DHA Suffa University
Department of Computer Science
CS -203L Computer Organization & Assembly Language
Fall 2017
Lab # 1

Lab 1 shows how to run the SPIM simulator. A small MIPS program is used as an example.

Starting SPIM

MIPS processors have 32 general purpose registers, each holding 32 bits. They also have registers other than general purpose ones.

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 system, any Windows OS will be similar.

To start SPIM click on its icon, QtSPIM by first clicking on the start button.

The SPIM user interface

On windows machines, the opening screen is as below. The screen is divided into four parts:

Register Display: This shows the contents (bit patterns in hex) of all 32 general purpose registers, the floating point registers, and a few others.

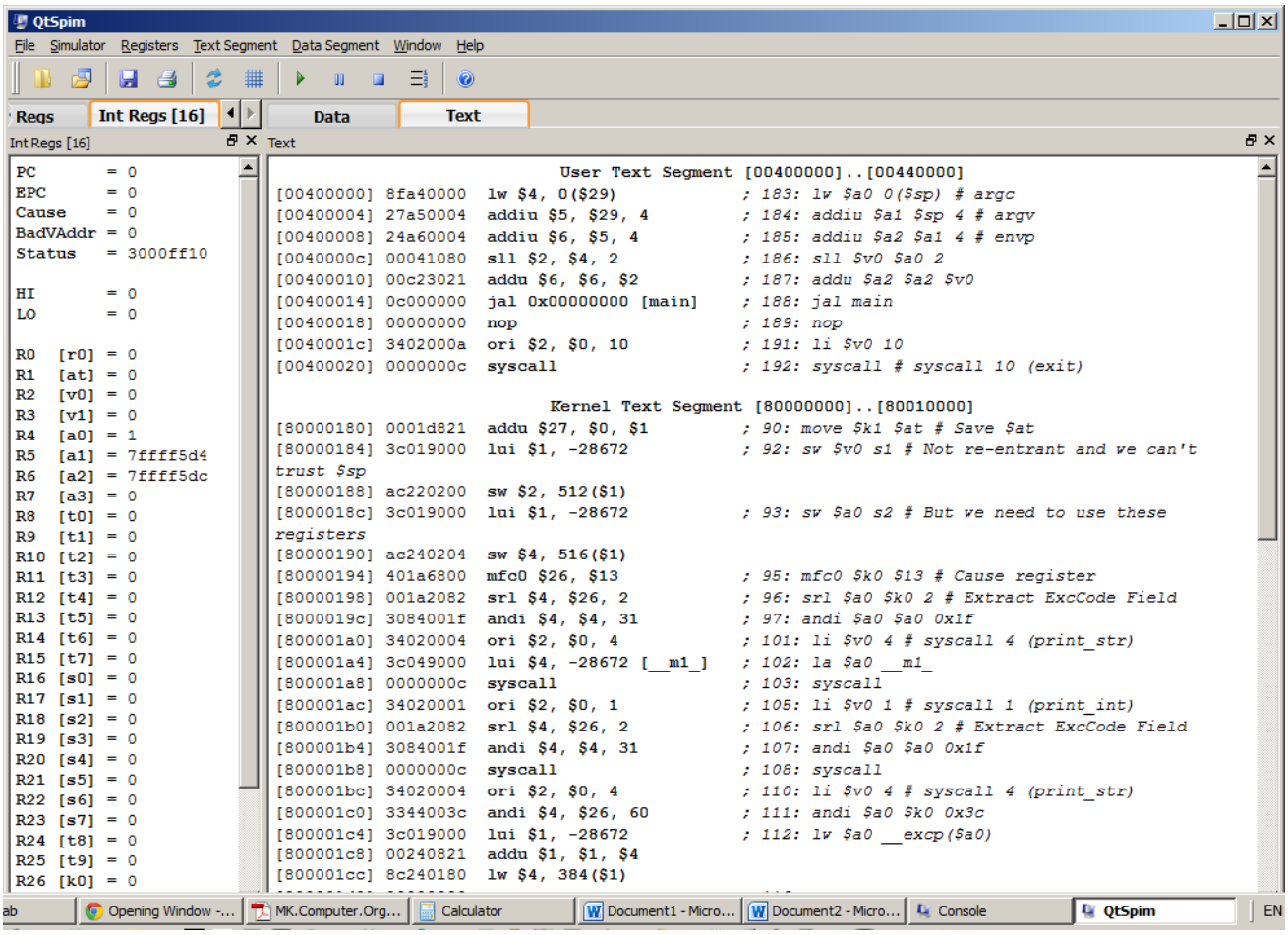
Text Display: This shows the assembly language program source, the machine instructions (bit patterns in hex) they correspond to, and the addresses of their memory locations.

Data and Stack Display: This shows the sections of MIPS memory that hold ordinary data and data which has been pushed onto a stack.

SPIM Messages: This shows messages from the simulator (often error messages).

Text is the name for machine language bit patterns intended for eventual execution. The word "program" is often ambiguous, so "text" is used. "Text" is a machine language program that may be executed by a process.

Character output from the simulated computer is in the SPIM console window, shown as an icon in the picture.


QtSpim
File Simulator Registers Text Segment Data Segment Window Help
Regs
Int Regs [16]
Data
Text
Int Regs [16]
PC = 0
EPC = 0
Cause = 0
BadVAddr = 0
Status = 3000ff10
HI = 0
LO = 0
R0 [r0] = 0
R1 [at] = 0
R2 [v0] = 0
R3 [v1] = 0
R4 [a0] = 1
R5 [a1] = 7ffff5d4
R6 [a2] = 7ffff5dc
R7 [a3] = 0
R8 [t0] = 0
R9 [t1] = 0
R10 [t2] = 0
R11 [t3] = 0
R12 [t4] = 0
R13 [t5] = 0
R14 [t6] = 0
R15 [t7] = 0
R16 [s0] = 0
R17 [s1] = 0
R18 [s2] = 0
R19 [s3] = 0
R20 [s4] = 0
R21 [s5] = 0
R22 [s6] = 0
R23 [s7] = 0
R24 [t8] = 0
R25 [t9] = 0
R26 [k0] = 0
Text
User Text Segment [00400000]..[00440000]
[00400000] 8fa40000 lw $4, 0($29) ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004 addiu $5, $29, 4 ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004 addiu $6, $5, 4 ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080 sll $2, $4, 2 ; 186: sll $v0 $a0 2
[00400010] 00c23021 addu $6, $6, $2 ; 187: addu $a2 $a2 $v0
[00400014] 0c000000 jal 0x00000000 [main] ; 188: jal main
[00400018] 00000000 nop ; 189: nop
[0040001c] 3402000a ori $2, $0, 10 ; 191: li $v0 10
[00400020] 0000000c syscall ; 192: syscall # syscall 10 (exit)
Kernel Text Segment [80000000]..[80010000]
[80000180] 0001d821 addu $27, $0, $1 ; 90: move $k1 $at # Save $at
[80000184] 3c019000 lui $1, -28672 ; 92: sw $v0 s1 # Not re-entrant and we can't
trust $sp
[80000188] ac220200 sw $2, 512($1)
[8000018c] 3c019000 lui $1, -28672 ; 93: sw $a0 s2 # But we need to use these
registers
[80000190] ac240204 sw $4, 516($1)
[80000194] 401a6800 mfc0 $26, $13 ; 95: mfc0 $k0 $13 # Cause register
[80000198] 001a2082 srl $4, $26, 2 ; 96: srl $a0 $k0 2 # Extract ExcCode Field
[8000019c] 3084001f andi $4, $4, 31 ; 97: andi $a0 $a0 0x1f
[800001a0] 34020004 ori $2, $0, 4 ; 101: li $v0 4 # syscall 4 (print_str)
[800001a4] 3c049000 lui $4, -28672 [__m1_] ; 102: la $a0 __m1_
[800001a8] 0000000c syscall ; 103: syscall
[800001ac] 34020001 ori $2, $0, 1 ; 105: li $v0 1 # syscall 1 (print_int)
[800001b0] 001a2082 srl $4, $26, 2 ; 106: srl $a0 $k0 2 # Extract ExcCode Field
[800001b4] 3084001f andi $4, $4, 31 ; 107: andi $a0 $a0 0x1f
[800001b8] 0000000c syscall ; 108: syscall
[800001bc] 34020004 ori $2, $0, 4 ; 110: li $v0 4 # syscall 4 (print_str)
[800001c0] 3344003c andi $4, $26, 60 ; 111: andi $a0 $k0 0x3c
[800001c4] 3c019000 lui $1, -28672 ; 112: lw $a0 __excp($a0)
[800001c8] 00240821 addu $1, $1, $4
[800001cc] 8c240180 lw $4, 384($1)
Opening Window -...
MK.Computer.Org...
Calculator
Document1 - Micro...
Document2 - Micro...
Console
QtSpim
EN

Writing an assembly source program

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.

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. An editor like Notepad will work. You will probably want to use a better editor, but as a common ground I'll use Notepad. (I like the freeware Crimson Editor by Ingyu Kang, but any text editor will be fine). Use whatever editor you use for your usual programming language.

```
## Program to add two values
        .text
        .globl main
        main:
        li $t1, 5 #loading an integer value 5 into register $t1.
        li $t2, 7 #loading an integer value 7 into register $t2.
        add $t0, $t1, $t2 #adding values of $t1 & $t2, and putting the result in $t0.
        li $v0, 1
        move $a0, $t0 #copying the value of register of $t2 into $a0.
        syscall
        li $v0, 10 #exit code
        syscall
## 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.

Assembling and loading a 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.

File Open Selection

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

If there are mistakes in addup.s, 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.

Running a 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.

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

Explanation of the Program

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.

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.)
Assembly Language Statement
The layout of a machine instruction is part of the architecture of a processor chip. Without knowing the layout you can't tell what the instruction means. Even if you know the layout, it is hard to remember what the patterns mean and hard to write machine instructions.
A statement in pure assembly language corresponds to one machine instruction. Assembly language is much easier to write than machine language. Here is the previous machine instruction and the assembly language that it corresponds to:
machine instruction assembly language statement
0000 0001 0010 1011 1000 0000 0010 0000 add $t0, $t1, $t2
The instruction means: add the integers in registers $t1 and $t2 and put the result in register $t0. To create the machine instruction from the assembly language statement a translation program called an assembler is used.
Humans find assembly language much easier to use than machine language for many reasons.
It is hard for humans to keep track of those ones and zeros.
By using symbols programmers gain flexibility in describing the computation.
Task:
Solve the following expression using Assembly Language:
(i) (9-8)*12+(16-15)-(5*2)

(ii) ((52*8)*12+8)-(18+116)+(15*2)

(iii) ((32+52)+58*3)+57