CS 140

Class Notes 1

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 multiplicand) 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 multiplicand 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 straightforward 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

$$p_0 \leftarrow x_0 \oplus y_0$$

1.1 Two's complement Multiplication - Robertson's Algorithm

Consider the case that we want to multiply two 8 bit numbers $X = x_0x_1...x_7$ and $Y = y_0y_1...y_7$. 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 once the first 1 in X is encountered. Also, since X is negative, the correction step (subtraction as the last step) is also performed.

A Word of Caution: A difference exists in the correction steps between multiplication of two integers and two fractions. In the case of two integers, the correction step involves subtraction and shift right. In the case of fractions, the correction step involves subtraction and setting $Q(7) \leftarrow 0$.

1.2 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 multiplicand 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 multiplicand before shifting and $\overline{1}$ means subtract multiplicand 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

Booth's algorithm results in reduction in the number of add/subtract steps needed (as compared to the Robertson's algorithm) if the multiplier contains runs (or sequences) of 1s or 0s. The worst case scenario occurs in booth's algorithm if $X = 010101..01$, where there are $n/2$ isolated 1s, which forces $n/2$ subtractions and $n/2$ additions. This is worse than the standard multiplication algorithm - which contains only $n/2$ additions.

2 Division

In a fixed point division of two numbers, a divisor V and a dividend D are given. The goal of division is to compute quotient Q and remainder R such that

$$D = Q * V + R$$

One of the simplest methods of division is the sequential digit-by-digit algorithm. Here, in step i, the quotient bit q_i is determined by comparing the value of $2^{-i}V$, which represents the divisor shifted i bits to the right, to the partial remainder R_i. The quotient bit q_i is set to 1 if 2^{-i} is less than R_i, else it is set to 0. Thus, the partial remainder is computed using the expression:

$$R_{i+1} = R_i - q_i * 2^{-i} * V$$

This is equivalent to

$$R_{i+1} = 2R_i - q_i * V$$

As it is clear from the basic approach described here, the value of the quotient bit is determined by performing a trial subtraction of the form $2R_i - V$. If this is positive, then $q_i = 1$, otherwise $q_i = 0$. Note that when $q_i = 0$, the result of the trial subtraction is $2R_i - V$, but the new partial remainder should be $R_{i+1} = 2R_i$. Based on how this discrepancy is handled in the computations of q_i and R_{i+1}, two distinct division algorithms can be proposed:

2.1 Restoring Division

Here, at every step, the operation

$$R_{i+1} = 2R_i - V$$

is performed. When the result of the subtraction is negative, a restoring addition is performed as follows:

$$R_{i+1} = R_{i+1} + V$$

In other words, the partial remainder is restored to the correct value if the quotient bit is 0.

2.2 Non-Restoring Division

It is based on the observation that a restoring step of the form

$$R_i = R_i + V$$

followed by the next partial remainder calculation step

$$R_{i+1} = 2R_i - V$$

can be merged into a single operation

$$R_{i+1} = 2R_i + V$$

Thus, when the quotient bit $q_i = 1$, then the next partial remainder is computed by performing a subtraction. However, when the quotient bit $q_i = 0$, rather than restoring the partial remainder, the next step is the addition of the divisor to the partial remainder, and not a subtraction.

A Word of Caution: In the case of non-restoring division, note that when the last quotient bit is 0 (that is, $q_0 = 1$), then as a result of the trial subtraction, the partial remainder is negative. Hence, a correction step is necessary to restore the remainder value. Note that this is necessary only for the last step.

The restoring and non-restoring division techniques described here are applicable to unsigned integers as well as sign-magnitude numbers.