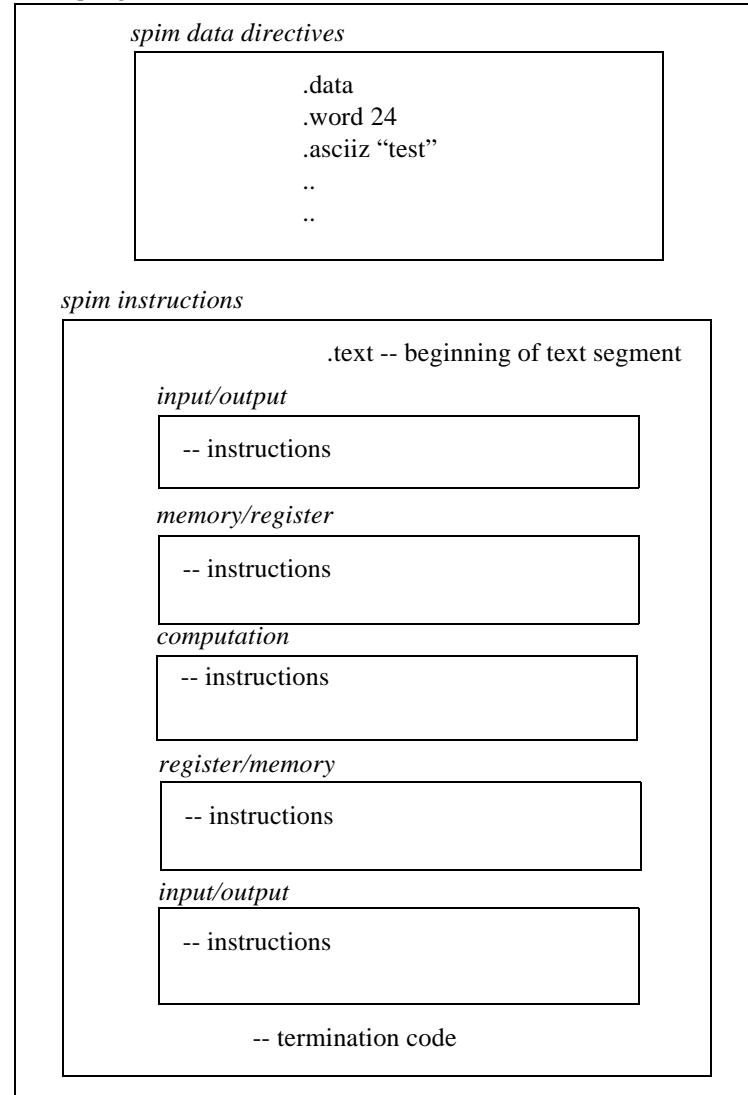
To ease the task of writing one's first program this section presents the basic elements of a SPIM program captured in the program template of Figure 2. A typical program will have two components corresponding to data directives and instructions. The data segment description contains data directives such as those for storing data, reserving space, and storing character strings corresponding to messages to be printed to the screen. The data segment directives are followed by the SPIM instructions. The use of the *.text* directive signifies the end of the data directives and the beginning of instructions that are to be encoded and stored in the text segment. Within the text segment there are some common groups of instructions for input, output, moving data between memory and registers, and of course for processing the data stored in registers. The basic components of a typical SPIM program are described in the following.

#### **The SPIM Registers**

There are 32 SPIM registers that can be referenced with the labels \$0, \$1, ..., \$31. Register \$0 is special - it is hardwired to the value 0. Thus, the contents of register \$0 cannot be changed. The instruction set designers chose to make the value of 0 available after extensive analysis of the compiled assembly language code from sets of benchmark applications. The value 0 is surprisingly useful in synthesizing a number of useful operations. The SPIM registers are known as *general purpose* registers as opposed to registers such as the PC which are used for specific purposes and cannot be used by user programs. However, there are conventions followed by the compilers and assemblers wherein some of the registers are utilized for special purposes and it is not advisable to use these same registers for application programming. For example, \$1 is used for building 32-bit constants and \$2, \$3, \$4, and \$5 are used in procedure and function calls. How can we use registers without conflicting with the conventions for their use by the assembler? Not surprisingly the designers have adopted a naming convention that enables programmers to avoid using registers that may have special uses.

In addition to a numeric value as \$3 or \$23, each register can be identified by a symbolic name. For example register \$1 can also be referenced as \$at. Register \$28 can also be referenced as \$gp. The symbolic names for all 32 registers can be found in Table 2 of the SPIM documentation. Among them we will find two sets of registers with symbolic names - \$t0, \$t1, ..., \$t9 and \$s0, \$s1, ..., \$s7. At this point we can use these registers in our programs without concern about conflict with use by the assembler. Later we will have to distinguish between these two sets of registers when we discuss procedure and function calls. For now these 18 registers should suffice for any programs we will write. When using symbolic register names the SPIM instructions will now appear as follows.

*SPIM program*



**FIGURE 2. Example structure of a SPIM program**

```
add $t3, $t0, $t1
lw $t6, $t8
```

*Example:* The availability of register \$0 hardwired to the value of 0 makes it convenient to initialize registers to specific values. For example, we can initialize the contents of \$t0 to the value 0 or to the value 16 as follows.

```
add $t0, $0, $0 #initialize $t0 to the value 0
addi $t0, $0, 16 #initialize $t0 to the value 16
```

*Example:* The availability of the value 0 is also useful in moving values between registers. For example, suppose we wish to move the contents of register \$t0 to \$t7. The following instruction will achieve this goal. This example specifies register \$0 using its symbolic name, namely \$zero.

```
add $t7, $t0, $zero
```

In fact the SPIM ISA provides a “move” instruction that can be used as follows.

```
move $t0, $t7
```

This instruction states that the contents of \$t7 are to be copied into register \$t0.

### Use of Labels

Labels in a SPIM program are a “programmer friendly” way of identifying memory locations. To this point we have accessed memory locations based on the numerical values of the memory addresses. For example, in the use of the load instruction,

```
lw $t0, $t1
```

the assumption was that the contents of register \$t1 contained the address of a memory location. How was this address placed in this register? How did we know what address to use? In a real machine with 32-bit addresses it is clearly infeasible to expect the programmer to know and keep track of vast number of physical addresses even if we do understand how data directives are processed. Therefore we need a

tractable way in which to identify memory locations and thus arrive at the use of labels. Consider the following sequence of data directives.

```
.data
L1: .word 0x22
L2: .ascii "Print me!"
    .align 2
L3: .space 32
```

L1, L2, and L3 are labels. The value of a label is a memory address and it is associated with a data directive. For example, L1 is the label associated with the `.word` directive. The value of L1 is the address of the data processed by the `.word` directive. In the above case it is the address of the memory location that contains the value 0x22. Since there are no preceding data directives this is the first word in the data segment and the value of L1 is the start of the data segment which is 0x10010000. The value of L2 is associated with the `.ascii` directive. This string starts at the next address of the data segment which can be determined to be 0x10010004. The character string "Print me!" requires 9 bytes for the characters and one byte for the null termination character for a total of 10 bytes. The following `.align 2` directive causes the assembler to move to the next word boundary whose address is 0x10010010. Therefore the value of L3 is 0x10010010. Generally starting from the first data directive it is possible to determine the values of all labels used in the data segment description.

Now consider the following modification to the above sequence of directives.

```
.data
L1: .word 0x22, 0x32, 0x44, 0x66
L2: .ascii "Print me!"
    .align 2
L3: .space 32
```

Now what is the value of L1? It is unchanged being associated with the first value following the data directive. How about the memory address of the location that contains the value 0x32? This value is stored in the next word and therefore its address can be inferred to be L1+4 since each word requires 4 bytes and memory is byte addressed. Proceeding in this fashion we note that the value 0x66 specified in the above data directives is stored in memory location L1+12. We can think of labels as a way of "marking" memory locations so that we can refer to these locations by their label rather than by trying to keep track of numerical address values. It can be verified in the

modified example above the values of L1, L2 and L3 are now 0x10010000, 0x10010010, 0x1001001c respectively.

Other than serving as markers for memory locations what purpose do labels serve? With such a way of marking or identifying memory locations with symbolic names the task of loading and storing data from/to memory using the `lw` and `sw` instructions becomes quite a bit easier as described in the next section.

### Addressing Modes

The manner in which addresses for memory accesses are constructed is referred to as the *addressing mode*. Both `lw` and `sw` instructions as described earlier provide the address of a memory location in a register. This mode is referred to as *register indirect* mode and requires that a sequence of instructions first construct the memory address in a register and that this register be used in the load or store instruction.

A very useful alternative mode in which to address memory is *indexed addressing* or base-offset addressing. In this mode the address of a memory location is specified by providing a base address and an offset from the base address. For example, a `lw` instruction that employs the indexed addressing mode would appear as follows.

```
lw $t0, 4($t1)
```

During the execution of this instruction the contents of register `$t1` is added to the value 4. This sum is used as the address to memory and the contents of that location is fetched and placed in register `$t0`. The `sw` instruction is similarly defined.

```
sw $t0, 4($t1)
```

This is the form of the `lw` and `sw` instructions used in the SPIM ISA. In general, the value 4 can be replaced by any constant. However, the use of labels provides a more powerful motivation for this style of `lw/sw` instructions. Consider the following code sequence.

```
.data
L1: .word 0x22, 0x32, 0x44, 0x66
L2: .ascii "Print me!"
    .align 2
L3: .space 32
```

The first data value in the above sequence can be stored register `$t0` with the following instruction.

lw \$t0, L1(\$0)

Consider the implementation of this instruction. The address is constructed by the sum of the contents of register \$0 and the offset L1. Since the contents of \$0 is always the value 0 the resulting sum is L1. The contents of memory at this address, 0x22, are fetched into register \$t0. Conversely if we wish to store a value into the data segment we might utilize the following instruction.

sw \$t0, L3(\$0)

The address is computed as  $L3+0 = L3$ . The contents of \$t0 are stored at this address where 32 bytes have been reserved by the use of the .space directive.

The SPIM lw and sw instructions use this indexed addressing format. If the offset is omitted it is assumed to be zero. Such an addressing mode makes it convenient to address regions of the data segment without having to keep track of numerical addresses. Indexed addressing is particularly useful in the structured accesses to elements of arrays. Some examples will help illustrate the utility of indexed addressing.

*Example:* Imagine an array of 8 values stored in memory and a code sequence that must compute the sum of every other value. Register \$t0 is used to keep track of the offset from the starting address of the array. Register \$t2 is used to store the sum. Note that since we use labels we do not have to know the numerical address of the start of the data segment!

```
.data
L1: .word 2,4,6,8,10,12,14,16 # the array
L2: .space 4 #reserve space to store the sum
.text
add $t0, $0, $0 # initialize $t0=0
add $t1, $0, $0 # initialize the sum = 0
lw $t2, L1($t0) # load first value
add $t1, $t1, $t2 # add to sum
addi $t0, $t0, 8 # computer addr of two elements down
lw $t2, L1($t0) # load second value
add $t1, $t1, $t2 # add to sum
addi $t0, $t0, 8 # computer addr of two elements down
lw $t2, L1($t0) # load third value
add $t1, $t1, $t2 # add to sum
addi $t0, $t0, 8 # computer addr of two elements down
lw $t2, L1($t0) # load fourth value
add $t1, $t1, $t2 # add to sum
```



*Example:* The use of labels also makes it easy to reserve storage in the data segment for the placement of values. For example, in the preceding code segment the sum of the four elements of the array is computed in register \$t1. Note that the data segment contains a directive for the reservation of four bytes which is one word. This location is marked with the label L2. Since it is marked we can reference it in a SW instruction and therefore we can access that location. The instruction

sw \$t2, L2(\$0)

can be used to store the final value of sum in the data segment. Again, without having to worry about numerical address values. However, from the operation of the datapath we know that the instructions must use numerical values for the address not symbols! The use of labels is a programming convenience to ease the writing of assembly programs. The process of translating the labels into numerical values is one of the many additional functions performed by the assembler when the instructions are encoded.

### Input/Output

SPIM provides facilities for writing to the console window and reading from the console window. The basic philosophy is the use of a special instruction, `syscall`, that invokes another program that actually performs the input/output operation. This is best illustrated by an example. Consider the case where we wish to print the contents of register \$t0 to the console window as an integer value. This operation is comprised of three steps.

1. Place the contents of \$t0 in register \$a0 (for example using the instruction `add $a0, $t0, $0`)
2. Place a code signifying “print integer” in register \$v0. The code is 1 for printing integers to the screen (for example using the instruction `addi, $v0, $0, 1`)
3. Execute the instruction `syscall`.

A code sequence produced by these steps is given below.

```
.text
addi $v0, $0, 1      # place the value 1 in $v0
add $a0, $t0, $0    # place the contents of $t0 in $a0
syscall
```

When a `syscall` instruction is executed a special handler program will start executing. This handler always first examines the contents of register \$v0 looking for a code to determine the service to be performed. On seeing the value 1 the handler will proceed to print the contents of \$a0 to the console window as an integer value.

This approach of placing a code value in `$v0` and an argument (if any) in `$a0` and invoking the handler provides a uniform way in which to get access to a range of system functions. Other functions supported in SPIM and their respective control codes are provided in Table 1 in the SPIM documentation.

*Example:* A very common task is to print a character string to the screen. For example, suppose we want to print the string “Enter Number : ” to the console window. The following block of code will realize this behavior. The code for printing a character string to the screen is 4. This code value is placed on register `$v0`. The register `$a0` should contain the starting address of the character string. The starting address is marked with the label `str` as shown below. A new instruction, load address (`la`) can be used to place the value of `str` (which is an address) into the register `$a0`. Now when the `syscall` instruction is executed the handler program will examine `$v0` for the code finding a value of 4 which corresponds to the “print string” operation. In this case the contents of `$a0` will be expected to be the starting address of the string and the string will be printed to the console window. The string is terminated in memory by a null termination character. This is how the handler can determine the ending address of the string.

```
        .data
str:    .asciiz "Enter Number : "

        .text
addi $v0, $0, 4    # code for print string
la $a0, str        # $a0 = string address
syscall           # will cause the string to be
                  # printed to the console
```

*Example:* A natural extension to the preceding code sequence is to follow it by a sequence of instructions that reads the value typed into the console by the user. Such a code sequence is shown below. The execution of the `syscall` now causes the value typed into the console window to be

placed in register \$v0 from which the value can be moved to another register or stored in memory as shown.

```

        .data
str:    .asciiz "Enter Number : "
L1:    .word 0x0          # has the same effect as .space 4

        .text
addi $v0, $0, 4          # code for print string
la $a0, str              # $a0 = string address
syscall                  # print string

addi $v0, $0, 5          # code for reading from console
syscall
sw $v0, L2($0)          # store the value read
                        # from the screen into
                        # the data segment
    
```

### Instructions

There are a host of useful instructions which are listed in the SPIM documentation. Some of the more useful instructions and their application are listed here.

Instruction	Meaning
li \$t0, 0x4678	Place the value 0x4678 in register \$t0
la \$v0, L1	If L1 is a label, this instruction will load the value of L1 into register \$v0. This is a useful way of placing addresses into registers
move \$t0, \$t1	This instruction will copy the value in register \$t1 into register \$t0
lui \$t0, 0x4355	Set the upper 16 bits of register \$t0 to the value 0x4355.
li \$t0, 44	Place the value 44 (or equivalently 0x2c) in register \$t0. This is not a native instruction. The assembler will generally translate these into instructions such as addi \$t0, \$0, 44

---

## *Getting Started*

This section provides a few short, complete programs with blocks of code that correspond to operations that are often encountered in the course of writing SPIM programs. Use the SPIM simulator to execute these small programs and help generate an intuition about programming at this level prior to the construction of larger, non-trivial programs. Within the simulator use breakpoints and single step mode (refer to the class tutorial notes) to examine the contents of memory and the registers at various points in the execution of the program to make sure that you understand effect of each instruction.

### **Getting Familiar with the Data Segment**

The simplest place to start is to create a few simple programs with data directives and to assemble these programs. We can examine the data segment and ensure we understand the effect of the data directives. Consider the following two examples.

*Program:* Figure 3 shows a sequence of data directives. Create a text file with this program. From the SPIM simulator load the program (from File menu item) and open the data segment window (from the Window menu item). This window will appear as shown in Figure 3. Study carefully the layout of the data segment window. Each line contains four words. The first column contains the address of the first word shown in that line. For example the address of the word that contains the value 0x00000025 is 0x10010000. Correlate the data directives to the contents of the data segment as shown and verify that the remaining values appearing in the data segment correspond to the values specified by the directives shown in the program.

There are several examples of the use of data segment directives in the web problems associated with the memory module. These programs can be similarly tested and used to generate an understanding of the use of data directives.

### **Getting Familiar with the Text Segment**

Every program should be properly terminated. The correct termination sequence is shown below and should appear at the end of every program.

```
li $v0, 10          # code for program end
syscall
```

```

        .data
here:   .word 0x25, 100
        .asciiz "Sample text"
        .align 2
there:  .space 6
        .byte 85
    
```

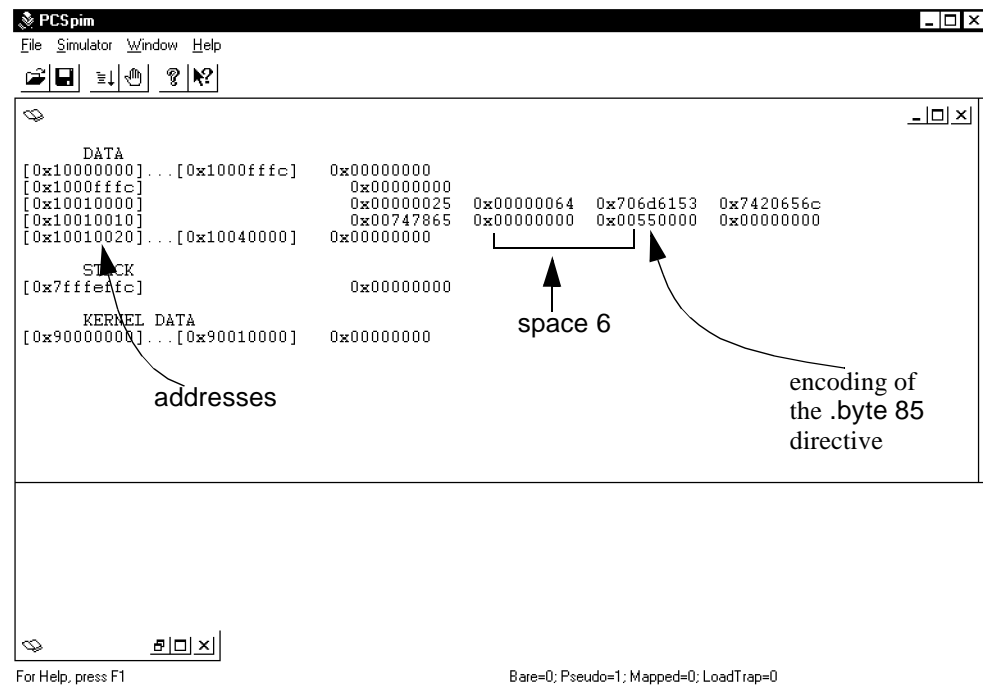


FIGURE 3. An example of data segment usage

*Program:* The sequence of instructions shown in Figure 4 loads two values into registers \$t0 and \$t1 and performs the bit-by-bit logical OR and logical AND operation between the contents of the two registers. Execute this program and examine the contents of the registers (from the Window menu item in

```

.data
L1: .word 0x2345      # some arbitrary value
L2: .word 0x33667    # some arbitrary value

.text
lw $t0, L1($0)      #load the first value
lw $t1, L2($0)      # load the second value
and $t2, $t0, $t1   # compute the bit-by-bit AND
or $t3, $t0, $t1    # compute the bit-by-bit OR
li $v0, 10          # code for program end
syscall
    
```

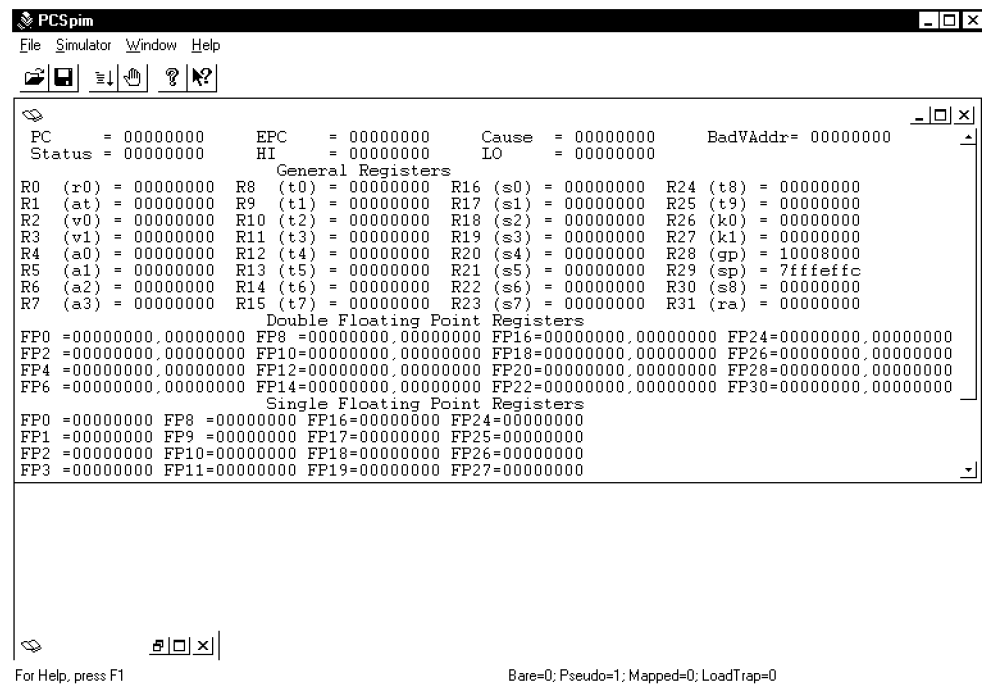


FIGURE 4. The register view in SPIM

SPIM) after the program completes execution. Are the contents of \$t2 and \$t3 what you expect it to be? Are they the same values that you derived manually?

*Program:* This program is another example of the use of logic functions and shift operations. Two values are placed in the data segment and various logical operations are performed on them. Run this program in single step mode and check the contents of each register after each logical operation. Perform the same operations manually and check your answers.

```
.data
label: .word 0x0f000000
label1: .word 0xffffffff
.text

lw $t0, label($0)
lw $t1, label1($0)
and $t2, $t0, $t1    #should produce 0x0f000000 in $t2
andi $t3, $t1, 0xffff #what value will be in $t3?
srl $t2, $t2, 12     # what do we have in $t2 now?

move $t0, $0        #initialize $t0 to 0
lui $t0, 0x0f00     # store a 32-bit constant into $t0
andi $t0, 0x0000

li $v0, 10          # program termination
syscall
```

*Program:* This program constructs a 32-bit constant value 0x78965432 in register \$t4. Run this program checking the contents of the registers before and after program execution. It is also useful to execute the program in single step mode examining the registers after each step.

```
.text
add $t4, $0, $0,    #initialize $t4 to 0
lui $t4, 0x7896     #load 0x7896 into the upper
                   #16 bits of $t4
ori $t4, $t4, 0x5432 # perform the logical OR
                   #between $t0 and the value
                   #0x5432
li $v0, 10          # code for program end
syscall
```

*Program:* The following program prompts the user for a value, doubles the number and prints the result to the screen. After the program has completed execution check the contents of the data segment. From

the code it is apparent that the value of the label `res` is `0x10010014`. Check the contents of memory location `0x10010014` and ensure that the answer is indeed correct.

```
.data
str1: .ascii "Enter the number: "
      .align 2          #move to a word boundary
res:  .space 4         # reserve space to store result

.text
li $v0, 4             # code to print string
la $a0, str1
syscall
li $v0, 5             # code to read integer
syscall
move $t0, $v0        # move the value read to $t0

add $t1, $t0, $t0    # multiply by 2
sw $t1, res($0)     # store result in memory

li $v0, 1             # code to print integer
move $a0, $t1        # the value to be printed in $a0
syscall              # print to the screen
li $v0, 10           # code for program end
syscall
```

*Program:* This program introduces the use of the `lb` instruction. Rather than loading a word from memory the `lb` instruction loads the value stored in the byte at that address. It follows that the addresses can be aligned on a byte address and do not need to be on a word address.

```
.data
value: .word 0x12345678

.text
la $s3, value        # place address in $s3
lw $t0, 0($s3)
lb $t1, 0($s3)       #these instructions load
lb $t2, 1($s3)       #the byte at that address
lb $t3, 2($s3)       #rather than the word
lb $t4, 3($s3)

li $v0, 10           # terminate program
syscall
```



---

### *Writing SPIM Programs*

This chapter has provided numerous small snippets of SPIM code to perform specific operations. The examples are intended to introduce a way of thinking about writing programs at this level and demonstrating how simple operations in Java or C translate into larger sequences (not necessarily more difficult) of instructions at this level. The task of writing SPIM programs is one of coalescing such snippets into larger meaningful components that perform useful computations.

Writing SPIM programs contains a few essential elements.

1. *Design the algorithm:* How is the computation to be performed? Pseudo code with variables is a common way of expressing the algorithm.
2. *Perform register allocation:* Decide which registers will be used to hold the values corresponding to the variables in the algorithm. Remember assembly instructions only manipulate hardware entities such as registers and memory locations.
3. *Perform memory allocation:* Design the layout of the data segment and implement this layout using data directives. Use labels throughout particularly at the start of data structures such as arrays.
4. *Write the program:* Once the preceding three steps are complete the programs almost write themselves!

Write small blocks of code and debug them individually before aggregating them into larger program components. Make use of breakpoints and single step modes within the SPIM simulator. A little patience up front will be returned 10-fold in hours spent debugging your programs.

