**LSU EE 3755** **Homework 3** Solution **Due: 15 October 2012**

**Problem 1:** The module below performs subtraction naïvely, using two adders. If the synthesis program does not see it, the resulting hardware will use the two adders. Re-write the module so that it uses one adder, even before the synthesis program applies optimization.

```
module subtract(diff, a, b);
   input a, b;
   output diff;

   wire [31:0] a, b, diff;

   wire [31:0] bnot = ~ b;   // Perform bitwise-negation.
   wire        cout1, cout2; // Ignore values.
   wire [31:0] bneg;

   // Ripple_add_32 ports: carry-out, sum, addend, addend, carry-in.
   ripple_add_32 a1(cout1, bneg, bnot, 32'd1, 1'b0);
   ripple_add_32 a2(cout2, diff, a,    bneg,  1'b0);
endmodule
```

The solution appears below. In the module above the first ripple adder, `a1`, is used only to add one to `bnot`, part of the process of computing the two's complement representation of `-b`. In the solution below the carry-in input of what was the second adder is used to add that one, thus eliminating an adder.

```
// SOLUTION
module subtract(diff, a, b);
   input a, b;
   output diff;

   wire [31:0] a, b, diff;
   wire [31:0] bnot = ~ b;   // Perform bitwise-negation.
   wire        cout;         // Ignore value.

   ripple_add_32 a2(cout, diff, a, bnot,  1'b1);
endmodule
```

**Problem 2:** Memorize the Boolean expressions to compute the generate and propagate signals in a carry lookahead adder. Show a Boolean expression needed to generate carry in 5 of a flat carry lookahead adder, where carry in 0 is the carry in to the least significant bit. In your Boolean expression use $p_i$ for the propagate signal from bit $i$, for $i \geq 0$ and use $g_i$ for the generate signal from bit $i$, for $i \geq 0$.

Solution:

$$c_5 = g_4 + g_3p_4 + g_2p_3p_4 + g_1p_2p_3p_4 + g_0p_1p_2p_3p_4,$$

where $+$ is logical or and juxtaposition is logical and.

**Problem 3:** Show the work for long-hand multiplication of $6b_{16} \times 27_{16}$ using radix-16 multiplication (four bits at a time). See the examples in lecture set 7, `http://www.ece.lsu.edu/ee3755/2012f/l07.v.html`.

Solution appears below. The work is shown in binary (to the left) even though with radix 16 it would be easier to do the work in hexadecimal.

```
            0110 1011  = 0x6b                          =  107
          * 0010 0111  = 0x27                          =   39
---------------------
       0010 1110 1101  = 0x6b * 0x7 = 0x2ed  = 749 *  1 =  749
+ 0000 1101 0110       = 0x6b * 0x2 = 0xd6   = 214 * 16 = 3424
---------------------
     1 0000 0100 1101              = 0x104d =           = 4173
```

**Problem 4:** Show the work for long-hand multiplication of $6b_{16} \times 27_{16}$ using radix-4 Booth recoding. Remember that with Booth recoding some of the partial products can be negative, so remember to sign extend as long as necessary. Please make an effort to arrive at the correct answer, which is $104d_{16}$.

Solution appears below. For this discussion see the "Radix-4 Booth Table" from Note Set 7, at `http://www.ece.lsu.edu/ee3755/2012f/l07.v.html`.

The first step is to prepare the multiples of the multiplicand needed by the radix-4 Booth algorithm, that appears first below, followed by the multiplication. Being a radix-4 algorithm the multiplier is examined two bits at a time, (since $\log_2 4 = 2$), starting at the least significant bits. Multiplier bits are labeled `MB` in the Radix-4 Booth Table and in the work below. For the first partial product (labeled with an `a:` on the left) the multiplier bits are `MB = 11`, and since it's the first partial product the carry in bit is zero, that is, `C = 0`. Based on the Booth table the partial product should be -1 times the multiplicand and there should be a carry out. (There is a carry out whenever the MSB of `MB` is 1.) The value of -1 times the multiplicand is found in the multiples table, notice that the sign bit is extended as far to the left as needed. For the second partial product the next two multiplier bits are used, `01`; the carry in is the carry out from the previous step, `1`. According to the Booth table we use two times multiplicand, this multiple needs to be shifted by two bits more (since it's the second partial product). Similar reasoning provides the partial product lines `c:` and `d:`. The sum of the partial products is on line `s:`, the partial product sum's carry out, shown as an `X`, is ignored. (The carries shown in the Booth table have nothing to do with the carries obtained when adding the partial products.)

The calculation is performed below for 20 bits, but only 16 bits are necessary to represent the product of two 8-bit numbers.

```
Multiples of the multiplicand, used in partial products.
  1 * 0x6b =  0000 0000 0110 1011
 -1 * 0x6b =  1111 1111 1001 0101  Negative, extend sign as necessary.
  2 * 0x6b =  0000 0000 1101 0110
 -2 * 0x6b =  1111 1111 0010 1010  Negative, extend sign as necessary.

Perform the Multiplication:

                    0110 1011  = multiplicand  = 0x6b  = 107
                  * 0010 0111  = multiplier    = 0x27  =  39
     ------------------------
a:   1111 1111 1111 1001 0101    MB = 11, C = 0  =>  x = -1, c = 1
b:   0000 0000 0011 0101 10      MB = 01, C = 1  =>  x = 2,  c = 0
c:   1111 1111 0010 1010         MB = 10, C = 0  =>  x = -2, c = 1
d:   0000 0001 1010 11           MB = 00, C = 1  =>  x = 1, c = 0
   -------------------------
s: X 0000 0001 0000 0100 1101  = 0x4d01
```