Lecture 5:
Twos Complement Integer Arithmetic

Negation

The *negation* of a twos complement integer can be formed by following the rules:

1. Take the Boolean complement of each bit of the integer (including the sign bit). That is, invert each bit.
2. Treating the result as an unsigned binary integer, add 1.

For instance, negating 18 and -18 for an 8 bit word are as follows

$$\begin{array}{rcll} +18 & = & 00010010 & \text{(twos complement)} \\ \text{bitwise complement} & = & 11101101 & \\ & & \underline{+\qquad\quad 1} & \\ & & 11101110 = -18 & \end{array}$$

$$\begin{array}{rcll} -18 & = & 11101110 & \text{(twos complement)} \\ \text{bitwise complement} & = & 00010001 & \\ & & \underline{+\qquad\quad 1} & \\ & & 00010010 = +18 & \end{array}$$

There are two special cases to consider

Zero Whenever you negate zero, you will receive a carry that is outside the range of the word size. It is ignored.

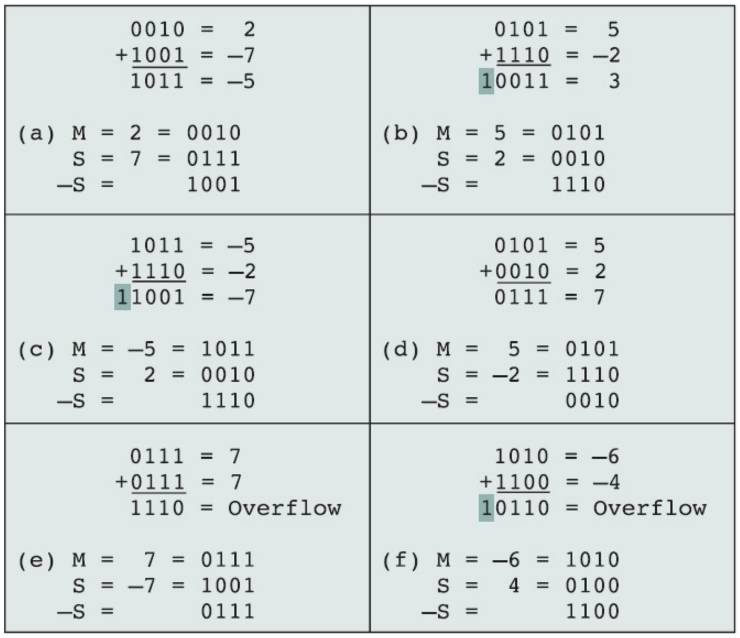
$$\begin{array}{rcll} 0 & = & 00000000 & \text{(twos complement)} \\ \text{bitwise complement} & = & 11111111 & \\ & & \underline{+\qquad\quad 1} & \\ & & 100000000 = 0 & \end{array}$$

Maximum Negative Value The range of an n bit word is -2^{n-1} to $2^{n-1}-1$. Hence, -2^{n-1} cannot be negated; its negation is itself. So its positive representation is not possible.

$$\begin{array}{rcll} +128 & = & 10000000 & \text{(twos complement)} \\ \text{bitwise complement} & = & 01111111 & \\ & & \underline{+\qquad\quad 1} & \\ & & 10000000 = -128 & \end{array}$$

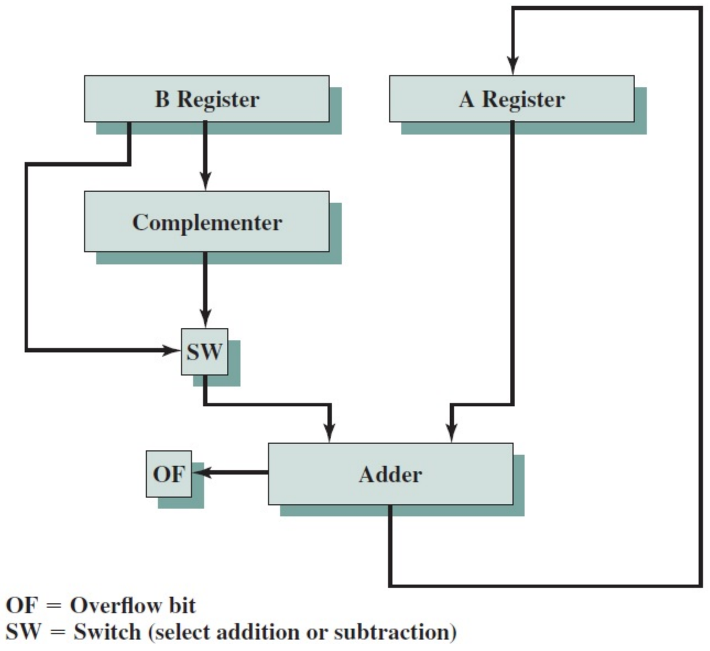
Addition and Subtraction

Addition and subtraction of two complement integers can be implemented with a simple algorithm. To perform addition, you just need to do standard addition, but it ignores the carry bit that exceeds the length of the word. To perform subtraction, you need to find the twos complement negation of the subtrahend, and then, add it to the minuend. Furthermore, you need to look not for a possible *overflow* error. An overflow error can only occur when both operands have the same sign. If the result of the operation has a different sign than the operands, there is an overflow error.



```
  0010 =  2                    0101 =  5
 +1001 = -7                   +1110 = -2
  1011 = -5                   10011 =  3

(a) M = 2 = 0010              (b) M = 5 = 0101
    S = 7 = 0111                  S = 2 = 0010
   -S =     1001                 -S =     1110

  1011 = -5                    0101 = 5
 +1110 = -2                   +0010 = 2
 11001 = -7                    0111 = 7

(c) M = -5 = 1011             (d) M =  5 = 0101
    S =  2 = 0010                 S = -2 = 1110
   -S =      1110                -S =      0010

  0111 = 7                     1010 = -6
 +0111 = 7                    +1100 = -4
  1110 = Overflow             10110 = Overflow

(e) M =  7 = 0111             (f) M = -6 = 1010
    S = -7 = 1001                 S =  4 = 0100
   -S =      0111                -S =      1100
```

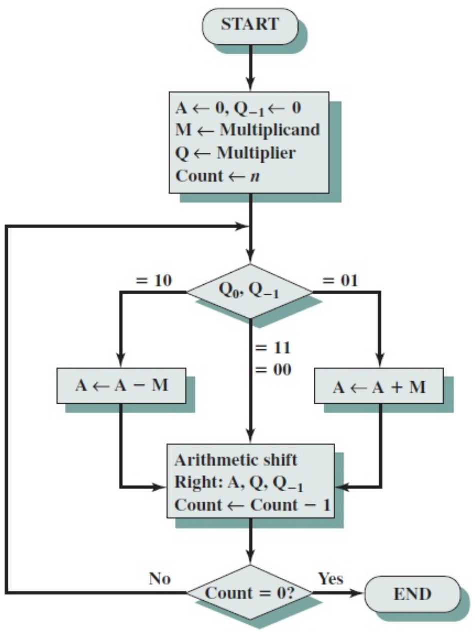
A diagram of the algorithm is as follows



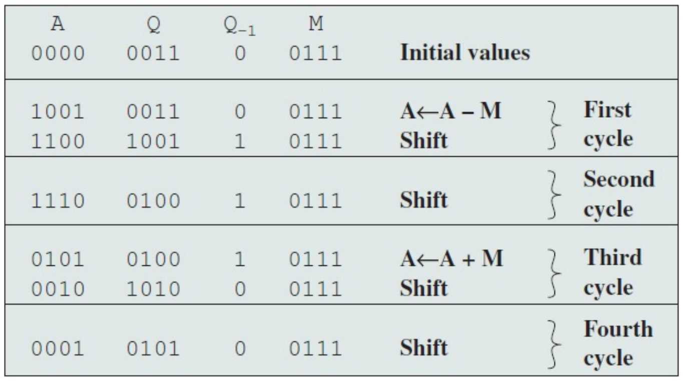
The diagram illustrates that the complement of the second register, B, (which would be the subtrahend if subtraction is the operation) is found, then both are feed to a switch which determines which one to use based on the operation being performed. Afterwards, the selected form of the second operand and the first operand are feed into a adder algorithm.

Multiplication

Multiplication is a more complicated operation to perform. There are several multipliation algorithms out there, but there are a few key factors to know such as the product of two n bit words is at most $2n$ bits and multiplication is a sum of partial products. For unsigned binary integers, multiplication is achieved by using an adder and a shift operation. However, for twos complement (or, in general, when dealing with negative values) the algorithm becomes more difficult. One of the common common algorithms used to perform twos complement multiplication is the Booth's algorithm.



The algorithm starts by multiplier and multiplicand are placed in the Q and M regsters respectively. There is also a 1 bit regster placed logically to the right of the least significant bit (Q_0) of the Q register and designated Q_{-1}; its use is explained shortly. Now, as each bit is examined, the bit to its right is also examined. If the two bits are the same (1-1 or 0-0), then all of the bits of the A, Q, and Q_{-1} registers are shifted to the right 1 bit. If the two bits differ, then the multiplicand is added to or subtracted from the A register, depending on whether the two bits are $0-1$ or $1-0$. Following the addition or subtraction, the right shift occurs. In either case, the right shift must maintain the sign bit of A and Q; hence, an arithmetic shift. For instance,



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 following example, shows that the Booth's algorithm works for all sign combinations of operands

```
    0111
  × 0011      (0)
11111001     1-0
0000000      1-1
000111       0-1
00010101      (21)
```

(a) $(7) \times (3) = (21)$

```
    0111
  × 1101      (0)
11111001     1-0
0000111      0-1
111001       1-0
11101011      (-21)
```

(b) $(7) \times (-3) = (-21)$

```
    1001
  × 0011      (0)
00000111     1-0
0000000      1-1
111001       0-1
11101011      (-21)
```

(c) $(-7) \times (3) = (-21)$

```
    1001
  × 1101      (0)
00000111     1-0
1111001      0-1
000111       1-0
00010101      (21)
```

(d) $(-7) \times (-3) = (21)$