

COE 308 – Computer Architecture

Assignment 3 SOLUTION: Procedures in MIPS Assembly Language

For the following problems, the table holds C code functions. Assume that the first function listed in the table is called first. You will be asked to translate these C code routines into MIPS assembly.

a.	<pre>int compare(int a, int b) { if (sub(a, b) >= 0) return 1; else return 0; } int sub(int a, int b) { return a - b; }</pre>
b.	<pre>int fib_iter(int a, int b, int n) { if (n == 0) return b; else return fib_iter(a+b, a, n-1); }</pre>

1. Implement the C code in the table in MIPS assembly. What is the total number of MIPS instructions needed to execute the function?

a.	<pre>compare: addi \$sp, \$sp, -4 # allocate frame = 4 bytes sw \$ra, 0(\$sp) # save return address jal sub # call sub li \$t0, 0 # result = 0 bltz \$v0, exit # if sub(a,b)<0 goto exit li \$t0, 1 # result = 1 exit: move \$v0, \$t0 # \$v0 = result lw \$ra, 0(\$sp) # restore return address addi \$sp, \$sp, 4 # free stack frame jr \$ra # return to caller sub: sub \$v0, \$a0, \$a1 # result = a - b jr \$ra # return to caller 11 or 12 instructions (depending whether bltz is taken or not). Includes the call and return from sub</pre>
----	---

b.	<pre> int fib_iter(int a, int b, int n) { if (n == 0) return b; else return fib_iter(a+b, a, n-1); } fib_iter: bne \$a2, \$0, else # if (n != 0) goto else move \$v0, \$a1 # result = b jr \$ra # return to caller else: addiu \$sp, \$sp, -4 # allocate frame = 4 bytes sw \$ra, 0(\$sp) # save return address move \$t0, \$a0 addu \$a0, \$a0, \$a1 # \$a0 = a+b move \$a1, \$t0 # \$a1 = a addiu \$a2, \$a2, -1 # \$a2 = n-1 jal fib_iter # recursive call lw \$ra, 0(\$sp) # restore return address addiu \$sp, \$sp, 4 # free stack frame jr \$ra # return to caller Total number of instructions = n * 11 + 3 11 instructions for each recursive call/return (if n>0) +3 instructions if (n == 0) </pre>
----	---

2. Functions can often be implemented by compilers “in-line”. An in-line function is when the body of the function is copied into the program space, allowing the overhead of the function call to be eliminated. Implement an “in-line” version of the above C code in MIPS assembly. What is the reduction in the total number of MIPS assembly instructions needed to complete the function?

a.	<pre> compare: sub \$t0, \$a0, \$a1 li \$v0, 0 bltz \$t0, exit li \$v0, 1 exit: jr \$ra 4 or 5 instructions (whether bltz is taken or not) </pre>
b.	<p>Due to recursive nature of the code, not possible for the compiler to in-line the function call.</p>

3. For each function call, show the contents of the stack after the function call is made. Assume that the stack pointer is originally at address 0x7ffffffc.

a.	<p>after calling function compare: $\\$sp = \\$sp - 4 = 0x7ffffff8$</p> <p>0x7ffffff8: return address of compare</p>
b.	<p>suppose that fib_iter was called with $n = 4$</p> <p>0x7ffffff8: return address of caller ($n=4$) 0x7ffffff4: return address of 1st recursive call ($n=3$) 0x7ffffff0: return address of 2nd recursive call ($n=2$) 0x7fffffec: return address of 3rd recursive call ($n=1$) 0x7fffffe8: return address of 4th recursive call ($n=0$)</p> <p>The return address of the 4 recursive calls is the same. It is the address of the 'lw' instruction that comes immediately after the recursive 'jal fib_iter' instruction</p>

The following problems refer to a function `f` that calls another function `func`. The function declaration for `func` is "int func(int a, int b);". The code for function `f` is as follows:

a.	<pre>int f(int a, int b, int c) { return func(func(a, b), c); }</pre>
b.	<pre>int f(int a, int b, int c) { return func(a, b) + func(b, c); }</pre>

4. Translate function `f` into MIPS assembly code, using the MIPS calling convention. If you need to use register `$t0` through `$t7`, use the lower-numbered registers first.

a.	<pre>int f(int a, int b, int c) { return func(func(a, b), c); }</pre> <pre>f: addiu \$sp, \$sp, -8 # allocate frame = 8 bytes sw \$ra, 0(\$sp) # save return address sw \$a2, 4(\$sp) # save c jal func # call func(a,b) move \$a0, \$v0 # \$a0 = result of func(a,b) lw \$a1, 4(\$sp) # \$a1 = c jal func # call func(func(a,b),c) lw \$ra, 0(\$sp) # restore return address addiu \$sp, \$sp, 8 # free stack frame jr \$ra # return to caller</pre>
----	---

b.	<pre> int f(int a, int b, int c) { return func(a, b) + func(b, c); } f: addiu \$sp, \$sp, -12 # allocate frame = 12 bytes sw \$ra, 0(\$sp) # save return address sw \$a1, 4(\$sp) # save b sw \$a2, 8(\$sp) # save c jal func # call func(a,b) lw \$a0, 4(\$sp) # \$a0 = b lw \$a1, 8(\$sp) # \$a1 = c sw \$v0, 4(\$sp) # save result of func(a,b) jal func # call func(b,c) lw \$t0, 4(\$sp) # \$t0 = result of func(a,b) addu \$v0, \$t0, \$v0 # \$v0 = func(a,b)+func(b,c) lw \$ra, 0(\$sp) # restore return address addiu \$sp, \$sp, 12 # free stack frame jr \$ra # return to caller </pre>

5. Right before your function `f` of Problem 4 returns, what do you know about contents of registers `$t5`, `$s3`, `$ra`, and `$sp`? Keep in mind that we know what the entire function `f` looks like, but for function `func` we only know its declaration.

Register `$ra` is equal to the return address in the caller function, registers `$sp` and `$s3` have the same values they had when function `f` was called, and register `$t5` can have an arbitrary value. For `$t5`, note that although our function `f` does not modify it, function `func` is allowed to modify it so we cannot assume anything about `$t5` after function `func` has been called.

For the following problems, the table has an assembly code fragment that computes a Fibonacci number. However, the entries in the table have errors, and you will be asked to fix these errors.

	<pre> fib: addi \$sp, \$sp, -12 sw \$ra, 8(\$sp) sw \$s1, 4(\$sp) sw \$a0, 0(\$sp) slti \$t0, \$a0, 3 beq \$t0, \$0, L1 addi \$v0, \$0, 1 j exit L1: addi \$a0, \$a0, -1 jal fib addi \$s1, \$v0, \$0 addi \$a0, \$a0, -1 jal fib add \$v0, \$v0, \$s1 exit: lw \$a0, 8(\$sp) lw \$s1, 0(\$sp) lw \$ra, 4(\$sp) addi \$sp, \$sp, 12 jr \$ra </pre>

6. The MIPS assembly program above computes the Fibonacci of a given input. The integer input is passed through register \$a0, and the result is returned in register \$v0. In the assembly code, there are few errors. Correct the MIPS errors.

a.	<pre> FIB: addi \$sp, \$sp, -12 sw \$ra, 8(\$sp) sw \$s1, 4(\$sp) sw \$a0, 0(\$sp) slli \$t0, \$a0, 3 beq \$t0, \$0, L1 addi \$v0, \$0, 1 j EXIT L1: addi \$a0, \$a0, -1 jal FIB addi \$s1, \$v0, \$0 addi \$a0, \$a0, -1 jal FIB add \$v0, \$v0, \$s1 EXIT: lw \$a0, 0(\$sp) lw \$s1, 4(\$sp) lw \$ra, 8(\$sp) addi \$sp, \$sp, 12 jr \$ra </pre>
----	---

7. For the recursive Fibonacci MIPS program above, assume that the input is 4. Rewrite the Fibonacci program to operate in a non-recursive manner. Restrict your register use to registers \$s0 - \$s7. What is the total number of instructions used to execute your non-recursive solution versus the recursive version of the factorial program?

	<p>According to MIPS convention, we should preserve \$s0 and \$s1. We could have used \$t0 and \$t1 without preserving their values. For input 4, we have 23 instructions in non-recursive Fib versus 73 instructions to execute recursive Fib.</p>
a.	<pre> fib: addiu \$sp, \$sp, -8 # allocate stack frame sw \$s0, 0(\$sp) # save \$s0 sw \$s1, 4(\$sp) # save \$s1 li \$s0, 1 # prev value in Fib sequence li \$v0, 1 # curr value in Fib sequence blt \$a0, 3, EXIT # if (n < 3) goto exit LOOP: addu \$s1, \$v0, \$s0 # next = curr + prev move \$s0, \$v0 # prev = curr move \$v0, \$s1 # curr = next addiu \$a0, \$a0, -1 # n = n - 1 bge \$a0, 3, LOOP # Loop if (n >= 3) EXIT: lw \$s0, 0(\$sp) # restore \$s0 lw \$s1, 4(\$sp) # restore \$s1 addiu \$sp, \$sp, 8 # free stack frame jr \$ra # return to caller </pre>

In this exercise, you will be asked to write a MIPS assembly program that converts strings into the number format as specified in the table.

a.	Positive integer decimal string
b.	String of hexadecimal digits

8. Write a program in MIPS assembly language to convert an ASCII number string with the conditions listed in the table above, to an integer. Your program should expect register \$a0 to hold the address of a null-terminated string containing some combination of the digits 0 through 9. Your program should compute the integer value equivalent to this string of digits, then place the number in register \$v0. If a nondigit character appears anywhere in the string, your program should stop with the value -1 in register \$v0.

a.	<pre> str2int: # convert string to integer li \$t6, 0x30 # \$t6 = '0' li \$t7, 0x39 # \$t7 = '9' li \$v0, 0 # initialize \$v0 = 0 move \$t0, \$a0 # \$t0 = pointer to string lb \$t1, (\$t0) # load \$t1 = digit character LOOP: blt \$t1, \$t6, NoDigit # char < '0' bgt \$t1, \$t7, NoDigit # char > '9' subu \$t1, \$t1, \$t6 # convert char to integer mul \$v0, \$v0, 10 # multiply by 10 add \$v0, \$v0, \$t1 # \$v0 = \$v0 * 10 + digit addiu \$t0, \$t0, 1 # point to next char lb \$t1, (\$t0) # load \$t1 = next digit bne \$t1, \$0, LOOP # branch if not end of string jr \$ra # return integer value NoDigit: li \$v0, -1 # return -1 in \$v0 jr \$ra </pre>	

b.	hexstr2int:	# convert hex string to int
	li \$t4, 0x41	# \$t4 = 'A'
	li \$t5, 0x46	# \$t7 = 'F'
	li \$t6, 0x30	# \$t6 = '0'
	li \$t7, 0x39	# \$t7 = '9'
	li \$v0, 0	# initialize \$v0 = 0
	move \$t0, \$a0	# \$t0 = pointer to string
	lb \$t1, (\$t0)	# load \$t1 = digit character
	LOOP:	
	blt \$t1, \$t6, NoDigit	# char < '0'
	bgt \$t1, \$t7, HEX	# check if hex digit
	subu \$t1, \$t1, \$t6	# convert to integer
	j Compute	# jump to Compute integer
	HEX:	
	blt \$t1, \$t4, NoDigit	# char < 'A'
	bgt \$t1, \$t5, NoDigit	# char > 'F'
	addiu \$t1, \$t1, -55	# convert: 'A'=10, 'B'=11, etc
	sll \$v0, \$v0, 4	# multiply by 16
	add \$v0, \$v0, \$t1	# \$v0 = \$v0 * 16 + digit
	addiu \$t0, \$t0, 1	# point to next char
	lb \$t1, (\$t0)	# load \$t1 = next digit
	bne \$t1, \$0, LOOP	# branch if not end of string
	jr \$ra	# return integer value
	NoDigit:	
	li \$v0, -1	# return -1 in \$v0
	jr \$ra	