University of Pittsburgh

CS/COE 447 Spring 2010 Exam 2

There are a total of 100 points. You are allowed to use the Green Card (or a copy of it) that comes with the text. Also, two figures from Appendix C (C.5.10 and C.5.11) have been given to you as a separate handout.

We can't answer questions like *What do you want for this question?* or *I don't understand this question.* It makes the room too loud, and it isn't fair, since some people would get extra information. Please just use your best judgment.

Show your work for partial credit.

Each question is on its own page, to give you plenty of room. Don't feel you need to fill up each page; just write what you need to.

Good luck!!

1. (9 points)

   (a) Translate -33 decimal (minus 33) into an 8-bit two's complement binary integer
   0xFFDF 11011111

   (b) Viewing 11110010 as an 8-bit two's complement integer, what decimal value does it represent?
   -14

   (c) Viewing 10000011 as an 8-bit unsigned binary integer, what decimal value does it represent?
   $128 + 3 = 131$

2. (10 points) Viewing the following as 8-bit two's complement binary integers, perform the indicated operation. What decimal values are represented by the operands and results? Is there signed overflow? Please explain.

   Please show your work.

```
          11111100          decimal value:  -4


         +11110000          decimal value: -16
         ----------
          11101100
result:                     decimal value: -20


Signed overflow? Yes/No  Explanation:
```

   No overflow. In 8 bits, the range of numbers is -128 to + 127. -20 is within this range, so it fits in 8 bits. Also, you added two negative numbers and ended up with a negative number, which indicates no signed overflow (remembering that you throw away the carry when adding 2 two's complement numbers).

3. (12 points) Represent 19.75 decimal in the IEEE 754 single-precision floating point format. **Please show your work**. Give your answer in **binary AND in hex.**

```
19.75 = 10011.11
normalized = 1.001111 * 2E4
bias = 127 + 4 = 10000011
mantissa=0011110000...

binary:
 0 10000011 00111100000000....
 0100 0001 1001 1110 00000000....
Hex:
 0x419e0000
```

4. (12 points) For each of the following, state whether there is overflow, and **briefly explain your answers.**

```
   li $t0,0x10000000
   addiu $t0,$t0,-1
Overflow? ____yes______  Explanation:
addiu is an unsigned operation.
The unsigned numbers being added are:
   0x10000000
   0xFFFFFFFF

Since the second one is the largest unsigned number, adding anything
to it results in unsigned overflow.

   li $t0,0x10000000
   addi $t0,$t0,0xFFFFFFFF
Overflow? __No______  Explanation:

addi is a signed operation.

0x10000000 is a positive number
0xFFFFFFFF is -1
When you add a positive and a negative number, you cannot get signed overflow.

   li $t0,0x80000000
   addi $t0,$t0,0xFFFFFFFF
Overflow? ____yes_____  Explanation:
Again, addi is a signed number.

0x80000000 is the smallest negative number (the one with the greatest
absolute value), namely -2E31.

0xFFFFFFFF is a negative number (in fact, it is -1).  If you add a
negative number to the smallest negative number, the result doesn't fit.
```

5. (7 points) For some of the source instructions below, the assembler must create more than one primative MIPS instruction.

   (a) Briefly state why, in such cases, more than one instruction is needed.

   The immediate field within an addi instruction is only 16 bits. The assembler can use a single instruction only if there are no 1s in bits 15 through 31 for positive numbers, and if there are no 0s in bits 15 through 31 for negative numbers.

   The special cases are a 0 in bit 15 for negative numbers, and a 1 in bit 15 for positive numbers. For addi (and even addiu), the hardware sign extends the 15th bit. In these cases, the sign extension would be wrong.

   (b) Put a checkmark next to the instructions for which the assembler must create more than one primative MIPS instruction.

```
addi $t4,$zero,0xffff7fff  Yes - bit 15 is a 0, yet the number is
   negative.  Sign extension would be all 0s

addi $t6,$zero,0xffff8000  No - this is a negative number, and there
   are no zeros in bits 15 through 31.

addi $t1,$zero,0x00010000 Yes - this is a positive number and there is a 1 in bit

addi $t5,$zero,0xfffeffff Yes - this is a negative number and there is a 0 in bit

addi $t3,$zero,0x00007fff No - this is positive and bits 15 through 31
    are 0

addi $t1,$zero,0x0000ffff Yes - this is positive and there is a 1 in
    bit 15
```

6. (12 points) Trace Booth's algorithm on the values **11** * −**13**. Specifically, fill in the form below, which is the same format as on Lab 10.

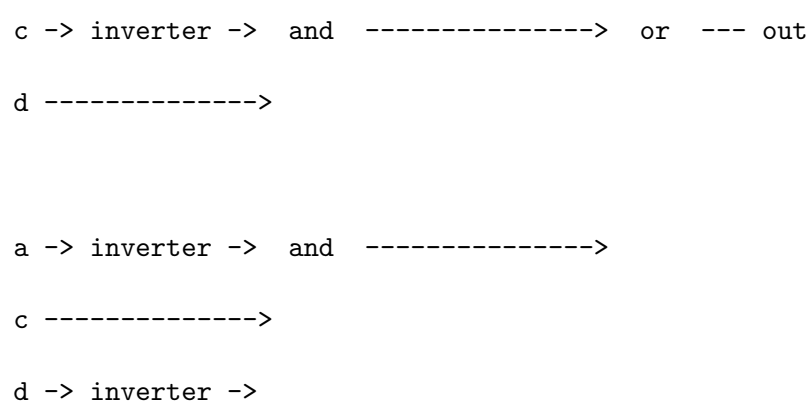
```
N=8 (8 bit numbers)
M=11=00001011     -M=11110101
R=-13=11110011

Note that multiplication of two N-bit binary numbers results in a
2N-bit product

S(step)    M(multiplicand)   P product                Notes
-------------------------------------------------------------------
                                                 Initialize Values
           00001011      0000000001110011 0
-------------------------------------------------------------------
8         same for       11110101 11110011 0        sub
      rest of column     11111010 11111001 1        shift
-------------------------------------------------------------------
7                        11111101 01111100 1        shift only

-------------------------------------------------------------------
6                        0000 1000 0111 1100 1      add
                         0000 0100 0011 1110 0      shift
-------------------------------------------------------------------
5                        0000 0010 0001 1111 0      shift only

-------------------------------------------------------------------
4                        1111 0111 0001 1111 0      sub
                         1111 1011 1000 1111 1      shift
-------------------------------------------------------------------
3                        1111 1101 1100 0111 1       shift only

-------------------------------------------------------------------
2
                         1111 1110 1110 0011 1      shift only
-------------------------------------------------------------------
1
                         1111 1111 0111 0001 1      shift only
-------------------------------------------------------------------
0
                         1111 1111 0111 0001       extract the product
-------------------------------------------------------------------
```

7. (8 points) Consider the boolean equation **out = C'D + A'CD'**

   (a) Give the truth-table representation for this boolean equation.

ACD	out
000	0
001	1
010	1
011	0
100	0
101	1
110	0
111	0

   (b) Using AND, OR, and NOT gates, draw the circuit for the above boolean equation.



Or, you could ‘‘read’’ the circuit directly from the and gates, and have 3 and gates (corresponding to the three 1's in the out column)

and 1 or gate.

8. (10 points) Refer to Figures C.5.10 and C.5.11, which have been given to you as a separate handout. Suppose our datapath is only 4 bits.

   Suppose we are executing slt $t0,$t1,$t2, and that $t1 = 1101 binary and $t2 = 0001 binary. What values do the following have? Show your work, for partial credit (i.e., write down what is being added to what, and show the carries).

```
Ainvert   0
Binvert   1
Operation  3
CarryIn to ALU0  1
a0  1
b0  1
Output of the adder of ALU0 0
CarryOut of ALU0  1
Result0 1
Output of the adder of ALU1 0
CarryOut of ALU1 1
Result1 0
Output of the adder of ALU2 1
CarryOut of ALU2 1
Result2 0
Output of the adder of ALU3 1
CarryOut of ALU3 1
Result3 0
Set 1
```

9. (20 points) Write a subroutine **reverse** that uses the stack to reverse the values of an array of 1-word integers. The address of the first element of the array is passed to the subroutine in $a0, and the number of elements in the array is passed in $a1. You do **not** need to write a main program to call your subroutine, and you do not need to print the elements of the array.

```
# $a0 - address of the first element of the array
# $a1 - the number of array elements
reverse:
   add $t2,$a0,$zero # $t2 pointer to the array
   add $t0,$zero,$zero # counter - number of elements processed so far
loop1:
   slt $t1,$t0,$a1 # counter < number of elements?
   beq $t1,$zero,endloop1 # no: exit loop
   addi $sp,$sp,-4
   lw $t3,0($t2)  # the next array element
   sw $t3,0($sp)  # push it on the stack
   addi $t2,$t2,4 # point to the next element
   addi $t0,$t0,1 # increment counter
   j loop1
   # now, the elements have all been pushed on the stack, with the
   # last element being on the top of the stack

endloop1:
   add $t2,$a0,$zero # address of the first array element
   add $t0,$zero,$zero # reset counter to 0
loop2:
   slt $t1,$t0,$a1 # counter < number of elements?
   beq $t1,$zero,endloop2 # no: exit loop
   lw $t3,0($sp)  # the element at the top of the stack
   addi $sp,$sp,4 # pop
   sw $t3,0($t2)  # store the element in the array
   addi $t2,$t2,4 # point to the next element
   addi $t0,$t0,1 # increment counter
   j loop2
endloop2:
   jr $ra  # return from the subroutine
```

10. (6 BONUS POINTS) This is a BONUS question, so you do not need to answer it. It will be graded very strictly.

    Suppose our datapath is only 4 bits.

    Suppose we are executing slt $t0,$t1,$t2.

    In each of the following cases, state whether the correct value for LESS0 (the LESS input to ALU0) is Set or Set', and explain your answer.

    (a) $t1 = 7 and $t2 = 6

    The correct value is set. 7 is not less than 6, so the answer should be LESS0 == 0. There is no overflow, because we are performing positive subtract positive, which is equivalent to positive + negative. Overflow is not possible in this case. When there is no overflow, then set has the correct value.

    (b) $t1 = **7** and $t2 = **−6**

    The correct value is set'. We are performing the operation 7 - (-6), which is equivalent to 7 + 6 == 13. In four bits, the largest two's complement value is 2e3 - 1, or 7. Since the value is 13, there is overflow. When there is overflow, we know the result of the addition operation has the incorrect sign. Thus, since the correct sign should be positive, we know that set will be 1. set' is thus the correct value. We can see this is true, since 7 is NOT less than -6.