CS232 Exam 1
September 28, 2007

Name: **Lou Piniella**

Section: noon   2pm (1214)   2pm (1103)   4pm   (circle one)

- This exam has 6 pages, including this cover.
- There are three questions, worth a total of 100 points.
- The last two pages are a summary of the MIPS instruction set, calling convention, and hexadecimal notation, which you may remove for your convenience.
- No other written references or calculators are allowed.
- Write clearly and show your work. State any assumptions you make.
- You have 50 minutes. Budget your time, and good luck!

Question	Maximum	Your Score
1	40	
2	20	
3	40	
Total	100	

Question 1: Write a recursive MIPS function (40 points)

The fundamental theorem of arithmetic states that every natural number greater than 1 can be written as a unique product of prime numbers. The number of prime factors of a number can be computed by the recursive function `count_factors` by passing 2 as the second argument. For example, the number 20 has prime factors (2, 2, 5) so `count_factors(20,2)` will return 3. Translate `count_factors` into a **recursive** MIPS assembly language function; iterative versions (*i.e.,* those with loops) will not receive full credit. You will not be graded on the efficiency of your code, but you must follow all MIPS conventions. Comment your code!!!

Note: pseudo instructions exist for the modulo (%) and divison (/) operations. See the reference at the end of the test. You can assume these work for unsigned integers.

```
unsigned
count_factors(unsigned num, unsigned factor) {
  if (num == 1) {
     return 0;
  }
  if ((num % factor) == 0) {
     return 1 + count_factors(num/factor, factor);
  }
  return count_factors(num, factor+1);
}
```

```
count_factors:
        bne     $a0, 1, cf_recurse        # if (num == 1)
        move    $v0, $0                   # return 0;
        jr      $ra

cf_recurse:
        sub     $sp, $sp, 4
        sw      $ra, 0($sp)

        rem     $t0, $a0, $a1             # if ((num % factor) == 0)
        bne     $t0, $0, cf_recurse1

        div     $a0, $a0, $a1
        jal     count_factors             # count_factors(num / factor, factor)
        addi    $v0, $v0, 1               # + 1
        j       cf_end

cf_recurse1:
        addi    $a1, $a1, 1
        jal     count_factors             # count_factors(num, factor + 1)

cf_end:
        lw      $ra, 0($sp)
        add     $sp, $sp, 4
        jr      $ra
```

Question 2: Concepts (20 points)

Write a short answer to the following questions. For full credit, answers should not be longer than **two sentences**.

Part a) Write the body of a function that takes an integer "in" and returns the same value with the "n"th bit inverted (*i.e.,* 0->1, 1->0). This can be done using just bit-wise logical operations and shifts. No control flow (*e.g.,* loops/ifs) are needed. (10 points)

Example: flip_nth_bit(0x04F2, 7)→ 0x0472

0 4 F 2
0000 0100 **1**111 0010

0000 0100 **0**111 0010
0 4 7 2

```
unsigned
flip_nth_bit(unsigned in, unsigned n) {
    return in ^ (1<<n);
          -- or --
    li    $t0, 1
    sll   $t0, $t0, $a1
    xor   $v0, $t0, $a0
    jr    $ra
}
```

Part b) What is an abstraction layer? How does it relate to instruction set architectures (ISAs)? (5 points)

An abstraction layer separates an interface from its implementation.

ISAs are an example of an abstraction layer because they describe the interface to hardware, without details of the implementation. This idea enables binary compatibility across generations of hardware.

Part c) Why does MIPS have multiple instruction formats (*e.g.,* R-type, I-type, J-type)? What is the motivation for including each format ? (5 points)

There are a limited number of bits in the instruction; the different instruction formats spend them differently.

R-types are used when instructions have two sources and a destination register (allocate the remainder of bits for extra opcode space: the func field). I-type are used when two registers and an immediate are needed. J-type enable using almost all of the instruction bits for an immediate.

Question 3: Understanding MIPS programs (40 points)

```
foo:        li      $t0, 1
            li      $v0, 0
A:          bge     $t0, $a1, B
            sll     $t1, $t0, 2
            add     $t1, $t1, $a0
            lw      $t2, 0($t1)
            lw      $t3, -4($t1)
            ble     $t2, $t3, C
            add     $t0, $t0, 1
            j       A
B:          li      $v0, 1
C:          jr      $ra
```

Part (a)

Translate the function `foo` above into a high-level language like C. Your function header should list the types of any arguments and return values. Also, your code should be as concise as possible, without any gotos or pointer arithmetic. We will not deduct points for syntax errors unless they are significant enough to alter the meaning of your code. (30 points)

```
int
foo(int *arr, int n) {
   for (int i = 1; i < n; i++) {
       if (arr[i] <= arr[i - 1]) {
           return 0;
       }
   }
   return 1;
}
```

Part (b)

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

foo takes an int array $a0 of length $a1 and returns true if the integers in the array are in ascending order.

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
	rem $t0, $t1, $t2
	div $t0, $t1, $t2
		$t0 = $t1 + $t2
		$t0 = $t1 – $t2
		$t0 = $t1 % $t2
		$t0 = $t1 / $t2

Logical	and $t0, $t1, $t2 or $t0, $t1, $t2 sll $t0, $t1, $t2 srl $t0, $t1, $t2 sra $t0, $t1, $t2	$t0 = $t1 & $t2 (Logical AND) $t0 = $t1 | $t2 (Logical OR) $t0 = $t1 << $t2 (Shift Left Logical) $t0 = $t1 >> $t2 (Shift Right Logical) $t0 = $t1 >> $t2 (Shift Right Arithmetic)
Register Setting	move $t0, $t1 li $t0, 100	$t0 = $t1 $t0 = 100

Data Transfer	lw $t0, 100($t1)
	lb $t0, 100($t1)
	sw $t0, 100($t1)
	sb $t0, 100($t1)
		$t0 = Mem[100 + $t1] 4 bytes
		$t0 = Mem[100 + $t1] 1 byte
		Mem[100 + $t1] = $t0 4 bytes
		Mem[100 + $t1] = $t0 1 byte

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, logical, 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 $s8/$fp $sp $ra

Pointers in C:
Declarartion: either *char *char_ptr* -or- *char char_array[]* for *char c*
Dereference: *c = c_array[i]* -or- *c =*c_pointer*
Take address of: *c_pointer = &c*

Hexadecimal Notation

C and languages with a similar syntax (such as C++, C# and Java) prefix hexadecimal numerals with "0x", e.g. "0x5A3". The leading "0" is used so that the parser can simply recognize a number, and the "x" stands for hexadecimal.

Hex	Bin	Dec
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15