EE 1193

INTRODUCTION TO HDL

MODULE-3: COMPUTER ARITHMETIC

Dr. Amit Acharyya, IITH

Overview

- In the previous module we explored a few ways that numbers can be represented in a digital computer, but we only briefly touched upon arithmetic operations that can be performed on those numbers.
- In this module we cover four basic arithmetic operations: addition, subtraction, multiplication, and division.
- We begin by describing how these four operations can be performed on fixed point numbers, and continue with a description of how these four operations can be performed on floating point numbers.
- Some of the largest problems, such as weather calculations, quantum mechanical simulations, and land-use modeling, tax the abilities of even today's largest computers. Thus the topic of high-performance arithmetic is also important. Therefore some of the algorithms and techniques used in speeding arithmetic operations are also important. However, it is out of scope of discussion in the class.

Fixed Point Addition and Subtraction

- The addition of binary numbers and the concept of overflow were briefly discussed in Module 2.
- Here, we cover addition and subtraction of both signed and unsigned fixed point numbers in detail.
- Since the two's complement representation of integers is almost universal in today's computers, we will focus primarily on two's complement operations.

TWO'S COMPLEMENT ADDITION AND SUBTRACTION

- In this section, we look at the addition of signed two's complement numbers. As we explore the *addition of signed numbers, we also implicitly cover subtraction as* well, as a result of the arithmetic principle: a - b = a + (-b).
- We can negate a number by complementing it (and adding 1, for two's complement), and so we can perform subtraction by complementing and adding. This results in a savings of hardware because it avoids the need for a hardware subtractor.

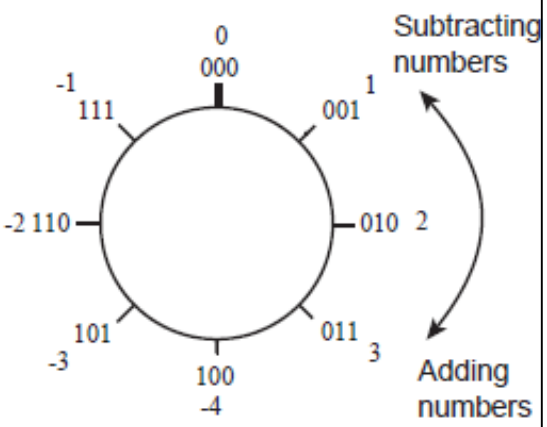


Figure: Number circle for 3-bit two's complement numbers.

We will need to modify the interpretation that we place on the results of addition when we add two's complement numbers. To see why this is the case, consider Figure. With addition on the real number line, numbers can be as large or as small as desired—the number line goes to ±infinity, so the real number line can accommodate numbers of any size.
On the other hand, as discussed in Module 2, computers represent data using a finite number of bits, and as a result can only store numbers within a certain range. For example, an examination of Table in mod-2 shows that if we restrict the size of a number to, for example, 3 bits, there will only be eight possible two's complement values that the number can assume. In Figure these values are arranged in a circle beginning with 000 and proceeding around the circle to 111 and then back to 000. The figure also shows the decimal equivalents of these same numbers.

- Some experimentation with the number circle shows that numbers can be added or subtracted by traversing the number circle clockwise for addition and counter-clockwise for subtraction.
- Numbers can also be subtracted by two's complementing the subtrahend and adding.
- Notice that overflow can only occur for addition when the operands ("addend" and "augend") are of the same sign.
- Furthermore, overflow occurs if a transition is made from +3 to -4 while proceeding around the number circle when adding, or from -4 to +3 while subtracting.

- Here are two examples of 8-bit two's complement addition, first using two positive numbers:

$$\begin{array}{lll} & 00001010 & (+10)_{10} \\ + & 00010111 & (+23)_{10} \\ \hline & 00100001 & (+33)_{10} \end{array}$$

- A positive and a negative number can be added in a similar manner:

$$\begin{array}{lll} & 00000101 & (+5)_{10} \\ + & 11111110 & (-2)_{10} \\ \hline \text{Discard carry} \rightarrow (1) & 00000011 & (+3)_{10} \end{array}$$

The carry produced by addition at the highest (leftmost) bit position is discarded in two's complement addition. A similar situation arises with a carry out of the highest bit position when adding two negative numbers:

$$\begin{array}{lll} & 11111111 & (-1)_{10} \\ + & 11111100 & (-4)_{10} \\ \hline \text{Discard carry} \rightarrow (1) & 11111011 & (-5)_{10} \end{array}$$

The carry out of the leftmost bit is discarded because the number system is **modular—** it "wraps around" from the largest positive number to the largest negative number as Figure shows. Although an addition operation may have a (discarded) carry-out from the MSB, this does not mean that the result is erroneous. The two examples above yield correct results in spite of the fact that there is a carry-out of the MSB. The next section discusses overflow in two's complement addition in more detail.

Overflow

- When two numbers are added that have large magnitudes and the same sign, an **overflow will occur if the result is too large to fit in the number of bits used in** the representation.
- Consider adding $(+80)_{10}$ and $(+50)_{10}$ using an eight bit format.
- The result should be $(+130)_{10}$, however, as shown below, the result is $(-126)_{10}$:

$$\begin{array}{rl} 0\,1\,0\,1\,0\,0\,0\,0 & (+80)_{10} \\ +\,0\,0\,1\,1\,0\,0\,1\,0 & (+50)_{10} \\ \hline 1\,0\,0\,0\,0\,0\,1\,0 & (-126)_{10} \end{array}$$

This should come as no surprise, since we know that the largest positive 8-bit two's complement number is $+(127)_{10}$, and it is therefore impossible to represent $(+130)_{10}$. Although the result 10000010_2 "looks" like 130_{10} if we think of it in unsigned form, the sign bit indicates a negative number in the signed form, which is clearly wrong. In general, if two numbers of opposite signs are added, then an overflow cannot occur. Intuitively, this is because the magnitude of the result can be no larger than the magnitude of the larger operand. This leads us to the definition of two's complement overflow.

- *If the numbers being added are of the same sign and the result is of the opposite sign, then an overflow occurs and the result is incorrect.*
- *If the numbers being added are of opposite signs, then an overflow will never occur. As an alternative method of detecting overflow for addition, an overflow occurs if and only if the carry into the sign bit differs from the carry out of the sign bit.*
- *If a positive number is subtracted from a negative number and the result is positive, or if a negative number is subtracted from a positive number and the result is negative, then an overflow occurs.*
- *If the numbers being subtracted are of the same sign, then an overflow will never occur.*

HARDWARE IMPLEMENTATION OF ADDERS AND SUBTRACTORS

- Up until now we have focused on algorithms for addition and subtraction.
- Now we will take a look at implementations of simple adders and subtractors.

- The adder is modeled after the way that we normally perform decimal addition by hand, by summing digits in one column at a time while moving from right to left.
- In this section, we review the **ripple-carry adder, and then take a look at a ripple-borrow subtractor.**
- **We then combine the two into a single addition/subtraction** unit.

- Figure below shows a 4-bit ripple-carry adder. Two binary numbers *A and B are added from right to left, creating a sum and a carry* at the outputs of each full adder for each bit position.
- Four 4-bit ripple-carry adders are cascaded in Figure below to add two 16-bit numbers.
- The rightmost full adder has a carry-in of 0.
- Although the rightmost full adder can be simplified as a result of the carry-in of 0, we will use the more general form and force *c0 to 0 in order to simplify subtraction later on*

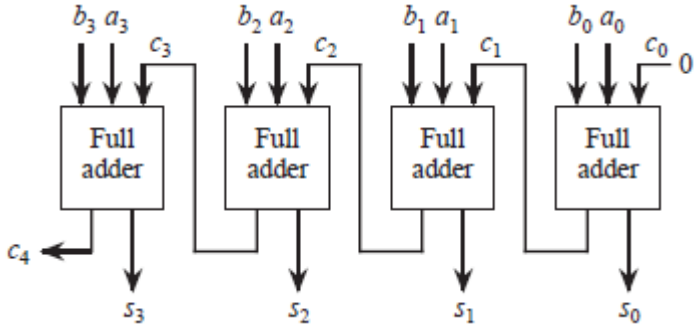


Figure : Ripple-carry adder.

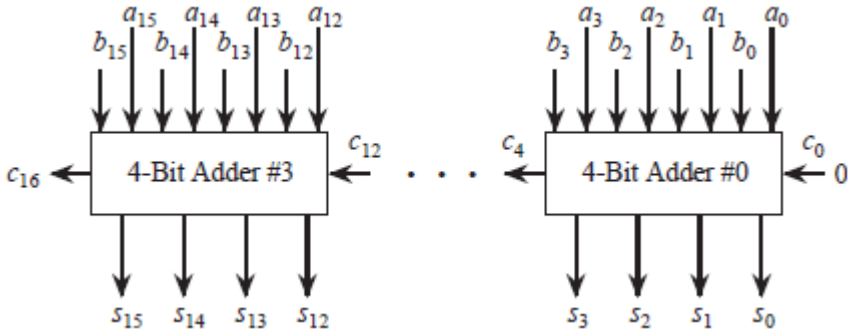
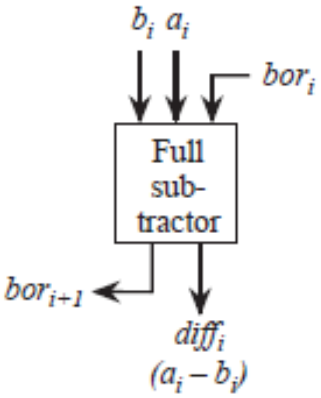


Figure A 16-bit adder is made up of a cascade of four 4-bit ripple-carry adders.

- **Subtraction of binary numbers proceeds in a fashion analogous to addition. We** can subtract one number from another by working in a single column at a time, subtracting digits of the **subtrahend b_i, *from the minuend ai, as we move from*** right to left.
- As in decimal subtraction, if the subtrahend is larger than the minuend or there is a borrow from a previous digit then a borrow must be propagated to the next most significant bit.
- Figure shows the truth table and a "black-box" circuit for subtraction.

a_i	b_i	bor_i	$diff_i$	bor_{i+1}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1



- Full subtractors can be cascaded to form **ripple-borrow subtractors in the same** manner that full adders are cascaded to form ripple-carry adders.
- Fig below illustrates a four-bit ripple-borrow subtractor that is made up of four full subtractors.

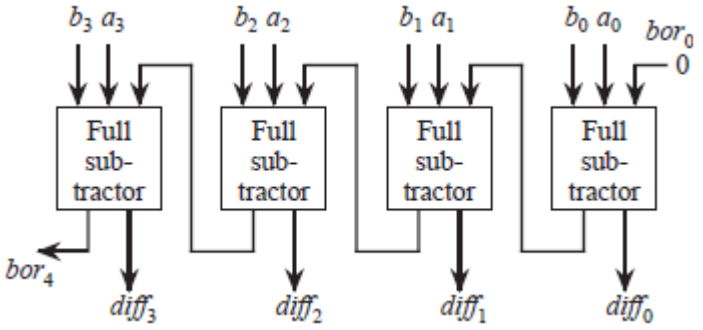


Figure: Ripple-borrow subtractor.

Figure: Truth table and schematic symbol for a ripple-borrow subtractor.

- As discussed before, an alternative method of implementing subtraction is to form the two's complement negative of the subtrahend and *add it to the minuend.*
- The circuit that is shown in Figure performs both addition and subtraction on four-bit two's complement numbers by allowing the *bi inputs to be* complemented when subtraction is desired. Add /SUBTRACT control line determines which function is performed. The bar over the ADD symbol indicates the ADD operation is active when the signal is low. That is, if the control line is 0, then the *ai and bi inputs are passed through to the adder, and the sum is* generated at the *si outputs. If the control line is 1, then the ai inputs are passed* through to the adder, but the *bi inputs are one's complemented by the XOR* gates before they are passed on to the adder.
- In order to form the two's complement negative, we must add 1 to the one's complement negative, which is accomplished by setting the *carry_in line (c0) to 1 with the control input.*
- *In this* way, we can share the adder hardware among both the adder and the subtractor.

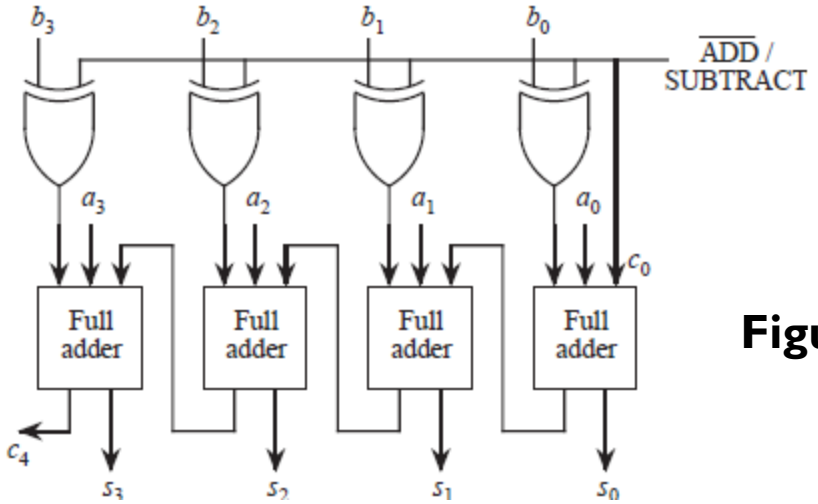


Figure: Addition / subtraction unit.

Fixed Point Multiplication and Division

- Multiplication and division of fixed point numbers can be accomplished with addition, subtraction, and shift operations.
- The sections that follow describe methods for performing multiplication and division of fixed point numbers in both unsigned and signed forms using these basic operations.
- We will first cover unsigned multiplication and division, and then we will cover signed multiplication and division.

UNSIGNED MULTIPLICATION

- Multiplication of unsigned binary integers is handled similar to the way it is carried out by hand for decimal numbers. Figure below illustrates the multiplication process for two unsigned binary integers.
- Each bit of the multiplier determines whether or not the multiplicand, shifted left according to the position of the multiplier bit, is added into the product.
- When two unsigned *n-bit numbers are* multiplied, the result can be as large as 2*n bits. For the example shown in Figure*, the multiplication of two four-bit operands results in an eight-bit product.
- When two signed *n-bit numbers are multiplied, the result can be as large as only* 2*(n-1)+1 = (2n-1) bits, because this is equivalent to multiplying two (n-1)-bit* unsigned numbers and then introducing the sign bit.

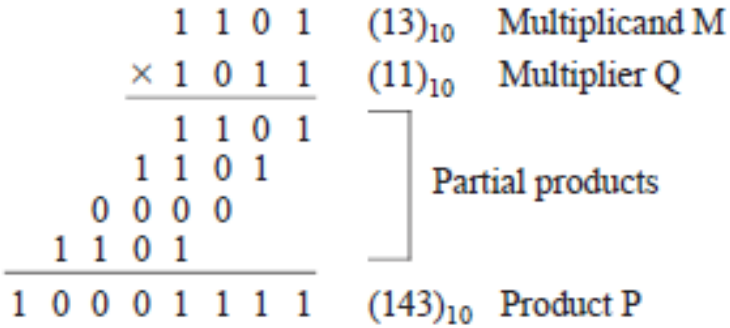


Figure: Multiplication of two unsigned binary integers.

- A hardware implementation of integer multiplication can take a similar form to the manual method. Figure below shows a layout of a multiplication unit for four-bit numbers, in which there is a four-bit adder, a control unit, three four-bit registers, and a one-bit carry register.
- In order to multiply two numbers, the multiplicand is placed in the M register, the multiplier is placed in the Q register, and the A and C registers are cleared to zero.
- During multiplication, the rightmost bit of the multiplier determines whether the multiplicand is added into the product at each step. After the multiplicand is added into the product, the multiplier and the A register are simultaneously shifted to the right.
- This has the effect of shifting the multiplicand to the left (as for the manual process) and exposing the next bit of the multiplier in position *q0*.

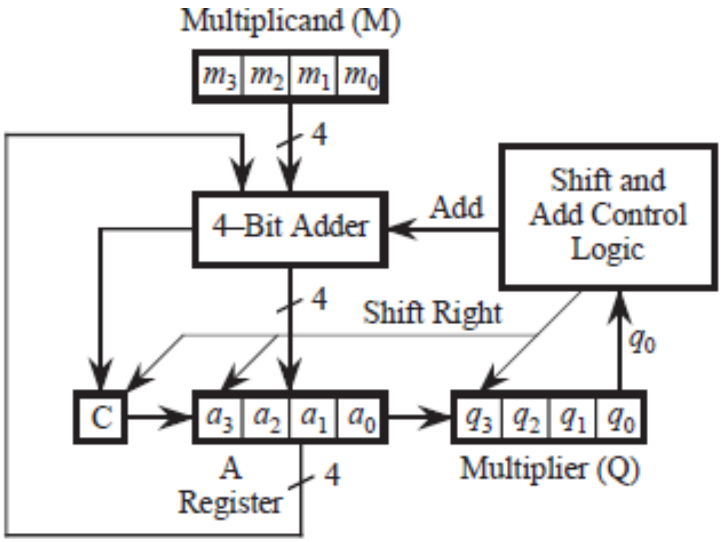


Figure: A serial multiplier.

- Figure below illustrates the multiplication process. Initially, C and A are cleared, and M and Q hold the multiplicand and multiplier, respectively.
- The rightmost bit of Q is 1, and so the multiplier M is added into the product in the A register.
- The A and Q registers together make up the eight-bit product, but the A register is where the multiplicand is added. After M is added to A, the A and Q registers are shifted to the right. Since the A and Q registers are linked as a pair to form the eight-bit product, the rightmost bit of A is shifted into the leftmost bit of Q.
- The rightmost bit of Q is then dropped, C is shifted into the leftmost bit of A, and a 0 is shifted into C.
- The process continues for as many steps as there are bits in the multiplier. On the second iteration, the rightmost bit of Q is again 1, and so the multiplicand is added to A and the C/A/Q combination is shifted to the right.
- On the third iteration, the rightmost bit of Q is 0 so M is not added to A, but the C/A/Q combination is still shifted to the right.
- Finally, on the fourth iteration, the rightmost bit of Q is again 1, and so M is added to A and the C/A/Q combination is shifted to the right.
- The product is now contained in the A and Q registers, in which A holds the high-order bits and Q holds the low-order bits.

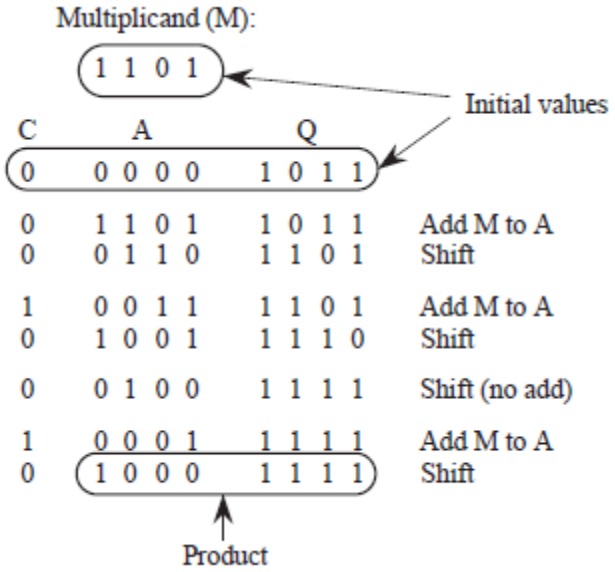


Figure: An example of multiplication using the serial multiplier.

UNSIGNED DIVISION

- In longhand binary division, we must successively attempt to subtract the divisor from the dividend, using the fewest number of bits in the dividend as we can.
- Figure below illustrates this point by showing that $(11)_2$ does not "fit" in 0 or 01, but *does fit in 011 as indicated by the pattern 001 that starts the quotient.*

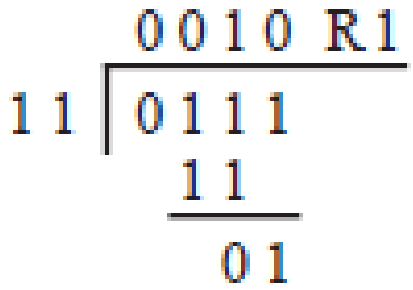


Figure: Example of base 2 division.

Computer-based division of binary integers can be handled similar to the way that binary integer multiplication is carried out, but with the complication that the only way to tell if the dividend does not "fit" is to actually do the subtraction and test if the remainder is negative. If the remainder is negative then the subtraction must be "backed out" by adding the divisor back in, as described next.

- In the division algorithm, instead of shifting the product to the right as we did for multiplication, we now shift the quotient to the left, and we subtract instead of adding.
- When two *n-bit unsigned numbers are being divided, the result is no* larger than *n bits*. Figure below shows a layout of a division unit for four-bit numbers in which there is a five-bit adder, a control unit, a four-bit register for the dividend Q, and two five-bit registers for the divisor M and the remainder A.
- Five-bit registers are used for A and M, instead of 4-bit registers as we might expect, because an extra bit is needed to indicate the sign of the intermediate result. Although this division method is for unsigned numbers, subtraction is used in the process and negative partial results sometimes arise, which extends the range from -16 through +15, thus there is a need for 5 bits to store intermediate results.

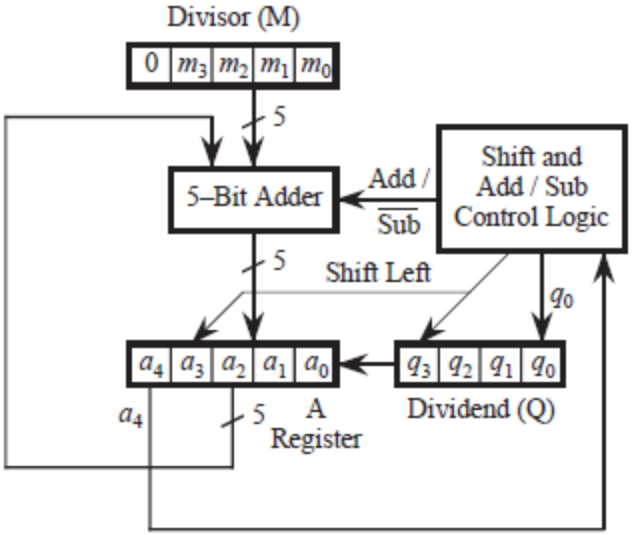


Figure: A serial divider.

- In order to divide two four-bit numbers, the dividend is placed in the Q register, the divisor is placed in the M register, and the A register and the high order bit of M are cleared to zero.
- The leftmost bit of the A register determines whether the divisor is added back into the dividend at each step. This is necessary in order to restore the dividend when the result of subtracting the divisor is negative, as described above.
- This is referred to as **restoring division, because the dividend is** restored to its former value when the remainder is negative.
- When the result is not negative, then the least significant bit of Q is set to 1, which indicates that the divisor "fits" in the dividend at that point.

- Figure illustrates the division process. Initially, A and the high order bit of M are cleared, and Q and the low order bits of M are loaded with the dividend and divisor, respectively.
- The A and Q registers are shifted to the left as a pair and the divisor M is subtracted from A. Since the result is negative, the divisor is added back to restore the dividend, and *q0 is cleared to 0*.
- *The process repeats by* shifting A and Q to the left, and by subtracting M from A. Again, the result is negative, so the dividend is restored and *q0 is cleared to 0*.
- *On the third iteration,* A and Q are shifted to the left and M is again subtracted from A, but now the result of the subtraction is not negative, so *q0 is set to 1*.
- *The process continues* for one final iteration, in which A and Q are shifted to the left and M is subtracted from A, which produces a negative result. The dividend is restored and *q0* is cleared to 0.
- The quotient is now contained in the Q register and the remainder is contained in the A register.

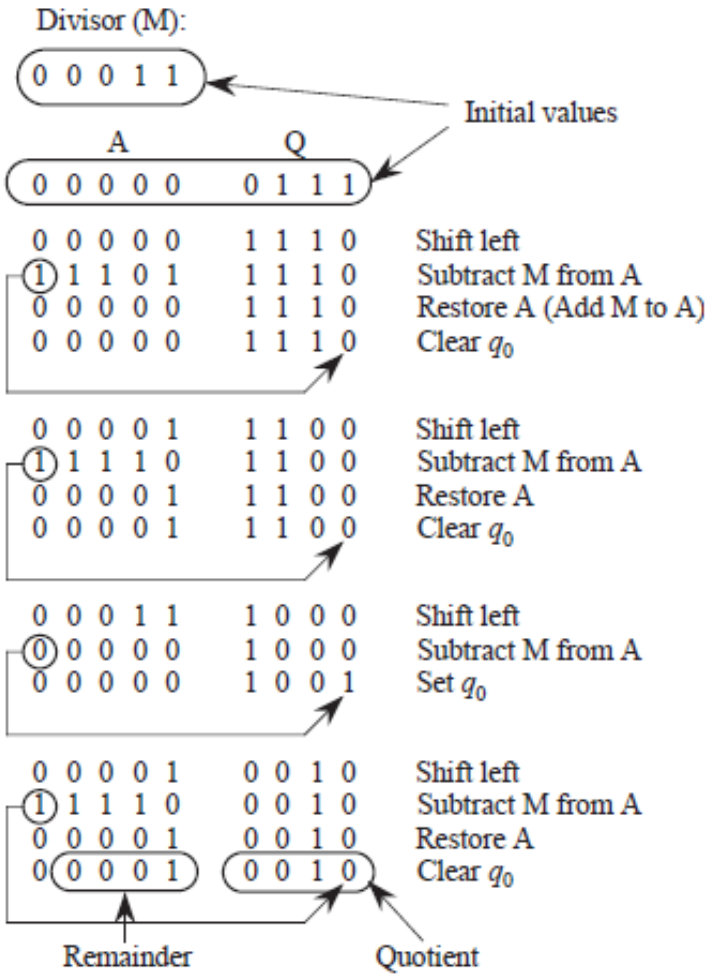


Figure: An example of division using the serial divider.

SIGNED MULTIPLICATION AND DIVISION

- If we apply the multiplication and division methods described in the previous sections to signed integers, then we will run into some trouble.
- Consider multiplying -1 by +1 using four-bit words, as shown in the left side of Figure beside.
- The eight-bit equivalent of +15 is produced instead of -1.
- What went wrong is that the sign bit did not get extended to the left of the result.
- This is not a problem for a positive result because the high order bits default to 0, producing the correct sign bit 0.
- A solution is shown in the right side of this Figure in which each partial product is extended to the width of the result, and only the rightmost eight bits of the result are retained.
- If both operands are negative, then the signs are extended for both operands, again retaining only the rightmost eight bits of the result.

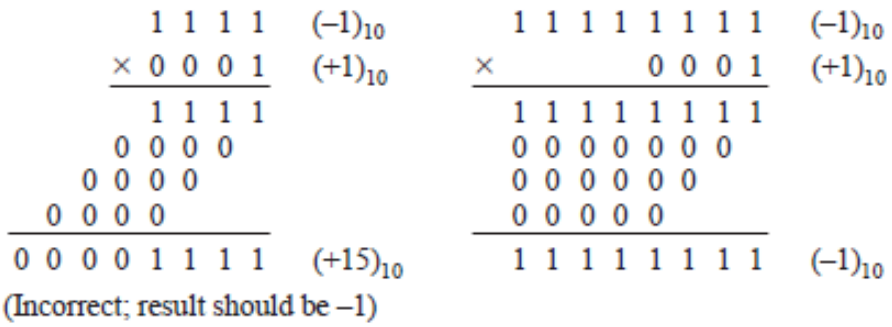


Figure: Multiplication of signed integers.

Signed division is more difficult. We will not explore the methods here, but as a general technique, we can convert the operands into their positive forms, perform the division, and then convert the result into its true signed form as a final step.

Floating Point Arithmetic

- Arithmetic operations on floating point numbers can be carried out using the fixed point arithmetic operations described in the previous sections, with attention given to maintaining aspects of the floating point representation.

- In the sections that follow, we explore floating point arithmetic in base 2 and base 10, keeping the requirements of the floating point representation in mind.

FLOATING POINT ADDITION AND SUBTRACTION

- Floating point arithmetic differs from integer arithmetic in that exponents must be handled as well as the magnitudes of the operands.
- As in ordinary base 10 arithmetic using scientific notation, the exponents of the operands must be made equal for addition and subtraction.
- The fractions are then added or subtracted as appropriate, and the result is normalized.
- This process of adjusting the fractional part, and also rounding the result can lead to a loss of precision in the result.
- Consider the unsigned floating point addition ($.101 \times 2^3 + .111 \times 2^4$) in which the fractions have three significant digits.
- We start by adjusting the *smaller exponent to be equal to the larger exponent,* and adjusting the fraction accordingly. Thus we have $.101 \times 2^3 = .010 \times 2^4$, losing $.001 \times 2^3$ of precision in the process. The resulting sum is:

$$(.010 + .111) \times 2^4 = 1.001 \times 2^4 = .1001 \times 2^5,$$

and rounding to three significant digits, $.100 \times 2^5$, and we have lost another 0.001×2^4 in the rounding process.

Why do floating point numbers have such complicated formats?

- We may wonder why floating point numbers have such a complicated structure, with the mantissa being stored in signed magnitude representation, the exponent stored in excess notation, and the sign bit separated from the rest of the magnitude by the intervening exponent field.
- There is a simple explanation for this structure. Consider the complexity of performing floating point arithmetic in a computer. Before any arithmetic can be done, the number must be unpacked from the form it takes in storage. (e.g. IEEE 754 floating point format.)
- The exponent and mantissa must be extracted from the packed bit pattern before an arithmetic operation can be performed; after the arithmetic operation(s) are performed, the result must be renormalized and rounded, and then the bit patterns are re-packed into the requisite format.
- The virtue of a floating point format that contains a sign bit followed by an exponent in excess notation, followed by the magnitude of the mantissa, is that two floating point numbers can be compared for >, <, and = without unpacking.
- The sign bit is most important in such a comparison, and it appropriately is the MSB in the floating point format. Next most important in comparing two numbers is the exponent, since a change of ± 1 in the exponent changes the value by a factor of 2 (for a base 2 format), whereas a change in even the MSB of the fractional part will change the value of the floating point number by less than that.
- In order to account for the sign bit, the signed magnitude fractions are represented as integers and are converted into two's complement form. After the addition or subtraction operation takes place in two's complement, there may be a need to normalize the result and adjust the sign bit.
- The result is then converted back to signed magnitude form.

FLOATING POINT MULTIPLICATION AND DIVISION

- Floating point multiplication and division are performed in a manner similar to floating point addition and subtraction, except that the sign, exponent, and fraction of the result can be computed separately.
- If the operands have the same sign, then the sign of the result is positive. Unlike signs produce a negative result.
- The exponent of the result before normalization is obtained by adding the exponents of the source operands for multiplication, or by subtracting the divisor exponent from the dividend exponent for division.
- The fractions are multiplied or divided according to the operation, followed by normalization.

- Consider using three-bit fractions in performing the base 2 computation: $(+.101 \times 2^{2}) \times (-.110 \times 2^{-3})$. The source operand signs differ, which means that the result will have a negative sign.
- We add exponents for multiplication, and so the exponent of the result is 2 + -3 = -1.
- We multiply the fractions, which produces the product .01111.
- Normalizing the product and retaining only three bits in the fraction produces $-.111 \times 2^{-2}$.
- Now consider using three-bit fractions in performing the base 2 computation: $(+.110 \times 2^{5}) / (+.100 \times 2^{4})$. The source operand signs are the same, which means that the result will have a positive sign.
- We subtract exponents for division, and so the exponent of the result is 5 – 4 = 1. We divide fractions, which can be done in a number of ways.
- If we treat the fractions as unsigned integers, then we will have 110/100 = 1 with a remainder of 10.
- What we really want is a contiguous set of bits representing the fraction instead of a separate result and remainder, and so we can scale the dividend to the left by two positions, producing the result: 11000/100 = 110.
- We then scale the result to the right by two positions to restore the original scale factor, producing 1.1.
- Putting it all together, the result of dividing $(+.110 \times 2^{5})$ by $(+.100 \times 2^{4})$ produces $(+1.10 \times 2^{1})$.
- After normalization, the final result is $(+.110 \times 2^{2})$.

SUMMARY of Module-3

- *Computer arithmetic can be carried out as we normally carry out decimal arithmetic by hand, while taking the base into account.*
- *A two's complement or a ten's complement representation is normally used for integers, whereas signed magnitude is normally used for fractions due to the difficulty of manipulating positive and negative fractions in a uniform manner.*