Implementing Computer Arithmetics

0.1 Addition and Subtraction

Figure 1 suggests the data paths and hardware elements needed to accomplish addition and subtraction. The central element is a binary adder, which is presented two numbers for addition and produces a sum $s_0 \ldots s_{n-1}$ and an overflow indication c_n. The binary adder treats the two numbers as unsigned integers. (A logical implementation of a serial adder was given earlier). For addition, the two numbers are presented to the adder from two registers, designated in this case as A (bits $a_{n-1} \ldots a_0$) and B (bits $b_{n-1} \ldots b_0$) registers, with *add/subtract* control line as 0 (for addition). The result may be stored in one of these registers or in a third register. The overflow is stored in the 1-bit overflow flag c_n (where 0 = no overflow, and 1 = overflow). The c_0 is kept as 0 for addition.

For subtraction, the subtrahend (B register) is passed through a twos complementer so that its twos complement is presented to the adder. The *add/subtract* line as 1 sends B bits after complementing as 2s complement. The c_0 is true for 2's complement. For addition, $S \leftarrow A + B$, and for subtraction, $S \leftarrow A - B$.

Note that Figure 1 only shows the data paths. Control signals are needed to control whether or not the complementer is used, depending on whether the operation is addition or subtraction.

For Signed numbers' the over flow (carry) can be directly calculated as

$$C = a_{n-1} b_{n-1} \bar{s}_{n-1} + \bar{a}_{n-1} \bar{b}_{n-1} s_{n-1} \tag{1}$$

The adder requires total n number of Full-adders. This adder/subtracter, where the bits are sequentially added, is called *ripple carry adder*. However, for high-speed addition different approach is used, where bits are

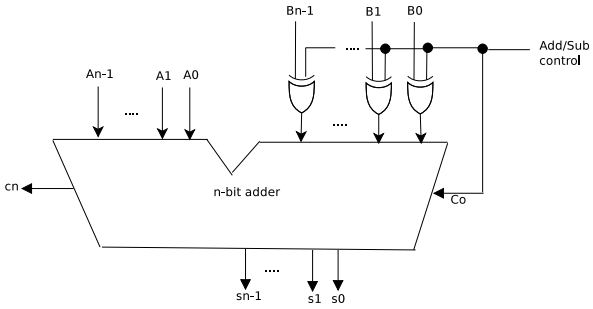

Figure 1: Adder/Subtractor.

added in parallel, and carry are calculated separately, in parallel to the addition.

0.2 Integer Multiplication

Compared with addition and subtraction, the multiplication is a complex operation, irrespective of whether performed in hardware or software. A wide variety of algorithms have been used in various computers for carrying out the multiplication. What is presented here is a feel for the type of approach typically taken. We begin with the simpler problem of multiplying two unsigned (nonnegative) integers.

The example in figure 2 illustrates the multiplication of unsigned binary integers, as might be carried out using paper and pencil. Consider multiplying two binary numbers 101 and 110. When we multiply A by B, the first is called *multiplicand* and second is called *multiplier*.

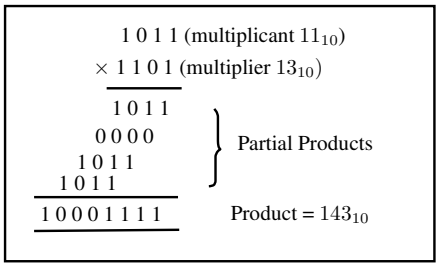

Figure 2: Multiplication using partial products.

Several important observations can be made about this:

1. Multiplication involves the generation of partial products, one for each digit in the multiplier. These partial products are then summed to produce the final product.

2. The partial products are easily defined. When the multiplier bit is 0, the partial product is 0. When the multiplier is 1, the partial product is the multiplicand.

3. The final product is produced by summing the partial products. For this operation, each successive partial product is shifted one position to the left relative to the preceding partial product.

4. Note hat, the multiplication of two n-bit binary integers results in a product of up to $2n$ bits in length (e.g., 11 * 11 = 1001). Hence, we need to keep twice the space to store the result.

When compared with the pencil-and-paper approach, there are several things we can do to make computerized multiplication more efficient. First, we can perform a running addition on the partial products rather than waiting until the end. This will not require separate storage of each individual partial product, hence fewer registers are needed. Second, we can save some time on the generation of partial products. For each 1 in the multiplier, an add and a shift operation is required; but for each 0, only a shift is required in the progressive sum.

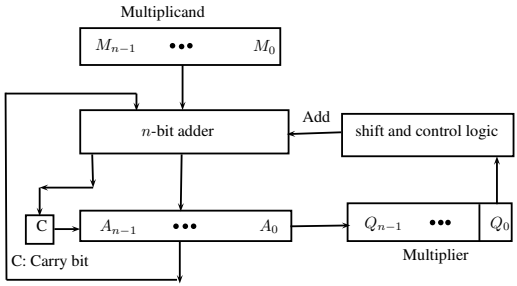

Figure 3: $n \times n$ - bit multiplier hardware unit.

Figure 3 shows a possible implementation employing these measures. The multiplier and multiplicand are loaded into two registers (Q and M), respectively. A third register A, is also needed and is initially set to 0. There is also a 1-bit C register, initialized to 0, which holds a potential carry bit resulting from addition of partial products.

The operation of the multiplication is as follows. Control logic reads the bits of the multiplier one at a time. If $Q_0 = 1$, then the multiplicand

is added to the A register and the result is stored again in the A register, with the C bit used for overflow. Then all of the bits of the C, A, and Q registers are shifted to the right one bit, so that the C bit goes into A_{n-1} , A_0 goes into Q_{n-1} , and Q_0 is lost. If Q_0 were 0, then no addition has been performed, just right shift. This process is repeated for each bit of the original multiplier, i.e., Q.

The resulting $2n$-bit product is contained in the A and Q registers. A flowchart explaining the sequence of these operations is shown in figure 4.

For two operands $|M| = n$-bit for multiplicand and $|Q| = n$-bit for multiplier, result size is $2n$-bits.

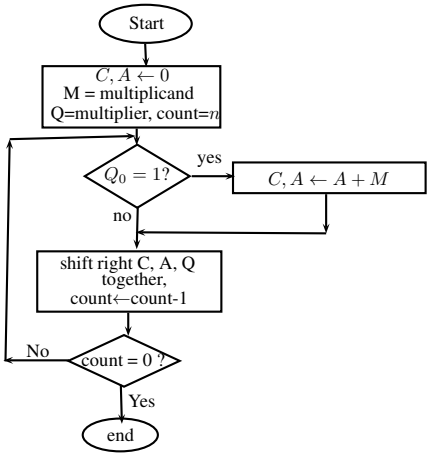

Figure 4: $\times n$ bit Multiplier Flow-chart.

0.3 Integer Division

Division is somewhat more complex than multiplication but is based on the same general principles. As before, the basis for the algorithm is the paper-and-pencil approach, and the operation involves repetitive shifting and addition or subtraction. Figure 5 shows an example of the long division of unsigned binary integers. It is instructive to describe the process in detail. First, the bits of the dividend are examined from left to right, until the set of bits examined represents a number greater than or equal to the divisor. This is referred to as the divisor being able to divide the number. Until this event occurs, 0s are placed in the quotient from left to right. When the event occurs, a 1 is placed in the quotient and the

divisor is subtracted from the partial dividend. The result is referred to as a partial remainder. From this point on, the division follows a cyclic pattern. At each cycle, additional bits from the dividend are appended to the partial remainder until the result is greater than or equal to the divisor. As before, the divisor is subtracted from this number to produce a new partial remainder. The process continues until all the bits of the dividend are exhausted.

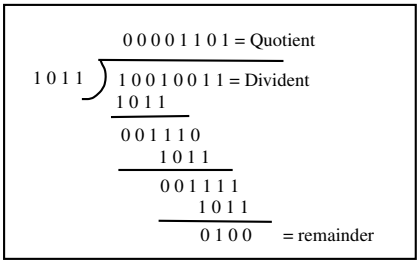

Figure 5: Division using pencil-and-paper example.

When divisor is subtracted from resultant dividend, result is shifted to right. Initial condition is: Register Q holds Dividend, M holds Divisor. Final condition is: Register A holds remainder, Q holds Quotient.

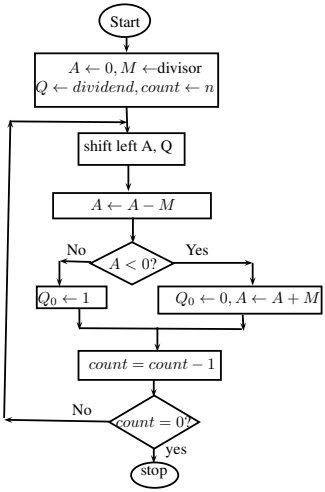

Figure 6: n by n-bit Division flow-chart.

Figure 6 shows a machine algorithm that corresponds to the long

division process. The divisor is placed in the M register, the dividend in the Q register. At each step, the A and Q registers together are shifted to the left 1 bit. M is subtracted from A to determine whether A divides the partial remainder. If it does, then Q_0 gets a 1 bit. Otherwise, Q_0 gets a 0 bit and M must be added back to A to restore the previous value. The count is then decremented, and the process continues for n steps. At the end, the quotient is in the Q register and the remainder is in the A register.

The figure 7 shows the implementation of two n-bit numbers. For two operands $|M| = n$ for divisor and $|Q| = n$ for dividend, the result size is n. When divisor is subtracted from resultant dividend, result is shifted to left. Initially register Q holds Dividend, M holds Divisor. And, finally when computation is over, register A holds remainder, and Q holds Quotient.

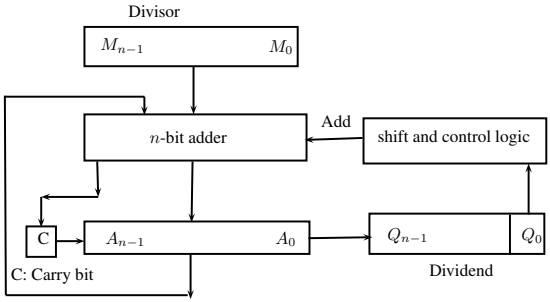

Figure 7: n by n-bit division implementation.

Exercises

1. Extend the block diagram given in figure 1 to perform the signed addition and subtraction using 2's complement method.

2. Give the logical justification for the equation 1 for computation of carry bit.

3. Construct a Gate logic to compute the carry output along with the bits $s_0 \ldots s_3$ in the figure 1.

4. Imagine that the bits a_{n-1} and b_{n-1} represent the sign bit for the two signed binary numbers being multiplied. How you will modify the following:

(a) block diagram in figure 3 to perform the signed n bit n bit multiplication?

(b) flow chart in figure 4 to perform the signed n bit n multiplication?

5. Compute the complexity of following arithmetic operations implemented for n-bit binary unsigned integers in this chapters. Assume that smallest operation performed in sequential order takes $O(1)$ time.

   (a) addition.

   (b) multiplication.

   (c) division.