Lab #1

Introduction to SPIM

SPIM is a simulator that can run programs for the MIPS Rx000 architectures. The simulator may load and execute assembly language programs. The process which a source file (assembly language) is translated to an executable file contains two stages:

- assembling (implemented by the assembler)
- linking (implemented by the linker)

An executable file must be loaded before the CPU can actually run that program. Figure 1. shows the relation between the processes involved in translation and loading and various types of files they manipulate. The assembler performs the translation of an assembly language module into machine code. A program may have several modules, each of them a part of the program. This is usually the case when you build an application from several files. The output of the assembler is an object module for each source module. Object modules contain machine code. The translation of a module is not complete if the module uses a symbol (a label) that is defined in a different module or is part of a library.

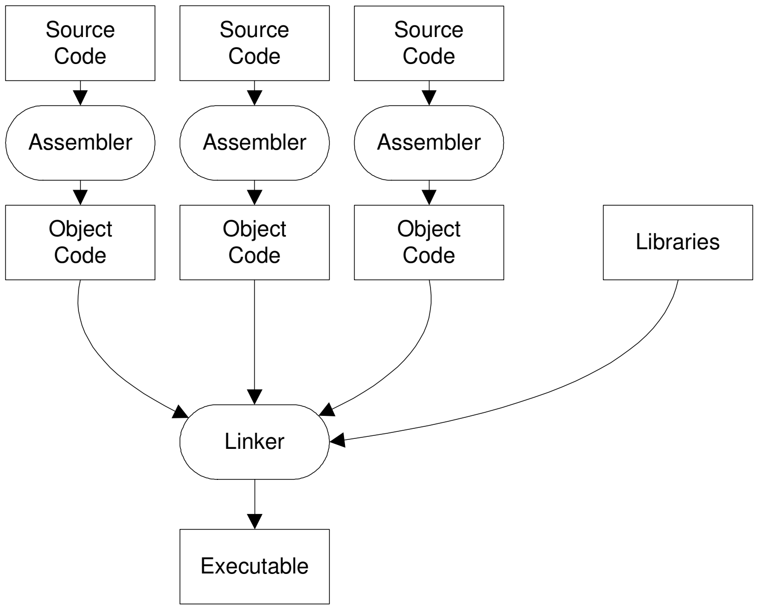


Figure 1. The Translation Process

The linker is program that resolves external references. In other words the linker will match a symbol used in a source module with it's definition found in a different module or in a library. The output of the linker is an executable file.

File suffixes indicate the type of the file. Files that contain assembly code have the suffix ".s' or ".asm". Compilers use the first convention. Files with object programs have the suffix ".o" and executables don't usually have any suffix.

The process of translation in SPIM is transparent to the user. This means that you don't have to deal with an assembler, a linker and a loader as separate programs. Provided you have written a correct assembly language program, the only thing you have to do is to start the simulator and then indicate what program you want to execute. The whole process of translation is hidden.

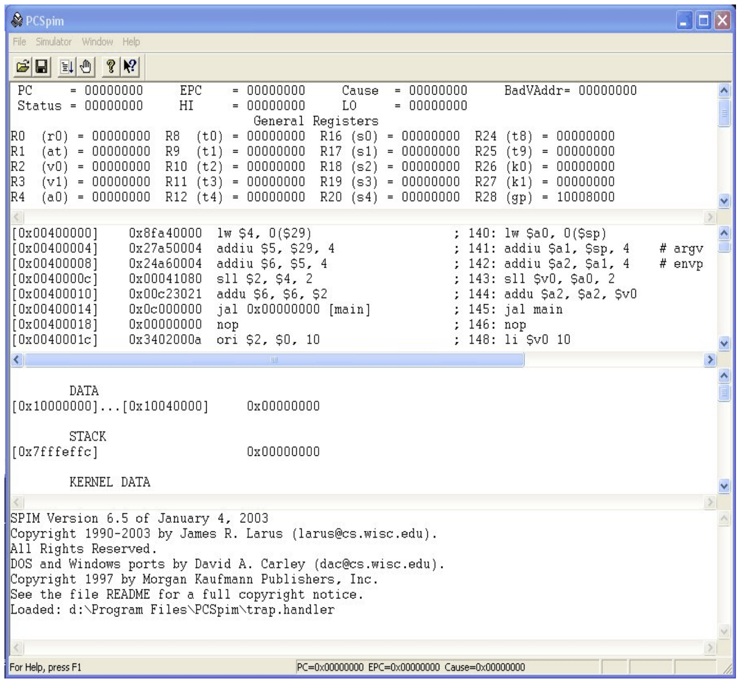
SPIM implements both a simple, terminal-style interface and a visual windowing interface. On Unix, the SPIM program provides the terminal interface and the **xspim** program provides the X window interface. On PCs, the SPIM program provides the console interface and PCSpim provides the Windows interface.

Section 1. First SPIM Program

The first example SPIM program puts bit patterns representing integers into two registers. Then it adds the two patterns together. The screen shots for this example are from a MS Win system. Unix and Linux should be close.

1.1 Start SPIM

First click on the start button => all programs => PCSpim. On windows machines, the opening screen is as below. The screen is divided into four parts:



```
 PC     = 00000000    EPC    = 00000000    Cause  = 00000000    BadVAddr= 00000000
 Status = 00000000    HI     = 00000000    LO     = 00000000
                              General Registers
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000000  R24 (t8) = 00000000
R1  (at) = 00000000  R9  (t1) = 00000000  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 00000000  R10 (t2) = 00000000  R18 (s2) = 00000000  R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 00000000  R19 (s3) = 00000000  R27 (k1) = 00000000
R4  (a0) = 00000000  R12 (t4) = 00000000  R20 (s4) = 00000000  R28 (gp) = 10008000
```

```
[0x00400000]    0x8fa40000  lw $4, 0($29)                 ; 140: lw $a0, 0($sp)
[0x00400004]    0x27a50004  addiu $5, $29, 4              ; 141: addiu $a1, $sp, 4    # argv
[0x00400008]    0x24a60004  addiu $6, $5, 4               ; 142: addiu $a2, $a1, 4    # envp
[0x0040000c]    0x00041080  sll $2, $4, 2                 ; 143: sll $v0, $a0, 2
[0x00400010]    0x00c23021  addu $6, $6, $2               ; 144: addu $a2, $a2, $v0
[0x00400014]    0x0c000000  jal 0x00000000 [main]         ; 145: jal main
[0x00400018]    0x00000000  nop                           ; 146: nop
[0x0040001c]    0x3402000a  ori $2, $0, 10                ; 148: li $v0 10
```

```
        DATA
[0x10000000]...[0x10040000]     0x00000000

        STACK
[0x7fffeffc]                    0x00000000

        KERNEL DATA
```

```
SPIM Version 6.5 of January 4, 2003
Copyright 1990-2003 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
DOS and Windows ports by David A. Carley (dac@cs.wisc.edu).
Copyright 1997 by Morgan Kaufmann Publishers, Inc.
See the file README for a full copyright notice.
Loaded: d:\Program Files\PCSpim\trap.handler
```

```
PC=0x00000000  EPC=0x00000000  Cause=0x00000000
```

Figure 2 SPIM Windows

1. Register Display: This shows the contents (bit patterns in hex) of all 32 general purpose registers, the floating point registers, and a few others.
2. Text Display: This shows the assembly language program source, the machine instructions (bit patterns in hex) they correspond to, and the addresses of their memory locations.
3. Data and Stack Display: This shows the sections of MIPS memory that hold ordinary data and data which has been pushed onto a stack.
4. SPIM Messages: This shows messages from the simulator (often error messages).

Messages from the simulated computer appear in the console window when an assembly program that is running (in simulation) writes to the (simulated) monitor. If a real MIPS computer were running you would see the same messages on a real monitor.

Messages from the simulator are anything the simulator needs to write to the user of the simulator. These are error messages, prompts, and reports.

Now that the simulator is running you need to assemble and load a program. Depending on the settings of the simulator, there already may be some machine instructions in simulated memory. These instructions assist in running your program. If you start the simulator from the Simulator menu this code will run, but it will be caught in an infinite loop. To stop it, click on Simulator; Break.

1.2 Editing A Program

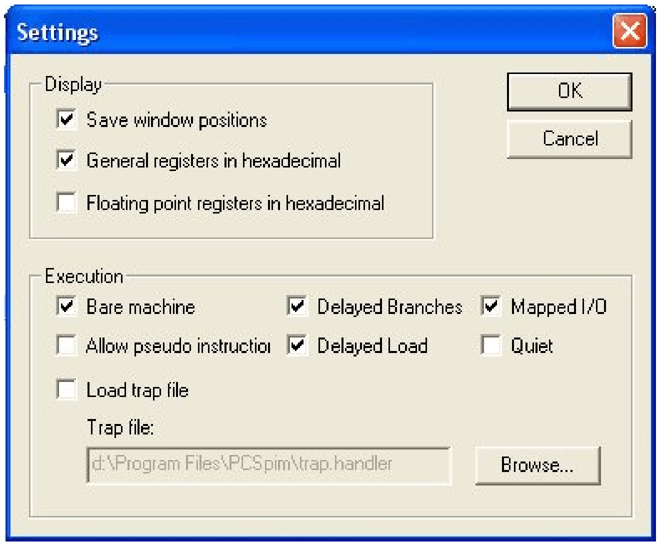
A source file (in assembly language or in any programming language) is the text file containing programming language statements created (usually) by a human programmer. Open **Notepad** to create a file called addup.asm. Type in the following program

```
## Program to add two plus three
        .text
        .globl  main

main:
        ori     $8,$0,0x2       # put two's comp. two into register 8
        ori     $9,$0,0x3       # put two's comp. three into register 9
        addu    $10,$8,$9       # add register 8 and 9, put result in 10

## End of file
```

The first "#" of the first line is in column one. The character "#" starts a comment; everything on the line from "#" to the right is ignored. Sometimes I use two in a row for emphasis, but only one is needed. Each of the three lines following main: corresponds to one machine instruction.

1.3 Setting Up SPIM

Each MIPS machine instruction is 32 bits (four bytes) long. The three lines after main: call for three machine instructions. The remaining lines consist of information for the assembler and comments (for the human). For this first program some SPIM options must be set. In the menu bar, click on Simulator then Settings to get the settings dialog. Select the options as in Figure 3. These settings simulate a bare machine with no user conveniences. Later we will include the conveniences.


Settings

Display
- [✓] Save window positions
- [✓] General registers in hexadecimal
- [ ] Floating point registers in hexadecimal

Execution
- [✓] Bare machine
- [✓] Delayed Branches
- [✓] Mapped I/O
- [ ] Allow pseudo instructio
- [✓] Delayed Load
- [ ] Quiet
- [ ] Load trap file

Trap file: d:\Program Files\PCSpim\trap.handler

Figure 3 SPIM Setup window

1.4 Loading the source program

Modern computers boot up to a user-friendly state. Usually there is some firmware (permanent machine code in EEPROM) in a special section of the address space. This starts running on power-up and loads an operating system. SPIM can simulate some basic firmware, but we have turned off that option.

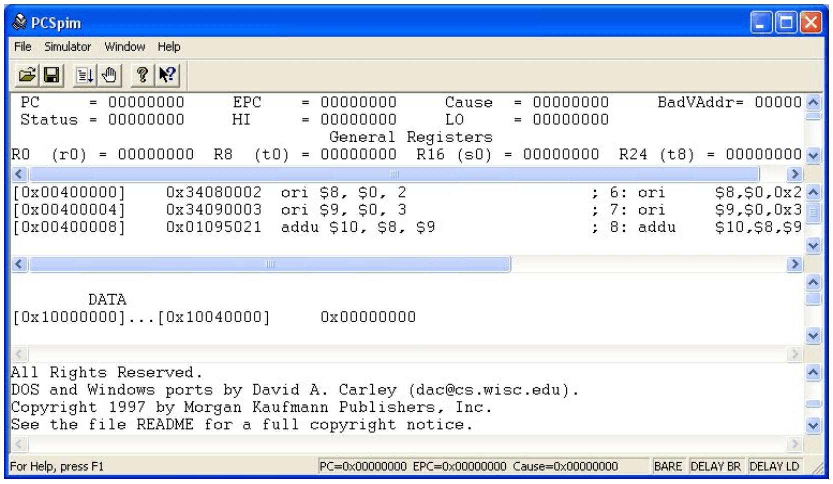
Load the program into the SPIM simulator by clicking File then Open. Click on the name (addup.asm) of your source file. You may have to navigate through your directories using the file dialog box.

If there are mistakes in addup.asm, SPIM's message display panel shows the error messages. Use your editor to correct the mistakes, save the file then re-open the file in SPIM.

1.5 Assembling the program

Loading the source file into SPIM does two things:

(1) The file is assembled into machine instructions, and
(2) The instructions are loaded into SPIM's memory. The text display shows the result.



```
 PC      = 00000000     EPC     = 00000000     Cause    = 00000000     BadVAddr= 00000
 Status  = 00000000     HI      = 00000000     LO       = 00000000
                                 General Registers
R0   (r0) = 00000000  R8   (t0) = 00000000  R16 (s0) = 00000000  R24 (t8) = 00000000

[0x00400000]    0x34080002  ori $8, $0, 2                   ; 6: ori     $8,$0,0x2
[0x00400004]    0x34090003  ori $9, $0, 3                   ; 7: ori     $9,$0,0x3
[0x00400008]    0x01095021  addu $10, $8, $9                ; 8: addu    $10,$8,$9

       DATA
[0x10000000]...[0x10040000]     0x00000000

All Rights Reserved.
DOS and Windows ports by David A. Carley (dac@cs.wisc.edu).
Copyright 1997 by Morgan Kaufmann Publishers, Inc.
See the file README for a full copyright notice.

For Help, press F1          PC=0x00000000  EPC=0x00000000  Cause=0x00000000    BARE  DELAY BR  DELAY LD
```

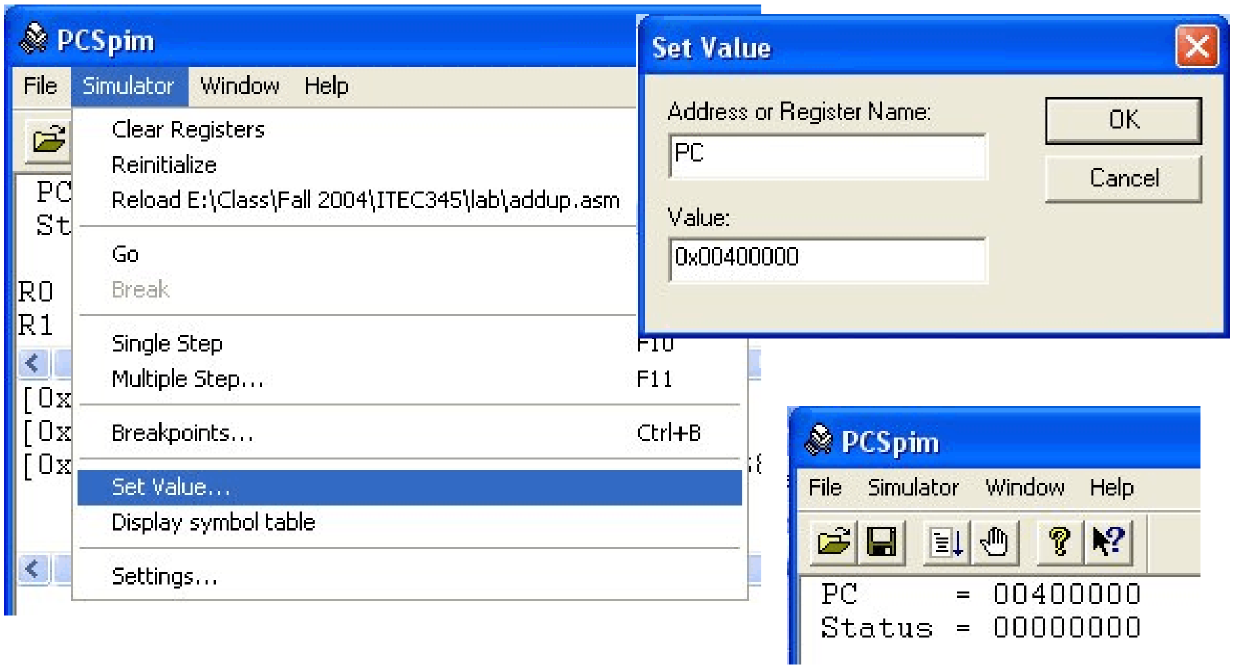
Figure 4. Assembler Result

The text display is the second window from the top. You should see some of the source file in it and the machine instructions they assembled into. The leftmost column are addresses in simulated memory.

1.6 Setting the PC

The program counter is the part of the processor that contains the address of the current machine instruction. (Actually, it contains the address of the first of the four bytes that make up the current instruction.) In the register display (top window) you see that the PC starts out at zero. This must be changed to 0x00400000, the address of the first instruction. To do this, select (click on) Simulator; Set Value in the menu bar.

In the set value dialog, type PC in the top text box and 0x00400000 in the bottom text box. Click on OK and the PC (in the register display) should change.



Address or Register Name: PC

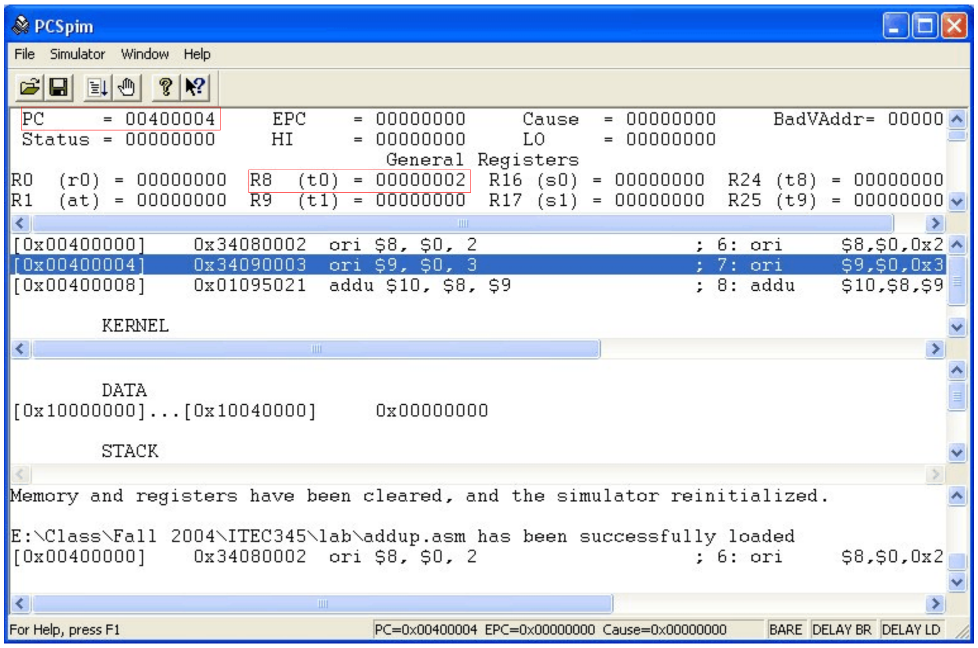
Value: 0x00400000

```
PC     = 00400000
Status = 00000000
```

Figure 6. Set the PC value

1.7 Running the program

Push F10 to execute one instruction. The first instruction executes, loading register eight with a 2 (see the register display). The PC advances to the next instruction 0x00400004 and the message display window shows the instruction that just executed.



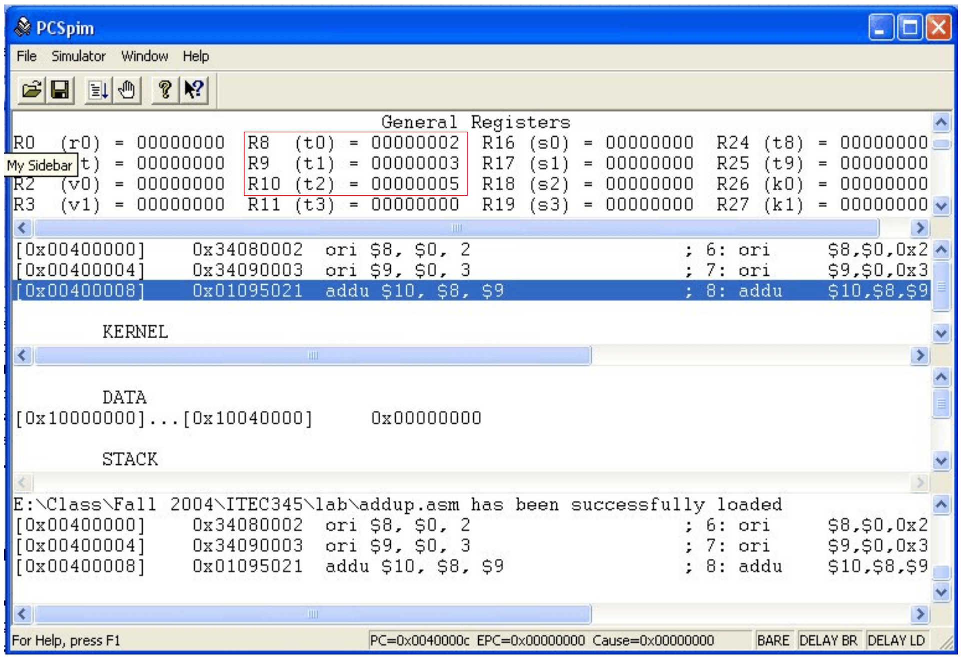
```
PC       = 00400004     EPC      = 00000000     Cause    = 00000000     BadVAddr= 00000
Status   = 00000000     HI       = 00000000     LO       = 00000000
                                  General Registers
R0  (r0) = 00000000  R8  (t0) = 00000002  R16 (s0) = 00000000  R24 (t8) = 00000000
R1  (at) = 00000000  R9  (t1) = 00000000  R17 (s1) = 00000000  R25 (t9) = 00000000

[0x00400000]    0x34080002  ori $8, $0, 2                   ; 6: ori    $8,$0,0x2
[0x00400004]    0x34090003  ori $9, $0, 3                   ; 7: ori    $9,$0,0x3
[0x00400008]    0x01095021  addu $10, $8, $9                ; 8: addu   $10,$8,$9

        KERNEL

        DATA
[0x10000000]...[0x10040000]     0x00000000

        STACK

Memory and registers have been cleared, and the simulator reinitialized.

E:\Class\Fall 2004\ITEC345\lab\addup.asm has been successfully loaded
[0x00400000]    0x34080002  ori $8, $0, 2                   ; 6: ori    $8,$0,0x2

For Help, press F1    PC=0x00400004  EPC=0x00000000  Cause=0x00000000    BARE  DELAY BR  DELAY LD
```

Figure 7. Register Content

Push F10 two more times to execute the remaining instructions. Each instruction is 32 bits (four bytes) long, so the PC changes by four each time. After the third instruction, register 8 will have the sum of two plus three.

1.8 Program's Result

The bit patterns for these small integers are easy to figure out. You may have to use the slider on the register display to see register ten.



```
                           General Registers
R0  (r0) = 00000000  R8  (t0) = 00000002  R16 (s0) = 00000000  R24 (t8) = 00000000
R1  (at) = 00000000  R9  (t1) = 00000003  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 00000000  R10 (t2) = 00000005  R18 (s2) = 00000000  R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 00000000  R19 (s3) = 00000000  R27 (k1) = 00000000

[0x00400000]    0x34080002  ori $8, $0, 2             ; 6: ori    $8,$0,0x2
[0x00400004]    0x34090003  ori $9, $0, 3             ; 7: ori    $9,$0,0x3
[0x00400008]    0x01095021  addu $10, $8, $9          ; 8: addu   $10,$8,$9

        KERNEL

        DATA
[0x10000000]...[0x10040000]     0x00000000

        STACK

E:\Class\Fall 2004\ITEC345\lab\addup.asm has been successfully loaded
[0x00400000]    0x34080002  ori $8, $0, 2             ; 6: ori    $8,$0,0x2
[0x00400004]    0x34090003  ori $9, $0, 3             ; 7: ori    $9,$0,0x3
[0x00400008]    0x01095021  addu $10, $8, $9          ; 8: addu   $10,$8,$9

PC=0x0040000c  EPC=0x00000000  Cause=0x00000000
```

Figure 8. Program's Result

If you push F10 again, the PC will point at a word in memory that contains bits not intended to be a machine instruction. However the simulator will try to execute those bits. A real processor would "crash" at this point. (This is sometimes called "falling off the end of a program"). The simulator prints an error message in the bottom panel. You are done and exit SPIM.

1.9 Program Explanation

There are various ways for a program executing on a real machine to return control to the operating system. But we have no OS, so for now we will single step instructions. Hopefully you are wondering how the program works. Here it is again:

```
## Program to add two plus three
        .text
        .globl  main

main:
        ori     $8,$0,0x2       # put two's comp. two into register 8
        ori     $9,$0,0x3       # put two's comp. three into register 9
        addu    $10,$8,$9       # add register 8 and 9, put result in 10
## End of file
```

The first line of the program is a comment. It is ignored by the assembler and results in no machine instructions.

.text is a directive. A directive is a statement that tells the assembler something about what the programmer wants, but does not itself result in any machine instructions. This directive tells the assembler that the following lines are ".text" -- source code for the program.

.globl main is another directive. It says that the identifier main will be used outside of this source file (that is, used "globally") as the label of a particular location in main memory.

Blank lines are ignored. The line main: defines a symbolic address (sometimes called a statement label). A symbolic address is a symbol (an identifier) that is the source code name for a location in memory. In this program, main stands for the address of the first machine instruction (which turns out to be 0x00400000). Using a symbolic address is much easier than using a numerical address. With a symbolic address, the programmer refers to memory locations by name and lets the assembler figure out the numerical address.

The symbol main is global. This means that several source files can use the symbol main to refer to the same location in storage. (However, SPIM does not use this feature. All our programs will be contained in a single source file.)

1.10 Questions

A. What is a source file?
B. What is a register?
C. What character, in SPIM assembly language, starts a comment?
D. How many bits are there in each MIPS machine instruction?
E. When you open a source file from the File menu of SPIM, what two things happen?
F. What is the program counter?
G. Say that you push F10 to execute one instruction. What amount is added to the program counter?
H. What is a directive, such as the directive .text?
I. What is a symbolic address?
J. Where was the first machine instruction placed in memory?
K. What machine instruction (bit pattern) did your first instruction (ori $8,$0,0x2) assemble into?

Section 2. 2nd Program

2.1 Another Program

Type in the following program using Notepad and save it as lab12.asm

```
      .data 0x10000000
msg1: .asciiz "Please enter an integer number: "
      .text
      .globl main
# Inside main there are some calls (syscall) which will change the
# value in $31 ($ra) which initially contains the return address
# from main. This needs to be saved.
main: addu $s0, $ra, $0 # save $31 in $16
      li $v0, 4 # system call for print_str
      la $a0, msg1 # address of string to print
      syscall
# now get an integer from the user
      li $v0, 5 # system call for read_int
      syscall # the integer placed in $v0
# do some computation here with the number
      addu $t0, $v0, $0 # move the number in $t0
      sll $t0, $t0, 2 # last digit of your SSN instead of 2
# print the result
      li $v0, 1 # system call for print_int
      addu $a0, $t0, $0 # move number to print in $a0
      syscall
# restore now the return address in $ra and return from main
      addu $ra, $0, $s0 # return address back in $31
      jr $ra # return from main
```

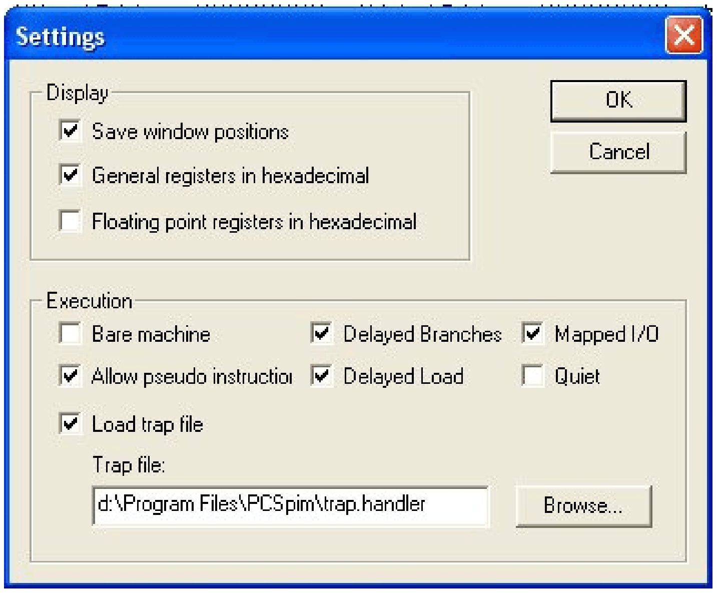
Before you continue make sure you have entered the last digit of your SSN instead of 2 in the instruction sll $t0, $t0, 2

2.2 Start SPIM

Start the SPIM simulator. Start > All Programs > PCSpim

2.3 Setup SPIM

Goto the Simulator > Setting dialog box. Make sure you set it up as shown


Settings

Display
- [✓] Save window positions
- [✓] General registers in hexadecimal
- [ ] Floating point registers in hexadecimal

Execution
- [ ] Bare machine
- [✓] Delayed Branches
- [✓] Mapped I/O
- [✓] Allow pseudo instructioı
- [✓] Delayed Load
- [ ] Quiet
- [✓] Load trap file

Trap file: d:\Program Files\PCSpim\trap.handler

Figure 9 SPIM setup for Lab12.asm

2.4 Load Program

Load the lab12.asm. You can use the File > Open sequence or click on the File Open button. If you make any mistake, check your program again.

2.5 Run Program

Run the program by pressing the F5 button or the Run button. You will be prompted to enter a PC. It should show the value 0x00400000 which is the correct PC value. Just click OK to run the program. The Console window should come to the foreground and it should display a message:

```
   Please enter an integer number:
```

Type in an integer number and press Enter. The program will show a number as the result of a calculation. To run the program again, press F5 again.

2.6 Do Some Experiment

You now try to figure out what program lab12.asm does. Run it several times with various input data. Use both positive and negative integers. Fill out the following table:

Input Number	Output Number

After you are done. Exit SPIM.

2.7 Figure out the relationship

What is the formula that describes the relation between the output and the input?

Section 3. Memory, Registers and Breakpoints

Using the simulator you will peek into the memory and into various general purpose registers. You will also execute a program step by step. Stepping may be very useful for debugging. Setting breakpoints in a program is another valuable debugging aide: you will be playing with these too.

3.1 Load Program

Start SPIM and load lab12.asm

3.2 Show Global symbols

Go to Simulator > Display Symbol Table. On the message window, you will see a listing of all global symbols. Global symbols are those that are preceded by the assembler directive '.globl'. For each symbol the address in memory where the labeled instruction is stored, is also printed.

Symbol	Address

Exit Simulator.

3.3 Modify lab12.asm

Modify lab12.asm as follows: replace the first line after the line labeled 'main' with a line that reads

```
   label1:  li $v0, 4   # system call for print_int
```

Save the program as lab13.asm. The only difference between the two programs is the label 'label1'

3.4 Load lab13.asm

Start the SPIM simulator, load the program lab13.asm and print the list of global symbols.

Symbol	Address

As you can see there is no difference between the listing you obtain at this step and the one at Step 3.2. The reason is that ‘label1’ is a local symbol. Local symbols are visible only within the module in which they are defined. A global symbol is visible inside and outside the module where it is defined. A global symbol can therefore be referenced from other modules. Exit Simulator.

3.5 Memory Content

We now know where the program is stored in memory. It is the address returned by Display Symbol Table for the symbol ‘main’. To see what is stored in memory starting with that address, look at the Text Display window (uses the vertical scroll bar). The window shows lines that contains (in this order):

- the address in memory
- the hexadecimal representation of the instruction
- the native representation of instructions (no symbolic names for registers)
- the textual instruction as it appears in the source file

3.6 Program in the Memory

Starting with the address of the symbol ‘__start’ and fill the table below.

Label	Address	Native Instruction	Source Instruction

3.7 Tracing a program

The Step (F10) command allows the user to execute a program step by step. The user can then see how a specific instruction has modified registers, memory, etc. Use step to fill out the following table

Label	Address	Native Instruction	Source Instruction

Why does the table differ than step 3.6?

3.8 Reload lab12.asm

Load again lab12.asm. You will get an error message indicating that some label(s) have multiple definitions. This happens because the program lab12.asm has already been loaded. If there is a need to reload a program, then the way to do it is

Simulator > Reinitialize
then Load lab12.asm

Reinitialize will clear the memory and the registers.

3.9 **Breakpoint**

Let’s assume you don’t want to step through the program. Instead, you want to stop every time right before some instruction is executed. This allows you to see what is in memory or in registers right before the instruction is executed.

Set a breakpoint at the second syscall in your program (find the address from step 3.7). To set a break point:

Simulator > Breakpoint or CTRL-B

A dialog box will open where you can add breakpoint addresses. Type the address in the address box and click on the Add button. You can add more than one breakpoint and you can also delete an existing breakpoint. When you are done click on the Close button.

Now you can run the program, up to the first breakpoint encountered (there is only one at this time). Use the F5 button. When a breakpoint is reached, a dialog box will pop up and ask whether you want to continue. Select No, because we want to stop execution and fill the ‘Before the syscall’ column of the following table

Register Number	Register Name	Before Syscall	After Syscall	Change
0	zero			
1	$at			
2	$v0			
3	$v1			
4	$a0			
5	$a1			
6	$a2			
7	$a3			
8	$t0			
9	$t1			
10	$t2			
11	$t3			

12	$t4			
13	$t5			
14	$t6			
15	$t7			
16	$s0			
17	$s1			
18	$s2			
19	$s3			
20	$s4			
21	$s5			
22	$s6			
23	$s7			
24	$t8			
25	$t9			
26	$k0			
27	$k1			
28	$gp			
29	$sp			
30	$fp			
31	$ra			

3.10 Next

Press step (F10) to have the syscall executed. Before you can do anything else you must supply an integer. This happens because the program executes a syscall, a call to a system function, in this
case one that reads an integer from the keyboard. Fill out the 'After the syscall' column of the above table. In the column 'Changed', mark with a star registers that have changed.

Some registers have changed during the syscall execution. Can you assume that syscall uses only these registers? Explain.