Comp 411 Computer Organization
Fall 2007

Problem Set #4 Solutions

Problem 1
a) The instructions that place values on the stack are:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw $ra 12($sp)</td>
<td>This places the return address on the stack</td>
</tr>
<tr>
<td>sw $a0 8($sp)</td>
<td>This places the first argument (n) on the stack</td>
</tr>
<tr>
<td>sw $a1 4($sp)</td>
<td>This places the second argument (a) on the stack</td>
</tr>
<tr>
<td>sw $a2 0($sp)</td>
<td>This places the third argument (b) on the stack</td>
</tr>
</tbody>
</table>

The argument values are placed on the stack, but they are never retrieved from it. Thus, it is not necessary that they be stored on the stack.

b) The resulting fragment would still work. If the branch in L08 evaluates to true, it is because $a0 (holding the value of n), was equal to 0. When n is 0, the function does not call itself again; it simply returns a. Thus, the return address in $ra is never overwritten by another call and is still valid.

c) ifib:

```
addu  $t0,$0,$0  # a = 0;
addiu $t1,$0,1   # b = 1;
while:
    beq  $a0,$0,return  # while (n > 0) {
    addu  $t2,$t0,$t1  # t = a + b;
    addu  $t0,$t1,$0  # a = b;
    addu  $t1,$t2,$0  # b = t;
    addiu $a0,$a0,-1  # n = n - 1;
    beq  $0,$0,while  # }
return:
    addu  $v0,$t0,$0
jr    $ra
```

d) Answers may vary, but the iterative one is probably faster, since it does not need to allocate space on the stack each loop (and thus uses less memory).

e) By looking through the stack, one can see that the return address being stored on the stack in each call frame is 0x00400058. This is the address of the instruction immediately after the jal call. So, the return address points to L14: lw $ra,12($sp). Since this line is 13 lines below the beginning of the function, it is 13*4 (or 0x32) bytes above it in memory. Thus start of the function is at 0x00400058 - 0x32 or 0x00400024

f) As shown at address 0x7ffeefd8 in the stack, n was 7.
g) Since the memory at stack address 0x7fffef94 appears to be uninitialized (instead of having a new value of the a argument), the last executed line was likely L03.

h) The last value for n in this stack trace was 3. Since the function terminates when n = 0, there will be three more call frames before the terminal case of n = 0 is reached. Since each call frame uses 4 words, with 4 bytes per word, another 3 * 4 * 4 (or 0x30) bytes are needed. The lowest memory address will then be 0x7fffef90 - 0x30 = 0x7fffef60.

Problem 2

a) This function must store three values on the stack- the return address, the input n, and the evaluation of the call to fib2(n-1). When compared to unoptimized fib (the code as given), it needs less (3 vs. 4). However, compared to optimized fib (from problem 1a) it needs more (3 vs. 1). Notice that fib2 terminates when n < 2, so its maximum call frame depth is less than fib which terminates when n = 0.

b) The function fib2 found by Bud is actually amazingly slower than fib. It recalculates many of the values several times when recursing. It is a terrible way to calculate the Fibonacci numbers. Never use it.

Consider the case when n = 4. On the initial call to the function, it will recursively calculate fib2(3) and fib2(2). However, the call to fib2(3) will also recursively call fib2(2)! Since these calls are evaluated independently, the data is not shared and the work is repeated.

The values for n = 2 are calculated twice!
c) \textbf{fib2:}

\begin{verbatim}
add $sp,$sp,-16
sw $ra,12($sp)  # save return address
sw $a0,8($sp)   # save n
slti $t0,$a0,2  # if (n<2)
beq $t0,$0,else
add $v0,$0,$a0  # return value = n
beq $0,$0,return
else:
  add $a0,$a0,-1
  jal fib2
  sw $v0,4($sp)  # save the return value of fib2(n-1) on stack
  lw $a0,8($sp)  # get back a copy on n
  add $a0,$a0,-2
  jal fib2
  lw $t0,4($sp)  # get back the return value from fib2(n-1)
  add $v0,$v0,$t0
return:
  lw $ra,12($sp)
  add $sp,$sp,16
  jr $ra
\end{verbatim}

\textbf{Problem 3}

a) \textbf{gcd:}

\begin{verbatim}
sub $sp,$sp,4  # save return address for the end
sw $ra,0($sp)  # if(x != y), start next calc
bne $a0,$a1,next
move $v0,$a0  # x = y, so return x

b end
next: slt $t0,$a0,$a1  # if(x < y)
beq $t0,$0,xminy
subu $a1,$a1,$a0  # else, calc y - x
b recur
xminy: subu $a0,$a0,$a1  # x was > y, so calc x - y
recur: jal gcd  # call gcd again
end: lw $ra,0($sp)  # gcd is finished, return to caller
addu $sp,$sp,4
j $ra
\end{verbatim}
b) An easy way to track the number of calls is to increment a counter register by one each time the function is entered. The number of calls to gcd is as follows:

<table>
<thead>
<tr>
<th>invocation</th>
<th>number of calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd(217, 147)</td>
<td>13</td>
</tr>
<tr>
<td>gcd(65, 127)</td>
<td>25</td>
</tr>
<tr>
<td>gcd(610, 377)</td>
<td>14</td>
</tr>
<tr>
<td>gcd(1024, 768)</td>
<td>4</td>
</tr>
<tr>
<td>gcd(775, 899)</td>
<td>11</td>
</tr>
</tbody>
</table>

Clearly, gcd(65, 127) results in the most calls to gcd.

c) An interesting fact about the Fibonacci numbers is that any two consecutive Fibonacci numbers are relatively prime, that is, their only common factor is 1. Thus, the gcd will always return a greatest common factor of 1 for any two consecutive Fibonacci numbers.

Since each Fibonacci number is the sum of the previous two Fibonacci numbers, the gcd function will subtract the numbers until it arrives at 1. So, when the function is called, \( F_N \) is found to be greater than \( F_{N-1} \), so \( F_N \) is replaced with \( F_N - F_{N-1} \). Notice that \( F_N - F_{N-1} \) is equal to \( F_{N-2} \) (after all \( F_N \) is defined to be \( F_{N-1} + F_{N-2} \)). Eventually, after \( N-1 \) calls, the numbers are both reduced to 1.

d) Answers will vary and may include mention of the \( a = qb + r \) proof of the algorithm, proof via line segments, or description of removing the remainder.