The University *of* North Carolina *at* Chapel Hill

Comp 411 Computer Organization
Spring 2012

Problem Set #3
Solution Set

Problem 1. "Compiler Appreciation"

There are many solutions to these problems. My solutions use as few stores and loads as possible, primarily to keep the code simple.

a)

```
lw $t0 y
lw $t1 x

sub $t0 $0 $t0    #make y = -1
sub $t0 $t0 $t1   #y - x
sw $t0 y
```

b) I'm assuming that i is an array offset, and not already adjusted for word alignment.

```
lw $t0 i         #get the values for mem offsets
addi $t1 $t0 -1
addi $t2 $t0 1

sll $t0 $t0 2    #insure word alignment
sll $t1 $t1 2
sll $t2 $t2 2

lw $t1 a($t1)    #fetch the values
lw $t2 a($t2)
add $t1 $t1 $t2 #do the add
sw $t1 a($t0)    #store the value
```

c)

```
       lw $t0 x
       lw $t1 y

       sgt $t2 $t0 $t1  #is x > y?
       bne $t2 $0 else  #looks like x is < y, so go to else case
       sub $t0 $t0 $t1  #set x = x - y
       j done           #the new x is done, go to store it
 else: sub $t0 $t1 $t0  #set x = y -x
 done: sw $t0 x         #store the new x
```

d) Since there is no guarantee that the loop will execute even one time, i must be checked before executing any loop code.

```
       lw $t0 i          #get i

       j test            #jump to the test
while: srl $t0 $t0 1     #shift right 1
test:  andi $t0 $t0 1    #and i with 1
       beq $t0 $0 while  #if test is met, run loop again
       sw $t0 i          #store final value
```

e) Since the loop will always run the first time, my code enters the loop with i equal to 0. Upon entering the loop structure, i is immediately incremented. For this reason, the loop runs from 1 to 9, since after the eighth time it will return to the top of the loop and begin the ninth and final run.

```
     addi $t0 $0 $0    #set 'i' to 0
for: addi $t0 $t0 1    #do the i++ part (the first time through, this will make i=1)
     sll $t1 $t0 2     #multiply for word alignment
     add $t2 $t0 $t0   #set i = i + i
     addi $t2 $t2 1    #set i = i + 1
     sw $t2 a($t1)     #store the new i in memory, using word aligned value
     slti $t1 $t0 9    #check if i < 9 (because we should stop the 10th time)
     bne $t1 $0 for    #if i is still < 9, do the loop again
```

f) Note that this is a fairly complicated address redirection. For the purpose of this class, we will be assuming that the array is filled with indices and not memory pointers.

```
lw $t0 x         #get x
sll $t0 $t0 2
lw $t1 a($t0)    #get value at a[x]
sll $t1 $t1 2
lw $t1 a($t1)    #get value at a[a[x]]
sw $t1 a($t0)    #store the result back in a[x]
```

Problem 2. "MIPS Calisthenics"

a)

```
add $t0 $0 $0
add $t1 $0 $0
add $t2 $0 $0
add $t3 $0 $0
add $t4 $0 $0
add $t5 $0 $0
add $t6 $0 $0
add $t7 $0 $0
add $t8 $0 $0
```

b)

```
      addi $t0 $0 0x100    #starting at address 0x100
loop: sw $0 0($t0)         #store 0 at the address
      addi $t0 $t0 4       #add 4 for the next address
      sge $t1 $t0 0x1fc    #check if address is less than 0x1fc
      bne $t1 $0 loop      #if not greater, clear the next address
```

c)

```
addi $t2 $t0 $0     #put value at $t0 in a temporary location
addi $t0 $t1 $0     #copy value at $t1 to $t0
addi $t1 $t2 $0     #copy original $t0 value from temporary location to $t1
```

d)

```
          add $t2 $0 $0      #set counter to 0}
          addi $t0 $0 0x100  #starting at address 0x100}
loop:     lw $t1 0($t0)      #get value at the current address}
          bne $t1 $0 nextaddr #if it is not 0, continue loop}
          addi $t2 $t2 1     #since the value is 0, we should count it}
nextaddr: addi $t0 $t0 4     #add 4 for the next address}
          sge $t1 $t0 0x1fc  #check if address is greater than 0x1fc}
          bne $t1 $0 loop    #if not greater, clear the next address}
```

Problem 3.

a) The return address is saved in $sp-8, the argument "n" is saved in $sp-4, and the value returned from the first call to fib (with n-1) is saved in $sp. All of these values must be saved; the return address must be saved because the function is not a leaf. The "n" argument must be saved because it is needed to construct the parameter for the second call to fib (with n-2). The value returned from the first fib call must be saved as well, so that it is available after the second call.

b) It would not work. If you saved the value returned from the first call to fib in a scratch register (like $a1 in this case). Subsequent, calls by non-leaf children would overwrite it, thus making it unavailable upon return.

c)

```
int fib(int n) {
    int a, b, t;
    if (n < 2)
        return n;
    else {
        a = 0;
        b = 1;
        n -= 1;
        while (n != 0) {
            t = a;
            a = b;
            b += t;
            n -= 1
        }
        return b;
    }
}
```

```
fib:        slti   $t0,$a0,2
            beq    $t0,$0,else
            add    $v0,$0,$a0
            beq    $0,$0,rtn
else:       addi   $t0,$0,0
            addi   $t1,$0,1
            beq    $0,$0,test
while:      add    $t2,$t0,$0
            add    $t0,$t1,$0
            add    $t1,$t1,$t2
test:       addi   $a0,$a0,-1
            bne    $a0,$0,while
            add    $v0,$0,$t1
rtn:        j      $ra
```

d) The iterative version is a leaf routine, and all variables can be allocated in registers, thus, no stack space is needed and it requires less memory. The assembly language implementation is also shorter, and faster since a Fibonacci number is only computed once, whereas the same Fibonacci numbers are computed several times in the recursive version. For example:

fib(5) = fib(4)+fib(3)
fib(5) = (fib(3)+fib(2))+(fib(2)+fib(1))
fib(5) = ((fib(2)+fib(1))+(fib(1)+fib(0))+((fib(1)+fib(0))+fib(1))
fib(5) = ((((fib(1)+fib(0))+ fib(1))+(fib(1)+fib(0)) + ((fib(1)+fib(0))+fib(1))

Note that fib(3) is computed twice, and fib(2) is computed 3 times. This redundancy only gets worse as n grows (it grows proportional to n2). Therefore, the iterative version is faster than the recursive one. Perhaps the iterative version is slightly easier to understand. There is some subtly in the iterative code— for example, the need for the t variable to manage the updating of the n-1 and n-2 Fibonacci numbers.

e) The trick here is to first find some stack frame for an instance of fib(). Each stack frame is composed of three words, the first word being the return address and the second word being the argument passed in. Notice that successive calls to fib() are with arguments one or two less than the caller's. If we look into this stack dump, we can see a 3-word pattern starting at location 0x7fffefe0. We can surmise that the contents of 0x7fffefe0 are the return address of the first self-call of fib (with argument n-1). From this we can figure out that the function fib() must be located at 0x00400024 (0x00400048 – 4*9).

f) The argument of the original call was 7. One indication that this is the initiating call is that the return address 0x0040007c is outside of the fib() routine.

g) This is a tricky question. If you examine the stack carefully, it appears that most stack frames do not have their 3rd element initialized. By examining the code, one can see that immediately upon return from the first self-call (fib(n-1)), the returned value ($v0) is stored on the stack, as is evident from the 1 stored in stack location 0x7fffef9c. You can also see that the return address of the next call is different from those previous, which indicates that the second call to fib(n-2) has already taken place. This call would be the second call of a callee whose argument was 2 (from stack location 0x7fffefa0), thus, fib is called with 2-2 = 0, and this 0 has not yet been stored onto the stack. Thus, the stack dump must have occurred in a call with fib(0) after the instruction with label L1.

h) The deepest stack recursion is determined argument. If fib is called with n, then a stack frame will be allocated for fib calls with arguments n-1 and n-2. Fib is called again until n is either 1 or 0. The second call to fib (n-2) reuses the same memory used by the first call (n-1). Thus, the depth of the stack is equal to 3 times the argument, in this case 21 locations. So the lowest memory location allocated on the stack in this location is 0x7fffef90. However, in the final call of fib, the third stack entry is never used, thus the lowest memory location referenced is 0x7fffef94.