

CSC 281 System Internals and Assembly Language, Assignment 1, Fall 2005

Goal: To learn the SPIM interface and basic assembly language constructs such as input/output and loops, you will modify a short program.

Step 0. Load SPIM. It can be found under Q:\InstructorFiles\Chase_Gene\assembly\pcspim.exe. You will also need to load the trap handler file. You don't need to know why at this time, but you have to do this to make it run properly. Here's how: In SPIM, under the Simulator menu option, select Settings. Click on Load Trap File and Browse. The file `trap.handler` is in the same directory. To see it, when you browse make sure you are selecting all files, not just ones with specific extensions.

Step 1. In SPIM, load the program Q:\InstructorFiles\Chase_Gene\assembly\demo1.s using the File menu option and selecting Open. Run it using the Simulator menu option and selecting Go. If everything is loaded correctly, including the trap handler, it should prompt you to start at address 0x00400000. Select OK for that. The program will prompt you to enter an integer, and then it will print out the first four multiples of that integer.

An aside for Dr. Chase's personal preferences. PCspim.zip is version 7.1 of PCspim, dated January 2005. It has these advantages over version 6.5: fewer bugs, and the console window stays open, so that you don't have to open it each time from the menu option labeled Window. It has these disadvantages: The help file is corrupted (although it is not much help anyway except for one thing that I make clear below), and sadly you cannot resize the windows that interest you the most to fill up most of your screen. The bugs that I know of do not affect anything we do in this course, so you are welcome to install PCspim-old.zip (version 6.5 or before) if the non-resizable windows bother you as they do me. Google for PCspim if you wish to download a Linux version.

Step 2. Now take some time to familiarize yourself with the `demo1.s` program that you just ran. The full text of it is on Page 3 below. Notice the following features of the code, since you will use in your assignments as well:

1. Comments, including filename, author, date, brief specification, high-level pseudocode specification, a list of which registers are used for what purposes, and comments in the code itself. The code is split into blocks that fit the structure of the code, and each block has a short comment. Some lines also have comments on the right, but generally you should not put these in. I just did so since you are beginners, but once you understand the basics, you will find such comments more annoying than helpful. The comments above each block of code, though, are essential.
2. The code starts with a data section where you put strings and variables. This program has strings, but no variables. If a string requires more than one line, use `.ascii` on each line except on the last line. When you say `.asciiz` (with a 'z') it tells the compiler to put a null-character after the string. This is how the operating system knows where the string ends. Note how carriage returns are stored in strings as `\n`, as in the Java or C or C++ programming language. For tabs in strings, use `\t`.
3. Next is the text section, containing program code. It uses "main:" as its label. Labels always begin in the first column. Your program will always begin at the label `main`. The operating system expects it.
4. The following MIPS assembler commands are used. Here are their meanings in English:

<code>addi registerOut, registerIn, number</code>	—add the constant number and registerIn, and put the result in registerOut
<code>la register, label</code>	—put the address of the label in the register
<code>syscall</code>	—do input or output as specified (see point 5 below)
<code>bgt register1, register2, label</code>	—if register1 value is greater than register2, go to label
<code>mul reg1, reg2, reg3</code>	—multiply reg2 * reg3 and put result in reg1
<code>add reg1, reg2, reg3</code>	—add reg2 + reg3 and put result in reg1
<code>j label</code>	—jump (go to) label
<code>jr register</code>	—jump to address in register
5. A programmer may not do input/output directly, but must request the service from the operating system using a system call (`syscall`). One selects a specific system call by putting integer codes in prespecified registers. For example, you print a string by putting 4 in register \$2 and the string address in register \$4. You know

the string address since the string is labeled, and the label tells you the address. For a full list of these system services, see page A-44 of your textbook linked from the course website. You do not need to memorize them because I will provide them for all exams even though they are not on the green summary sheet for all exams. For now, here are three worth knowing about at least:

- To print an integer, put the number 1 in \$v0, put the integer in \$a0, and do syscall
- To print a string, put the number 4 in \$v0, put the string address in \$a0, and do syscall
- To read an integer, put the number 5 in \$v0, do syscall, and the integer will be in \$v0

Syscall is not a MIPS assembly language command, but what we will come to call a “pseudo-operation.” It just tells the operating system to take over. The operating system (or kernel) is a program that takes care of input/output for you. Whenever you do a syscall, the operating system looks in register \$v0 to see what system service is desired, then it carries it out (using the data you placed in \$a0 if necessary), and then returns control to your program.

6. For-loops must be programmed at a low level, since there is no FOR instruction. Instead, one keeps the counter value and the final value in selected registers, and then tests after each iteration to see whether the counter has attained the final value yet. Other high-level constructs must also be built from scratch. To keep your assembly code simple, do not use complicated branching structures (sometimes called “spaghetti code”); implement only logic like FOR, REPEAT ... UNTIL, WHILE, SELECT ... CASE.
7. Page 4 shows the text for demo1a.s. If you load and run this, you will find that it runs identically to demo1.s. There are two differences. Names of registers are used instead of numbers of registers. The names and numbers of registers are listed on page A-24 linked from the course's website, and also on your green MIPS summary card for exams. For example, instead of using \$2, demo1a.s uses \$v0. The other difference is that the comments to the right of some of the lines of code have been removed. demo1a.s is a model of good style. **You should use named registers and comments that do not simply repeat the code.** Repeat! **Your homework should use demo1a.s NOT demo1.s as your model!**

Step 3. This time step through the program to see that it operates as you would expect. To do so, you should first reload the demo1a.s program using the Simulator menu option and selecting Reload. To step through the code, you can use the Simulator menu option and select Single Step, or you can use function key F10. As you run it, watch the two SPIM windows named General Registers and Messages. Make the Messages window as wide as possible so you can see not only the code that is run by the simulator, but your source code.

As you step through the program you will see four columns of data in the Messages window.

1: address of the current instruction 2: instruction in hexadecimal 3: current instruction 4: line number and source

When you first start stepping through, you are actually running operating system code, not your own source code. The sixth instruction is **jal main**. At this point, the simulator jumps to the label “main,” and is now ready to start your program.

You should also notice changes that are happening in the general registers. For example, after the instruction **addi \$2, \$0, 4** you should notice that register 2 contains the number 4.

Occasionally the current instruction differs from your source code. For example, the **la** instruction, to load an address of the prompt string is actually implemented using an **lui** instruction. More about this later.

Run the program a few times until you feel comfortable with what is happening throughout. Do you see how I/O works? Do you see where the multiples are being created, and which registers are keeping the user-entered number and the multiplier?

Step 4. Modify the program. You may use any ascii text editor, such Super Notetab (recommended because

Microsoft Notepad likes to add “.txt” to the end of files even if they already end in “.s”). You can find Super Notetab under the lab menu Messiah Programs | Computer Classes. Load the program into the text editor and save it on your own disk. Now might be a good time to test whether you can load and run your copy in SPIM. Then modify the demo1a.s program according to the following specifications:

Have the user enter two integers. Then print all of the integers between those two integers, including the two integers themselves. For example, if the user enters 3 and 6, your program prints 3, 4, 5, and 6, each on a separate line. If the second integer is less than the first integer, then nothing should be printed. (Printing “nothing” is not printing nothing!) If the two numbers are equal, print only the single number.

Test it in SPIM using the File menu option and selecting Open.

Step 5. Copy your modified program (so you can refer back to it if necessary), and modify it again, so that the specification is as in Step 4 but then it also prints the following (assuming the example is 3, 4, 5, and 6 again). Don't worry about overflow.

The sum of the numbers listed is 18
The product of the numbers listed is 360

Step 6. Adjust the comments accordingly and improve the readability of the program where possible. Readability is crucial! For example, do not write comments that merely explain the meaning of an instruction. Notice that in the following line of code, the comment is redundant. It doesn't help a person who already knows what the **addi** instruction does. Comments should be higher level. For example, if this is part of a For loop, you might comment: #increment loop counter.

```
addi $t1, $t1, 1      # increment $t1
```

Make sure you have clearly commented using the guidelines in Step 2, Point 1.

Submit a printout of a single program that incorporates the specifications of Steps 3–6. **And put the program on Drive Q:\StudentFiles** If something doesn't work properly, and you have tried all you can to fix it, mention in your comments how your program does function. Plan for help before the due date. Start early so there is time to consult with me or a Math Department helper.

```

#File: q:\InstructorFiles\Chase_Gene\assembly\demo1a.s. [Identical to demo1.s, but uses register names, not numbers, and uses
# improved commenting style. I have compressed it to fit on one page. Please be more generous with white space!]
#Author: Paul van Arragon Last Modified 01 Feb 2005 by Gene B. Chase.
#Specification: enter an integer x, print x, 2x, 3x and 4x.
#Pseudocode:      print: Enter an integer:
#                  read the integer x
#                  print: Multiples:
#                  initialize multiplier = 1 and limit = 4
#                  while multiplier <= limit do
#                      print x * multiplier
#                      increment multiplier
#Use of registers: $v0, $a0 used for system calls
#                  $t0      multiplier
#                  $t1      limit
#                  $t2      multiples 1x, 2x, 3x, 4x
#                  $s0      the user-entered integer x
#####
.data
prompt: .asciiz "Enter an integer: "
string: .asciiz "Multiples: "
cr:     .asciiz "\n"
tab:    .asciiz "\t"

.text
# Print out the prompt "Enter an integer: "
main:  addi    $v0, $zero, 4
       la     $a0, prompt
       syscall

# Read an integer and put it in $s0
       addi    $v0, $zero, 5
       syscall
       move   $s0, $v0

# Print out the string "Multiples: "
       addi    $v0, $zero, 4
       la     $a0, string
       syscall

# Initialize multiplier $t0 = 1 and limit $t1 = 4
       addi    $t0, $zero, 1
       addi    $t1, $zero, 4

# Exit loop if multiplier > limit
Loop:  bgt     $t0, $t1, Endloop

# Print a tab
       addi    $v0, $zero, 4
       la     $a0, tab
       syscall

# Put the multiple x * multiplier in $t2
       mul    $t2, $s0, $t0

# Print the multiple from $t2
       addi    $v0, $zero, 1
       add    $a0, $t2, $zero
       syscall

# Increment the multiplier and loop again
       addi    $t0, $t0, 1
       j      Loop

# Print a carriage return
Endloop: addi    $v0, $zero, 4
        la     $a0, cr
        syscall

# Return from main program
        jr     $ra

# [a bug in PCspim requires that every instruction ends with carriage return. Only on the last line is it not obvious, so I like to end with #]

```

#File: q:\InstructorFiles\Chase_Gene\assembly\case.s

#Author: Paul van Arragon, 19 Sept. 1994, modified 12 Sept. 1997 by pva and 28 Oct. 2002 by Gene Chase.

#Specification: implement switch ... case statement, Exercise 2.10, p. IMD 2.20-3 (on your CD or our class web site)

#Pseudocode:

```
# initialize g=8, h=7, i=6, and j=4
# user input k
# switch(k)
# Case 0: f=i+j (10)
# Case 1: f=g+h (15)
# Case 2: f=g-h (1)
# Case 3: f=i-j (2)
# print f
```

#Use of registers:

```
# $s0 is the variable f with the result to be printed
# $s1, $s2, $s3, $s4 are variables g, h, i, j with values 8, 7, 6, 4
# $s5 is the variable k, the user-entered value
```

Things to notice:

```
# JumpTable must be a word address, so we align it to
# an address ending in two zeroes.
```

```
# We can figure out the JumpTable address by looking at the assembled code. First look at the statements
# with labels L0, L1, L2, and L3. Their addresses are 0x0040007c, 84, 8c, 94. Then look for these values
# in the data section. Therefore the JumpTable must be at 0x10010040. Then look at the la $t4, JumpTable
# instruction. It compiled into two statements, since 0x10010040 is too big to put in one instruction. When
# you run it, you should see that the address gets in $t4.
```

```
# Aside: the la command is compiled into two instructions.
# lui loads the 0x10010000, and ori adds the 0x00000040
```

```
#####
```

```
.data
str1: .ascii "The answer is "
str2: .ascii "\n"
str3: .ascii "Enter the value for k. Enter 0, 1, 2, or 3: "
```

```
.align 2          #This ensures that JumpTable is a word address
JumpTable:
.word L0, L1, L2, L3
```

```
.text
.globl main
```

main:

```
# initialize the 4 in $t2 and variables g h i and j
```

```
addi $t2, $0, 4
addi $s1, $0, 8
addi $s2, $0, 7
addi $s3, $0, 6
addi $s4, $0, 4
```

```
#prompt for an integer
```

```
addi $v0, $zero, 4
la $a0, str3
syscall
```

```

#read an integer (the value for k) into $s5
    addi    $v0, $zero, 5
    syscall
    move   $s5, $v0                # normally I would use all moves or none, using add $s5, $zero, $v0 here.

#make sure the value entered is 0, 1, 2, or 3
    slt    $t3, $s5, $zero        #Test if k < 0
    bne    $t3, $zero, Exit       #if k<0, go to Exit
    slt    $t3, $s5, $t2         #Test if k < 4
    beq    $t3, $zero, Exit       #if k >= 4, go to Exit

#multiply k by four
    add    $t1, $s5, $s5         #Temp register $t1 = 2 * k
    add    $t1, $t1, $t1         #Temp register $t1 = 4 * k

#Get the address of the jump table
    la     $t4, JumpTable        #load the address of the label
    add    $t1, $t1, $t4         #t1 = address of JumpTable[k]
    lw     $t0, 0($t1)          #Temp register $t0 = JumpTable[k]

#Jump to the address of the label in JumpTable[k]
    jr     $t0                  #jump based on register $t0

#Switch statement cases
L0:    add    $s0, $s3, $s4       #case 0: i+j (10)
        j Exit
L1:    add    $s0, $s1, $s2       #case 1: g+h (15)
        j Exit
L2:    sub    $s0, $s1, $s2       #case 2: g-h (1)
        j Exit
L3:    sub    $s0, $s3, $s4       #case 3: i-j (2)
Exit:

#print str1 (The answer is)
    addi    $v0, $zero, 4
    la     $a0, str1
    syscall

#print answer
    addi    $v0, $zero, 1
    add    $a0, $s0, $0
    syscall

#print str2 (carriage return)
    addi    $v0, $zero, 4
    la     $a0, str2
    syscall

#return from main
    jr     $ra

#

```

System Internals and Assembly Language: Assignment 2

To get more practice with MIPS you will write a more complicated program that uses a nested loop and an if statement.

Modify your program from Assignment 1 so that instead of printing all the integers between the numbers input by the user, it only prints the ones that are prime. For example, if you input 3 and 6, it prints out 3 and 5, since of the numbers 3, 4, 5, and 6, only 3 and 5 are prime. Notice the following: your program must work for **all integers** positive, negative, and 0, although only the following numbers are prime: numbers greater than 1 which have only themselves and 1 as divisors.

For now, you should not worry about the printing of sums and products, so you can just leave that part as is.

The high-level pseudocode of your program is as follows:

```
Get two integers, let's call them integer1 and integer2
For each integer inclusively between integer1 to integer2
    if the integer is prime, print it
```

To write the code that checks whether an integer is prime, you may use a for loop within the for loop. The inner loop might take each integer greater than 1 and less than the counter, and divide the counter by it. Now check the remainder. If you ever find a remainder of zero, you have found a divisor, so the counter must not be prime. The pseudocode for this inner loop is

```
Set the prime flag to be true, since we will assume an integer is prime until we find a divisor
For all possible divisors of the integer from 2 to (integer - 1)
    divide the integer by the possible divisor
    if the remainder is zero, reset the prime flag to false
If the flag is true, the integer must be prime, so print it
```

To find the remainder after dividing, use the commands `div` and `mfhi`. For example, consider these two lines of code:

```
div $t1, $t2    #divides $t1 by $t2 putting the quotient in the Lo register and the remainder in the Hi register
mfhi $t3        #moves the remainder from the Hi register into $t3. Next you can check whether $t3 is zero.
```

Now modify your program to print out the sums and products of the primes as well. For example, with inputs 3 and 6, you would print
The sum of the primes listed is 8
The product of the primes listed is 15

Make sure your code is as simple as possible and well commented. Only after getting it to work, should you consider further modifications as listed below. Make sure the original simplicity of your code is not shrouded too much by these added details. In other words, you have to use structured programming techniques even though assembly language does not force you to do so as Java does. I have nothing against Italian food, but I hate spaghetti code.

Optional extra credit improvements (to receive this extra credit you must succeed at the improvement, not merely attempt it, and you must have the main assignment working correctly):

(+3%) 1. Improve efficiency of your code by exiting the loop as soon as you find a divisor. Your inner loop would be as follows:

```
Set the initial possible divisor to 2
Set the prime flag to be true since we will assume it is prime until we find a divisor
For each divisor less than the integer, as long as the prime flag is still true
    divide the integer by the possible divisor
    if the remainder is zero, reset the prime flag to false
If the flag is true, the integer must be prime, so print it
```

You could also try it in the following way. It verges on spaghetti code, but is slightly shorter since it doesn't use the prime flag:

```
Set the initial possible divisor to 2
If the possible divisor is equal to the integer the integer must be prime so print it and quit
Divide the integer by the possible divisor
If the remainder is zero, it must not be prime so quit
Increase the possible divisor by 1 and start over at the second line
```

(+3%) 2. Improve efficiency by only checking divisors up to the square root of the integer. For example, you can prove that 29 is prime since it is not divisible by 2, 3, 4, or 5. Hint: do not calculate square roots but use the idea that $a^2 > b$ if $a > \sqrt{b}$.

(+3%) 3. Do not try all integers as possible divisors, but only 2 and the odd numbers.