Signed Number Representations
CS221, Mock

So far we have discussed unsigned number representations. In particular, we have looked at the binary number system and shorthand methods in representing binary codes. With *m* binary digits, we can represent the 2^m unique patterns, from 000….0 to 111….1. When we try to represent signed quantities in the same m digits, we still only have 2^m patterns to work with. Unless we increase the number of digits available (i.e. make *m* larger), the representation of signed numbers will involve dividing these 2^m patterns into positive and negative portions. The two techniques we will look at to do this is the sign-magnitude representation and two's complement.

Sign/Magnitude Notation

Sign/magnitude notation is the simplest and one of the most obvious methods of encoding positive and negative numbers. Assign the leftmost (most significant) bit to be the **sign bit**. If the sign bit is 0, this means the number is positive. If the sign bit is 1, then the number is negative. The remaining bits are used to represent the magnitude of the binary number in the unsigned binary notation.

Example:

Binary	Value
0000	+0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7
1000	-0
1001	-1
1010	-2
1011	-3
1100	-4
1101	-5
1110	-6
1111	-7

Looking at the list you should notice an immediate peculiarity; there are two representations for zero! There is positive zero, and negative zero. This can cause complications for computers checking numbers for equality. Another disadvantage is performing addition or subtraction – we require a special consideration of both signs of the numbers to properly compute the operation. Because of these drawbacks, sign-magnitude is rarely used for representing integers.

Radix Complementation – Two's Complement

Radix complementation is used to represent signed quantities and is based on the ideas of modular arithmetic. In modular arithmetic, there is a value called the Modulus (*M*) which when added to or subtracted from a number, does not change its value.

If we are representing the integer A, where A is composed of n bits, then if A is positive the sign bit, A_{n-1} is zero. The remaining n-1 bits represent the magnitude of the number as in sign magnitude:

Binary	Two's Complement Value
0000	0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7

Using four bits, the largest positive number we can represent is +7 since the first bit must be a 0 to denote positive.

For a negative value for A, the sign bit, A_{n-1} is one instead of zero. The remaining n-1 bits are used to represent the negative integers from –1 to -2^{n-1}. Note that we will have the ability to represent one additional negative integer than positive integers, because we're using up one of the patterns starting with 0 to represent 0.

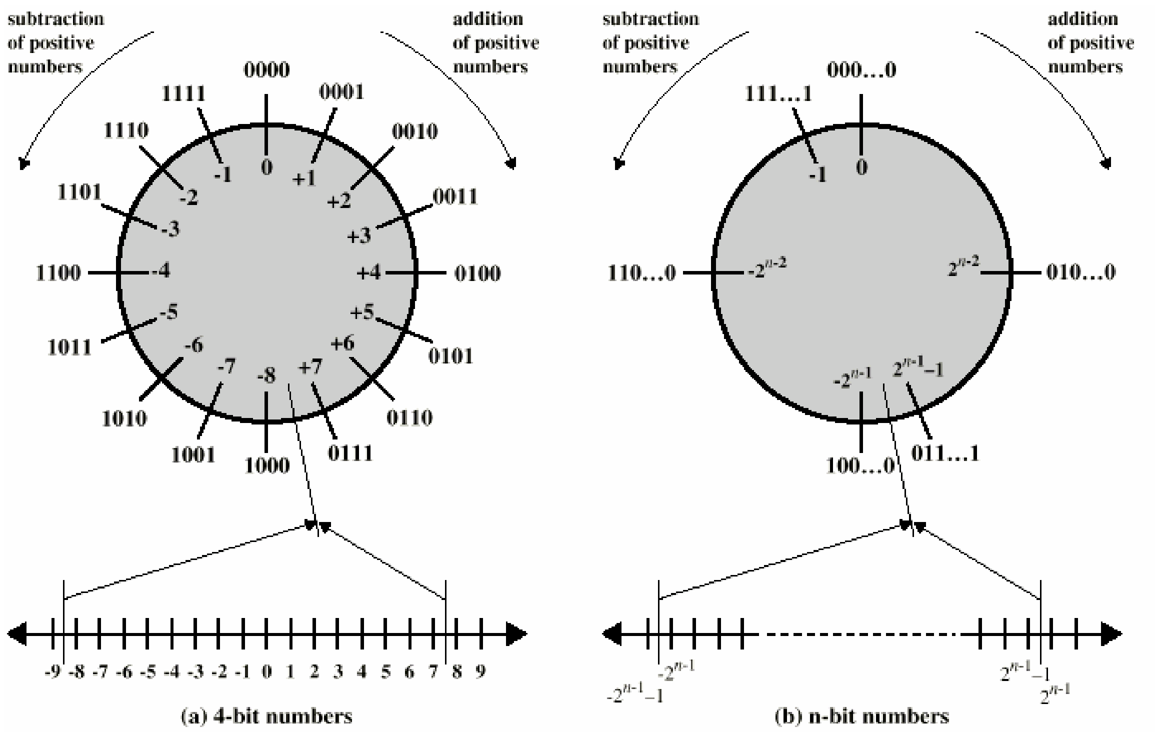
We would like to assign the negative integers to the available bit patterns in a way that facilitates straightforward arithmetic. A method that does this is to use the following formula:

$$Val = -2^{n-1} A_{n-1} + \sum_{i=0}^{n-2} 2^{i} A_{i}$$

Consider a positive number. In this case, A_{n-1} is zero. We end up with only the summation of the remaining terms.

Now consider a negative number. The A_{n-1} term is one, so we must add in -2^{n-1}. Now consider if all of the remaining bits are all one's. These will all add up to be $+2^{n-1}$ -1. It will be one smaller than the negative value; e.g. 1111 = yields $-2^3 + 7$, or $-8 + 7 = -1$. No matter how many bits we have, if they are all ones, we will end up with –1.

Now consider if all of the bits are one's except for the rightmost bit. We have the same case as before, except the positive value will be $+2^{n-1} - 2$ since we just subtracted one from the positive value. When we add this to the negative value, we end up with –2. The end result is we are counting backwards with the negative values, instead of counting forward as with positive values. This is shown in the "circle" below:



This representation has the benefit that if we start at any number on the circle, we can add positive k (or subtract negative k) from that number by moving k position clockwise or counterclockwise. If an arithmetic operation results in traversal of the point where the endpoints are joined, an incorrect answer is given. However, we are guaranteed that if we add a positive and a negative value together, we will result in a value that is possible to represent using the number of bits available.

A complete binary table for four bits is shown below:

Binary	Two's Comp Value	Binary	Two' Comp
0000	0	1111	-1
0001	+1	1110	-2
0010	+2	1101	-3
0011	+3	1100	-4
0100	+4	1011	-5
0101	+5	1010	-6
0110	+6	1001	-7
0111	+7	1000	-8

To summarize, two's complement lets us have only one representation for zero and allows us to easily perform arithmetic operations without special cases for sign bits.

If we are given a decimal value, A, that we want to represent in two's complement, there is an easy way to do it:

1. If A is positive, represent it using the sign-magnitude representation. The leftmost bit must be 0, and the remaining bits are the binary for the integer. Be careful there are enough bits available to represent the number.

2. If A is negative, first represent in binary +A.
   a. Flip all the 1's to 0's and the 0's to 1's
   b. Add 1 to the result using unsigned binary notation

If you are given a binary value in two's complement and want to know what the value is in decimal, then the process is to:

1. If the leftmost bit is 0, the number is positive. Compute the magnitude as an unsigned binary number.

2. If the leftmost bit is 1, the number is negative.
   a. Flip all the 1's to 0's and the 0's to 1's
   b. Add 1 to the result using unsigned binary notation
   c. Compute the value as if it were an unsigned binary value, say it is B. This is the magnitude of the negative number.
   d. The actual value is -B

Examples: Assume that we have 5 bits available.

What is –5 in twos complement?
    +5 in unsigned binary is 00101 (and +5 is 00101 in twos complement)
    Flip the bits to get 11010
    Now add 1: 11011
    The answer is 11011

What is –7 in two's complement?
    +7 in unsigned binary is 00111 (and +7 is 00111 in twos complement)
    Flip the bits to get 11000
    Now add 1: 11001

What is the decimal value of the two's complement binary value 11100?
    Leftmost bit is 1, so flip bits: 00011
    Add one : 00100
    This is 4, so the answer is –4.

What is the decimal value of the two's complement binary value 11011?
    Leftmost bit is 1, so flip the bits: 00100
    Add one: 00101
    This is 5, so the answer is –5

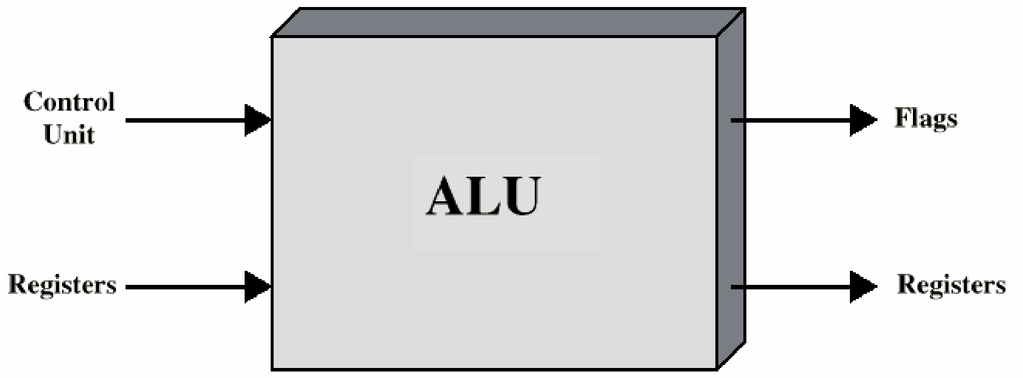
Why does this procedure of flipping the bits and adding one work? We won't prove it (a good way of doing so is by going over the Modulus definition of two's complement). However, to give you what is hopefully a convincing argument look once again at the table below:

Binary	Two's Comp Value	Binary	Two' Comp
0000	0	1111	-1
0001	+1	1110	-2
0010	+2	1101	-3
0011	+3	1100	-4
0100	+4	1011	-5
0101	+5	1010	-6
0110	+6	1001	-7
0111	+7	1000	-8

This table is arranged so that each row has the binary representation if the bits are flipped. For example, 0000 flipped is 1111. However, 0000=0 while 1111=-1. The numbers are off by one. If you look at each row, every value is off by one. This is why after we flip the bits of a negative representation, if we add one, we get back the magnitude of the negative value.

Computer Arithmetic

When the computer performs mathematical operations, these are executed inside the ALU (Arithmetic Logic Unit). To some degree, everything else inside the computer is there to service the ALU. ALU's today can handle integers and many also handle floating point (real) numbers. Some systems have a separate FPU (Floating Point Unit) instead of bundling this with the ALU. For example, older Intel processors had a separate math co-processor on a separate chip.



As shown in the picture above, the control unit determines when/what data the ALU gets. Registers feed data into the ALU, and the ALU in turn outputs the result of computations to registers and also to flags that will contain critical information regarding the status of the arithmetic operation (e.g., could indicate if it was successful). The flags are typically stored as bit values on a "flags" register.

Addition and Subtraction

Addition in two's complement proceeds just like normal addition you would do with decimal numbers, except instead you are adding with binary values. Each digit is added using the following rules

```
0 + 0 = 0
1 + 0 = 1
1 + 1 = 0 (carry of 1 to next column)
1 + 1 + 1 = 1 (carry of 1 to next column)
```

There may be a carry beyond the end of the calculation, which is ignored. However, we can't ignore the possibility of **overflow**. This occurs when we try to represent a value that is too large to hold in the number of bits available.

To check for overflow we use the following rules:

If adding two positive numbers, if the result is negative, there is overflow

If adding two negative numbers, if the result is positive, there is overflow

It is impossible to have overflow when adding a positive and negative number.

Here are some examples:

```
      0010        (+2)
+     1001        (-7)
      ------
      1011
```

There is no overflow since we added a positive and negative number.
To convert 1011 to decimal, note the leading 1 indicating a negative value.
Flip the bits to get 0100, add 1 to get 0101, and the result is –5.

```
      1111        (-1)
+     1001        (-7)
      ------
     11000
```

We discard the last carry of 1, giving us 1000.
There is no overflow since we added two negative numbers and got a negative one.
To convert 1000 to decimal, note the leading 1 indicating a negative value.
Flip the bits to get 0111 , add 1 to get 1000, and the result is –8.

```
      0001        (+1)
+     0111        (+7)
      ------
      1000
```

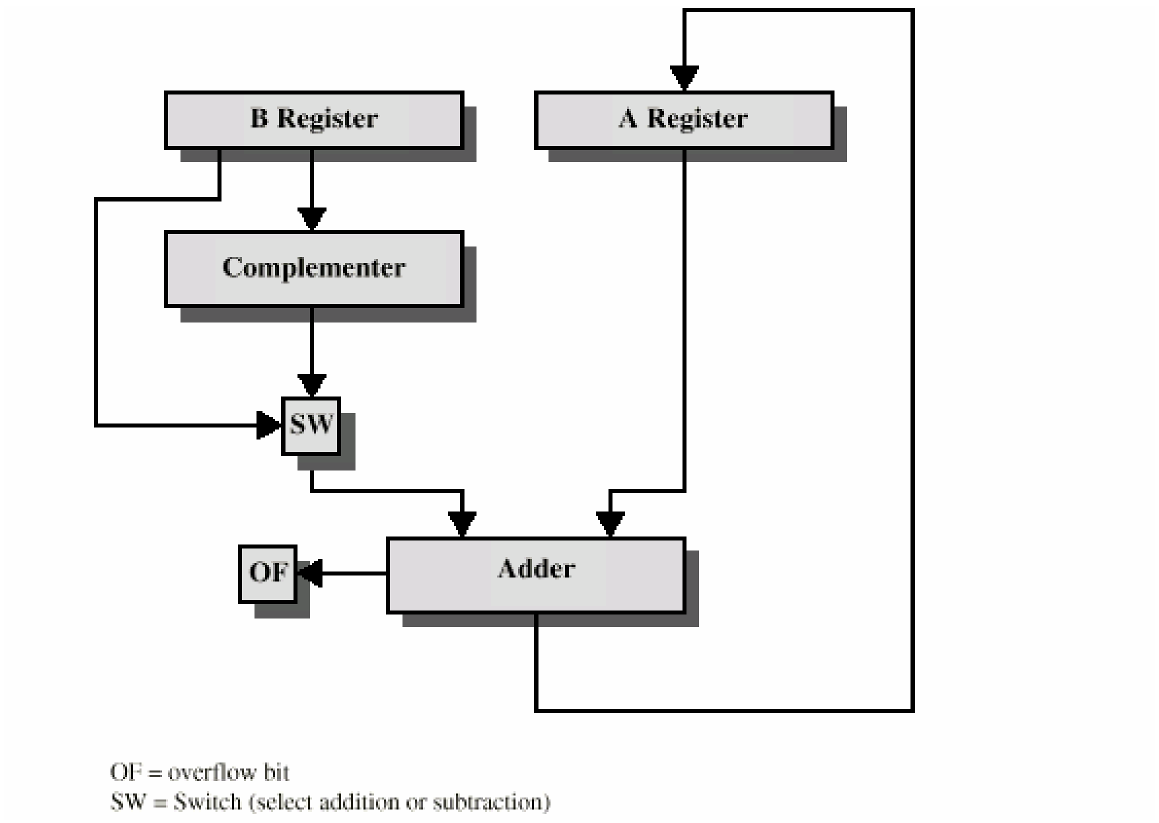
There is overflow since we added two positive numbers and got a negative result.

In the ALU, the results of the overflow (and the carry) are stored in the flags register. After performing an addition, we should check the flags register to make sure the resulting data is valid.

To perform subtraction, A-B, instead of building a separate circuit, we instead perform the operation A + -B. The value –B is computed by taking the two's complement and adding one to its value:

+3 = 0011	
flip bits:	1100
add one:	1101
1101 is –3 in two's complement	

We would then perform the addition routine as described above. One hardware design to perform these tasks are shown below:



Integer Multiplication

Multiplying two integers together is a bit more complicated than addition. Note that we could perform multiplication through repeated addition – the downside is that this will be very slow.

First, multiplying (or dividing) by two on binary numbers is easy. Multiplying or dividing a decimal number by ten is just a matter of moving the decimal point around. The same happens with binary numbers when multiplying or dividing by two:

```
e.g.    (7.5) 111.1  *   2  = 1111.0  (15)
        (7.5) 111.1  /   2  = 11.11   (3.25)
```

With integers, we have a fixed decimal point we can’t move around. So instead of moving the decimal point, we can get the same result by shifting the bits left or right, putting a 0 in at the end, where each bit shift results in a power of 2:

```
e.g.    00111 * 2       = 01110
        00111 * 4       = 11100
        00111 / 2       = 00011    (truncation of result)
        01000 / 8       = 00001
```

We’ll use the technique of shifting a little later.

How about multiplying by arbitrary values? We can use the same paper-and-pencil technique you probably learned in elementary school to multiple numbers, if they are both positive. Multiplication on unsigned binary numbers is shown below:

```
⌘          1011    Multiplicand (11 dec)
⌘      x   1101    Multiplier     (13 dec)
       ------------
⌘          1011    Partial products
⌘         0000     Note: if multiplier bit is 1 copy
⌘        1011      multiplicand (place value)
⌘       1011       otherwise zero
    ----------------
⌘     10001111     Product (143 dec)
    ----------------
⌘   Note: need double length result
```

We could build circuitry to perform these tasks. The problem arises when multiplying two’s complement values. In the example below, if we consider values as regular unsigned binary then everything works ok. However, to make it work for negative values, we must make the partial products negative:

```
Unsigned Integers                Two's Complement

        1001    (9)                      1001    (-7)
X       0011    (3)              X       0011    (3)
    --------                         --------
    00001001                         11111001
    00010010                         11110010
    --------                         --------
    00011011   (27)                  11101011    (-21)
```

The routine does not work at all if the multiplier (the 3 in the example above) is negative, because then the amount of shift doesn't make sense.

To resolve the dilemma we could either:

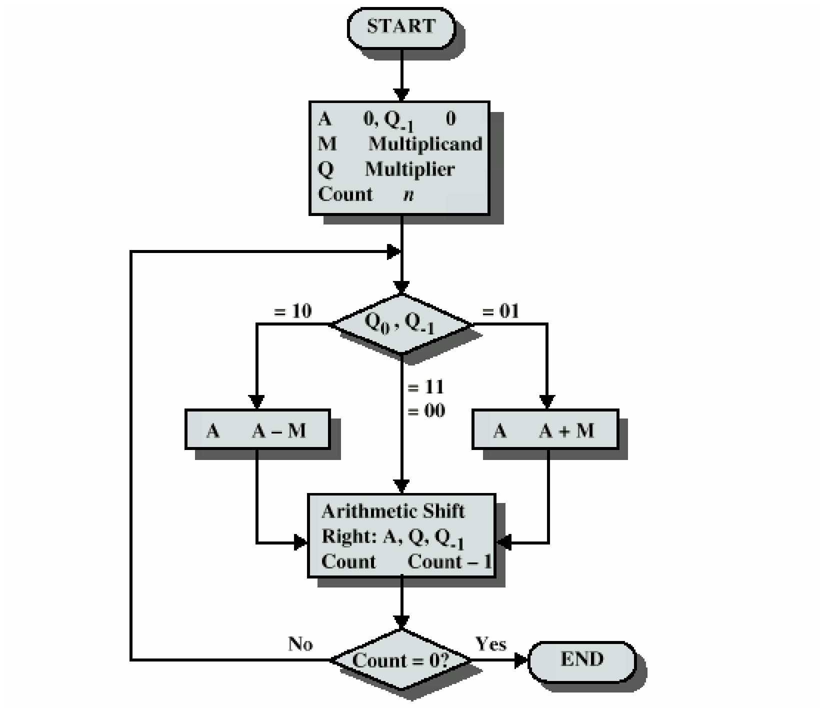
1. Convert both multiplier and multiplicand to positive, perform the multiplication, take the two's complement of the result iff the sign of the numbers differed.

2. Use a different method

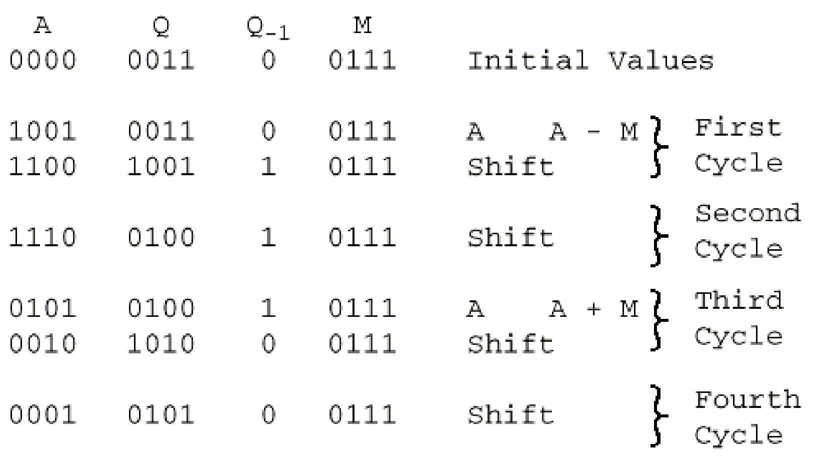
It turns out there is a different method that is faster and doesn't need this final transformation step. It is called Booth's algorithm and is widely implemented.

Booth's Algorithm

Booth's algorithm is shown in the flowchart below. The multiplier and multiplicand are placed in the Q and M registers, respectively. There is also a 1 bit register placed logically to the right of the least significant bit (Q0) of the Q register and designated Q-1. the results of the multiplication will appear in the A and Q registers.



An example is shown below:



A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial Values	
1001	0011	0	0111	A A - M	First Cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second Cycle
0101	0100	1	0111	A A + M	Third Cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth Cycle

In the example above, we are multiplying 7 * 3. An arithmetic shift is when we shift the bits, but leave the leftmost or rightmost bit as-is. For multiplication, this will preserve the sign bit. For example, an arithmetic shift right on 10010 results in 11001.   We have shifted all of the bits to the right, but kept the leftmost bit as a 1 instead of putting a zero in it. If the leftmost bit was a 0, it would remain 0 after the shift.

In the example, initially Q contains 3 and M contains 7. A and Q_{-1} are set to 0. Since the values are 4 bits, we will have 4 cycles. In the first cycle, Q_0Q_{-1} are equal to 10. Using the flowchart, this results in setting A = A-M , or 0-7 = -7. Next we arithmetically shift A,Q,and Q_{-1} as if they were all connected.   In the second cycle, Q_0Q_{-1} are equal to 11. Using the flowchart, we simply perform an arithmetic shift and continue to the third cycle. In the third cycle, Q_0Q_{-1} are equal to 01. Using the flowchart, we set A=A+M and arithmetic shift again. Finally, in the fourth cycle, we simply perform a shift.

The end result is in AQ which equals 00010101. This is 21 in decimal. You can verify that this algorithm also works for negative multipliers.

Why does this work? Here is a brief rationale. Consider a positive multiplier with one block of one's surrounded by zero's. Multiplication can be achieved by adding appropriately shifted copies of the multiplicand, as in the example of multiplying unsigned binary numbers:

$$\begin{aligned} M * (00011110) &= M * (2^4 + 2^3 + 2^2 + 2^1) \\ &= M * (16 + 8 + 4 + 2) \\ &= M * 30 \end{aligned}$$

Now observe that the block of ones, if the ones range from i to j, can be more succinctly described via:

$$2^j + 2^{j-1} + \ldots + 2^i = 2^{j+1} - 2^i$$

For example:

$$01111 = 2^3 + 2^2 + 2^1 + 2^0 = 15$$

This is the same as:

$$10000 - 00001 = 16 - 1 = 15$$

This means for our previous example we can compute:

$$\begin{aligned} M * (00011110) &= M * (2^5 - 2^1) \\ &= M * (32 - 2) \\ &= M * 30 \end{aligned}$$

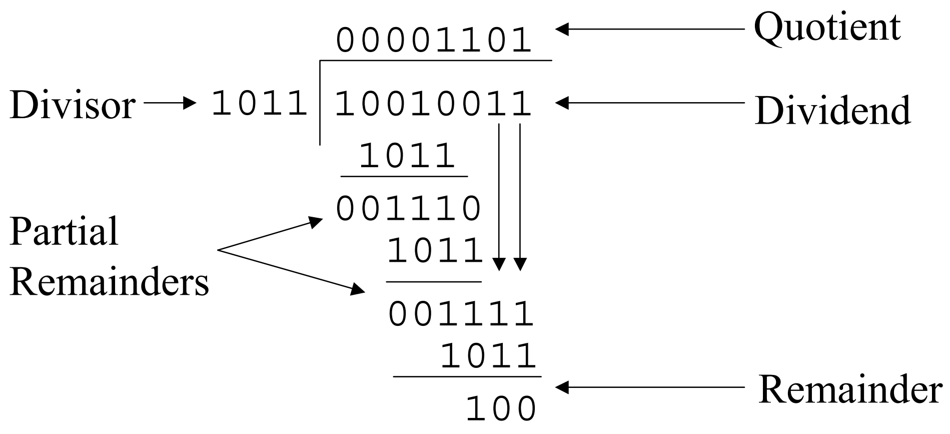
We can generalize this approach for cases with more than one block of ones:

$$\begin{aligned} M * (01110110) &= M * (2^6 + 2^5 + 2^4 + 2^2 + 2^1) \\ &= M * (2^7 - 2^4 + 2^3 - 2^1) \end{aligned}$$

Booth's algorithm conforms to this scheme by performing a subtraction when the first 1 of the block is encountered (10) and an addition when the end of the block is encountered (01). A similar argument applies for a negative multiplier.

Integer Division

Division is a bit more complex than multiplication but is based on the same principles as the paper-and-pencil division technique:



The book contains more information describing the steps that must be taken to perform integer division (and to perform it more efficiently, too).