CASE WESTERN RESERVE UNIVERSITY
ECES 318
HW 4

Fall 2015

Due Nov 12, 2015

NAME:

Problem	Scale	Score
1	50	
2	50	
3	50	
4	50	
total	200	

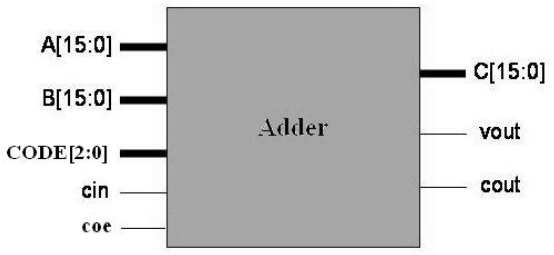
Problem 1. [50 points]

You are asked to design and write a VHDL model and verify its correctness for a circuit that is capable of adding and subtracting signed and unsigned numbers based on a 3-bit input code, carry output enable bit, and input carry bit. The code input will command the adder to perform one of six functions on the two 16-bit word inputs. The adder will output the result of the 16-bit function, a signed overflow bit, and a carry out bit if the carry output enable input bit is set. This circuit will be asynchronous, so it will not contain any memory elements to operate. The details about the input/output signals and control functions are outlined below.

```
TOP MODULE: Adder
 PRIMARY SIGNALS: BUS inputs/outputs and additional signals
      BUS signals: bit 15 is the most significant bit. All
      bus signals are named with upper case letters.
         INPUTS:                    OUTPUTS:
           A[15:0]                     C[15:0]
           B[15:0]
           CODE[2:0]

      Additional signals: named with lower case letters
           vout = signed overflow
           cout = carry output
           cin  = carry input
           coe  = carry output enable (active low)

    3-bit CODE[2:0] control signals
           add   : 000       signed addition      (A+B -> C)
           addu  : 001       unsigned addition    (A+B -> C)
           sub   : 010       signed subtraction   (A-B -> C)
           subu  : 011       unsigned subtraction (A-B -> C)
           inc   : 100       signed increment     (A+1 -> C)
           dec   : 101       signed decrement     (A-1 -> C)
```



```
 Here are the input/output:
  A = A(15:0);       B  = B(15:0)
  i = cin            oe = coe
  f = function       C = C(15:0)
  v = vout           cout = cout
```

A	B	i	oe	f	C	v	cout
0000	0001	0	0	add	0001	0	0
000F	000F	1	0	add	001F	0	0
7F00	0300	0	0	add	8200	1	0
FF00	0100	1	0	add	0001	0	1
8100	8000	1	1	add	0101	1	d
0000	0001	0	0	addu	0001	0	0
000F	000F	1	0	addu	001F	0	0
7F00	0300	0	0	addu	8200	0	0
FF00	0100	1	0	addu	0001	0	1
8100	8000	1	1	addu	0101	0	d
0000	0001	1	0	sub	FFFF	0	0
000F	000F	1	0	sub	0000	0	1
7F00	0300	1	0	sub	7C00	0	1
FF00	0100	1	0	sub	FE00	0	1
8100	8000	1	1	sub	0100	0	d
0000	0001	1	0	subu	FFFF	0	0
FF00	FCE0	1	0	subu	0220	0	1
7F00	0300	1	0	subu	7C00	0	1
FF00	0100	1	0	subu	FE00	0	1
8100	8000	1	1	subu	0100	0	d
0000	0100	0	0	inc	0001	0	0
0F00	0F00	1	0	inc	0F01	0	0
7FFF	0300	0	0	inc	8000	1	0
FF00	0100	1	0	inc	FF01	0	0
8100	8000	1	1	inc	8101	0	d
0000	0100	0	0	dec	FFFF	0	0
000F	000F	1	0	dec	000E	0	1
7F00	0300	0	0	dec	7EFF	0	1
FF00	0100	1	0	dec	FEFF	0	1
8000	8000	1	1	dec	7FFF	1	d

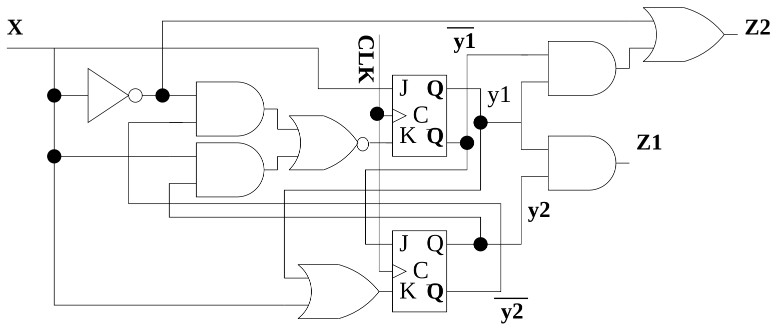
d = don't care

Signed overflow: When two operands of the same sign are added together and the result is of the opposite sign. Be sure that the signed overflow bit vout is only affected by signed operations.
Here is an example of signed overflow. You must determine all the cases when signed overflow will occur.

```
Ex:              01110000      8 bit signed int
                +01000001      8 bit signed int
                 ----------
                 10110001      vout=1
```

Problem 2. [50 points]

Analyze the logic circuit given below.



a) Write a structral VHDL Model for this design and simulate your circuit to verify correctness. (Explain your approach)
b) Write a behavioral VHDL model and verify that your model is equivalent to the given circuit.

Problem 3. [50 points]

Multiplication of signed operands, which generate a double-length product in the 2's-complement number system. The general strategy is the accumulate partial products by adding versions of the multiplicand as selected by the multiplier bits.
First, consider the case of a positive multiplier and a negative multiplicand, When we add a negative multiplicand to a partial product, we must extend the sign-bit value of the multiplicand to the left as far as the product will extend (as shown in the figure below). The sign extension of the multiplicand is hand-written.

					1	0	0	1	1	MULTIPLICAND (-13)
					0	1	0	1	1	MULTIPLIER (+11)
	1	**1**	**1**	**1**	1	0	0	1	1	
	1	**1**	**1**	1	0	0	1	1		
	0	**0**	0	0	0	0	0			
	1	1	0	0	1	1				
	0	0	0	0	0					
1	1	0	1	1	1	0	0	0	1	PRODUCT

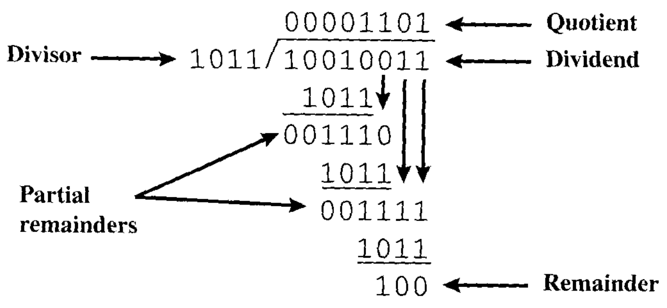
For a negative multiplier, a straightforward solution is to form the 2's complement of both multiplier and multiplicand and proceed as in the first case.
a) Write a VHDL file to model the above behavior for 5 bits multiplier and 5 bits multiplicand.
b) Use Modelsim to compile your VHDL file.
c) Use Modelsim to simulate your multiplier with the following cases: (-11*4, 14*-3, -11*-12)

Problem 4. [50 points]

Figure 1. shows an example of the long division of unsigned binary integers. First, the bits of the dividend are examined from left to right, until the set of bits examined represents a number greater than or equal to the divisor: this is referred to as the divisor being able to divide the number. Until this event occurs, Os are placed in the quotient from left to right. When the event occurs, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend. The result is referred to as a partial remainder. From this point on, the division follows a cyclic pattern. At each cycle, additional bits from the dividend are appended to the partial remainder until the result is greater than or equal to the divisor. As before, the divisor is subtracted from this number to produce a new partial remainder. The process continues until all the bits of the dividend are exhausted.

Figure 1. Example of division of unsigned binary integers.

Figure 2. shows a machine algorithm that corresponds to the long division process. The divisor is placed in the M register, the dividend in the Q register. At each step, the A and Q registers together are shifted to the left 1 bit. M is subtracted from A to determine whether A divides the partial remainder.3 If it does, then Q0 gets a 1 bit. Otherwise, Q0 gets a 0 bit and M must be added back to A to restore the previous value. The count is then decremented, and the process continues for n steps. At the end, the quotient is in the Q register and the remainder is in the A register.



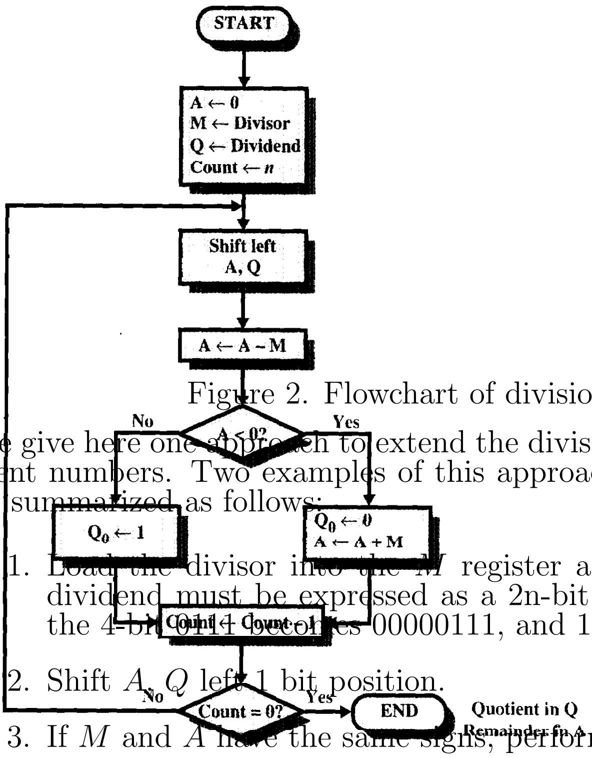


Figure 2. Flowchart of division of unsigned binary integers.

We give here one approach to extend the division algorithm to handle negative twos complement numbers. Two examples of this approach are shown in Figure 3. The algorithm can be summarized as follows:

1. Load the divisor into the M register and the dividend into the A, Q registers The dividend must be expressed as a 2n-bit twos complement number Thus for example, the 4-bit 0111 becomes 00000111, and 1001 becomes 11111001.
2. Shift A, Q left 1 bit position.
3. If M and A have the same signs, perform $A = A$ - M; otherwise, $A = A + M$.
4. The preceding operation is successful if the sign of A is the same before and after the operation.
    a. If the operation is successful or $A = 0$, then set $Q_0 = 1$.
    b. If the operation is unsuccessful and $A \neq 0$, then set $Q_0 = 0$ and restore the previous value of A.
5. Repeat steps 2 through 4 as many times as there are bit positions in Q.
6. The remainder is in A. If the signs of the divisor and dividend were the same, then the quotient is in Q; otherwise, the correct quotient is the twos complement of Q.

Figure 3. Example of twos complement division.

a) Write a VHDL file to model the above behavior for 4 bits Divisor and 4 bits divident.
b) Use Modelsim to compile your VHDL file.
c) Use Modelsim to simulate your division circuit with the following cases: 7/-2, 6/-2.

A	Q	M = 0011	A	Q	M = 1101
1111	1001	Initial value	1111	1001	Initial value
1111	0010	shift	1111	0010	shift
0010		add	0010		subtract
1111	0010	restore	1111	0010	restore
1110	0100	shift	1110	0100	shift
0001		add	0001		subtract
1110	0100	restore	1110	0100	restore
1100	1000	shift	1100	1000	shift
1111		add	1111		subtract
1111	1001	set $Q_0 = 1$	1111	1001	set $Q_0 = 1$
1111	0010	shift	1111	0010	shift
0010		add	0010		subtract
1111	0010	restore	1111	0010	restore

(c) (−7)/(3) (d) (−7)/(−3)