CS 320
Ch 10 – Computer Arithmetic

The ALU consists of c**ombinational logic. Processes all data in the CPU. ALL von Neuman machines have an ALU loop.**

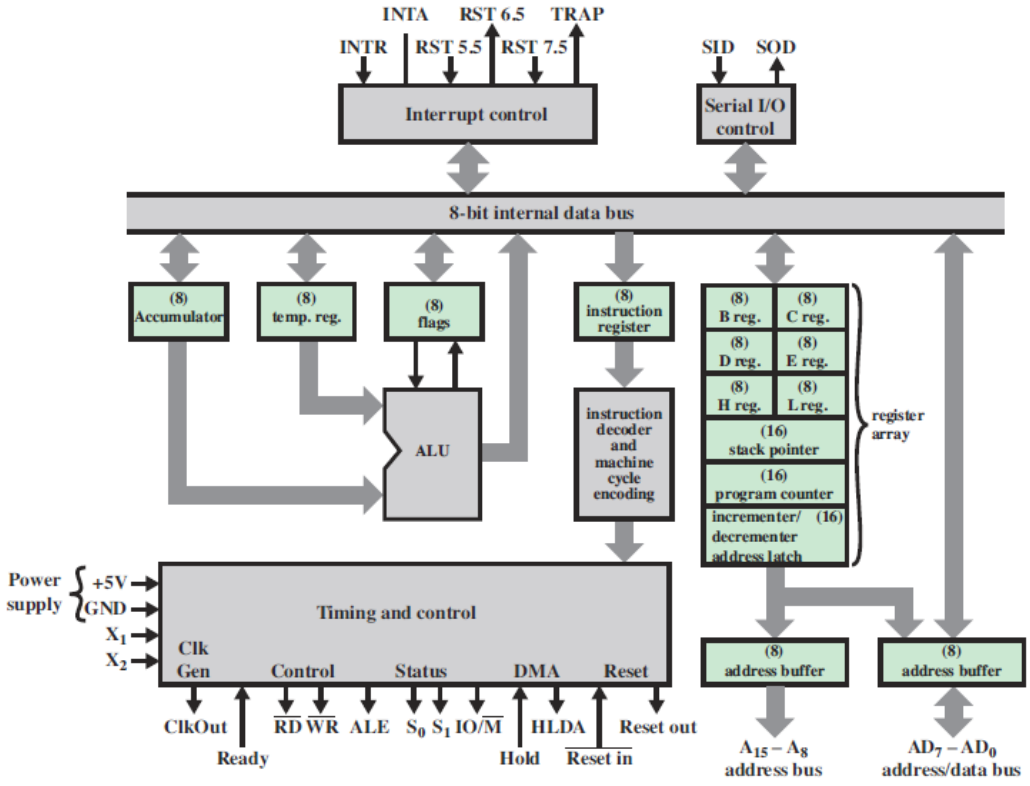


Figure 19.7 Intel 8085 CPU Block Diagram

Signed integers are typically represented in sign-magnitude or twos complement format. In sign-magnitude number representation. **LSB is sign bit 0 = +. All other bits represent positive magnitude.**
The disadvantages of the sign-magnitude representation are A) **Two values for zero. B) Does not correctly add a + and – number without changing sign and doing subtraction.**

The 2's complement number system has only o**ne representation for zero. Subtraction is equivalent to adding a + to a -.** $A_n = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} a_i 2^i$ **. a_{n-1} = 0 for positive number and 1 for negative numbers.**

For an 8-bit number, the range of numbers in sign-magnitude is **±127. In 2's complement -128 to +127.**

To go from an 8-bit number to a 16-bit number in 2's complement simply e**xtend zeros for +, extend 1's for -.**

To convert a number to its 2's complement representation: **If positive convert to binary; If negative, convert to binary, invert, and add 1.**
OR
For a negative number of n-bits subtract the positive number from 2^n.

Example, for n = 4 the number + 5 is 0101. To get -5 subtract +5 from 2^4
10000
<u>00101</u>
01011 or 1011 = -5 in four digits. You can do the same thing with base 10.
OR
To get the negative number begin with the LSB and copy down all zeros plus the first one. Thereafter invert all of the bits.
0101 = +5. The first bit is 1 so write that down and invert the remainder to get 1011 = -5

Note that fixed point arithmetic is not the same as integer arithmetic. **Fixed point has a fixed decimal anywhere. Integer has a fixed decimal to the left of the units digit.**

In 2's complement is it possible to generate a carry when adding a + and – number and if so, but this does this is not overflow?

```
  1100  = -4
  0100  = +4
 10000  = 0 with a carry but no overflow
```

Cannot have overflow if adding a positive and a negative.

Two number overflow only if adding two numbers of the same sign. Overflow occurs if the sign of the result and the carry are different.

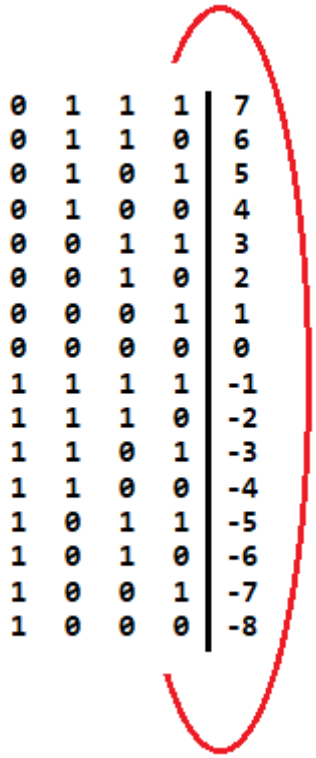
```
  0101  = +5
  0100  = +4
 01001  = overflow since cy = 0 and sign = 1

  1100  = -4
  1111  = -1
 11011  = -5 no overflow since sign and cy are same
```

Note that intermediate overflow in a series of operations where the final result comes out in the range of representable numbers is OK. Say +7 + 4 – 6 = +5. **Intermediate overflow is of no consequence.**

```
  0111  = +7
  0100  = +4
 01011  = intermediate overflow
  1010  = -6
 10101  = +5 overflow indicated but result OK
```

Twos complement arithmetic on a great circle

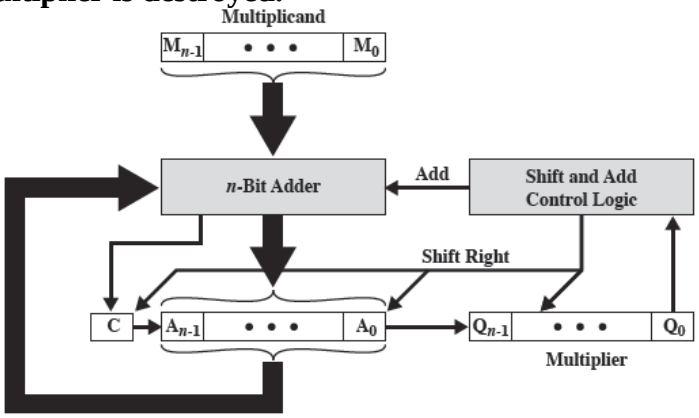


0	1	1	1	7
0	1	1	0	6
0	1	0	1	5
0	1	0	0	4
0	0	1	1	3
0	0	1	0	2
0	0	0	1	1
0	0	0	0	0
1	1	1	1	-1
1	1	1	0	-2
1	1	0	1	-3
1	1	0	0	-4
1	0	1	1	-5
1	0	1	0	-6
1	0	0	1	-7
1	0	0	0	-8

Equation for Overflow = $x_{n-1} \cdot y_{n-1} \cdot \overline{s_{n-1}} + \overline{x_{n-1}} \cdot \overline{y_{n-1}} \cdot s_{n-1}$

Overflow occurs only if adding two numbers of the same sign. Overflow occurs if the sign of the result and the carry are different.

Multiplication by shifting and adding:
Multiplier in Q, Multiplicand in M, produce in AQ. LSB of multiplier determines whether a shifted multiplicand is added to the product or not. In the end the multiplier is destroyed.



(a) Block Diagram

C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

(b) Example from Figure 9.7 (product in A, Q)

The unsigned multiplier does not work properly for 2's complement numbers. **It does if they are positive. If they are negative then the intermediate sums lose their sign if the multiplicand is negative and if the multiplier is negative the shifts no longer correspond to 1's and 0's as they do for positive numbers.**

Booth's algorithm presents a **way to do multiplication of 2's complement numbers and get correct results.**

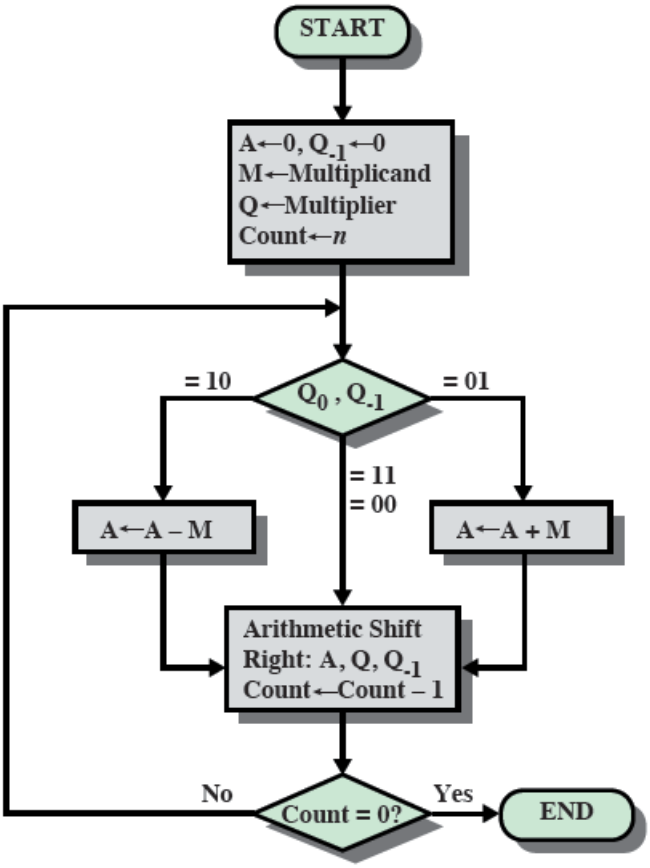
Booth's algorithm works by representing **negative 2's complement numbers as a positive number subtracted from a negative number.**

For a number such as `00011110` = 30_{10} can be written as $2^5 - 2^1 = 32 - 2 = 30$.

This result can be extended to any sequence of 1's in a number including a sequence of only a single 1.

For example 0010 = 2^2-2^1 = 4 – 2 = 2. Thus when the algorithm runs into a sequence 10 it subtracts the appropriate power of 2 and when it runs into a sequence 01 it adds the appropriate power of 2 to the product.

This flow chart shows how to multiply two 4-bit 2's complement numbers together. 0111 x 1101 (7 x -3 = -21).



An arithmetic shift is one that extends the sign bit.

An arithmetic left shift is not useful from a numeric point of view but possibly for bit manipulation problems.

```
  A    Q  Q-1 M
0000 1101 0 0111  Initialize
1001 1101 0 0111  A ← A – M
1100 1110 1 0111  Shift right arithmetic

0011 1110 1 0111  A ← A + M
0001 1111 0 0111  Shift right arithmetic

1010 1111 1 0111  A ← A – M
1101 0111 1 0111  Shift right arithmetic

1101 0111 1 0111  A unchanged
1110 1011 1 0111  Shift right arithmetic
AQ = 11101011 = -21
```

CS 320 **Name__ANSWERS_**
Arithmetic

1. Use Booth's algorithm to show how to multiply 5 x 6 = 30 using 8 bit numbers. Fill in the sequential tables below to show each step of the algorithm. The first thee table sequences have been filled in for you as an example.

Product	0	0	0	0	0	0	0	0	Multiplier 0110_
Multiplicand	0	0	0	0	0	1	0	1	0 → 0 so nothing
Product	0	0	0	0	0	0	0	0	

Product	0	0	0	0	0	0	0	0	Shift
Multiplicand	0	0	0	0	1	0	1	0	
Product	0	0	0	0	0	0	0	0	

Product	0	0	0	0	0	0	0	0	Multiplier 0110
Multiplicand	0	0	0	0	1	0	1	0	0 → 1 so subtract
Product	1	1	1	1	0	1	1	0	

Product	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**	**Shift**
Multiplicand	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	
Product	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**	

Product	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**	**Multiplier 0110**
Multiplicand	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**1 → 1 so nothing**
Product	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**	

Product	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**	**Shift**
Multiplicand	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	
Product	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**	

Product	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**	**Multiplier 0110**
Multiplicand	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**1 → 0 so add**
Product	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	

Product	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**	**Shift**
Multiplicand	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	
Product	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	

Product	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**	**Multiplier 0 0110**
Multiplicand	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0 → 0 so nothing**
Product	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	

Stallings 2012 edition

Go over division of binary numbers for yourself.

FLOATING POINT

There are three fields used to store floating point numbers: **Sign, Significand (S), and exponent (E). The significand is sometimes called the mantissa.**

Floating point numbers are stored in the form of $\pm S \times B^{\pm E}$. B is not stored. **The base B is implicit.**

Exponents typically get stored in a biased format. **A number is added to the actual exponent before it is stored. The number is typically $2^{k-1}-1$ where k is the number of bits in the exponent.**

In a biased representation all numbers can be treated as unsigned integers and their relative magnitudes will not change. This allows efficient comparison of exponents.

The significand, also called the mantissa, is stored as a normalized binary number. This means that the b**inary point and exponent are adjusted such that the first digit of the number is always a 1 and the binary point follows. Thus, all numbers are represented as $\pm 1.???? \times B^{\pm E}$.**

Since mantissa is normalized its first bit is always 1 and need not be stored. The bit is omitted and is said to be elided.

Smallest (in magnitude) positive and negative numbers:

1	8	23
sign	exponent	significand

The sign bit is the sign of the significand. The exponent is in twos complement form (biased) and can be in the range of -127 to +128. (Note that while an 8-bit twos complement number ranges from -128 to +127, in biased format the range changes to -127 to +128.) Since the significand always has at least a 1 bit (elided) in it, the positive and negative numbers with the smallest magnitude are -2^{-127} and $+2^{-127}$. Numbers between these two cannot be represented. This includes 0.

Largest (in magnitude) positive and negative numbers.

The significand is an unsigned (sign is separate) number that is 24 bits long but the MSB must be a 1. Thus the numbers which can be represented at 1.00...0 to 1.11....1 where each sequence is 24 bits long. The smallest number is 1 and the largest is $(2^{24}-1)/2^{23} = 2 - 2^{-23}$. The largest number is therefore $(2- 2^{-23})x2^{128}$ and the smallest number is $-(2- 2^{-23})x2^{128}$

Negative and positive underflow and overflow are illustrated below. **Note that 0 cannot be represented without special conditions.**

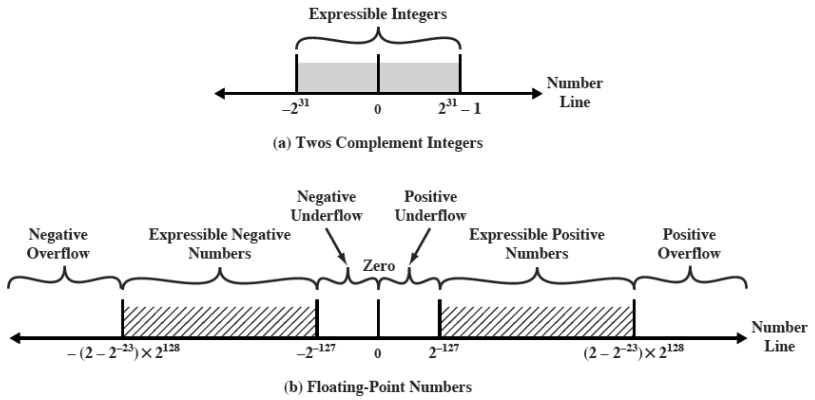


Figure 10.19 Expressible Numbers in Typical 32-Bit Formats

The floating point format described in the format above represent numbers between $\pm 3.4 \times 10^{38}$. Yet with 32 bits as integers we represent only $2^{32} = 4$ Billion numbers. Floating point format does not allow us to represent more numbers with the same number of bits. **The numbers are just spread out more and the same**

The density of integer numbers that can be represented compare to the density of the floating point numbers that can be represented. **Integers are evenly distributed. Floating point numbers are denser at the smaller end.**

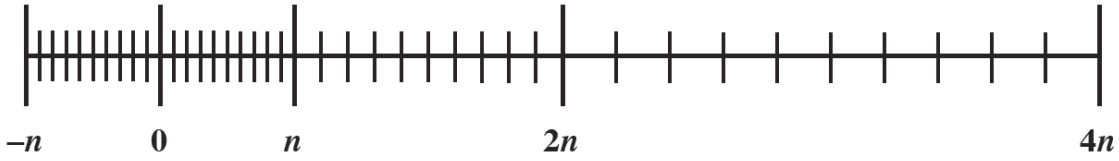


Figure 10.20 Density of Floating-Point Numbers

Range is the difference between the smallest negative number and the largest positive number.

Precision is the difference between numbers. Precision is like density.

To change the range of floating point representation m**ake the exponent larger are alter the base. This reduces the density.**

To change the precision m**ake the significand bigger. This reduces the size of the exponent. The density increases but the range is reduced.**

To increase both the precision and the range i**ncrease the number of bits for each.**

The IEEE 754 floating point format has some special representations for some numbers.
zero is represented when **Exponent and significand are both 0.**
A NaN is **Not a number. For example, the square root of a negative number.**

Stallings 2012 edition

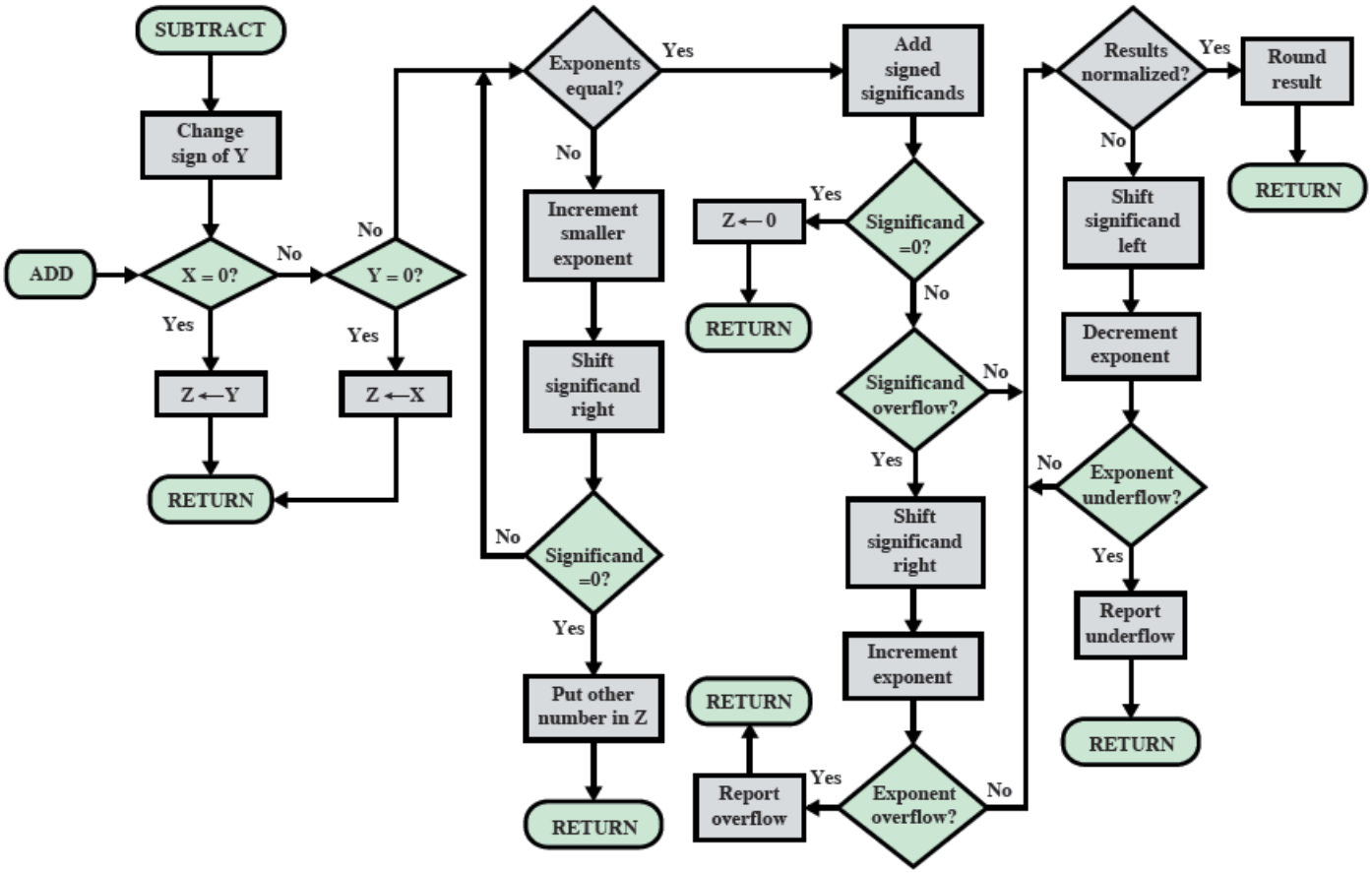


Figure 10.22 Floating-Point Addition and Subtraction (Z← X ± Y)

Floating point addition and subtraction
A) Exponents need to be the same to add or subtract the significands.
B) Subtraction is handled by changing the sign of the number to be subtracted.
C) The last column of the flow chart shows normalization.
D) Note that there checks for zeros for efficiency. Makes addition faster
.

The significant differences between addition/subtraction and multiplication/division flow charts are that you d**on't have to adjust exponent before operations.**

Guard bits **Additional bits padded to the significand prior to an arithmetic operation to increase precision.**

Example
IEEE single precision floating point representation uses 32 bits. There is a sign bit, an 8-bit exponent, and a 23 bit significand. The base is 2 and the exponent is biased by 127. Find the IEEE representation for 210_{10} = 00...011010010. **Normalize significand by moving the binary point to the left 7 places so that 11010010 = 1.1010010x2^7. Exponent is 7 + 127 = 134 = 10000110.**

1	8	23
0	11010010	101 0010 0000 0000 0000 0000

Stallings 2012 edition

CS 320 **Name_____**
In Class **February 23, 2018**

1. Find the IEEE representation for 321_{10} = 00...0101000001.
Normalize significand by moving the binary point to the left 7 places so that 101000001 = 1.01000001x2^8. Exponent is 8 + 127 = 135 = 10000111.

1	8	23
0	**10000111**	**010 0000 1000 0000 0000 0000**