UNIVERSITY OF CALIFORNIA—DAVIS

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

EEC180 — DIGITAL SYSTEMS II — WINTER 2023

Tobin Joseph

HOMEWORK 2

1. Write a Verilog module that implements an 8-bit Carry-Select adder using 4-bit adders, 2-to-1 multiplexers, and logic gates.

```
module fa (in1, in2, cin, sum, cout);

      //Declare inputs, outputs, and internal variables.
      input in1,
                  in2,
                  cin;
      output sum,
                  cout;

            // use Boolean expressions
      assign sum  = (in1 ^ in2) ^ cin;
      assign cout = (in1 & in2) | (in2 & cin) | (cin & in1);
endmodule



module add4bit(a,b,cin,sum,cout);

      //Declare inputs, outputs, and internal variables.
      input [3:0]a,b;
      input cin;
      output wire [3:0]sum;
      output cout;

      // instantiating four full-adder circuits - ripple carry
      fa FA1(a[0],b[0],cin,sum[0],cout1);
      fa FA2(a[1],b[1],cout1,sum[1],cout2);
      fa FA3(a[2],b[2],cout2,sum[2],cout3);
      fa FA4(a[3],b[3],cout3,sum[3],cout);
endmodule
```

```
module mux2_1(Y, D0, D1, S);

	output Y;
	input D0, D1, S;
	wire T1, T2, Sbar;

	not (Sbar, S);
	and (T1, D1, S), (T2, D0, Sbar);
	or (Y, T1, T2);

endmodule

module add8bitcarryselect(a,b,cin,sum,cout);

	//Declare inputs, outputs, and internal variables.
	input [7:0]a,b;
	input cin;
	output wire [7:0]sum;
	output cout;

	wire [7:0] sum0;
	wire [7:0] sum1;
	wire cout_0;
	wire cout_1;
	wire cin_0;
	wire cin_1;

	assign cin_0 = 1'b0;
	assign cin_1 = 1'b1;


	// instantiating 4-bit full-adder circuits 2x - ripple carry cin = 0
	add4bit A4B1(a[3:0],b[3:0],cin_0,sum0[3:0],cout1);
	add4bit A4B2(a[7:4],b[7:4],cout1,sum0[7:4],cout_0);

	// instantiating 4-bit full-adder circuits 2x - ripple carry cin = 1
	add4bit A4B3(a[3:0],b[3:0],cin_1,sum1[3:0],cout2);
	add4bit A4B4(a[7:4],b[7:4],cout2,sum1[7:4],cout_1);


	// instantiate 2 to 1 mux for carry out
	mux2_1 muxcout(cout, cout_0, cout_1, cin);
```

```
      // instantiate 2 to 1 mux for sum
      mux2_1 muxbit0(sum[0],sum0[0],sum1[0],cin);
      mux2_1 muxbit1(sum[1],sum0[1],sum1[1],cin);
      mux2_1 muxbit2(sum[2],sum0[2],sum1[2],cin);
      mux2_1 muxbit3(sum[3],sum0[3],sum1[3],cin);
      mux2_1 muxbit4(sum[4],sum0[4],sum1[4],cin);
      mux2_1 muxbit5(sum[5],sum0[5],sum1[5],cin);
      mux2_1 muxbit6(sum[6],sum0[6],sum1[6],cin);
      mux2_1 muxbit7(sum[7],sum0[7],sum1[7],cin);

endmodule
```

2. Write a Verilog module for a three-way magnitude comparator that outputs True if its three inputs are in this order a > b ≥ c.

```
//special case : single-bit three-way magnitude comparator for a > b >= c
module magcompabc(Y, a, b, c);

      output Y;
      input a, b, c;
      wire bbar, agtb, xorbc, bec;

      // for a > b >= c, the only possibility is a > b = C
      not (bbar, b);
      and (agtb, a, bbar);
      xor (xorbc, b, c);
      not (bec, xorbc);
      and (Y, agtb, bec);

endmodule

//two-bit magnitude comparator
module two_bit_comp(G, L, E, a, b);
    input [1:0]a,b;
    output G, L, E;

    assign G = ((a[1]) & (~b[1])) | (((a[1])~^(b[1])) & ((a[0]) & (~b[0])));
    assign L = ((~a[1]) & (b[1])) | (((a[1])~^(b[1])) & ((~a[0]) & (b[0])));
    assign E = (a[1]~^b[1]) & (a[0]~^b[0]);
endmodule

// four-bit magnitude comparator
module four_bit_comp(G, L, E, a, b);
    input [3:0] a,b ;
    output G, L, E;

    wire g1, g2, l1, l2, e1, e2;

    two_bit_comp comp1(g2, l2, e2, a[3:2], b[3:2]);
    two_bit_comp comp2(g1, l1, e1, a[1:0], b[1:0]);

    assign G = (g2) | (e2 & g1);
    assign L = (l2) | (e2 & l1);
    assign E = e1 & e2;
endmodule
```

```
// single output three-way magnitude comparator for a > b >= c
module abc_4bitcomp(Y, a, b, c);
      input [3:0] a,b,c;
      output Y;

      wire GAB, LAB, EAB, GBC, LBC, EBC, GEBC;

      four_bit_comp compab(GAB, LAB, EAB, a, b);
      four_bit_comp compbc(GBC, LBC, EBC, b, c);

      assign Y = GAB & (GBC | EBC);
endmodule
```

3. Our goal in this problem is to select an appropriate fixed-point notation. We would like to represent a relative pressure signal with a range from -10 PSI to 30 PSI with an accuracy of 0.01 PSI. Select the fixed-point representation that covers this range with the specified accuracy with a minimum number of bits.

Let's start with the integer part:

Requirement: Range from -10 PSI to 30 PSI.
We need to represent signed numbers and the magnitude has to accept the greater of |-10| or |30| which clearly is 30. To represent a magnitude of 30 in bits we will need n-bits such that

$$(2^n - 1) >= 30 > (2^{n-1} - 1).$$

We can say

$$30 < 31 = 2^5 - 1$$
$$30 > 15 = 2^4 - 1$$

So with n = 5, we have

$$(2^5 - 1) >= 30 > (2^{(5-1)} - 1)$$
$$31 >= 30 > 15$$

For the integer part, we have that we need 5 bits for the magnitude and a sign bit.

Let's process the non-integer part:

Requirement: An accuracy of 0.01 PSI.
We need to represent in binary the magnitudes with a precision of 0.01. The decimal 0.01 has no exact representation as a binary number, so we will have to use an approximation to represent numbers like that. Using rounding rules, the error between the binary representation and the actual decimal value should not exceed ±0.005 to round to 0.01. This way, when making the conversion to a decimal number with rounding rules, we should get the original decimal value with no errors.

We need to select an m such that

$$0.005 > 2^{-m} > 0.005/2$$
$$0.005 > 2^{-m} > 0.0025$$

With m = 8 we can say that $2^{-8} = 0.00390625$, then we have

$0.005 > 2^{-8} > 0.0025$
$0.005 > 0.00390625 > 0.0025$

To test the validity of our expression, let's verify the error introduced with m = 8 when converting to binary and how rounding can correct the error when converting back to decimal.

To calculate the binary representation of 0.01 we use the following algorithm:

To avoid the decimal separator, multiply the decimal number with the base raised to the power of decimals in the result:

$0.01 \times 2^8 = 3$

Divide by the base 2 to get the digits from the remainders:

Division by 2	Quotient	Remainder (Digit)	Bit #
(3)/2	1	1	0
(1)/2	0	1	1

$= (000000010)_2 / 2^8$

$= (0.00000010)_2$

Converting this back to a decimal and then rounding to the second decimal n-digit we get

$0.01 \cong (0.00000010)_2 = 0.0078125 \cong 0.01$

Doing the same up to 0.10 gives us:

$0.00 = (0.000000000)_2 = 0.00 = 0.00$
$0.01 \cong (0.00000010)_2 = 0.0078125 \cong 0.01$
$0.02 \cong (0.00000101)_2 = 0.01953125 \cong 0.02$
$0.03 \cong (0.00000111)_2 = 0.02734375 \cong 0.03$
$0.04 \cong (0.00001010)_2 = 0.0390625 \cong 0.04$
$0.05 \cong (0.00001100)_2 = 0.046875 \cong 0.05$
$0.06 \cong (0.00001111)_2 = 0.05859375 \cong 0.06$
$0.07 \cong (0.00010001)_2 = 0.06640625 \cong 0.07$
$0.08 \cong (0.00010100)_2 = 0.078125 \cong 0.08$

$0.09 \cong (0.00010111)_2 = 0.08984375 \cong 0.09$
$0.10 \cong (0.00011001)_2 = 0.09765625 \cong 0.10$

Spot-checking 10 aleatory values from 0.11 to 0.99 we get

$0.13 \cong (0.00100001)_2 = 0.12890625 \cong 0.13$
$0.24 \cong (0.00111101)_2 = 0.23828125 \cong 0.24$
$0.37 \cong (0.01011110)_2 = 0.3671875 \cong 0.37$
$0.43 \cong (0.01101110)_2 = 0.4296875 \cong 0.43$
$0.52 \cong (0.10000101)_2 = 0.51953125 \cong 0.52$
$0.60 \cong (0.10011001)_2 = 0.59765625 \cong 0.60$
$0.68 \cong (0.10101110)_2 = 0.6796875 \cong 0.68$
$0.77 \cong (0.11000101)_2 = 0.76953125 \cong 0.77$
$0.81 \cong (0.11001111)_2 = 0.80859375 \cong 0.81$
$0.95 \cong (0.11110011)_2 = 0.94921875 \cong 0.95$

With m = 8, the condition

$$0.005 > 2^{-8} > 0.0025$$

is satisfied as the binary representation, which is an approximate of the decimal value is enough to recover the original decimal value using standard rounding at the end of the conversion to decimal.

Putting together the results for the integer part and for the decimal part, we will have a fixed-point representation that covers this range with the specified accuracy with a minimum number of bits:

sign-bit | 5 bits integer magnitude . 8 bits decimal approximation

It can be two’s complement representation for ease of arithmetic with the numbers.

Thus, the total minimum number of bits needed for the fixed-point representation that covers this range with the specified accuracy is

$1 + 5 + 8 = 14$ bits

4. Let X be a 4-bit unsigned number. Let Y be an unsigned number such that $Y = X^m$.
Consider the following hypothesis: “We need at least 4m bits to represent Y.”
The above hypothesis is false. Find the smallest value of m where the hypothesis fails.
Note: You can use a calculator to help you find the answer.

First, we need to find the maximum number that can be represented with a 4-bit unsigned number, which is

$$X_{MAX} = (1111)_b = 15 = 2^4 - 1$$

From this, we have that

$$Y_{MAX} = X_{MAX}{}^m$$

The hypothesis establishes that “We need at least 4m bits to represent Y.” meaning that the maximum value of Y must be $<= 2^{4m} - 1$ and be $> 2^{(4m-1)} - 1$, that is using one less bit should not be enough.

$$(2^{4m} - 1) >= Y_{MAX} > (2^{(4m-1)} - 1)$$

Combining the equations we can say

$$(2^{4m} - 1) >= X_{MAX}{}^m > (2^{(4m-1)} - 1)$$
$$(2^{4m} - 1) >= (2^4 - 1)^m > (2^{(4m-1)} - 1)$$
$$((2^4)^m - 1) >= (2^4 - 1)^m > ((2^{4m} / 2) - 1)$$
$$(16^m - 1) >= 15^m > (((2^4)^m / 2) - 1)$$
$$(16^m - 1) >= 15^m > ((16^m / 2) - 1)$$

This side of the relation is always true for $m > 0$ as the base 16 is greater than the base 15:

$$(16^m - 1) >= 15^m$$

The other side of the relationship we have to test.

$$15^m > ((16^m / 2) - 1)$$

For m = 1

$$15^1 > ((16^1 / 2) - 1)$$
$$15 > 7$$

It stands.

Using a calculator for m = 10

$15^{10} > ((16^{10} / 2) - 1)$
$576,650,390,625 > 549,755,813,887$

It stands.

Using a calculator for m = 11

$15^{11} > ((16^{11} / 2) - 1)$
$8,649,755,859,375 < 8,796,093,022,207$

It fails at m = 11. This means that at m = 11, X_{MAX}^{m} can be represented with a number using less than 4m bits - 1.

Thus, the smallest value of m where the hypothesis fails is m = 11.