

CSSE3001/7000 2007 – Tutorial 2

2.5 [15] <§2.5> Consider the following code used to implement the instruction:

```
sllv $s0, $s1, $s2
```

which uses the least significant 5 bits of the value in register `$s2` to specify the amount register `$s1` should be shifted left:

```
.data
mask: .word 0xfffff83f
.text
start: lw $t0, mask
      lw $s0, shifter
      and $s0, $s0, $t0
      andi $s2, $s2, 0x1f
      sll $s2, $s2, 6
      or $s0, $s0, $s2
      sw $s0, shifter
shifter: sll $s0, $s1, 0
```

Add comments to the code and write a paragraph describing how it works. Note that the two `lw` instructions are pseudo-instructions that use a label to specify a memory address that contains the word of data to be loaded. Why do you suppose that writing “self-modifying code” such as this is a bad idea (and oftentimes not actually allowed)?

2.9 [12] <§2.5> Implement the following lines of C code in MIPS:

```
int a = 27;
struct
{
    unsigned int data0 : 8;
    unsigned int data1 : 8;
    unsigned int data2 : 8;
    unsigned int valid : 1;
} bits;

bits.data0 = a;
bits.data1 = bits.data0;
bits.data2 = 'd';
bits.valid = 1;
```

Assume that the address of `struct bits` is in `$s0` and the memory address of `a` is stored in `$s1`. (For the less C-proficient, the `struct` defines 25 contiguous bits, the first 8 bits called `data0`, etc., as shown:

struct (25 bits) =

valid (1 bit)	data2 (8 bits)	data1 (8 bits)	data0 (8 bits)
---------------	----------------	----------------	----------------

Also, the ASCII value of `d` is decimal 100.)

2.16 [30] <§2.7> Write a MIPS procedure to compute the n th Fibonacci number $F(n)$ where:

$$F(n) = \begin{cases} 0, & \text{if } n = 0; \\ 1, & \text{if } n = 1; \\ F(n-1) + F(n-2), & \text{otherwise.} \end{cases}$$

Base your algorithm on the straightforward but hopelessly inefficient recursive procedure below:

```
int fib(int n)
{
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

2.17 [30] <§2.7> Write a program as in Exercise 2.16, except this time base your program on the following procedure and optimize the tail call so as to make your implementation efficient:

```
int fib_iter (int a, int b, int count)
{
    if (count == 0)
        return b;
    else
        return fib_iter(a + b, a, count - 1);
}
```

Here, the first two parameters keep track of the previous two Fibonacci numbers computed. To compute $F(n)$, you have to make the procedure call `fib_iter(1, 0, n)`.

Note: tail recursion is when the recursion is in the last executable statement, and need not actually return. Eliminating tail recursion is a key optimisation in making functional languages like Scheme viable. See end of the tutorial for an example.

2.18 [20] <§2.7> Estimate the difference in performance between your solution to Exercise 2.16 and your solution to Exercise 2.17.

2.27 Assume $\$s3 = i$, $\$s4 = j$, $\$s5 = @A$ (i.e., the address of array A). Below is the MIPS code:

```
Loop: addi $s4,$s4,1      # j = j + 1?
      add  $t1,$s3,$s3   # $t1 = 2 * i
      add  $t1,$t1,$t1   # $t1 = 4 * i
      add  $t1,$t1,$s5   # $t1 = @ A[i]
      lw   $t0,0($t1)   # $t0 = A[i]
      addi $s3,$s3,4    # i = i + 1?
      slti $t1,$t0,10   # $t1 = $t0 < 10?
      beq  $t1,$0, Loop # goto Loop if >=
      slti $t1,$t0, 0   # $t1 = $t0 < 0?
      bne  $t1,$0, Loop # goto Loop if <
```

Below is part of the corresponding C code:

```
do j = j + 1
while ( _____ );
```

What C code properly fills in the blank condition in the C `while` loop?

- A: `[i++] >= 10?`
- B: `A[i++] >= 10 | A[i] < 0?`
- C: `A[i++] >= 10 & A[i] < 0?`
- D: `A[i++] >= 10 || A[i] < 0?`
- E: `A[i++] >= 10 && A[i] < 0?`
- F: None of the above

2.30 [12] <§2.3, 2.6, 2.9> The following code fragment processes two arrays and produces an important value in register `$v0`. Assume that each array consists of 2500 words indexed 0 through 2499, that the base addresses of the arrays are stored in `$a0` and `$a1` respectively, and their sizes (2500) are stored in `$a2` and `$a3`, respectively. Add comments to the code and describe in one sentence what this code does. Specifically, what will be returned in `$v0`?

```
                sll  $a2, $a2, 2
                sll  $a3, $a3, 2
                add  $v0, $zero, $zero
                add  $t0, $zero, $zero
outer:          add  $t4, $a0, $t0
                lw   $t4, 0($t4)
                add  $t1, $zero, $zero
inner:          add  $t3, $a1, $t1
                lw   $t3, 0($t3)
                bne  $t3, $t4, skip
                addi $v0, $v0, 1
skip:          addi $t1, $t1, 4
                bne  $t1, $a3, inner
                addi $t0, $t0, 4
                bne  $t0, $a2, outer
```

2.31 [10] <§2.3, 2.6, 2.9> Assume that the code from Exercise 2.30 is run on a machine with a 2 GHz clock that requires the following number of cycles for each instruction:

Instruction	Cycles
<code>add, addi, sll</code>	1
<code>lw, bne</code>	2

In the worst case, how many seconds will it take to execute this code?

2.37 [25] <§2.10> As discussed on page 107 (Section 2.10, “Assembler”), pseudo-instructions are not part of the MIPS instruction set but often appear in MIPS programs. For each pseudo-instruction in the following table, produce a minimal sequence of actual MIPS instructions to accomplish the same thing. You may need to use `$at` for some of the sequences. In the following table, `big` refers to a specific

number that requires 32 bits to represent and `small` to a number that can fit in 16 bits.

Pseudoinstruction	What it accomplishes
<code>move \$t1, \$t2</code>	<code>\$t1 = \$t2</code>
<code>clear \$t0</code>	<code>\$t0 = 0</code>
<code>li \$t1, small</code>	<code>\$t1 = small</code>
<code>li \$t2, big</code>	<code>\$t2 = big</code>
<code>beq \$t1, small, L</code>	if (<code>\$t1=small</code>) go to L
<code>beq \$t2, big, L</code>	if (<code>\$t2=big</code>) go to L
<code>ble \$t3, \$t5, L</code>	if (<code>\$t3 <= \$t5</code>) go to L
<code>bgt \$t4, \$t5, L</code>	if (<code>\$t4 > \$t5</code>) go to L
<code>bge \$t5, \$t3, L</code>	if (<code>\$t5 >= \$t3</code>) go to L
<code>addi \$t0, \$t2, big</code>	<code>\$t0 = \$t2 + big</code>
<code>lw \$t5, big(\$t2)</code>	<code>\$t5 = Memory [\$t2+big]</code>

Tail recursion example:

Some recursive procedures can be implemented iteratively without using recursion. Iteration can significantly improve performance by removing the overhead associated with procedure calls. For example, consider a procedure used to accumulate a sum:

```
int sum (int n, int acc) {
    if (n > 0)
        return sum(n - 1, acc + n);
    else
        return acc;
}
```

Consider the procedure call `sum(3, 0)`. This will result in recursive calls to `sum(2, 3)`, `sum(1, 5)`, and `sum(0, 6)`, and then the result 6 will be returned four times. This recursive call of `sum` is referred to as a tail call, and this example use of tail recursion can be implemented very efficiently (assume `$a0 = n` and `$a1 = acc`):

```
sum: beq $a0, $zero, sum_exit #go to sum_exit if n is 0
     add $a1, $a1, $a0        #add n to acc
     addi $a0, $a0, -1       #subtract 1 from n
     j    sum                 #go to sum
sum_exit:
     move $v0, $a1           #return value acc
     jr   $ra                #return to caller
```
