CS232 Midterm Exam 1
February 24, 2003

Name: Ozzy Osbourne

- This exam has 6 pages, including this cover and the cheat sheet on the next page.
- You have 50 minutes; budget your time!
- No written references or calculators are allowed.
- To make sure you receive full credit, please write clearly and show your work.
- We will not answer questions regarding course material.

Question	Maximum	Your Score
1	30	
2	30	
3	40	
Total	100	

MIPS Instructions

These are some of the most common MIPS instructions and pseudo-instructions, and should be all you need. However, you are free to use *any* valid MIPS instructions or pseudo-instruction in your programs.

Category	Example Instruction	Meaning

Arithmetic	add $t0, $t1, $t2
	sub $t0, $t1, $t2
	addi $t0, $t1, 100
	mul $t0, $t1, $t2
	div $t0, $t1, $t2
	rem $t0, $t1, $t2
		$t0 = $t1 + $t2
		$t0 = $t1 – $t2
		$t0 = $t1 + 100
		$t0 = $t1 × $t2
		$t0 = $t1 / $t2
		$t0 = $t1 mod $t2

Register Setting	move $t0, $t1 li $t0, 100	$t0 = $t1 $t0 = 100
Data Transfer	lw $t0, 100($t1) sw $t0, 100($t1)	$t0 = Mem[100 + $t1] Mem[100 + $t1] = $t0

Branch	beq $t0, $t1, Label
	bne $t0, $t1, Label
	bge $t0, $t1, Label
	bgt $t0, $t1, Label
	ble $t0, $t1, Label
	blt $t0, $t1, Label
		if ($t0 = $t1) go to Label
		if ($t0 ≠ $t1) go to Label
		if ($t0 ≥ $t1) go to Label
		if ($t0 > $t1) go to Label
		if ($t0 ≤ $t1) go to Label
		if ($t0 < $t1) go to Label

Set	slt $t0, $t1, $t2 slti $t0, $t1, 100	if ($t1 < $t2) then $t0 = 1 else $t0 = 0 if ($t1 < 100) then $t0 = 1 else $t0 = 0
Jump	j Label jr $ra jal Label	go to Label go to address in $ra $ra = PC + 4; go to Label

The second source operand of the arithmetic and branch instructions may be a constant.

Register Conventions

The *caller* is responsible for saving any of the following registers that it needs, before invoking a function:

$t0-$t9 $a0-$a3 $v0-$v1

The *callee* is responsible for saving and restoring any of the following registers that it uses:

$s0-$s7 $ra

Performance

Formula for computing the CPU time of a program *P* running on a machine *X*:

$$CPU\ time_{X,P} = Number\ of\ instructions\ executed_P \times CPI_{X,P} \times Clock\ cycle\ time_X$$

CPI is the average number of clock cycles per instruction, or just:

$$CPI = Number\ of\ cycles\ needed / Number\ of\ instructions\ executed$$

Question 1: Understanding MIPS programs (30 points)

```
flathead:
      addi  $t0, $a2, 1
loop:
      bge   $t0, $a1, exit
      mul   $t1, $t0, 4
      add   $t1, $t1, $a0
      lw    $t2, 0($t1)
      sub   $t1, $t1, 4
      sw    $t2, 0($t1)
      addi  $t0, $t0, 1
      j     loop
exit:
      jr    $ra
```

Part (a)

Translate the *flathead* function above into a high-level language like C or Java. You should include a header that lists the types of any arguments and return values. Also, your code should be as concise as possible, without any gotos or explicit pointers. We will not deduct points for syntax errors unless they are significant enough to alter the meaning of your code. (20 points)

This function does stuff with $a0, $a1 and $a2, which indicates that there are three arguments. Multiplying ($a2+1) by 4 suggests that $a2 is an index, and adding that to $a0 for a "lw" means $a0 is an array of 32-bit values (most likely integers or floating-point numbers). The loop exits when $t0 ≥ $a1, suggesting that $a1 is some kind of maximum index. Since nothing is saved in $v0 before the "jr," there is no return value.

The problem says not to use gotos, so you should have translated the loop instructions into a higher-level "for" or "while" structure. You should also show array manipulations, instead of pointer arithmetic and dereferences.

```
void flathead(int a0[], int a1, int a2)
{
     int t0 = a2 + 1;

     while (t0 < a1) {
          a0[t0-1] = a0[t0];
          t0++;
     }
}
```

Part (b)

Describe briefly, in English, what this function does. (10 points)

This function takes an array $a0 *and shifts elements* $a2+1 *through* $a1 *into positions* $a2 *through* $a1–1. *In other words it shifts those elements "downward" or "leftward."*

You can also think of this as deleting element $a2.

Question 2: Performance (30 points)

Below is a C function that returns the index of the smallest element in an integer array *V[]*, which contains *n* items. A translation of this function into MIPS assembly language is shown to the right. Registers $a0 and $a1 correspond to *V[]* and *n*. The index of the minimal element is placed in $v0 as the return value.

```
int minimum(int V[], int n)
{
     int min, i;

     min = V[0];
     for (i = 1; i < n; i++)
          if (V[i] < min)
             min = V[i];
     return min;
}
```

```
minimum:
        lw    $t0, 0($a0)      # min = V[0]
        addi  $t1, $0, 1       # i = 1
loop:
        bge   $t1, $a1, exit   # i >= n ?
        mul   $t2, $t1, 4
        add   $t2, $t2, $a0
        lw    $t2, 0($t2)      # $t2 = V[i]
        bge   $t2, $t0, next   # V[i] >= min ?
        add   $t0, $t2, $0     # min = V[i]
next:
        addi  $t1, $t1, 1      # i++
        j     loop
exit:
        add   $v0, $t0, $0     # return min
        jr    $ra
```

Say that we run this program on a 500MHz processor, with a clock cycle time of 2ns. The CPIs for different types of MIPS instructions are given below; you can assume `mul` and `bge` each count as one instruction.

Instruction type	CPI
adds	4
mul	10
loads	5
branches and jumps	3

Part (a)

Assume *minimum* is passed an array with the numbers from 101 to 1 in descending order (101, 100, 99, …, 3, 2, 1). What would be the exact CPU time for the function call, in nanoseconds? (15 points)

On the right we've shown the number of cycles required for each instruction. For an input array with 101 elements, the loop body would execute 100 times. Furthermore, since the array elements are in descending order, `add $t0, $t2, $0` *will be executed every iteration. In other words, each line of the loop will execute 100 times. The* `bge` *at the top of the loop will fail on the 101st try, and the function will exit.*

```
minimum:
        lw    $t0, 0($a0)        5
        addi  $t1, $0, 1         4
loop:
        bge   $t1, $a1, exit     3
        mul   $t2, $t1, 4       10
        add   $t2, $t2, $a0      4
        lw    $t2, 0($t2)        5
        bge   $t2, $t0, next     3
        add   $t0, $t2, $0       4
next:
        addi  $t1, $t1, 1        4
        j     loop               3
exit:
        add   $v0, $t0, $0       4
        jr    $ra                3
```

The total number of cycles required works out to be:

$(5 + 4) + 100(3 + 10 + 4 + 5 + 3 + 4 + 4 + 3) + (3 + 4 + 3)$
$= 9 + 100(36) + 10$
$= 3619$ *cycles*

With a 2ns clock cycle time, this is 7238ns.

Question 2 continued

Part (b)
It is possible to replace `mul $t2, $t1, 4` by the following two instructions.

```
add  $t2, $t1, $t1
add  $t2, $t2, $t2
```

What would be the exact CPU time in nanoseconds if we made this modification to the original code, and called the *minimum* function with the same input array 101, 100, 99, …, 3, 2, 1? (10 points)

The original mul instruction required 10 clock cycles, and we propose to replace that with two adds that take 4 cycles each. This results in each loop iteration taking 2 cycles less than before; for the same input array, the loop body executes 100 times and we save $2 \times 100 = 200$ *cycles.*

The CPU time works out to be 3419 *cycles* × 2 *ns / cycle* = 6838 *ns.*

Part (c)
What is the performance increase or decrease of the modified code in Part (b) compared to the original? You may leave your answer as a fraction. (5 points)

The program in Part (b) would be faster, by 7238 / 6838 times. (This works out to about 1.06 times.)

Question 3: Writing a nested function (40 points)

Here is a C function *count*. Its arguments *A[]* and *n* are an integer array and the number of items in the array. The code passes each element of *A[]* to another function *test,* and counts the number of times *test* returns 1.

```
int count(int A[], int n)
{
     int i;
     int num = 0;

     for (i = 0; i < n; i++) {
          if (test(A[i]) == 1) {
               num++;
          }
     }
     return num;
}
```

Translate this into a MIPS assembly language function. Argument registers $a0 and $a1 will correspond to *A[]* and *n*, and the return value should be placed in $v0 as usual.

- Assume that we already have the MIPS function *test*, which takes an integer argument in $a0 and returns 0 or 1 in $v0.
- You will not be graded on the efficiency of your code, but you *must* follow all MIPS conventions.

The hard part here is preserving registers correctly. The count *function acts as both a caller and callee. It has to save all the callee-saved registers that it uses and restore them before returning, but it is also responsible for preserving any caller-saved registers that it needs across the call to* test. *Regardless of whether you use the caller-saved registers, callee-saved registers or a combination of them, you'll have to push stuff onto the stack.*

One example implementation is shown on the next page. It uses caller-saved registers for the temporary values i *and* num, *so those must be saved and restored before and after the call to* test. *In addition the base address of the array and the array size, which are passed as arguments $a0 and $a1, must also be saved. Finally, recall that the callee-saved register $ra has to be preserved because of the nested* jal.

Question 3 continued

```
count:
      subu  $sp, $sp, 20
      sw    $ra, 0($sp)       # Save $ra before jal
      li    $t0, 0            # $t0 = i
      li    $t1, 0            # $t1 = num
loop:
      bge   $t0, $a1, exit    # while (i < n)
      sw    $t0, 4($sp)       # Save caller-saved registers
      sw    $t1, 8($sp)       #   that we need
      sw    $a0, 12($sp)
      sw    $a1, 16($sp)
      mul   $t2, $t0, 4       # We don't need to save $t2
      addu  $t2, $a0, $t2
      lw    $a0, 0($t2)       # $a0 = A[i]
      jal   testfunc          # $v0 = testfunc(A[i])
      lw    $t0, 4($sp)       # Restore saved registers
      lw    $t1, 8($sp)
      lw    $a0, 12($sp)
      lw    $a1, 16($sp)
      beq   $v0, $0, next
      addi  $t1, $t1, 1       # num++
next:
      addi  $t0, $t0, 1       # i++
      j     loop
exit:
      move  $v0, $t1          # return num
      lw    $ra, 0($sp)       # Restore $ra
      addiu $sp, $sp, 20
      jr    $ra
```