CS 140

Class Notes 2

1 Multiplication

Consider two unsigned binary numbers X and Y. We want to multiply these numbers. The basic algorithm is similar to the one used in multiplying the numbers on pencil and paper. The main operations involved are *shift* and *add*.

Recall that the 'pencil-and-paper' algorithm is inefficient in that each product term (obtained by multiplying each bit of the multiplier to the multiplicant) has to be saved till all such product terms are obtained. In machine implementations, it is desirable to add all such product terms to form the *partial product*. Also, instead of shifting the product terms to the left, the partial product is shifted to the right before the addition takes place. In other words, if P_i is the partial product after i steps and if Y is the multiplicant and X is the multiplier, then

$$P_i \leftarrow P_i + x_j * Y$$

and

$$P_{i+1} \leftarrow P_i * 2^{-1}$$

and the process repeats.

Note that the multiplication of signed magnitude numbers require a straight forward extension of the unsigned case. The magnitude part of the product can be computed just as in the unsigned magnitude case. The sign p_0 of the product P is computed from the signs of X and Y as

$$p_0 \leftarrow x_0 \oplus y_0$$

2 Two's complement Multiplication - Robertson's Algorithm

Consider the case that we want to multiply two 8 bit numbers $X = x_0 x_1 ... x_7$ and $Y = y_0 y_1 ... y_7$. Depending on the sign of the two operands X and Y, there are 4 cases to be considered :

- $x_0 = y_0 = 0$, that is, both X and Y are positive. Hence, multiplication of these numbers is similar to the multiplication of unsigned numbers. In other words, the product P is computed in a series of add-and-shift steps of the form

  $$P_i \leftarrow P_i + x_j * Y$$

  $$P_{i+1} \leftarrow P_i * 2^{-1}$$

  Note that all the partial product are non-negative. Hence, leading 0s are introduced during right shift of the partial product.

- $x_0 = 1, y_0 = 0$, that is, X is negative and Y is positive. In this case, the partial product is always positive (till the sign bit x_0 is used). In the final step, a subtraction is performed. That is,

$$P \leftarrow P - Y$$

- $x_0 = 0, y_0 = 1$, that is, X is positive and Y is negative. In this case, the partial product is positive and hence leading 0s are shifted into the partial product until the first 1 in X is encountered. Multiplication of Y by this 1, and addition to the result causes the partial product to be negative, from which point on leading 1s are shifted in (rather than 0s).

- $x_0 = 1, y_0 = 1$, that is, both X and Y are negative. Once again, leading 1s are shifted into the partial product whenever the partial product is negative. Also, since X is negative, the correction step (subtraction as the last step) is also performed.

3 Booth's Algorithm

Recall that the preceding multiplication algorithms (Robertson's algorithm) involves scanning the multiplier from right to left and using the current multiplier bit x_i to determine whether the multiplicant Y be added, subtracted or add 0 (do nothing) to the partial product. In Booth's algorithm, two adjacent bits $x_i x_{i+1}$ are examined in each step. If $x_i x_{i+1} = 01$, then Y is added to the partial product, while if $x_i x_{i+1} = 10$, Y is subtracted from P_i (partial product). If $x_i x_{i+1} = 00$ or 11, then neither addition nor subtraction is performed. Thus, Booth's algorithm effectively skips over sequences of 1s and 0s in X. As a result, the total number of addition/subtraction steps required to multiply two numbers decrease (however, at the cost of extra hardware).

The process of inspecting the multiplier bits required by booth's algorithm can be viewed as encoding the multiplier using three digits 0, 1 and $\overline{1}$, where 0 means shift the partial product to the right (that is, no addition or subtraction is performed), while 1 means add multiplicant before shifting and $\overline{1}$ means subtract multiplicant from the partial product before shifting. The number thus produced is called a *signed digit number* and this process of converting a multiplier X into a signed digit form is called as *multiplier recoding.* To generate X^* from X, append the number X with a 0 to the right (that is, start with $X = x_0 x_1 ... x_{n-1} 0$). Then use the following table to generate X^* from X:

x_i	x_{i+1}	x_i^*
0	0	0
0	1	1
1	0	$\overline{1}$
1	1	0