

Assignment 3 is a **team project**. I have created shared directories for each team, so you should find a second directory on Q: with your last name as part of the directory name. You may base your work on the Assignment 2 code from both teammates. Some of you already used subroutines on Assignment 2. That will make this assignment easier. You may consult on all parts of the assignment, as long as any author is given credit in the comments of whatever routines he or she worked on. If both of your names are on all three parts, you will earn the same grade, but if one of you does one of the subroutines by yourself, or the main program by yourself, then the grades earned may not be the same.

This assignment is just as much about teamwork, style, format, and following instructions as it is about being able to use subroutines properly in assembly language. Plan ahead so that your work will be clear as well as accurate. I have put G in this handout at places so that you can use them as a checklist to make sure that you have followed instructions.

Assignment 3: Modify the program of Assignment 2 so that it uses two functions: `isPrime` and `getData`.

- G The first function checks for primeness. It must do the same thing that the Java function `private boolean isPrime (int n) {...}` would do. The function takes a single integer parameter `n` and returns 0 if `n` is not prime and 1 if `n` is prime (since assembly does not have a true boolean to return). The returned value will then be used in the same way that the prime flag was used in Assignment 2.
- G The second function will dialogue with the user to receive two numbers representing the range of numbers (in either order) between which the primes should be listed, inclusively. The function should return the two integers which the function asks the user for. It might correspond to the Java `private Pair getData () {...}` assuming that `Pair` is two `ints`. (Read the checklist below if you need to be reminded how to return two integers instead of just one.)
- G A working program may not earn an A if you do not adhere to conventions for writing functions.
- G Notice from the example at the end of this assignment that a subroutine has its own author, date, specifications, pseudocode, and register usage!
- G Each procedure should have its own `.data` and `.text` sections for any RAM locations that only it uses, although if any data must be “global” to all procedures, you must put that data in the main program.
- G Each of the three functions (`main`, `isPrime`, `getData`) must be in a file by itself, named respectively `Assignment3.s`, `isPrime.s` and `getData.s`. They will be in your joint directory on Q:.

You run your program by merging them with the batch file `combine.bat`, which I will demo in class, before submitting the result to SPIM. SPIM cannot load multiple files. This is the first year that I am trying this, so you may find it easier to do everything in one file and then separate them into three files just before you submit your project.

I recommend that you save your code frequently under different names before the final version, like `lab3-1gc.s`, `lab3-2gc.s`, `lab3-3kp.s`, and `lab3-4kp.s` so that if you make a mistake, you can “roll back” to an earlier version. I also recommend that if each of you make changes to the same code at the same time, you avoid saving your changes on top of each other, perhaps by incorporating your initials in the file names that you save when you are working alone, as I did with “kp” and “gc” in those sample file names. Then merge correct pieces of code soon and often when you are together. It is possible that you can work independently on each function, and then just get together to figure out how to combine them into your program, but you are also welcome to consult and copy from each other on everything from the very start.

If neither of you successfully solved Assignment 2, then with a 10% penalty, I will supply you with an excellent working version of Assignment 2 if you ask a day in advance.

¹ Based on Lab 3 © 1998 Paul van Arragon. All rights reserved. Used with permission. v4 © 2006 G. Chase

Checklist for writing functions or procedures

- A. In the main program before calling the procedure or function, you must—
1. save any registers if they are of type “temporary” if you need them again after the procedure or function call. First adjust the stack pointer (\$sp) by subtracting 4 for each register, then store your registers in position 0(\$sp), 4(\$sp), ... If the only values you reuse are in saved registers, this step is unnecessary.
 2. save also \$ra on the stack so that the main program can jump back to the program that called it.
 3. place parameters in \$a0, \$a1, \$a2, \$a3, and on the stack if necessary because there are too many. By convention, you may not use any other registers to pass values to the function or procedure.
 4. then jal to the function or procedure.
- B. After returning from the procedure or function:
5. restore the “temporary” registers and \$ra and restore the original stack pointer.
 6. if it was a function, assume the value is returned in \$v0 and \$v1 if necessary.
- C. In the procedure or function that is called you must—
7. save any registers if they are of type “saved” if you will be using them. Save them on the stack by adjusting the stack pointer as described above in Step 1.
 8. do your calculation, assuming parameters are in \$a0, \$a1, \$a2, \$a3, and the stack if necessary. By convention, you may not use any other values from the calling program.
 9. if you are a function, place the value to be returned in \$v0 and \$v1 if necessary.
 10. restore the “saved” registers and the stack pointer.
 11. jr \$ra to get back to the calling program.

Appendix section A.6 says this again. Your green “cheat sheet” has a list of MIPS register usage conventions. Your text says it yet another time.

Example: (See demo2.s. Parts of its code are on the next page.)

Consider demo1.s. For the sake of illustration, suppose we wanted to do the multiplication in a separate function. We must replace the line mul \$t2, \$s0, \$t0 with all the steps of a function call. We would modify our code as follows. The code appears below.

1. save \$t0 and \$t1 since they are temporary and get reused. Adjust the stack by - 8. We no longer need \$t2 at all, so it doesn't get saved.
2. don't bother saving \$ra since this program doesn't return anywhere.
3. put the parameters \$t0 and \$s0 in \$a0 and \$a1 so the function knows what to multiply.
4. jal to the function.
5. restore \$t0 and \$t1 and adjust the stack by 8.
6. assume the product is in \$v0, so that's what we print instead of \$t2.
7. suppose the function works with registers \$t0 and \$s1. It must save register \$s1 first thing.
8. multiply the parameter values in \$a0 and \$a1.
9. put the result in \$v0.
- 10 restore register \$s1.
- 11 jr \$ra.

Note two cases of intentional extra work.

The calling program saves \$t1 even though the function never uses \$t1.

The function saves \$s1 even though the caller doesn't use register \$s1.

Why do we do it? Do you see the wisdom of the saving conventions? How would you write the code so that this extra work is unnecessary, and yet follow our conventions?

The following code illustrates how function calling works. It is in Q:\InstructorFiles\Chase_Gene\assembly\demo2.s. Note that demo2.s as found on Q: does not work, intentionally. Read the comments in the program to see how to fix it. Assignment 3 has nothing to do with any multFunction. This example merely gives you another model of how to call functions. You are to modify Assignment 2, not demo2.s.

Here is how the function multFunction is called.

```

#Save $t0 and $t1, adjusting stack by -8
addi $sp, $sp, -8
sw $t1, 4($sp)
sw $t0, 0($sp)

# put the parameters from $t0 and $s0 into $a0 and $a1 and jal to the function
move $a0, $t0
move $a1, $s0
jal multFunction

#restore $t0 and $t1, adjusting stack by 8
lw $t0, 0($sp)
lw $t1, 4($sp)
addi $sp, $sp, 8

#print the multiple (which is in returned in $v0)
add $a0, $v0, $zero #Load the integer to be printed in register $a0.
addi $v0, $zero, 1 #Enter system call code 1 for printing integers.
syscall #Print the integer.

```

Here is the function:

```

multFunction:
#Save $s1, adjusting stack by -4
addi $sp, $sp, -4
sw $s1, 0($sp)

#Put the parameters in the registers that you want them.
move $s1, $a0
move $t0, $a1

#The function body: multiply the parameters; place result in $v0 per convention.
mul $v0, $t0, $s1

#Restore $s1, adjusting stack by 4, and return to calling code
lw $s1, 0($sp)
addi $sp, $sp, 4
jr $ra

```

Now what follows on page 4 is the demo that I will be doing on how to use `combine.bat`.

Things to note: The integers that should be entered should be positive and less than byte-sized, because we are putting them into a byte-array. We don't have a "read byte" command to execute, so negative integers don't have a sign in bit 7 as they should if they were negative bytes.

```

# This is version 1.2 of the file found at:
# q:\InstructorFiles\Chase_Gene\Assembly\swapDriver.s
# Paul van Arragon 29 Sep 1994; mod: Gene B. Chase,
# 27 Sep 2001 for style; 1 October 2006 for small
# corrections (text page number, spelling, comments)
# Specification: Demonstrate use of procedure swap:
# The array is initialized in memory to be the bytes
# 0,1,2,3,...,10. That is, A[i] contains i.
# The user enters new values for A[4] and A[5] which
# are then swapped.

#main pseudocode:
# initialize array A to [0,1,2,3,4,5,6,7,8,9,10].
# read a value into A[4] and A[5] in place of the 4
# and 5.
# call swap(4), meaning swap A[4] with A[5].
# print "The array values, after swapping are: "
# For i := 0 to 10 do print(A[i])

#use of registers:
# $t1 for-loop counter
# $t2 for-loop final value
# $s0 array address
# $a0 is the parameter to pass the address of
# the array
# $a1 is index (offset) of element to be swapped
# with its successor.
# The rest is as described in your text.
#####
.data
str1: .asciiz "Enter an integer for array[4]: "
str2: .asciiz "Enter an integer for array[5]: "
str3: .asciiz "The array values, after swapping are:"
space: .asciiz " "
nl: .asciiz "\n"

# Note that we don't need to align the array, since
# we use bytes instead of words
array: .byte 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
.text
main:
# Put array address in $s0
la $s0, array

# prompt for an integer
addi $v0, $zero, 4
la $a0, str1
syscall

# read an integer
addi $v0, $zero, 5
syscall

# put the integer from $v0 in array[4]
sb $v0, 4($s0)

# prompt for a second integer
addi $v0, $zero, 4
la $a0, str2
syscall

# read the second integer
addi $v0, $zero, 5
syscall

# put the second integer from $v0 in array[5]
sb $v0, 5($s0)

# call the swap procedure
move $a0, $s0 #arg1 is array address
addi $a1, $zero, 4 #arg2 is subscript 4, to swap
# array[4] & array[5]
addi $sp, $sp, -4 #adjust stack pointer
sw $ra, 0($sp) #save return address
jal swap #call the procedure
lw $ra, 0($sp) #retrieve return address
addi $sp, $sp, 4 #readjust stack

# print str3 (The array values are: )
addi $v0, $zero, 4
la $a0, str3
syscall

# print the array
# For i =0 to 10 do print(array[i])

# initialize for loop counter and end values
addi $t1, $zero, 0
addi $t2, $zero, 10

# create address of array[i] in $t3
Loop:
add $t3, $t1, $s0

# print array[i]
addi $v0, $zero, 1
lb $a0, 0($t3)
syscall

# print a space between numbers
addi $v0, $zero, 4
la $a0, space
syscall

# increment and test part of the for loop:
addi $t1, $t1, 1
ble $t1, $t2, Loop

# print nl (new line)
addi $v0, $zero, 4
la $a0, nl
syscall

# return from main to operating system
jr $ra
#####
# File: swapFunction.s
# Author: Gene B. Chase, 1 October 2006. vl.1
# Procedure swap. This one varies from the text
# swap procedure in three important ways:
# * We are swapping bytes k and k+1, not words, so I
# don't multiply k by 4.
# * We use saved registers instead of temporary
# registers, so I can demo saving them.
# * BIG: a0 and a1 are the address of array and the
# offset into array, not the two addresses of
# elements to be swapped as on text page 124!
# pseudocode (Procedure gets its own specifications,
# pseudocode, and register usage!)
# load A[k] into temp1
# load A[k+1] into temp2
# store temp1 into A[k+1]
# store temp2 into A[k]

# save the callee-saved registers
swap:
addi $sp, $sp, -8
sw $s0, 4($sp)
sw $s1, 0($sp)

# create the address of A[4] in $t1
add $t1, $a0, $a1

# swap the bytes using lb (not lw)
lb $s0, 0($t1)
lb $s1, 1($t1)
sb $s1, 0($t1)
sb $s0, 1($t1)

# restore the registers
lw $s0, 4($sp)
lw $s1, 0($sp)
addi $sp, $sp, 8

# return to the caller of swap
jr $ra
#

```