Date:_____

Quiz for Chapter 3 Arithmetic for Computers

Not all questions are of equal difficulty. Please review the entire quiz first and then budget your time carefully.

Name:_____

Course:_____

Solutions in RED

1. [9 points] This problem covers 4-bit binary multiplication. Fill in the table for the Product, Multplier and Multiplicand for each step. You need to provide the DESCRIPTION of the step being performed (shift left, shift right, add, no add). The value of M (Multiplicand) is 1011, Q (Multiplier) is initially 1010.

Product	Multiplicand	Multiplier	Description	Step
0000 0000	0000 1011	1010	Initial Values	Step 0
0000 0000	0000 1011	1010	0 => No add	Step 1
0000 0000	0001 0110	1010	Shift left M	Step 2
0000 0000	0001 0110	0101	Shit right Q	Step 3
0001 0110	0001 0110	0101	1 => Add M to Product	Step 4
0001 0110	0010 1100	0101	Shift left M	Step 5
0001 0110	0010 1100	0010	Shift right Q	Step 6
0001 0110	0010 1100	0010	0 => No add	Step 7
0001 0110	0101 1000	0010	Shift left M	Step 8
0001 0110	0101 1000	0001	Shift right Q	Step 9
0110 1110	0101 1000	0001	1 => Add M to Product	Step 10
0110 1110	1011 0000	0001	Shift left M	Step 11
0110 1110	1011 0000	0000	Shift right Q	Step 12
0110 1110	1011 0000	0000	0 => No add	Step 13
0110 1110	0110 0000	0000	Shift left M	Step 14
0110 1110	0110 0000	0000	Shift Right Q	Step 15

2. [6 points] This problem covers floating-point IEEE format.

(a) List four floating-point operations that cause NaN to be created?

Four operations that cause Nan to be created are as follows:

(1) Divide 0 by 0
(2) Multiply 0 by infinity
(3) Any floating point operation involving Nan
(4) Adding infinity to negative infinity

(b) Assuming single precision IEEE 754 format, what decimal number is represent by this word:

```
1 01111101 00100000000000000000000
```

(Hint: remember to use the biased form of the exponent.)

The decimal number

= (+1)* (2^(125-127))*$(1.001)_2$
= (+1)*(0.25)*(0.125)
= 0.03125

3. [12 points] The floating-point format to be used in this problem is an 8-bit IEEE 754 normalized format with 1 sign bit, 4 exponent bits, and 3 mantissa bits. It is identical to the 32-bit and 64-bit formats in terms of the meaning of fields and special encodings. The exponent field employs an excess-7coding. The bit fields in a number are (sign, exponent, mantissa). Assume that we use *unbiased rounding to the nearest even* specified in the IEEE floating point standard.

(a) Encode the following numbers the 8-bit IEEE format:

(1) 0.0011011_{binary}

This number = $1.1011_2 * 2^{-3}$
= $(+1) * (1.1011)_2 * 2^{(4-7)}$
= 0 0100 110 in the 8-bit IEEE 754
(after applying unbiased rounding to the nearest even)

(2) $16.0_{decimal}$

This number = $(1000.0)_2$
= $(1.000)_2 * 2^{(10-7)}$
= 0 1010 000 in the 8-bit IEEE 754

(b) Perform the computation $1.011_{binary} + 0.0011011_{binary}$ showing the correct state of the guard, round and sticky bits. There are three mantissa bits.

```
  1.0110   (The sticky bit is set to 1)
+ 0.0011   (The least two significant bits are the guard and round bits)
-----------------------------------------------------------------------
  1.1001
```

After rounding with sticky bit, the answer then is $(1.101)_2$

(c) Decode the following 8-bit IEEE number into their decimal value: 1 1010 101

The decimal value is (+1)*$(2)^{(10-7)}$*1.625 = 13.0

(d) Decide which number in the following pairs are greater in value (the numbers are in 8-bit IEEE 754 format):

(1) 0 0100 100 and 0 0100 111

The first number = $2^{(8-7)}$*(1.1)2 = 3.0
The second number = $2^{(8-7)}$*(1.111)2 = 3.75

The second number is greater in value

(2) 0 1100 100 and 1 1100 101

The first number = $-2^{(12-7)}*(1.1)2 = -48.0$
The second number = $-2^{(8-7)}*(1.101)2 = -52.0$

The first number is greater in value.

(e) In the 32-bit IEEE format, what is the encoding for negative zero?

The representation of negative zero in 32-bit IEEE format is

`1 00000000 00000000000000000000000`

(f) In the 32-bit IEEE format, what is the encoding for positive infinity?

The representation for positive infinity in 32-bit IEEE format is

`0 11111111 00000000000000000000000`

4. [9 points] The floating-point format to be used in this problem is a normalized format with 1 sign bit, 3 exponent bits, and 4 mantissa bits. The exponent field employs an excess-4 coding. The bit fields in a number are (sign, exponent, mantissa). Assume that we use *unbiased rounding to the nearest even* specified in the IEEE floating point standard.

(a) Encode the following numbers in the above format:

(1) 1.0_{binary}

The encoding for the above number is `0 011 0000`

(2) 0.0011011_{binary}

The encoding for the above number is `0 000 1011`

(b) In one sentence for each, state the purpose of guard, rounding, and sticky bits for floating point arithmetic.

The guard bit is an extra bit that is added at the least significant bit position during an arithmetic operation to prevent loss of significance

The round bit is the second bit that is used during a floating point arithmetic operation on the rightmost bit position to prevent loss of precision during intermediate additions.

The sticky bit keeps record of any 1's that have been shifted on to the right beyond the guard and round bits

(c) Perform rounding on the following **fractional** binary numbers, use the bit positions in *italics* to determine rounding (use the rightmost 3 bits):

(1) Round to positive infinity: $+0.100101\mathit{110}_{binary}$

The result is `+0.100110`

Name:_____

(2) Round to negative infinity: -0.001111001_{binary}

The result is -0.010000

(4) Unbiased to the nearest even: $+0.100101100_{binary}$

The result is $+0.100110$

(5) Unbiased to the nearest even: -0.100100110_{binary}

The result is -0.100100

(d) What is the result of the square root of a negative number?

In the IEEE 754 standard (defined since 1985), the square root of negative number results in an exception being generated by the floating point arithmetic unit

5. [6 points] Prove that Sign Magnitude and One's Complement addition cannot be performed correctly by a single unsigned adder. Prove that a single n-bit unsigned adder performs addition correctly for all pairs of n-bit Two's Complement numbers for n=2. You should ignore overflow concerns and the n+1 carry bit. (For an optional added challenge, prove for n=2 by first proving for all n.)

Take the example of adding -1 to 2. Furthermore, assume that we have 3-bits to represent these numbers in signed magnitude and one's complement.

In sign magnitude representation,

-1 = 101
2 = 010

Adding 101 to 010 using an unsigned adder results in 111 which in signed magnitude representation is -3 (the correct value should be 001)

In one's complement representation,

-1 = 110
2 = 010

Adding 010 to 001 using an unsigned adder results in 111 which in one's complement representation is 0 (the correct value should be 001).

2's complement addition for all pairs of n=2

Operand 1	Operand 2	Result
00 (0)	00 (0)	00 (0)
00 (0)	01 (1)	01 (1)
00 (0)	10 (-2)	10 (-2)
00 (0)	11 (-1)	11 (-1)
01 (1)	01 (1)	10 (-2)

Name:_____

01 (1)	10 (-2)	11 (-1)
01 (1)	11 (-1)	00 (0)
10 (-2)	10 (-2)	00 (0)
10 (-2)	11 (-1)	01 (1)

In the above table, the range of numbers representable through 2 bits are -2 to 1.

6. [4 points] Using 32-bit IEEE 754 single precision floating point with one(1) sign bit, eight (8) exponent bits and twenty three (23) mantissa bits, show the representation of -11/16 (-0.6875).

The representation of -0.6875 is:

```
1 01111110 01100000000000000000000000
```

7. [3 points] What is the smallest positive (not including +0) representable number in 32-bit IEEE 754 single precision floating point? Show the bit encoding and the value in base 10 (fraction or decimal OK).

The smallest positive representable number 32-bit IEEE 754 single precision floating point value is

```
0 00000000 00000000000000000000001
```

Its decimal value is $= (+1) * (2^{(0-127)})*(1.00000000000000000000001)$
$= 5.87747175411e-39$

8. [12 points] Perform the following operations by converting the operands to 2's complement binary numbers and then doing the addition or subtraction shown. Please show all work in binary, operating on 16-bit numbers.

(a) 3 + 12

```
        0000 0000 0000 0011  (3)
+       0000 0000 0000 1100  (12)
-----------------------------------
        0000 0000 0000 1111
```

The decimal value of the result is 15

(b) 13 – 2

```
        0000 0000 0000 1101  (13)
+       1111 1111 1111 1110  (-2)
-----------------------------------
        0000 0000 0000 1011
```

The decimal value of the result is 11 (we ignored the carry here)

(c) 5 – 6

```
        0000 0000 0000 0101 (5)
+       1111 1111 1111 1010 (-6)
-----------------------------------
        1111 1111 1111 1111
```

The decimal value of the result is -1

Name:_____

(d) –7 – (-7)

```
      1111 1111 1111 1001 (-7)
+     0000 0000 0000 0111 (7)
-------------------------------
      0000 0000 0000 0000
```

The decimal value of the result is 0 (we ignored the carry here)

9. [9 points] Consider 2's complement 4-bit signed integer addition and subtraction.

(a) Since the operands can be negative or positive and the operator can be subtraction or addition, there are 8 possible combinations of inputs. For example, a positive number could be added to a negative number, or a negative number could be subtracted from a negative number, etc. For each of them, describe how the overflow can be computed from the sign of the input operands and the carry out and sign of the output. Fill in the table below:

Sign (Input 1)	Sign (Input 2)	Operation	Sign (Output)	Overflow (Y/N)
+	+	+	+	N
+	+	+	-	Y
+	+	-	+	N
+	+	-	-	N
+	-	+	+	N
+	-	+	-	N
+	-	-	+	N
+	-	-	-	Y
-	+	+	+	N
-	+	+	-	N
-	+	-	+	N
-	+	-	-	N
-	-	+	+	Y
-	-	+	-	N
-	-	-	+	N
-	-	-	-	N

(b) Define the WiMPY precision IEEE 754 floating point format to be:

where each 'X' represents one bit. Convert each of the following WiMPY floating point numbers to decimal:

(a) 00000000

$(+1)*(2^{(0-7)})*(1.0) = 0.03125$

(b) 11011010

$(-1)*(2^{(5-7)})*(1.1010) = -0.40625$

(c) 01110000

Name:_____

$(+1)*(2^{(7-7)})(1.0000) = 1.0000$

10. [8 points] This problem covers 4-bit binary unsigned division (similar to Fig. 3.11 in the text). Fill in the table for the Quotient, Divisor and Dividend for each step. You need to provide the DESCRIPTION of the step being performed (shift left, shift right, sub). The value of Divisor is 4 (0100, with additional 0000 bits shown for right shift), Dividend is 6 (initially loaded into the Remainder).

Quotient	Divisor	Remainder	Description	Step
0000	0100 0000	0000 0110	Initial Values	Step 0
0000	0100 0000	1100 0110	Rem = Rem – Div	Step 1
0000	0100 0000	0000 0110	Rem < 0 => +Div, sll Q, Q0 = 0	Step 2
0000	0010 0000	0000 0110	Shift Div to right	Step 3
0000	0010 0000	1110 0110	Rem = Rem – Div	Step 4
0000	0010 0000	0000 0110	Rem < 0 => +Div, sll Q, Q0 = 0	Step 5
0000	0001 0000	0000 0110	Shift Div to right	Step 6
0000	0001 0000	1111 0110	Rem = Rem – Div	Step 7
0000	0001 0000	0000 0110	Rem < 0 => +Div, sll Q, Q0 = 0	Step 8
0000	0000 1000	0000 0110	Shift Div to right	Step 9
0000	0000 1000	1111 1110	Rem = Rem – Div	Step 10
0000	0000 1000	0000 0110	Rem < 0 => +Div, sll Q, Q0 = 0	Step 11
0000	0000 0100	0000 0110	Shift Div to right	Step 12
0000	0000 0100	0000 0010	Rem = Rem – Div	Step 13
0001	0000 0100	0000 0010	Rem > 0 => sll Q, Q0 = 1	Step 14
0001	0000 0010	0000 0010	Shift Div Right	Step 15

11. [8 points] Why is the 2's complement representation used most often? Give an example of overflow when:

(a) 2 positive numbers are added

Assuming a 8-bit 2's complement representation, an overflow occurs when +127 is added to itself:

```
        0111111
+       0111111
-------------------
        1111110
```

Here an overflow occurred due to carry into the sign bit

(b) 2 negative numbers are added

Assuming a 8-bit 2's complement representation, an overflow occurs when -127 is added to itself:

```
        1000001
+       1000001
-------------------
        0000010
```

Here an overflow occurred due to carry out of the sign bit and the sign of the result is different from the signs of either operands.

(c) A-B where B is a negative number

If A = +127 and B = -127, an overflow occurs similar to case (a)

Name:_____

The two's complement system is used because it does not need to examine the sign of each operand before performing any arithmetic operation. Unlike the signed magnitude representation it does not have two different representations for zero

12. [14 points] We're going to look at some ways in which binary arithmetic can be unexpectedly useful. For this problem, all numbers will be 8-bit, signed, and in 2's complement.

(a) For x = 8, compute x & (−x). (& here refers to bitwise-and, and − refers to arithmetic negation.)

```
x       =   00001000
-x      =   11111000
-------------------------
x & (-x) = 00001000
```

(b) For x = 36, compute x & (−x).

```
x    =   00100100
-x   =   11011100
-------------------------
x & (-x) = 00000100
```

(c) Explain what the operation x & (−x) does.

It gets the least significant bit position of x which is set to 1 and raises it to power 2.

(d) In some architectures (such as the PowerPC), there is an instruction `adde rX=rY,rZ`, which performs the following:

```
rX = rY + rZ + CA
```

where `CA` is the carry flag. There is also a negation instruction, `neg rX=rY` which performs:

```
rX = 0 - rY
```

Both `adde` and `neg` set the carry flag. gcc (the GNU C Compiler) will often use these instructions in the following sequence in order to implement a feature of the C language:

```
neg r1=r0
adde r2=r1,r0
```

Explain, simply, what the relationship between `r0` and `r2` is (Hint: `r2` has exactly two possible values), and what C operation it corresponds to. Be sure to show your reasoning.

The only two possible values for r2 are 0 and 1. If r0 = 0, then r2 = 1. If r0 is not 0, then r2 = 0. These two instructions can be used to test the branch condition: if (x == 0) ...