EE486: Advanced Computer Arithmetic
Homework #4 Solutions (25 pts)

The main purpose of having you do problem 1 and 4 is for you to see the small details that might be needed. The crucial difference between the two is in the sign of the result and the way the number is padded to the MSB side.

(5pts) Problem 4.1 - Modified Booth 4 for 2's complement numbers

Pad the LSB side with one zero, extend the MSB side with as many bits as required (by repeating the MSB) to make the total number of bits a multiple of 4. If the number of bits is already a multiple of 4 do not extend it. To understand this extension let us first see the following Lemma.

Lemma 1 *If a binary number x is represented in two's complement form by the bit string $x_n x_{n-1} \cdots x_1 x_0$, then $x = (-1)x_n 2^n + \sum_{i=0}^{n-1} x_i 2^i$.*

Proof:

- If x is positive, then $x_n = 0$ and the statement is true by the definition of binary number representation.
- If x is negative, then $x_n = 1$ and the absolute value of x is equal to the one's complement of the $x_n x_{n-1} \cdots x_0$ bits plus one. This means:

$$
\begin{aligned}
x &= -(\sum_{i=0}^{n}(1-x_i)2^i + 1) \\
&= -(1-x_n)2^n - \sum_{i=0}^{n-1}(1-x_i)2^i - 1 \\
&= 0 - \sum_{i=0}^{n-1} 2^i + \sum_{i=0}^{n-1} x_i 2^i - 1 \\
&= -2^n + 1 + \sum_{i=0}^{n-1} x_i 2^i - 1 \\
&= (-1)x_n 2^n + \sum_{i=0}^{n-1} x_i 2^i
\end{aligned}
$$

  The fact that $x_n = 1$ in this case was used in the last line above.

This concludes the proof since in both cases, the statement is true.$\diamond$

Basically, you can think of the MSB of a 2's complement number as if having a negative value. The modified Booth recoders also assume that the MSB of each group is having a negative value. So, if the MSB of the number falls as an MSB of a group in the Booth recoding, the algorithm will work correctly. If on the other hand, the MSB of the number falls within a group for the Booth recoder, we must do the sign extension described above. Can you see why this will be correct?

Now with that sign extension done, the following table can be used to perform Booth4

b_8	b_4	b_2	b_1	c_i	v	c_o
0	0	0	0	0	0	0
0	0	0	0	1	1	0
0	0	0	1	0	1	0
0	0	0	1	1	2	0
0	0	1	0	0	2	0
0	0	1	0	1	3	0
0	0	1	1	0	3	0
0	0	1	1	1	4	0
0	1	0	0	0	4	0
0	1	0	0	1	5	0
0	1	0	1	0	5	0
0	1	0	1	1	6	0
0	1	1	0	0	6	0
0	1	1	0	1	7	0
0	1	1	1	0	7	0
0	1	1	1	1	8	0
1	0	0	0	0	-8	1
1	0	0	0	1	-7	1
1	0	0	1	0	-7	1
1	0	0	1	1	-6	1
1	0	1	0	0	-6	1
1	0	1	0	1	-5	1
1	0	1	1	0	-5	1
1	0	1	1	1	-4	1
1	1	0	0	0	-4	1
1	1	0	0	1	-3	1
1	1	0	1	0	-3	1
1	1	0	1	1	-2	1
1	1	1	0	0	-2	1
1	1	1	0	1	-1	1
1	1	1	1	0	-1	1
1	1	1	1	1	-0	1

Logic equations for each possibility can be written in terms of the input bits and then they can be used to select the required bit pattern from the multiplicand or its 3, 5 and 7 times multiples or their complements. There is no need to really generate the 2, 4 and 6 times multiples since they are shifted versions of the one, two and three times multiples. As you have labored through the problem, I guess you understand why high order Booth recoders become cumbersome and have a delay of their own that might offset the gain of reducing the levels in the pratial product reduction tree. Another big problem is the need for the "hard" multiples. Redundant Booth 3 and 4 solve to some extent the issue of hard multiples but they are still big structures that might not be very good from a performance perspective.

For redundant Booth, see the PhD thesis of Gary Bewick (1994), *"Fast Multiplication: Algorithms and Implementation"* at:

`http://arith.stanford.edu/phds.html`

For performance and area comparisons of the different algorithms, see the PhD thesis of Hesham Al-Twaijry (1997), *"Area and Performance Optimized CMOS Multipliers"* at the same website. (That is the website of the dissertations done with Prof. Flynn)

(5pts) Problem 4.4 - Booth 4 for Sign Magnitude Numbers

The easiest way to answer this problem is to use a standard "Modified Booth 4" multiplier as the one presented in the first problem above and to compute the product of the *unsigned* magnitudes of both numbers and determine the sign of the product using $S_p = S_a$ XOR S_b.

The padding to the LSB side remains a single zero. The padding on the MSB side however is different here. You need to pad as many zeros as required to make the number of bits a multiple of four. If the number of bits is already a multiple of four you pad an additional group with all zeros. Can you see why you must do this? Remember that the table presented above assumes the MSB bit of a group to be negatively valued while the magnitudes are treated as unsigned.

Obviously, another way of dealing with the padding in both problem 1 and here is to design a special recoder for the LSB and MSB side groups that takes into account the end effects. In practice this is what is done instead of introducing unnecessary parital products at the MSB side.

(5pts) Problem 4.8 - Using ROMs

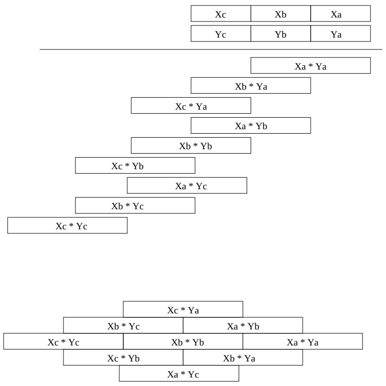

Figure 1: Partial Product Array and Rearranged Product Matrix

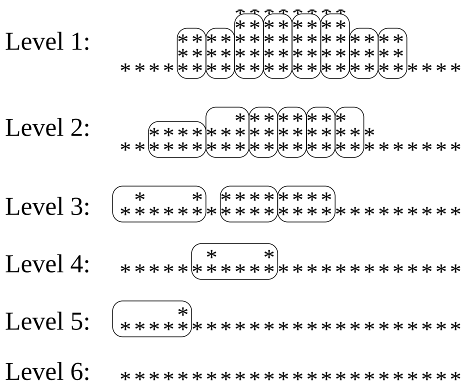

Figure 2: Reducing the PPA with 256*8 bit ROMs

(5pts) Problem 4.11 - Sign Extension Logic

For sign-extended 8 bit numbers represented as:

$$SSSSSSSSX_8X_7X_6X_5X_4X_3X_2X_1X_0$$

with $X_9 = S$, this is equal to:

$$\begin{aligned}
&= X_0 + 2X_1 + 2^2X_2 + 2^3X_3 + \cdots + 2^8X_8 + 2^9X_9 + \cdots + 2^{14}X_9 - 2^{15}X_9 \\
&= (X_0 + 2X_1 + \cdots) + (2^{15} - 2^9)X_9 - 2^{15}X_9 \\
&= (X_0 + \cdots) - 2^9X_9 \\
&= (-X_9)X_8X_7X_6X_5X_4X_3X_2X_1X_0
\end{aligned}$$

Thus, the sign extension can be replaced by $(-X_9)$ in the most significant bit required. The problem, then, is how to represent $(-X_9)$ properly. For the A row, it is clear that at least 2 bits of sign extension are required for overlap with the B row:

$$\begin{array}{l}
0000(-A_9)A_9A_9A_8A_7\cdots \\
0000(-B_9)B_8B_7B_6B_5\cdots
\end{array}$$

Now, $-A_9$ as shown can be rewritten as

$$-2^{11} + (2^{11} - 2^{11}A_9) \tag{1}$$

However, $(2^{11} - 2^{11}A_9)$ is simply the one's complement of A_9, and this row can be rewritten into two rows:

$$\begin{array}{cccccccc}
0 & 0 & (-1) & 0 & 0 & 0 & 0 & \cdots \\
0 & 0 & \overline{A_9} & A_9 & A_9 & A_8 & A_7 & \cdots
\end{array}$$

Similarly, for the B row, $-B_9$ can be rewritten as

$$\begin{aligned}
&= -2^{11} + (2^{11} - 2^{11}B_9) \\
&= (-2^{13} + 2^{12} + 2^{11}) + (2^{11} - 2^{11}B_9) \\
&= (-2^{13} + 2^{12} + 2^{11}) + 2^{11}\overline{B_9} \qquad (2)
\end{aligned}$$

The same procedure follows for the C row and the D row, rewriting each of the (-S) terms using numbers of larger weights:

$$Crow \;:\; (-2^{15} + 2^{14} + 2^{13}) + (2^{13} - 2^{13}C_9) \tag{3}$$

$$Drow \;:\; (-2^{16} + 2^{15}) + (2^{15} - 2^{15}D_9) \tag{4}$$

If we add up the A through D rows,i.e. equations 1 to 4, we get:

$$-2^{16} + 2^{15}\overline{D_9} + 2^{14} + 2^{13}\overline{C_9} + 2^{12} + 2^{11}\overline{B_9} + 2^{11}\overline{A_9} + \cdots$$

Since the width of the tree is only 16 bits (15-0), we can ignore the term -2^{16}.

The final result is :

				$\overline{A_9}$	A_9	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
			1	$\overline{B_9}$	B_8	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0		Y_1
	1	$\overline{C_9}$	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0		Y_3		
$\overline{D_9}$	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0		Y_5				
E_7	E_6	E_5	E_4	E_3	E_2	E_1	E_0		Y_7						
S_{15}	S_{14}	S_{13}	S_{12}	S_{11}	S_{10}	S_9	S_8	S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0

You can refer to appendix A of the PhD thesis of Gary Bewick (mentioned in problem 1) to see a more general proof. Some of you also pointed to another proof on the webpages of the EE371 class in lecture 11 on multipliers at:

`http://www.stanford.edu/class/ee371/lectures.html`

That second proof assumes that the whole triangle is full with 1 and sums it to get: 0101011. If any of the sign bits is not one but rather is a zero, this vector must be rectified by adding one (that is the complement of the sign bit) at the corresponding location. You end up with the same result.

(5pts) Problem 4.13 - (5,5,4) Implementation

There are many possible solutions to the problem. The best solutions take 4 stages of CSA's and use 6 counters.