Binary Arithmetic

Bob Brown
Computer Science Department
Southern Polytechnic State University

Introduction

Arithmetic is at the heart of the digital computer, and the majority of arithmetic performed by computers is binary arithmetic, that is, arithmetic on base two numbers. Decimal and floating-point numbers, also used in computer arithmetic, depend on binary representations, and an understanding of binary arithmetic is necessary in order to understand either one.

Computers perform arithmetic on fixed-size numbers. The arithmetic of fixed-size numbers is called finite-precision arithmetic. The rules for finite-precision arithmetic are different from the rules of ordinary arithmetic.

The sizes of numbers which can be arithmetic operands are determined when the architecture of the computer is designed. Common sizes for integer arithmetic are eight, 16, 32, and recently 64 bits. It is possible for the programmer to perform arithmetic on larger numbers or on sizes which are not directly implemented in the architecture. However, this is usually so painful that the programmer picks the most appropriate size implemented by the architecture. This puts a burden on the computer architect to select appropriate sizes for integers, and on the programmer to be aware of the limitations of the size he has chosen and on finite-precision arithmetic in general.

We are considering binary arithmetic in the context of building digital logic circuits to perform arithmetic. Not only do we have to deal with the fact of finite-precision arithmetic, we must consider the complexity of the digital logic. When there is more than one way of performing an operation we choose the method which results in the simplest circuit.

Finite-Precision Arithmetic

Consider what it would be like to perform arithmetic if one were limited to three-digit decimal numbers. Neither negative numbers nor fractions could be expressed directly, and the largest possible number that could be expressed is 999. This is the circumstance in which we find ourselves when we perform computer arithmetic because the number of bits is fixed by the computer's architecture. Although we can usually express numbers larger than 999, the limits are real and small enough to be of practical concern. Working with unsigned 16-bit binary integers, the largest number we can express is $2^{16}-1$, or 65,535. If we assume a signed number, the largest number is 32,767.

There are other limitations. Consider again the example of three-digit numbers. We can add 200 + 300, but not 600 + 700 because the latter sum is too large to fit in three

digits. Such a condition is called *overflow* and it is of concern to architects of computer systems. Because not all operations which will cause overflow can be predicted when a computer program is written, the computer system itself must check whether overflow has occurred and, if so, provide some indication of that fact.

Tanenbaum points out that the algebra of finite-precision is different from ordinary algebra, too. Neither the associative law nor the distributive law applies. Two examples from Tanenbaum illustrate this. If we evaluate the expression

$$a + (b - c) = (a + b) - c$$

using a = 700, b= 400, and c=300, the left-hand side evaluates to 800, but overflow occurs when evaluating $a + b$ in the right-hand side. The associative law does not hold.

Similarly if we evaluate

$$a \times (b - c) = a \times b - a \times c$$

using a = 5, b = 210, and c = 195, the left-hand side produces 75, but in the right-hand side, $a \times b$ overflows and distributive law does not hold.

These two examples show the importance of understanding the limitations on computer arithmetic. This understanding is important to programmers as well as designers of computers.

Addition

The rules for binary addition are the same as those for any positional number system. One adds the digits column-wise from the right. If the sum is greater than B–1 for base B, a carry into the next column is generated. In the case of binary numbers, a sum greater than one generates a carry. Here is the binary addition table:

				1
0	0	1	1	+1
+0	+1	+0	+1	+1
0	1	1	10	11

The first three entries are self-explanatory. The third entry is $1+1=10_2$, or one plus one is two; we have a sum of zero and a carry of one into the two's place. The fourth entry is $1+1+1=11_2$, or three ones are three. The sum is one and there is a carry into the two's place.

Now we will add two binary numbers with more than one bit each so you can see how the carries "ripple" left, just as they do in decimal addition.

```
  1 1 1
  0 0 1 1 0
+ 0 1 1 1 1
  1 0 1 0 1
```

The three carries are shown on the top row. Normally, you would write these down as you complete the partial sum for each column. Adding the rightmost column

produces a one with no carry; adding the next column produces a zero with one to carry. Work your way through the entire example from right to left.

One can also express the rules of binary addition with a *truth table*. This is important because there are techniques for designing electronic circuits that compute functions expressed by truth tables. The fact that we can express the rules of binary addition as a truth table implies that we can design a circuit which will perform addition on binary numbers, and that turns out to be the case.

We only need to write the rules for one column of bits; we start at the right and apply the rules to each column in succession until the final sum is formed. Call the bits of the addend and augend **A** and **B**, and the carry in from the previous column $\mathbf{C}_i$. Call the sum **S** and the carry out $\mathbf{C}_o$. The truth table for one-bit binary addition looks like this:

A	B	C_i	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

This says if all three input bits are zero, both **S** and $\mathbf{C_o}$ will be zero. If any one of the bits is one and the other two are zero, **S** will be one and $\mathbf{C_o}$ will be zero. If two bits are ones, **S** will be zero and $\mathbf{C_i}$ will be one. Only if all three bits are ones will both **S** and $\mathbf{C_o}$ be ones.

Negative Numbers

For pencil-and-paper arithmetic we could represent signed binary numbers with plus and minus signs, just as we do with decimal numbers. With computer circuits, our only symbols are zero and one. We must devise a way of representing negative numbers using only zeroes and ones. There are four possible approaches: signed magnitude, one's complement, two's complement, and excess 2^{n-1}. The first three of these take advantage of the fact that in computers numbers are represented in fixed-size fields. The leftmost bit is considered the sign bit. In signed-magnitude representation, a zero in the sign bit indicates a positive number, and a one indicates a negative number. The one's complement is formed by complementing each bit of the binary number. Again a zero in the sign bit indicates a positive number and a one indicates a negative number. Signed-magnitude and excess 2^{n-1} numbers are used in floating point, and will be discussed there. One's complement arithmetic is obsolete.

Two's complement numbers are used almost universally for integer representation of numbers in computers. In the binary number system, we can express any non-negative integer as the sum of coefficients of powers of two:

$$a_{n-1}\times 2^{n-1} + a_{n-2}\times 2^{n-2} + \ \ldots + a_1\times 2^1 \ + a_0\times 2^0 \ = \sum_{i=0}^{n-1} a_i 2^i$$

One way of looking at two's complement numbers is to consider that the leftmost bit, or sign bit, represents a negative coefficient of a power of two and the remaining bits represent positive coefficients which are added back. So, an *n*-bit two's complement number has the form

$$-2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

Consider 10000000, an eight-bit two's complement number. Since the sign bit is a one, it represents -2^7 or –128. The remaining digits are zeroes, so 10000000 = –128. The number 10000001 is –128+1 or –127. The number 10000010 is –126, and so on. 11111111 is –128 + 127 or –1.

Now consider 01111111, also an eight-digit two's complement number. The sign bit still represents -2^7 or –128, but the coefficient is zero, and this is a positive number, +127.

The two's complement representation has its own drawback. Notice that in eight bits we can represent –128 by writing 10000000. The largest positive number we can represent is 01111111 or +127. Two's complement is *asymmetric about zero.* For any size binary number, there is one more negative number than there are positive numbers. This is because, for any binary number, the number of possible bit combinations is even. We use one of those combinations for zero, leaving an odd number to be split between positive and negative. Since we want zero to be represented by all binary zeros and we want the sign of positive numbers to be zero, there's no way to escape from having one more negative number than positive.

If you think of a two's complement number as a large negative number with positive numbers added back, you could conclude that it would be difficult to form the two's complement. It turns out that there's a method of forming the two's complement that is very easy to do with either a pencil or a computer:

- Take the complement of each bit in the number to be negated. That is, if a bit is a zero, make it a one, and vice-versa.
- To the result of the first step, add one as though doing unsigned arithmetic.

Let's do an example: we will find the two's complement representation of –87. We start with the binary value for 87, or 01010111. Here are the steps:

01010111	original number
10101000	each bit complemented, or "flipped"
+ 1	add 1 to 10101000
10101001	this is the two's complement, or –87.

We can check this out. The leftmost bit represents –128, and the remaining bits have positive values which are added back. We have –128 + 32 + 8 + 1, or –128 + 41 = –87. There's another way to check this. If you add equivalent negative and positive numbers, the result is zero, so –87 + 87 = 0. Does 01010111 + 10101001 = 0? Perform the addition and see.

In working with two's complement numbers, you will often find it necessary to adjust the length of the number, the number of bits, to some fixed size. Clearly, you can expand the size of a positive (or unsigned) number by adding zeroes on the left, and you can reduce its size by removing zeroes from the left. If the number is to be considered a two's complement positive number, you must leave at least one zero on the left in the sign bit's position.

It's also possible to expand the size of a two's complement negative number by supplying one-bits on the left. That is, if 1010 is a two's complement number, 1010 and 11111010 are equal. 1010 is –8+2 or –6. 11111010 is –128+64+32+16+8+2 or –6. Similarly you can shorten a negative number by removing ones from the left so long as at least one one-bit remains.

We can generalize this notion. A two's complement number can be expanded by replicating the sign bit on the left. This process is called *sign extension.* We can also shorten a two's complement number by deleting digits from the left so long as at least one digit identical to the original sign bit remains.

Addition of Signed Numbers

Binary addition of two's complement signed numbers can be performed using the same rules given above for unsigned addition. If there is a carry out of the sign bit, it is ignored.

Since we are dealing with finite-precision arithmetic, it is possible for the result of an addition to be too large to fit in the available space. The answer will be truncated, and will be incorrect. This is the overflow condition discussed above. There are two rules for determining whether overflow has occurred:

- If two numbers of opposite signs are added, overflow cannot occur.
- If two numbers of the same sign are added, overflow has occurred if and only if the result is of the opposite sign.

Subtraction

Addition has the property of being commutative, that is, $a+b = b+a$. This is not true of subtraction. 5 – 3 is not the same as 3 – 5. For this reason, we must be careful of the order of the operands when subtracting. We call the first operand, the number which

is being diminished, the minuend; the second operand, the amount to be subtracted from the minuend, is the subtrahend. The result is called the difference.

```
   51    minuend
– 22     subtrahend
  29     difference.
```

It is possible to perform binary subtraction using the same process we use for decimal subtraction, namely subtracting individual digits and borrowing from the left. This process quickly becomes cumbersome as you borrow across successive zeroes in the minuend. Further, it doesn't lend itself well to automation. Jacobowitz describes the "carry" method of subtraction which some of you may have learned in elementary school, where a one borrowed in the minuend is "paid back" by adding to the subtrahend digit to the left. This means that one need look no more than one column to the left when subtracting. Subtraction can thus be performed a column at a time with a carry to the left, analogous to addition. This is a process which can be automated, but we are left with difficulties when the subtrahend is larger than the minuend or when either operand is signed.

Since we can form the complement of a binary number easily and can add signed numbers easily, the obvious answer to the problem of subtraction is to take the two's complement of the subtrahend, then add add it to the minuend. We aren't saying anything more than that 51–22 = 51+(–22). Not only does this approach remove many of the complications of subtraction by the usual method, it means we don't have to build special circuits to perform subtraction. All we need is a circuit which can form the bitwise complement of a number and an adder.

Multiplication

A simplistic way to perform multiplication is by repeated addition. In the example below, we could add 42 to the product register 27 times. In fact, some early computers performed multiplication this way. However, one of our goals is speed, and we can do much better using the familiar methods we have learned for multiplying decimal numbers. Recall that the multiplicand is multiplied by each digit of the multiplier to form a partial product, then the partial products are added to form the total product. Each partial product is shifted left to align on the right with its multiplier digit.

```
   42   multiplicand
×  27   multiplier
  294   first partial product (42 × 7)
 84     second partial product (42 × 2)
1134    total product.
```

Binary multiplication of unsigned (or positive two's complement) numbers works exactly the same way, but is even easier because the digits of the multiplier are all either

zero or one. That means the partial products are either zero or a copy of the multiplicand, shifted left appropriately. Consider the following binary multiplication:

```
   0111    multiplicand
× 0101     multiplier
  ----
   0111    first partial product (0111 × 1)
  0000     second partial product (0111 × 0)
 0111      third partial product (0111 × 1)
0000       fourth partial product (0111 × 0)
-------
0100011    total product.
```

Notice that no true multiplication is necessary in forming the partial products. The fundamental operations required are shifting and addition. This means we can multiply unsigned or positive integers using only shifters and adders.

With pencil-and-paper multiplication, we form all the partial products, then add them. It isn't necessary to do that; we could simply keep a running sum. When the last partial product is added, the running sum will be the total product. We can now state an algorithm for binary multiplication suitable for a computer implementation:

- If the rightmost digit of the multiplier is a one, copy the multiplicand to the product, otherwise set the product to zero
- For each successive digit of the multiplier, shift the multiplicand left one bit; then, if the multiplier digit is a one, add the shifted multiplicand to the product. The algorithm terminates when all the digits of the multiplier have been examined.

Since the underlying arithmetic operation is addition, the possibility of overflow exists. We handle the possibility a little differently in multiplication. If two *n*-bit numbers are multiplied, the largest possible product is 2*n* bits. Multiplication is usually implemented such that the register which receives the product is twice as large as the operand registers. In that case, overflow cannot occur.

Notice also that if the multiplier is *n* bits long, the multiplicand will have been shifted left *n* bits by the time the algorithm terminates. For this reason, multiplication algorithms make a copy of the multiplicand in a register 2*n* bits wide. Examination of the bits of the multiplier is often performed by shifting a copy of the multiplier right one bit at a time. This is because shift operations often save the last bit "shifted out" in a way that is easy to examine.

Unfortunately, this algorithm does not work for signed numbers. If the multiplicand is negative, the partial products must be sign-extended so that they form 2*n*-bit negative numbers. If the multiplier is negative, the situation is even worse; the bits of the multiplier no longer specify an appropriately-shifted copy of the multiplicand. One way around this dilemma would be to take the two's complement of negative operands, perform the multiplication, then take the two's complement of the product if the multi-

plier and multiplicand are of different signs. This approach would require a considerable amount of time before and after the actual multiplication, and so is usually rejected in favor of a faster but less straightforward algorithm. One such algorithm is Booth's Algorithm, which is discussed in detail in Stallings.

Division

As with the other arithmetic operations, division is based on the paper-and-pencil approach we learned for decimal arithmetic. We will show an algorithm for unsigned long division that is essentially similar to the decimal algorithm we learned in grade school. Let us divide 0110101 (53_{10}) by 0101 (5_{10}). Beginning at the left of the dividend, we move to the right one digit at a time until we have identified a portion of the dividend which is greater than or equal to the divisor. At this point a one is placed in the quotient; all digits of the quotient to the left are assumed to be zero. The divisor is copied below the partial dividend and subtracted to produce a partial remainder as shown below.

```
                  1        quotient
divisor  0101 ⟌0110101     dividend
              0101
                 1         partial remainder
```

Now digits from the dividend are "brought down" into the partial remainder until the partial remainder is again greater than or equal to the divisor. Zeroes are placed in the quotient until the partial remainder is greater than or equal to the divisor, then a one is placed in the quotient, as shown below.

```
                 101
0101 ⟌0110101
     0101↓↓
        110
```

The divisor is copied below the partial remainder and subtracted from it to form a new partial remainder. The process is repeated until all bits of the dividend have been used. The quotient is complete and the result of the last subtraction is the remainder.

```
                1010
0101 ⟌0110101
     0101   |
        110 |
       0101↓
          11
```

This completes the division. The quotient is 1010_2 (10_{10}) and the remainder is 11_2 (3_{10}), which is the expected result. This algorithm works only for unsigned numbers, but it is possible to extend it to two's complement numbers. As with the other algorithms, it can be implemented using only shifting, complementation, and addition.

Summary

The four arithmetic operations on binary numbers are performed in much the same way as they are performed on decimal numbers. By using two's complement to represent negative numbers, we can perform all four operations using only circuits which shift, complement, and add.

Computers operate on numbers of fixed size. For this reason, the rules of finite-precision arithmetic apply to computer arithmetic. Programmers, as well as designers of computer equipment, must be aware of the limitations of finite-precision arithmetic.

Bibliography

Jacobowitz, Henry, *Computer Arithmetic*, John F. Rider Publisher, 1962.

Stallings, William, *Computer Organization and Architecture, Fourth Edition,* Prentice-Hall, 1996.

Tanenbaum, Andrew S., *Structured Computer Organization, Third edition,* Prentice-Hall, 1990.

Youse, Bevan K., *The Number System,* Dickenson Publishing Company, 1965.

Exercises

1. Add the following unsigned binary numbers:

00101011	01100001	01000010	01111111	01111111
01001100	00001111	01010101	00000001	00000010

2. Convert the following unsigned binary numbers to two's complement:

01101111	00011000	00110101	01011011	01011000

3. Compute 01101010×01110111. Show your work.

4. Divide 01011 into 011011000. Show your work.

5. Convert 137 and 42 to binary. Compute the binary equivalent of $137 - 42$. Show your work.

6. Compute $42 - 137$ in binary. Show your work