The UNIVERSITY *of* NORTH CAROLINA *at* CHAPEL HILL

Comp 411 Computer Organization
Spring 2012

Problem Set #2
Issued Wednesday, 2/1/12; Due Wednesday, 2/15/12

Homework Information: Some of the problems are probably too long to be done the night before the due date, so plan accordingly. Late homework will not be accepted. Feel free to get help from others, but the work you hand in should be your own.

Problem 1. "Compiler Appreciation"

Translate the following code fragments (written in C) to MIPS assembly language. Use the general approach shown in lecture (allocate variables into low, directly addressable, memory addresses). You don't have to write optimized assembly language unless you are into that sort of thing. Just show the executable code—you can assume that the necessary storage allocation for variables and arrays has already been done (meaning that you can refer to them by their label). Furthermore, you can assume that all variables have been allocated into the lower 32K bytes of memory, and that all variables and arrays are C integers, i.e., 32-bit values.

Example:

```
x = 0
for ( i = 0;  i < 10; i++)
        x = x + a[ i ];
```

```
        sw     $0,x
        sw     $0,i
for:
        lw     $t0,x
        lw     $t1,i
        sll    $t2,$t1,2
        lw     $t2,a($t2)
        addu   $t0,$t0,$t2
        sw     $t0,x
        addiu  $t1,$t1,1
        sw     $t1,i
        slti   $t2,$t1,10
        bne    $t2,$0,for
```

```
(A) y = -y - x;

(B) a[i] = a[i+1] + a[i-1];

(C) if (x > y)
        x = x - y;
    else
        x = y - x;

(D) while ((i & 1) == 0)
         i = i >> 1;

(E) for (i = 1; i < 10; i++)
        a[i] = i + i + 1;

(F)  a[x] = a[a[x]];
```

Problem 2. "MIPS Calisthenics"

Write MIPS code fragments to perform the following simple tasks.

(A) Clear registers t0-t8
(B) Clear memory locations (words) 0x100 to 0x1fc, inclusive
(C) Swap the contents of registers $t0 and $t1
(D) Count how many memory locations (words) in the address range from 0x100 to 0x1fc have a contents of 0

Problem 3. "Stack Detective"

Consider the following *recursive* C function to compute the n^{th} Fibonacci number.

```
int fib(int n) {
    if (n < 2)
        return n;
    else
        return fib(n-1) + fib(n-2);
}
```

After compiling, the following assembly code is generated:

```
fib:    addi  $sp,$sp,-12
L01:    sw    $ra,8($sp)
L02:    sw    $a0,4($sp)
L03:    slti  $t0,$a0,2
L04:    beq   $t0,$0,L07
L05:    add   $v0,$0,$a0
L06:    beq   $0,$0,L15
L07:    addi  $a0,$a0,-1
L08:    jal   fib
L09:    sw    $v0,($sp)
L10:    lw    $a0,4($sp)
L11:    addi  $a0,$a0,-2
L12:    jal   fib
L13:    lw    $t0,($sp)
L14:    addu  $v0,$v0,$t0
L15:    lw    $ra,8($sp)
L16:    addi  $sp,$sp,12
L17:    jr     $ra
```

a) Explain how each of the 3 words allocated on the stack are used? Could this number be reduced? If so, explain how, if not explain why.

b) Suppose that the statement labeled `L09` was replaced with `add $a1,$0,$v0`, and the two statements labeled `L13` and `L14` were replaced with the single statement `add $v0,$v0,$a1`. Would the resulting fragment still work? Explain.

c) Write an iterative version of the fib() function based on the following Fibonacci code fragment:

```
int x, y;

main() {
  x = 0;
  y = 1;
  while (y < 100) {
    int t = x;
    x = y;
    y = t + y;
  }
}
```

d) Discuss the differences between your iterative fib() implementation and the given recursive one. Which is faster? Shorter? Uses less memory? Easier to understand?

Suppose that at some point during the execution of the given recursive `fib()` function the computer is interrupted and the stack is examined and found to contain the following:

Memory Address	Memory Contents
0x7fffefec	0x00380008
0x7fffefe8	0x00030002
0x7fffefe4	0x000000f1
0x7fffefe0	0x0040007c
0x7fffefdc	0x00000007
0x7fffefd8	0x5f36c89e
0x7fffefd4	0x00400048
0x7fffefd0	0x00000006
0x7fffefcc	0x8d197d50
0x7fffefc8	0x00400048
0x7fffefc4	0x00000005
0x7fffefc0	0xb89f3675
0x7fffefbc	0x00400048
0x7fffefb8	0x00000004
0x7fffefb4	0x0941c475
0x7fffefb0	0x00400048
0x7fffefac	0x00000003
0x7fffefa8	0xeb3ee605
0x7fffefa4	0x00400048
0x7fffefa0	0x00000002
0x7fffef9c	0x00000001
0x7fffef98	0x00400058
0x7fffef94	0x00000001
$sp ® 0x7fffef90	0x5c4ee709

If you use the MIPS simulator/assembler as an aid in answering the following questions (which might be a good idea, though it is not necessary), you need to be aware of the following caveat. The simulator assumes that all of memory, outside the loaded .text and .data segments is filled with zeros. In reality, this is usually not the case. Upon power up, memory locations are filled with apparently random values, and over the lifetime of a program the uninitialized values on the stack reflect the activation records of previously called procedures. Therefore, you need to consider that some of the values shown in the above stack dump may reflect uninitialized memory locations.

e) At what memory address can the function `fib()` be found?

f) What argument (value of *n*) was passed to the originating call?

g) What is the label of the last executed instruction before the machine was interrupted?

h) What would have been the lowest memory location referenced by the stack pointer during this particular invocation?