Computer Arithmetic

- **Number systems**
- **Integers**
- **Floats**

Number Systems

- There are 10 types of people: those who understand binary and those who don't.
- Normally, we are using a positional number system with the base 10, but the base can be changed. (Most common are 2,8,16)
- A number $\ldots a_3 a_2 a_1 a_0 . a_{-1} a_{-2} \ldots$ with base b equals
$$\sum_i (a_i \cdot b^i)$$
- Example:
$$1001.101_2 = 2^3 + 2^0 + 2^{-1} + 2^{-3} = 9.625_{10}$$

Converting Between Number Systems

- Converting from any number system to the base 10 is simple

Converting Between Number Systems

- Converting from any number system to the base 10 is simple
- The other way around: converting N from base 10 to base b:
  - Every number N can be expressed as
$$N = b \cdot N_1 + R_0 \qquad \text{with } R_0 < b$$

Converting Between Number Systems

- Converting from any number system to the base 10 is simple
- The other way around: converting N from base 10 to base b:
  - Every number N can be expressed as
    $$N = b \cdot N_1 + R_0 \qquad \text{with } R_0 < b$$
  - Same is true for N_1:
    $$N_1 = b \cdot N_2 + R_1 \qquad \text{with } R_1 < b$$

Converting Between Number Systems

- Converting from any number system to the base 10 is simple
- The other way around: converting N from base 10 to base b:
  - Every number N can be expressed as
    $$N = b \cdot N_1 + R_0 \qquad \text{with } R_0 < b$$
  - Same is true for N_1:
    $$N_1 = b \cdot N_2 + R_1 \qquad \text{with } R_1 < b$$
  - We can write N as:
    $$N = b \cdot (b \cdot N_2 + R_1) + R_0 = N_2 \cdot b^2 + R_1 \cdot b^1 + R_0 \cdot b^0$$
  - We proceed until $N_{m-1} = b \cdot N_m + R_{m-1}$ with $\mathrm{N_m} < b$

Converting Between Number Systems

- Converting from any number system to the base 10 is simple
- The other way around: converting N from base 10 to base b:
  - Every number N can be expressed as
    $$N = b \cdot N_1 + R_0 \qquad \text{with } R_0 < b$$
  - Same is true for N_1:
    $$N_1 = b \cdot N_2 + R_1 \qquad \text{with } R_1 < b$$
  - We can write N as:
    $$N = b \cdot (b \cdot N_2 + R_1) + R_0 = N_2 \cdot b^2 + R_1 \cdot b^1 + R_0 \cdot b^0$$
  - We proceed until $N_{m-1} = b \cdot N_m + R_{m-1}$ with $\mathrm{N_m} < b$

  - Now, N can be expressed as
    $$N_m \cdot b^m + R_{m-1} \cdot b^{m-1} + \cdots + R_2 \cdot b^2 + R_1 \cdot b^1 + R_0 \cdot b^0$$
  - This corresponds to the number
    $$(N_m R_{m-1} \dots R_2 R_1 R_0)_b$$

Converting After the Radix Point

- Each number F with $0 < F < 1$ can be expressed in basis b as
$$0.b_{-1}b_{-2}\dots$$

Converting After the Radix Point

- Each number F with $0 < F < 1$ can be expressed in basis b as

$$0.b_{-1}b_{-2} \dots$$

- F has the decimal value of

$$(b_{-1} \cdot b^{-1}) + (b_{-1} \cdot b^{-2}) \dots = b^{-1}\left(b_{-1} + b^{-1}(b_{-2} + \cdots)\right)$$

Converting After the Radix Point

- Each number F with $0 < F < 1$ can be expressed in basis b as

$$0.b_{-1}b_{-2}\ldots$$

- F has the decimal value of

$$(b_{-1} \cdot b^{-1}) + (b_{-1} \cdot b^{-2}) \ldots = b^{-1}\big(b_{-1} + b^{-1}(b_{-2} + \cdots)\big)$$

- Multiplying by b gives

$$b \cdot F = b_{-1} + b^{-1}\big(b_{-2} + b^{-1}(\ldots)\big)$$

- b_{-1} is exactly the integer part of $b \cdot F$

Converting After the Radix Point

- Each number F with $0 < F < 1$ can be expressed in basis b as
$$0.b_{-1}b_{-2}\ldots$$
- F has the decimal value of
$$(b_{-1} \cdot b^{-1}) + (b_{-1} \cdot b^{-2}) \ldots = b^{-1}\left(b_{-1} + b^{-1}(b_{-2} + \cdots)\right)$$

- Multiplying by b gives
$$b \cdot F = b_{-1} + b^{-1}\left(b_{-2} + b^{-1}(\ldots)\right)$$
- b_{-1} is exactly the integer part of $b \cdot F$

- Analogous to before, we can say that
$$b \cdot F = b_{-1} + F_1$$
- Calculating $b \cdot F_1$ gives us b_{-2} and F_2 and so on …

Examples

	Quotient	Remainder
$\frac{21}{2} =$	10	1
$\frac{10}{2} =$	5	0
$\frac{5}{2} =$	2	1
$\frac{2}{2} =$	1	0
$\frac{1}{2} =$	0	1

$$10101_2 = 21_{10}$$

	Prod.	Int.
$0.81 \cdot 2 =$	1.62	1
$0.62 \cdot 2 =$	1.24	1
$0.24 \cdot 2 =$	0.48	0
$0.48 \cdot 2 =$	0.96	0
$0.96 \cdot 2 =$	1.92	1
$0.92 \cdot 2 =$	1.84	1
…		

$$0.110011_2 \approx 0.81$$

Hexadecimal

- Base 16
- Often used to look at binary code as a byte can be displayed as a two digit hexadecimal number
- Extremely easy to convert between binary and hex

0000 = 0	0100 = 4	1000 = 8	1100 = C
0001 = 1	0101 = 5	1001 = 9	1101 = D
0010 = 2	0110 = 6	1010 = A	1110 = E
0011 = 3	0111 = 7	1011 = B	1111 = F

Computer Arithmetic

- Number systems
- Integers
- Floats

Representing Numbers

- With binary numbers, we can represent arbitrary numbers:

$$-1101.0101_2 = -13.3125_{10}$$

- But: On a computer, we are limited to only 0 and 1:
  - No sign-symbol
  - No radix point
- Limited space
- In the remainder:
  - LSB = Least Significant Bit
  - MSB = Most Significant Bit

Representing Negative Numbers
Sign-Magnitude Method

- Sign-Magnitude Method
  - Use MSB as sign

```
+18 = 0001 0010
-18 = 1001 0010
```

  - Drawback: Two representations of zero

```
+0 = 0000 0000
-0 = 1000 0000
```

  - Thus, it is more complicated to check for zero (very often used)

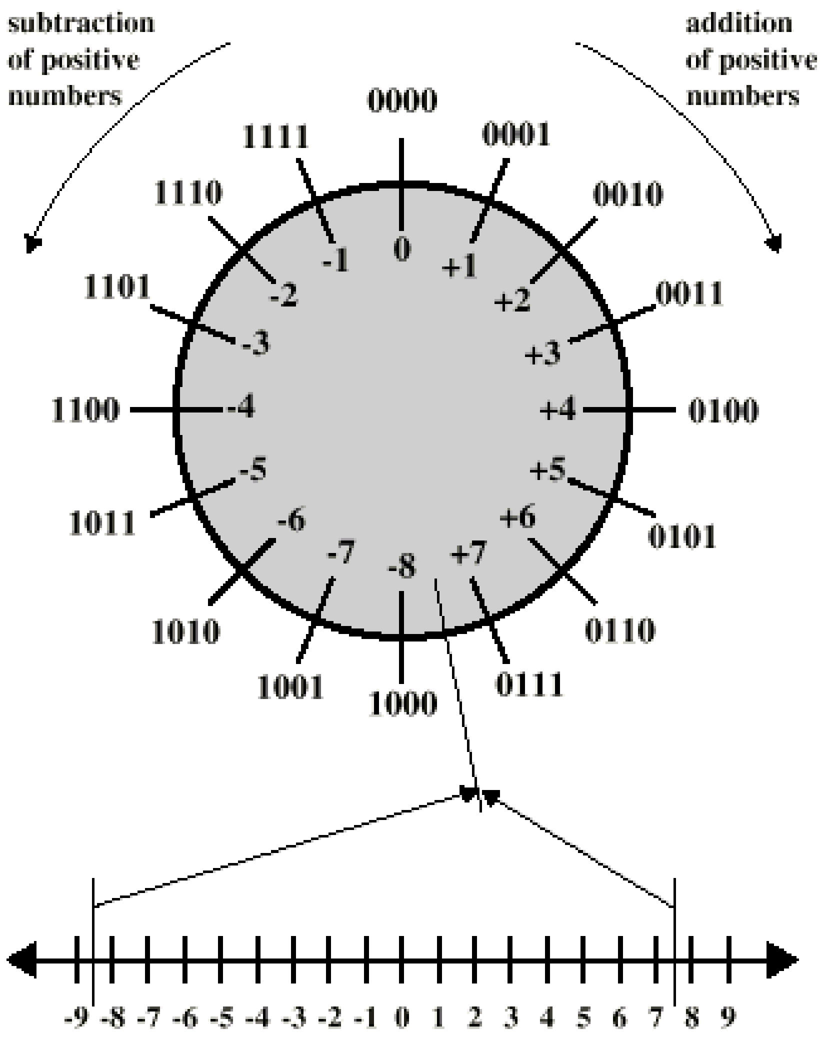
Representing Negative Numbers
Twos Complement

- Only one zero
- Arithmetic works pretty easy (We see that later)
- Negating is fairly simple
  - Compliment number
  - Add 1

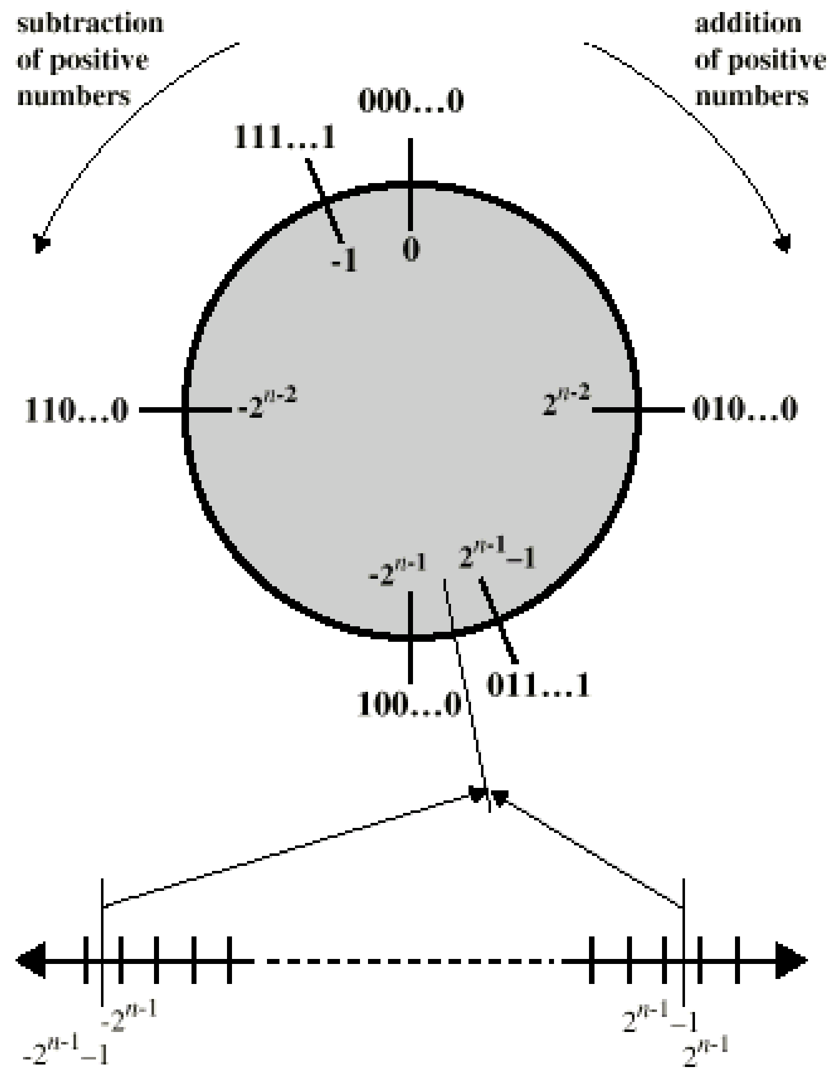
```
       +18 = 0001 0010

Compliment   1110 1101
Add 1                1
             ---------
       -18 = 1110 1110
```

- Range $2^{n-1} - 1$ through -2^{n-1}

Geometric Depiction of Twos Complement Numbers



(a) 4-bit numbers



(b) n-bit numbers

Why Does It Work Now?

- We have two operations to built a negative number:
- $y = 0 – x$
- Or: Select y in such a way, that $x + y = 0$

Why Does It Work Now?

- We have two operations to built a negative number:
- $y = 0 - x$
- Or: Select y in such a way, that $x + y = 0$
- Now, the twos compliment works, as we have a limited number of bits to represent an integer

```
   18 =  0001 0010
+ -18 =  1110 1110
-----    ---------
    0 = 10000 0000
          0000 0000
```

- More formally: The twos representation of a number equals

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

Addition and Subtraction

- Subtraction is done by adding the twos complement of the subtrahend to the minuend
- 5 - 7:

```
 5    0101
-7   +1001
      ----
      1110
```

-4 - 1:

```
-4    1100
-1   +1111
      ----
     11011
```

- Overflow Rule

An overflow only occurs iff both numbers have the same sign AND the sign changes!

Addition and Subtraction

- 5 - 7:

```
 5    0101
-7   +1001
      ----
      1110
```

-4 - 1:

```
-4    1100
-1   +1111
      ----
     11011
```

- 5 + 4:

```
 5    0101
 4   +0100
      ----
      1110
```

-7-6:

```
-7    1001
-6   +1010
      ----
     10011
```

An overflow only occurs iff both number have the same sign AND the sign changes!

Addition and Subtraction

- 5 - 7:

```
 5    0101
-7   +1001
      ----
      1110
```

No Overflow possible

-4 - 1:

```
-4    1100
-1   +1111
      ----
     11011
```

Sign doesn't change

- 5 + 4:

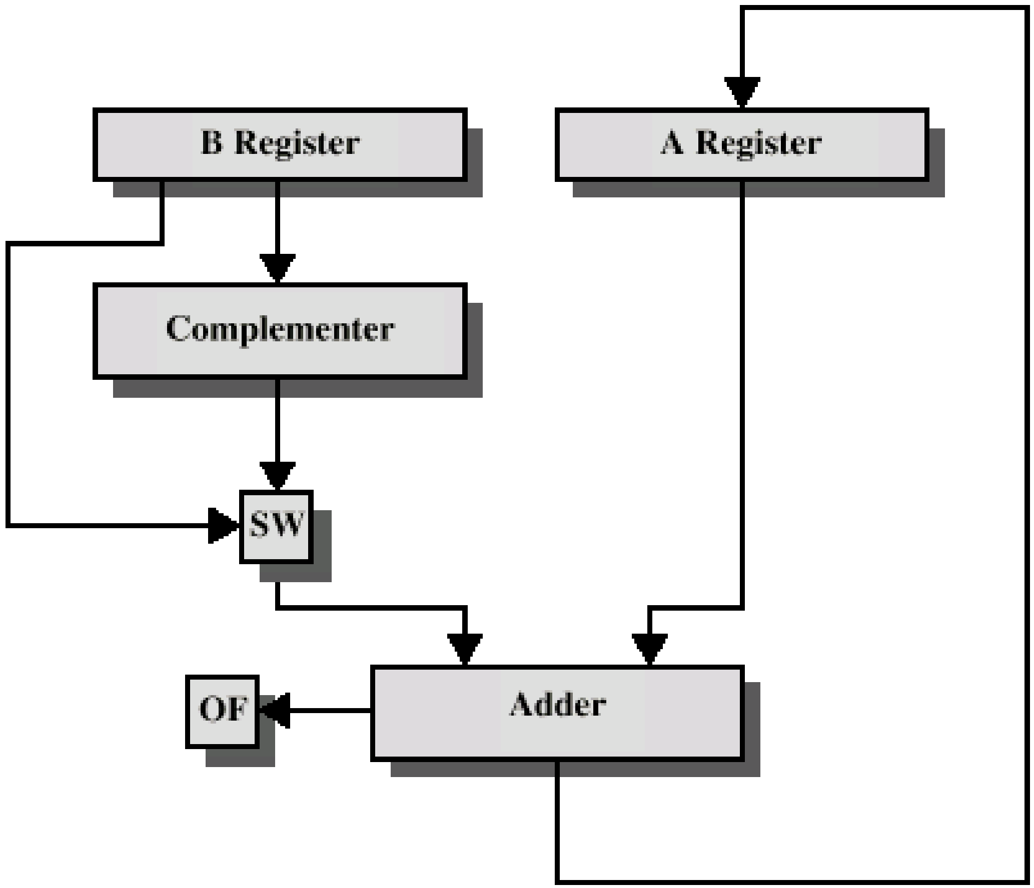
```
 5    0101
 4   +0100
      ----
      1110
```

-7-6:

```
-7    1001
-6   +1010
      ----
     10011
```

An overflow only occurs iff both number have the same sign AND the sign changes!

Block Diagram for a Hardware Adder



OF = overflow bit
SW = Switch (select addition or subtraction)

Multiplication

- More complicated than adding
- Several methods exist. We first concentrate on `uints`

- We look at a small example

Multiplication

```
1011 x 1101
```

Multiplicand (11) x Multiplier (13)

Multiplication

```
1011 x 1101
```

Multiplicand (11) x Multiplier (13)

Multiplication

```
1011 x 1101
-----------
      1011
```

Multiplicand (11) x Multiplier (13)

Multiplication

```
1011 x 1101
       1011
```

Multiplicand (11) x Multiplier (13)

Multiplication

```
1011 x 1101
       1011
      0000
```

Multiplicand (11) x Multiplier (13)

Multiplication

```
1011 x 1101
      1011
     0000
```

Multiplicand (11) x Multiplier (13)

Multiplication

```
1011 x 1101
      1011
     0000
    1011
```

Multiplicand (11) x Multiplier (13)

Multiplication

```
1011 x 1101
      1011
     0000
    1011
```

Multiplicand (11) x Multiplier (13)

Multiplication

```
1011 x 1101
-----------
       1011
      0000
     1011
    1011
```

Multiplicand (11) x Multiplier (13)

Multiplication

```
1011 x 1101
       1011
      0000
     1011
    1011
```

Multiplicand (11) x Multiplier (13)

Partial Products

Multiplication

```
1011 x 1101
-----------
       1011
      0000
     1011
    1011
-----------
   10001111
```

Multiplicand (11) x Multiplier (13)

Partial Products

Final Result (143)

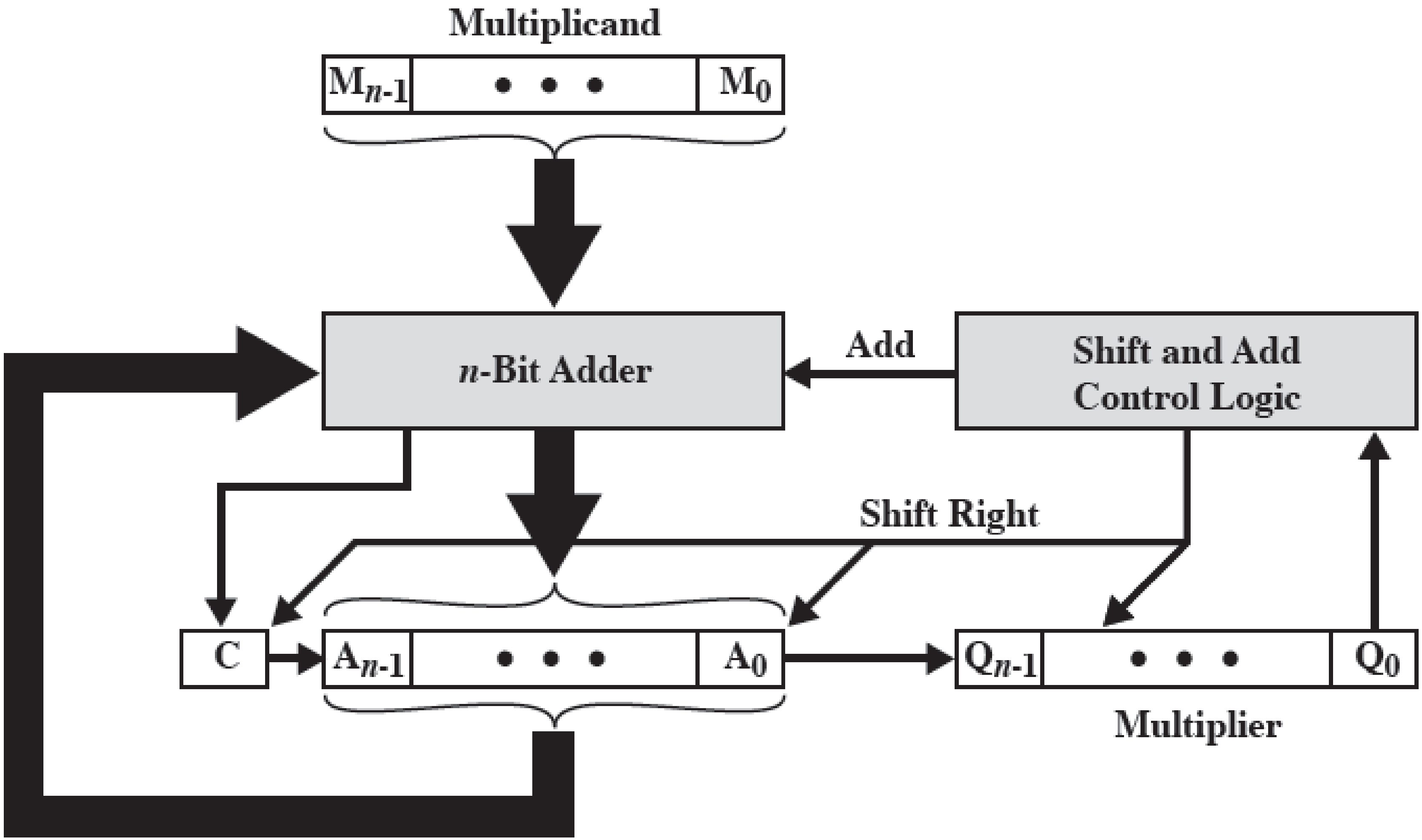
Observations

- The partial products are either 0 or the multiplicand
- Basically only two operations needed: Add and Shift

- The multiplication of two n-bit binary integers results in a product of up to $2n$ bits in length
- We don’t need to store the each partial sum

- Possible Implementation:
  - Multiplier and multiplicand are loaded into two registers (Q and M)
  - A, a third register, is initially set to 0.
  - The 1-bit C register, stores a carry bit of the multiplication
  - In order to save registers, Q will hold the results in the end, thus the multiplier gets destroyed

Hardware Implementation



Add and Shift

C	A	Q	M	
0	0000	1101	1011	Initial Values

Add and Shift

C	A	Q	M	
0	0000	1101	1011	Initial Values

Add and Shift

```
C    A        Q           M
0   0000    1101        1011    Initial Values
  + 1011                        Add
```

Add and Shift

```
C    A        Q          M
0   0000   1101         1011   Initial Values
  + 1011                       Add
0   1011
```

Add and Shift

C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add

Add and Shift

C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift

Add and Shift

C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift

Add and Shift

C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift

No Add, shift only

Add and Shift

C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift
0	0010	1111	1011	Shift

Add and Shift

C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift
0	**0010**	**1111**	**1011**	**Shift**

Add and Shift

C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift
0	0010	1111	1011	Shift
+	1011			Add

Add and Shift

```
C    A       Q           M
0    0000    1101        1011    Initial Values
0    1011    1101        1011    Add
0    0101    1110        1011    Shift
0    0010    1111        1011    Shift
   + 1011                        Add
0    1101
```

Add and Shift

C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift
0	0010	1111	1011	Shift
0	**1101**	**1111**	**1011**	**Add**

Add and Shift

C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift
0	0010	1111	1011	Shift
0	1101	1111	1011	Add
0	0110	1111	1011	Shift

Add and Shift

C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift
0	0010	1111	1011	Shift
0	1101	1111	1011	Add
0	**0110**	**1111**	**1011**	**Shift**

Add and Shift

C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift
0	0010	1111	1011	Shift
0	1101	1111	1011	Add
0	0110	1111	1011	Shift
+	1011			Add

Add and Shift

C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift
0	0010	1111	1011	Shift
0	1101	1111	1011	Add
0	**0110**	**1111**	**1011**	**Shift**
+	**1011**			**Add**
1	0001			

Add and Shift

C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift
0	0010	1111	1011	Shift
0	1101	1111	1011	Add
0	0110	1111	1011	Shift
1	**0001**	**1111**	**1011**	**Add**

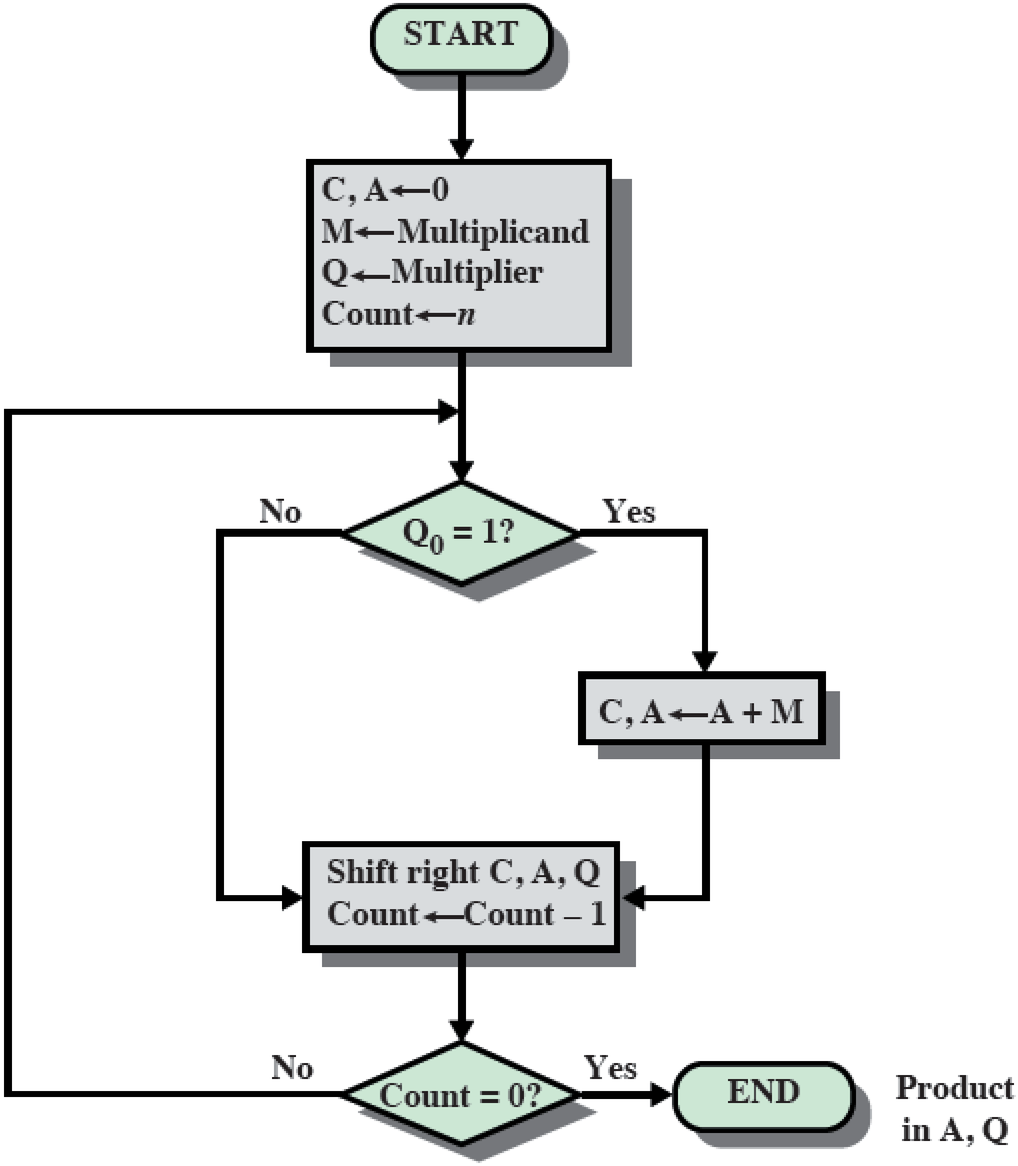
Add and Shift

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

Add and Shift

C	A	Q	M	
0	0000	1101	1011	Initial Values
0	1011	1101	1011	Add
0	0101	1110	1011	Shift
0	0010	1111	1011	Shift
0	1101	1111	1011	Add
0	0110	1111	1011	Shift
1	0001	1111	1011	Add
0	1000	1111	1011	Final Result

Flow Chart of the Implementation



Twos Complement Multiplication

- With Addition and Subtraction we treated twos complements as unsigned integers, which doesn't work anymore
- The multiplication of $-7 \cdot 3$, when interpreted as unsigned int corresponds to the multiplication of $9 \cdot 3$

```
    1001 (9)
   ×0011 (3)
--------------------
00001001 1001 × 2^0
00010010 1001 × 2^1
--------------------
00011011 (27)
```

- The result is 27 and not -21

Twos Complement Multiplication

- What if we extend the negative number to a $2n$ bit negative number?
- Extending a signed int is done by filling the new bits with the sign

```
0011 -> 00000011
```

```
1001 -> 11111001
```

- Our Example from before:

```
    1001 (-7)
   ×0011 (3)
--------------------------------
11111001 (-7) × 2^0 = (-7)
11110010 (-7) × 2^1 = (-14)
--------------------------------
11101011 (-21)
```

- Now it seems to work

Twos Complement Multiplication

- Unfortunately, it doesn't!
- Calculating $3 \cdot (-7)$:

```
0011 -> 00000011
```

```
1001 -> 11111001
```

- Our Example from before:

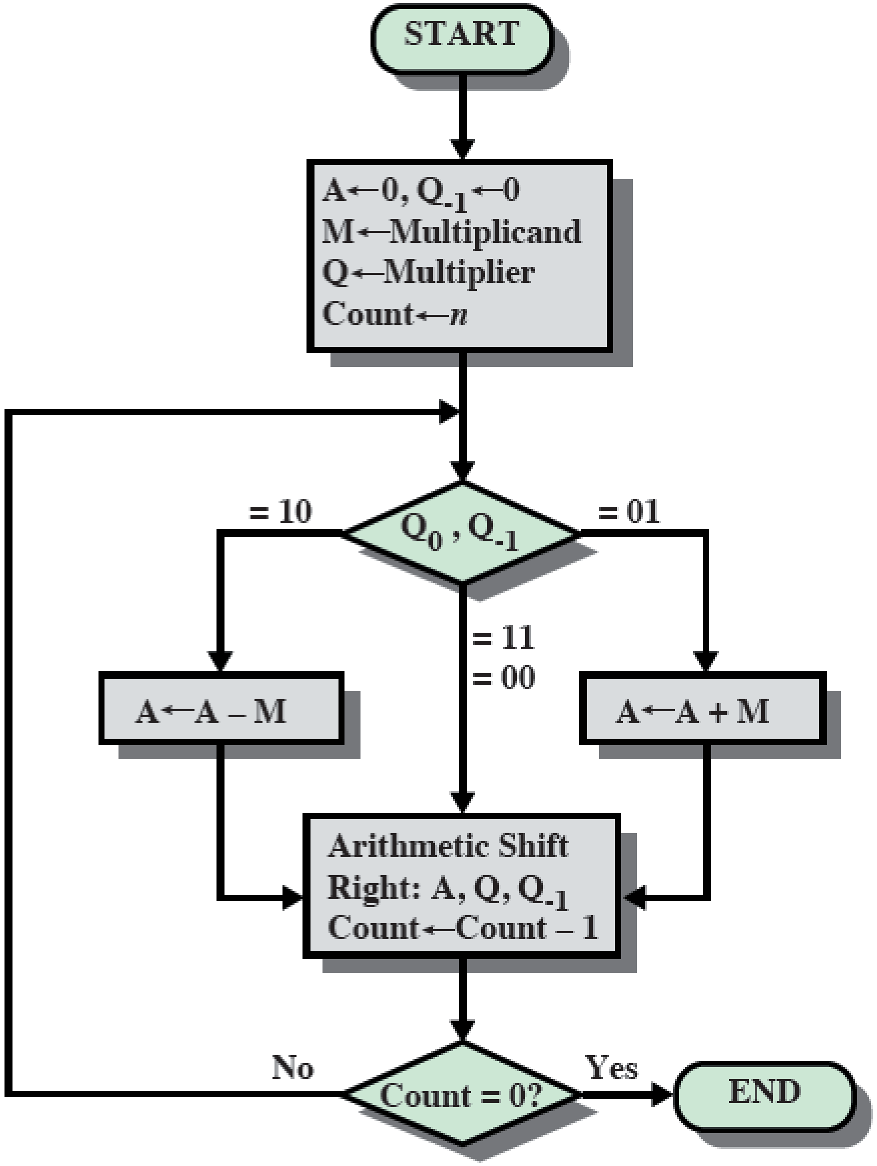
```
    0011 (3)
   ×1001 (-7)
---------------------
00000011 (3) × 2^0 = (3)
00000000
00000000
00011000 (3) x 2^3 = (24)
---------------------
00011011 (27)
```

- Damn, still not working!

The Solution: Booth's Algorithm

- As before, Multiplicand is in M, Multiplier in Q
- We have an additional 1-Bit register Q_{-1}
- The Algorithm now decides to add or subtract M from A
  - Adding: $Q_0,Q_{-1}= 0,1$
  - Subtracting: $Q_0,Q_{-1}= 1,0$
  - Only Shifting: $Q_0,Q_{-1}= 1,1$ or $0,0$

- The shift is an arithmetic shift, i.e., it preserves the sign



Booth's Algorithm

- Multiplying $-3 \cdot 7$:

A	Q	Q-1	M	
0000	1101	0	0111	Initial Values

Booth's Algorithm

- Multiplying $-3 \cdot 7$:

A	Q	Q-1	M	
0000	1101	0	0111	Initial Values

Booth's Algorithm

- Multiplying $-3 \cdot 7$:

```
  A        Q    Q-1        M
_________________________________________________
 0000    1101   0        0111   Initial Values
_________________________________________________
+1001    1101   0        0111   1,0 -> Subtract
_____
 1001
```

Booth's Algorithm

- Multiplying $-3 \cdot 7$:

A	Q	Q-1	M	
0000	1101	0	0111	Initial Values
1001	1101	0	0111	1,0 -> Subtract

Booth's Algorithm

- Multiplying $-3 \cdot 7$:

A	Q	Q-1	M	
0000	1101	0	0111	Initial Values
1001	1101	0	0111	1,0 -> Subtract
1100	1110	1	0111	Shift

Booth's Algorithm

- Multiplying $-3 \cdot 7$:

A	Q	Q-1	M	
0000	1101	0	0111	Initial Values
1001	1101	0	0111	1,0 -> Subtract
1100	1110	1	0111	Shift

Booth's Algorithm

- Multiplying $-3 \cdot 7$:

A	Q	Q-1	M	
0000	1101	0	0111	Initial Values
1001	1101	0	0111	1,0 -> Subtract
1100	1110	1	0111	Shift
+0111	1110	1	0111	0,1 -> Add
0011				

Booth's Algorithm

- Multiplying $-3 \cdot 7$:

A	Q	Q-1	M	
0000	1101	0	0111	Initial Values
1001	1101	0	0111	1,0 -> Subtract
1100	1110	1	0111	Shift
0011	**1110**	**1**	**0111**	**0,1 -> Add**

Booth's Algorithm

- Multiplying $-3 \cdot 7$:

A	Q	Q-1	M	
0000	1101	0	0111	Initial Values
1001	1101	0	0111	1,0 -> Subtract
1100	1110	1	0111	Shift
0011	1110	1	0111	0,1 -> Add
0001	1111	0	0111	Shift

Booth's Algorithm

- Multiplying $-3 \cdot 7$:

A	Q	Q-1	M	
0000	1101	0	0111	Initial Values
1001	1101	0	0111	1,0 -> Subtract
1100	1110	1	0111	Shift
0011	1110	1	0111	0,1 -> Add
0001	**1111**	**0**	**0111**	**Shift**

Booth's Algorithm

- Multiplying $-3 \cdot 7$:

A	Q	Q-1	M	
0000	1101	0	0111	Initial Values
1001	1101	0	0111	1,0 -> Subtract
1100	1110	1	0111	Shift
0011	1110	1	0111	0,1 -> Add
0001	1111	0	0111	Shift
+1001	1111	0	0111	1,0 -> Subtract
1010				

Booth's Algorithm

- Multiplying $-3 \cdot 7$:

A	Q	Q-1	M	
0000	1101	0	0111	Initial Values
1001	1101	0	0111	1,0 -> Subtract
1100	1110	1	0111	Shift
0011	1110	1	0111	0,1 -> Add
0001	1111	0	0111	Shift
1010	**1111**	**0**	**0111**	**1,0 -> Subtract**

Booth's Algorithm

- Multiplying $-3 \cdot 7$:

A	Q	Q-1	M	
0000	1101	0	0111	Initial Values
1001	1101	0	0111	1,0 -> Subtract
1100	1110	1	0111	Shift
0011	1110	1	0111	0,1 -> Add
0001	1111	0	0111	Shift
1010	1111	0	0111	1,0 -> Subtract
1101	0111	1	0111	Shift

Booth's Algorithm

- Multiplying $-3 \cdot 7$:

A	Q	Q-1	M	
0000	1101	0	0111	Initial Values
1001	1101	0	0111	1,0 -> Subtract
1100	1110	1	0111	Shift
0011	1110	1	0111	0,1 -> Add
0001	1111	0	0111	Shift
1010	1111	0	0111	1,0 -> Subtract
1101	**0111**	**1**	**0111**	**Shift**

Booth's Algorithm

- Multiplying $-3 \cdot 7$:

A	Q	Q-1	M	
0000	1101	0	0111	Initial Values
1001	1101	0	0111	1,0 -> Subtract
1100	1110	1	0111	Shift
0011	1110	1	0111	0,1 -> Add
0001	1111	0	0111	Shift
1010	1111	0	0111	1,0 -> Subtract
1101	0111	1	0111	Shift
1110	1011	1	0111	1,1 -> Shift only

Booth's Algorithm

- Multiplying $-3 \cdot 7$:

A	Q	Q-1	M	
0000	1101	0	0111	Initial Values
1001	1101	0	0111	1,0 -> Subtract
1100	1110	1	0111	Shift
0011	1110	1	0111	0,1 -> Add
0001	1111	0	0111	Shift
1010	1111	0	0111	1,0 -> Subtract
1101	0111	1	0111	Shift
1110	1011	1	0111	Final Result

Booth's Algorithm: Why Does it Work?

- Lets see what happens when we multiply M with a number having a block of 1's, e.g.: 00011110:
$$M \cdot 00011110 \;=\; M \cdot (2^4 + \;2^3 + \;2^2 + \;2^1) = \;M \cdot 30$$

Booth's Algorithm: Why Does it Work?

- Lets see what happens when we multiply M with a number having a block of 1's, e.g.: 00011110:

$$M \cdot 00011110 \;=\; M \cdot (2^4 +\; 2^3 +\; 2^2 +\; 2^1) =\; M \cdot 30$$

- Such a block can be reduced to:

$$2^n + 2^{n-1} + \cdots + 2^{n-K} =\; 2^{n+1} - 2^{n-K}$$

Booth's Algorithm: Why Does it Work?

- Lets see what happens when we multiply M with a number having a block of 1's, e.g.: 00011110:

$$M \cdot 00011110 \;=\; M \cdot (2^4 + \;2^3 + \;2^2 + \;2^1) = \;M \cdot 30$$

- Such a block can be reduced to:

$$2^n + 2^{n-1} + \cdots + 2^{n-K} = \;2^{n+1} - 2^{n-K}$$

- That means for our example:

$$M \cdot 00011110 \;=\; M \cdot (2^5 - \;2^1) = \;M \cdot 30$$

Booth's Algorithm: Why Does it Work?

- Lets see what happens when we multiply M with a number having a block of 1's, e.g.: 00011110:

$$M \cdot 00011110 \;=\; M \cdot (2^4 + \; 2^3 + \; 2^2 + \; 2^1) = \; M \cdot 30$$

- Such a block can be reduced to:

$$2^n + 2^{n-1} + \cdots + 2^{n-K} = \; 2^{n+1} - 2^{n-K}$$

- That means for our example:

$$M \cdot 00011110 \;=\; M \cdot (2^5 - \; 2^1) = \; M \cdot 30$$

- This works also for blocks of size 1:

$$M \cdot 2^k = \; M \cdot \left(2^{k+1} - 2^k\right)$$

Booth's Algorithm: Why Does it Work?

- Booth's Algorithm uses exactly this trick
- Whenever such a block is "opened" (Q_0,Q_{-1}= 1,0) we subtract
- Whenever a such a block is "closed" (Q_0,Q_{-1}= 0,1) we add
- Within a block or outside, we only shift

- Does this work with negative multipliers?
  - Yes, there is a proof in the book
  - We just look at an example

- Normally, it is even more efficient than the unsinged method (worst case are the same number of additions/subtractions)

Negative Multipliers

- Let's Multiply M with -6 (11111010):

$$-6 \ = \ -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^1$$

- Now, using the openings and closing of Booth's algorithm:

11111010

- Openings: 2^3 and 2^1
- Closings: 2^2

- Together (openings are subtracted, closings added):

$$M \cdot (-\,2^3 + \ 2^2 \ - \ 2^1) = \ M \cdot (-\,2^3 + \ 2^1) = \ M \cdot (-6)$$

Divisions

- $\frac{147}{11} = 13$ with a remainder of 4:

```
       10010011 / 1011
```

Quotient (Result)

Divisions

- $\frac{147}{11} = 13$ with a remainder of 4:

```
    10010011 / 1011
-1011                   1011 > 1



                        Quotient (Result)
```

Divisions

- $\frac{147}{11} = 13$ with a remainder of 4:

```
    10010011 / 1011
-1011

____________________________________________
    0                      Quotient (Result)
```

Divisions

- $\frac{147}{11} = 13$ with a remainder of 4:

```
    10010011 / 1011
 -1011                  1011 > 10

    0                   Quotient (Result)
```

Divisions

- $\frac{147}{11} = 13$ with a remainder of 4:

```
    10010011 / 1011
 -1011



    00                  Quotient (Result)
```

Divisions

- $\frac{147}{11} = 13$ with a remainder of 4:

```
    10010011 / 1011
  -1011                 1011 > 100


    00                  Quotient (Result)
```

Divisions

- $\frac{147}{11} = 13$ with a remainder of 4:

```
    10010011 / 1011
  -1011




    000                   Quotient (Result)
```

Divisions

- $\frac{147}{11} = 13$ with a remainder of 4:

```
    10010011 / 1011
   -1011                  1011 > 1001

    000                   Quotient (Result)
```

Divisions

- $\frac{147}{11} = 13$ with a remainder of 4:

```
    10010011 / 1011
   -1011



    0000                  Quotient (Result)
```

Divisions

- $\frac{147}{11} = 13$ with a remainder of 4:

```
    10010011 / 1011
    -1011                   1011 < 10010



    0000                    Quotient (Result)
```

Divisions

- $\frac{147}{11} = 13$ with a remainder of 4:

```
    10010011 / 1011
   -1011                1011 < 10010

   ________________________________________
    00001               Quotient (Result)
```

Divisions

- $\frac{147}{11} = 13$ with a remainder of 4:

```
      10010011 / 1011
      -1011
         111                    Partial Remainder

      00001                     Quotient (Result)
```

Divisions

- $\frac{147}{11} = 13$ with a remainder of 4:

```
    10010011 / 1011
    -1011
      1110                Partial Remainder
     -1011                1011 < 1110



    00001                 Quotient (Result)
```

Divisions

- $\frac{147}{11} = 13$ with a remainder of 4:

```
    10010011 / 1011
    -1011
      1110                Partial Remainder
     -1011                1011 < 1110



    000011                Quotient (Result)
```

Divisions

- $\frac{147}{11} = 13$ with a remainder of 4:

```
    10010011 / 1011
    -1011
      1110                 Partial Remainder
     -1011
      0011                 Partial Remainder

    000011                 Quotient (Result)
```

Divisions

- $\frac{147}{11} = 13$ with a remainder of 4:

```
      10010011 / 1011
      -1011
        1110                Partial Remainder
       -1011
        00111               Partial Remainder
        -1011               1011 > 111

      000011                Quotient (Result)
```

Divisions

- $\frac{147}{11} = 13$ with a remainder of 4:

```
    10010011 / 1011
    -1011
      1110                 Partial Remainder
     -1011
      00111                Partial Remainder
       -1011               1011 > 111

    0000110                Quotient (Result)
```

Divisions

- $\frac{147}{11} = 13$ with a remainder of 4:

```
    100100111 / 1011
    -1011
       1110                Partial Remainder
      -1011
       001111              Partial Remainder
         -1011             1011 < 1111
_______________________________________
    0000110                Quotient (Result)
```

Divisions

- $\frac{147}{11} = 13$ with a remainder of 4:

```
    100100111 / 1011
    -1011
      1110                    Partial Remainder
     -1011
      001111                  Partial Remainder
        -1011                 1011 < 1111

    000011101                 Quotient (Result)
```

Divisions

- $\frac{147}{11} = 13$ with a remainder of 4:

```
    100100111 / 1011
    -1011
      1110                  Partial Remainder
     -1011
      001111                Partial Remainder
       -1011
         100                Partial Remainder
    00001101                Quotient (Result)
```

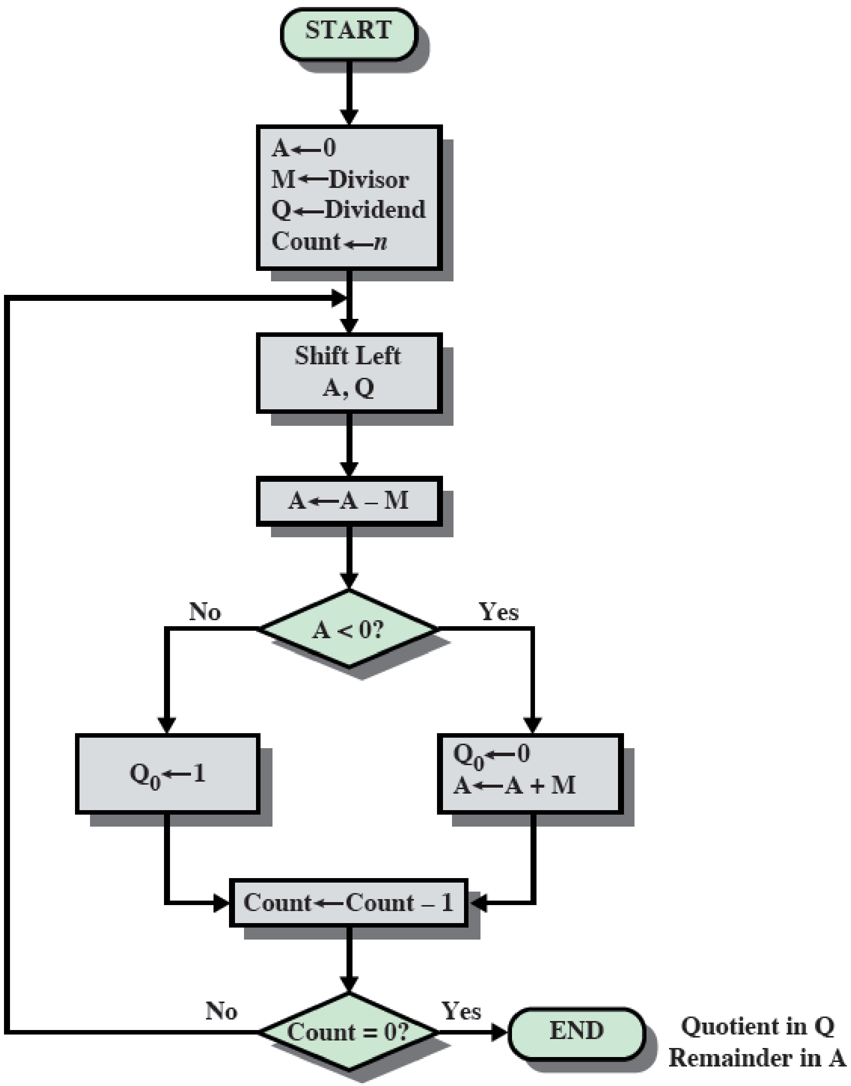
Divisions

- $\frac{147}{11} = 13$ with a remainder of 4:

```
     10010011 / 1011
     -1011
       1110                  Partial Remainder
      -1011
       001111                Partial Remainder
         -1011
            100              Remainder
     00001101                Final Result
```

Flow Chart of the Division

- The Divisor is placed in M, the Dividend in Q.
- A stores the partial Remainders
- At the end, Q holds the Quotient, A the remainder.



Step for Step: 7/3

A	Q	M	
0000	0111	0011	Initial Values

Step for Step: 7/3

A	Q	M	
0000	0111	0011	Initial Values
0000	1110	0011	Shift

Step for Step: 7/3

A	Q	M	
0000	0111	0011	Initial Values
0000	1110	0011	Shift
+1101			Substract, result negative
1101			

Step for Step: 7/3

A	Q	M	
0000	0111	0011	Initial Values
0000	1110	0011	Shift
+1101			Substract, result negative
1101			
0000	1110	0011	Restore, set Q0 = 0

Step for Step: 7/3

A	Q	M	
0000	0111	0011	Initial Values
0000	1110	0011	Shift
+1101			Substract, result negative
1101			
0000	**1110**	**0011**	**Restore, set Q0 = 0**

Step for Step: 7/3

```
  A     Q     M
-----------------------------------------
 0000  0111  0011  Initial Values
-----------------------------------------
 0000  1110  0011  Shift
+1101              Substract, result negative
 ----
 1101
 0000  1110  0011  Restore, set Q0 = 0
-----------------------------------------
 0001  1100  0011  Shift
```

Step for Step: 7/3

A	Q	M	
0000	0111	0011	Initial Values
0000	1110	0011	Shift
+1101			Substract, result negative
1101			
0000	1110	0011	Restore, set Q0 = 0
0001	1100	0011	Shift
+1101			Substract, result negative
1110			

Step for Step: 7/3

A	Q	M	
0000	0111	0011	Initial Values
0000	1110	0011	Shift
+1101			Substract, result negative
1101			
0000	1110	0011	Restore, set Q0 = 0
0001	1100	0011	Shift
+1101			Substract, result negative
1110			
0001	1100	0011	Restore, set Q0 = 0

Step for Step: 7/3

A	Q	M	
0000	0111	0011	Initial Values
0000	1110	0011	Shift
+1101			Substract, result negative
1101			
0000	1110	0011	Restore, set Q0 = 0
0001	1100	0011	Shift
+1101			Substract, result negative
1110			
0001	**1100**	**0011**	**Restore, set Q0 = 0**

Step for Step: 7/3

A	Q	M	
0000	0111	0011	Initial Values
0000	1110	0011	Shift
+1101			Substract, result negative
1101			
0000	1110	0011	Restore, set Q0 = 0
0001	1100	0011	Shift
+1101			Substract, result negative
1110			
0001	1100	0011	Restore, set Q0 = 0
0011	1000	0011	Shift

Step for Step: 7/3

```
 A      Q      M
 0000   0111   0011   Initial Values
 0000   1110   0011   Shift
+1101                 Substract, result negative
 1101
 0000   1110   0011   Restore, set Q0 = 0
 0001   1100   0011   Shift
+1101                 Substract, result negative
 1110
 0001   1100   0011   Restore, set Q0 = 0
 0011   1000   0011   Shift
+1101                 Substract, result positive
 0000
```

Step for Step: 7/3

A	Q	M	
0000	0111	0011	Initial Values
0000	1110	0011	Shift
+1101			Substract, result negative
1101			
0000	1110	0011	Restore, set Q0 = 0
0001	1100	0011	Shift
+1101			Substract, result negative
1110			
0001	1100	0011	Restore, set Q0 = 0
0011	1000	0011	Shift
+1101			Substract, result positive
0000	1001	0011	Keep, set Q0 = 1

Step for Step: 7/3

A	Q	M	
0000	0111	0011	Initial Values
0000	1110	0011	Shift
+1101			Substract, result negative
1101			
0000	1110	0011	Restore, set Q0 = 0
0001	1100	0011	Shift
+1101			Substract, result negative
1110			
0001	1100	0011	Restore, set Q0 = 0
0011	1000	0011	Shift
+1101			Substract, result positive
0000	**1001**	**0011**	**Keep, set Q0 = 1**

Step for Step: 7/3

A	Q	M	
0000	0111	0011	Initial Values
0000	1110	0011	Shift
+1101			Substract, result negative
1101			
0000	1110	0011	Restore, set Q0 = 0
0001	1100	0011	Shift
+1101			Substract, result negative
1110			
0001	1100	0011	Restore, set Q0 = 0
0011	1000	0011	Shift
+1101			Substract, result positive
0000	1001	0011	Keep, set Q0 = 1
0001	0010	0011	Shift

Step for Step: 7/3

```
 A     Q     M
 0000  0111  0011  Initial Values
 0000  1110  0011  Shift
+1101              Substract, result negative
 1101
 0000  1110  0011  Restore, set Q0 = 0
 0001  1100  0011  Shift
+1101              Substract, result negative
 1110
 0001  1100  0011  Restore, set Q0 = 0
 0011  1000  0011  Shift
+1101              Substract, result positive
 0000  1001  0011  Keep, set Q0 = 1
 0001  0010  0011  Shift
+1101              Substract, result negative
 1110
```

Step for Step: 7/3

A	Q	M	
0000	0111	0011	Initial Values
0000	1110	0011	Shift
+1101			Substract, result negative
1101			
0000	1110	0011	Restore, set Q0 = 0
0001	1100	0011	Shift
+1101			Substract, result negative
1110			
0001	1100	0011	Restore, set Q0 = 0
0011	1000	0011	Shift
+1101			Substract, result positive
0000	1001	0011	Keep, set Q0 = 1
0001	0010	0011	Shift
+1101			Substract, result negative
1110			
0001	0010	0011	Restore, set Q0 = 0

Step for Step: 7/3

A	Q	M	
0000	0111	0011	Initial Values
0000	1110	0011	Shift
+1101			Substract, result negative
1101			
0000	1110	0011	Restore, set Q0 = 0
0001	1100	0011	Shift
+1101			Substract, result negative
1110			
0001	1100	0011	Restore, set Q0 = 0
0011	1000	0011	Shift
+1101			Substract, result positive
0000	1001	0011	Keep, set Q0 = 1
0001	0010	0011	Shift
+1101			Substract, result negative
1110			
0001	0010	0011	Remainder, Result

Negative Numbers

- This approach could be extended to work with negative numbers
- In practice, it is calculated only with positive numbers
- At the end, the signs of the result and the remainder are assigned

Computer Arithmetic

- Number systems
- Integers
- Floats

Fixed Point Representation

- With a fixed-point notation it is possible to represent a range of positive and negative integers centered on or near 0
- By assuming a fixed binary or radix point, this format allows the representation of numbers with a fractional component as well
- Limitations:
  - Very large numbers cannot be represented nor can very small fractions
  - The fractional part of the quotient in a division of two large numbers could be lost

Floating Point Numbers

- Aim: We want to store extremely large and small numbers with as little overhead as possible

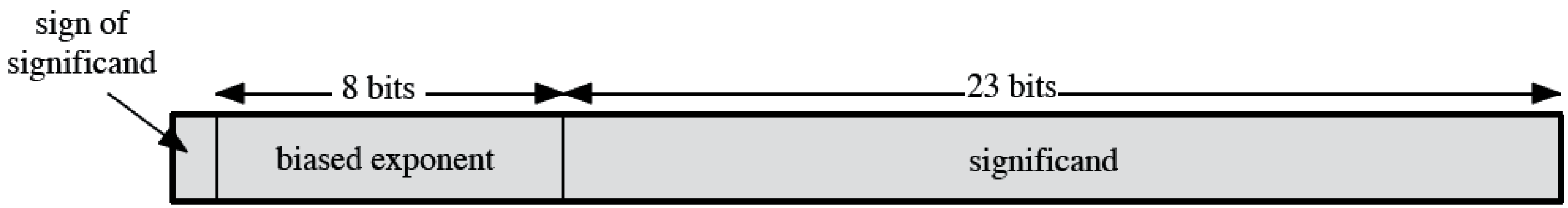
- Questions:
  - Do we really care about all digits in a number?

- Answer:
  - Most of the time, we don’t.

- Idea:
  - Store as number of fixed length (Significand S) with an exponent E:

$$\pm S \cdot B^{\pm E}$$

Typical 32-Bit Representation



(a) Format

$$1.1010001 \times 2^{10100} = 0\ 10010011\ 10100010000000000000000 = 1.6328125 \times 2^{20}$$
$$-1.1010001 \times 2^{10100} = 1\ 10010011\ 10100010000000000000000 = -1.6328125 \times 2^{20}$$
$$1.1010001 \times 2^{-10100} = 0\ 01101011\ 10100010000000000000000 = 1.6328125 \times 2^{-20}$$
$$-1.1010001 \times 2^{-10100} = 1\ 01101011\ 10100010000000000000000 = -1.6328125 \times 2^{-20}$$

(b) Examples

Biased Exponent

- The exponent value is stored in k bits.

- The representation used is known as a biased representation:
  - A fixed value, called the bias, is subtracted from the field to get the true exponent value.
  - Typically, the bias equals $2^{k-1} - 1$

- Example for 8 Bit Exponent:
  - A bit can represent numbers from 0 through 255.
  - The Bias is $2^7 - 1 = 127$
  - The true exponent values are in the range -127 to +128

- Why aren't we using the twos complement for E?

The Significand

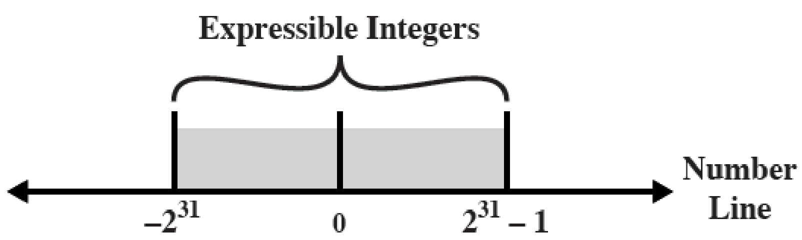
- The final portion of the word
- Any floating-point number can be expressed in many ways:
  - All these numbers are equivalent:

$$0.110 \cdot 2^5$$
$$110 \cdot 2^2$$
$$0.0110 \cdot 2^6$$

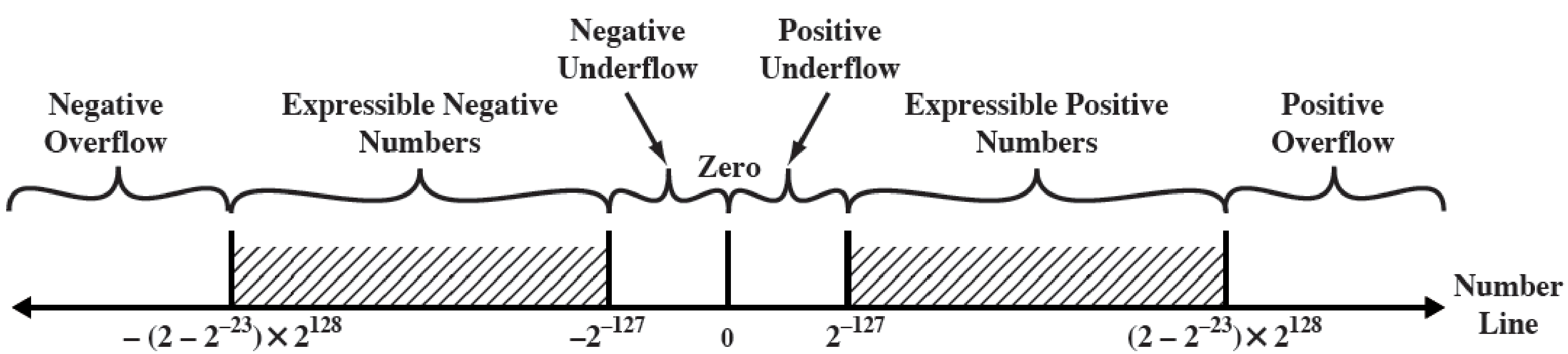
- *Normal number*
  - The most significant digit of the significand is nonzero

$$\pm 1.bbb \ldots b \cdot 2^{\pm E}$$

Expressible Numbers



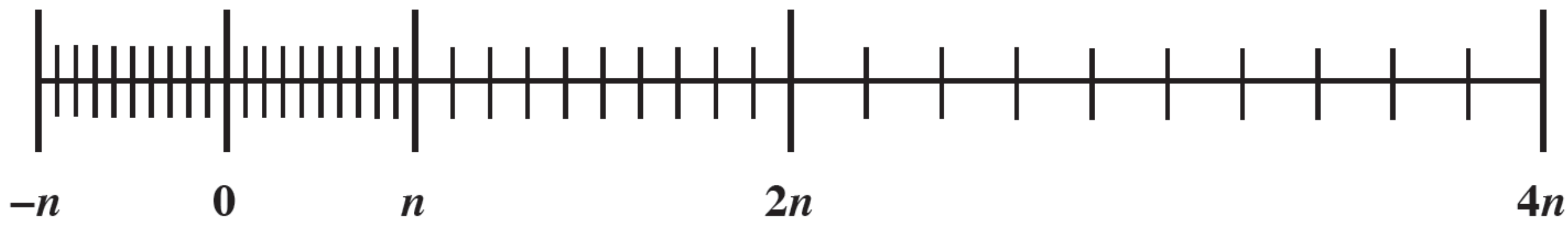
(a) Twos Complement Integers



(b) Floating-Point Numbers

- Floats cover an enormous range of numbers
- But not with the same density

Density of Floating-Point Numbers



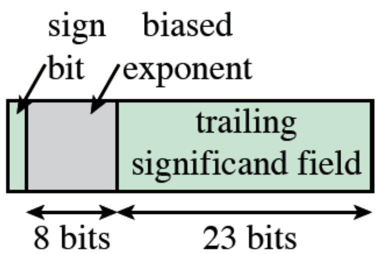
- The closer to zero, the finer the coverage, the further away, the sparser
- Float often don't produce exact numbers
- But this is okay:
  - If you look at your bank account, you are interested in all numbers
  - If you look at the US national debt, you don't care about the single dollar anymore

IEEE Standard 754

- Adopted in 1985 and revised in 2008
- Most important floating-point representation is defined
- Ensures the portability of programs from one processor to another
- Standard has been widely adopted and is used on virtually all contemporary processors and arithmetic coprocessors
- IEEE 754-2008 covers both **binary** and decimal floating-point representations

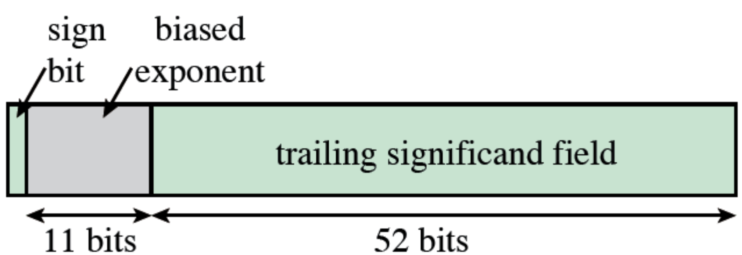
The Binary Basic Formats

Single



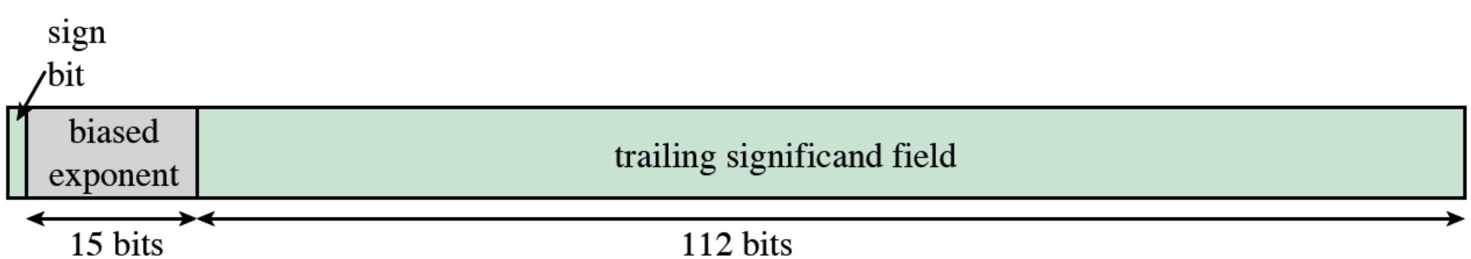
(a) binary32 format

Double



(b) binary64 format

Quadruple



(c) binary128 format

Binary Format Parameters

Parameter	binary32	Format binary64	binary128
Storage width (bits)	32	64	128
Exponent width (bits)	8	11	15
Exponent bias	127	1023	16383
Maximum exponent	127	1023	16383
Minimum exponent	– 126	– 1022	– 16382
~ normal number range (base 10)	$10^{-38}, 10^{+38}$	$10^{-308}, 10^{+308}$	$10^{-4932}, 10^{+4932}$
Trailing significand width (bits)*	23	52	112
Number of exponents	254	2046	32766
Number of fractions	2^{23}	2^{52}	2^{112}
Number of values	$1.98 \cdot 2^{31}$	$1.99 \cdot 2^{63}$	$1.99 \cdot 2^{128}$
Smallest positive normal number	2^{-126}	2^{-1022}	2^{-16362}
Largest positive normal number	$2^{128} - 2^{104}$	$2^{1024} - 2^{971}$	$2^{16384} - 2^{16271}$
Smallest subnormal magnitude	2^{-149}	2^{-1074}	2^{-16494}

Special Values for 32Bit float

	Sign	Biased exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	– 0
plus infinity	0	all 1s	0	∞
minus infinity	1	all 1s	0	$-\infty$
quiet NaN	0 or 1	all 1s	$\neq 0$; first bit $= 1$	qNaN
signaling NaN	0 or 1	all 1s	$\neq 0$; first bit $= 0$	sNaN
positive normal nonzero	0	$0 < e < 255$	f	$2^{e-127}(1.\mathrm{f})$
negative normal nonzero	1	$0 < e < 255$	f	$-2^{e-127}(1.\mathrm{f})$
positive subnormal	0	0	$f \neq 0$	$2^{e-126}(0.\mathrm{f})$
negative subnormal	1	0	$f \neq 0$	$-2^{e-126}(0.\mathrm{f})$

- For other binary floats, only the numbers for the exponent changes accordingly
- The different exceptions are explained later

Additional Formats of IEEE 754

- Extended precision formats
  - Provide additional bits in the exponent (extended range) and in the significand (extended precision)
  - Lessens the chance of a final result that has been contaminated by excessive roundoff error
  - Lessens the chance of an intermediate overflow aborting a computation whose final result would have been representable in a basic format
  - Intel FPUs use internally 80-Bit representations

- Extendable Precision Format
  - Precision and range are defined under user control
  - May be used for intermediate calculations but the standard places no constraint or format or length

Floating Point Arithmetic

Floating Point Numbers	Arithmetic Operations
$X = X_s \cdot B^{X_E}$ $Y = Y_s \cdot B^{Y_E}$	$\left.\begin{aligned} X + Y &= (X_S \cdot B^{X_E - Y_E} + Y_S) \cdot B^{Y_E} \\ X - Y &= (X_S \cdot B^{X_E - Y_E} - Y_S) \cdot B^{Y_E} \end{aligned}\right\} X_E \leq Y_E$ $X \cdot Y = (X_S \cdot Y_S) \cdot B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_S}{Y_S}\right) \cdot B^{X_E - Y_E}$

- **Exponent overflow**: A positive exponent exceeds the maximum possible exponent value.
- **Exponent underflow**: A negative exponent is less than the minimum possible exponent value.
- **Significand underflow**: In the process of aligning significands, digits may flow off the right end of the significand.
- **Significand overflow**: The addition of two significands of the same sign may result in a carry out of the most significant bit.

Examples

$$X = 0.3 \cdot 10^2 = 30$$
$$Y = 0.2 \cdot 10^3 = 200$$

$$X + Y = (0.3 \cdot 10^{2-3} + 0.2) \cdot 10^3 = 0.23 \cdot 10^3 = 230$$

$$X - Y = (0.3 \cdot 10^{2-3} - 0.2) \cdot 10^3 = (-0.17) \cdot 10^3 = -170$$

$$X \cdot Y = (0.3 \cdot 0.2) \cdot 10^{2+3} = 0.06 \cdot 10^5 = 6000$$

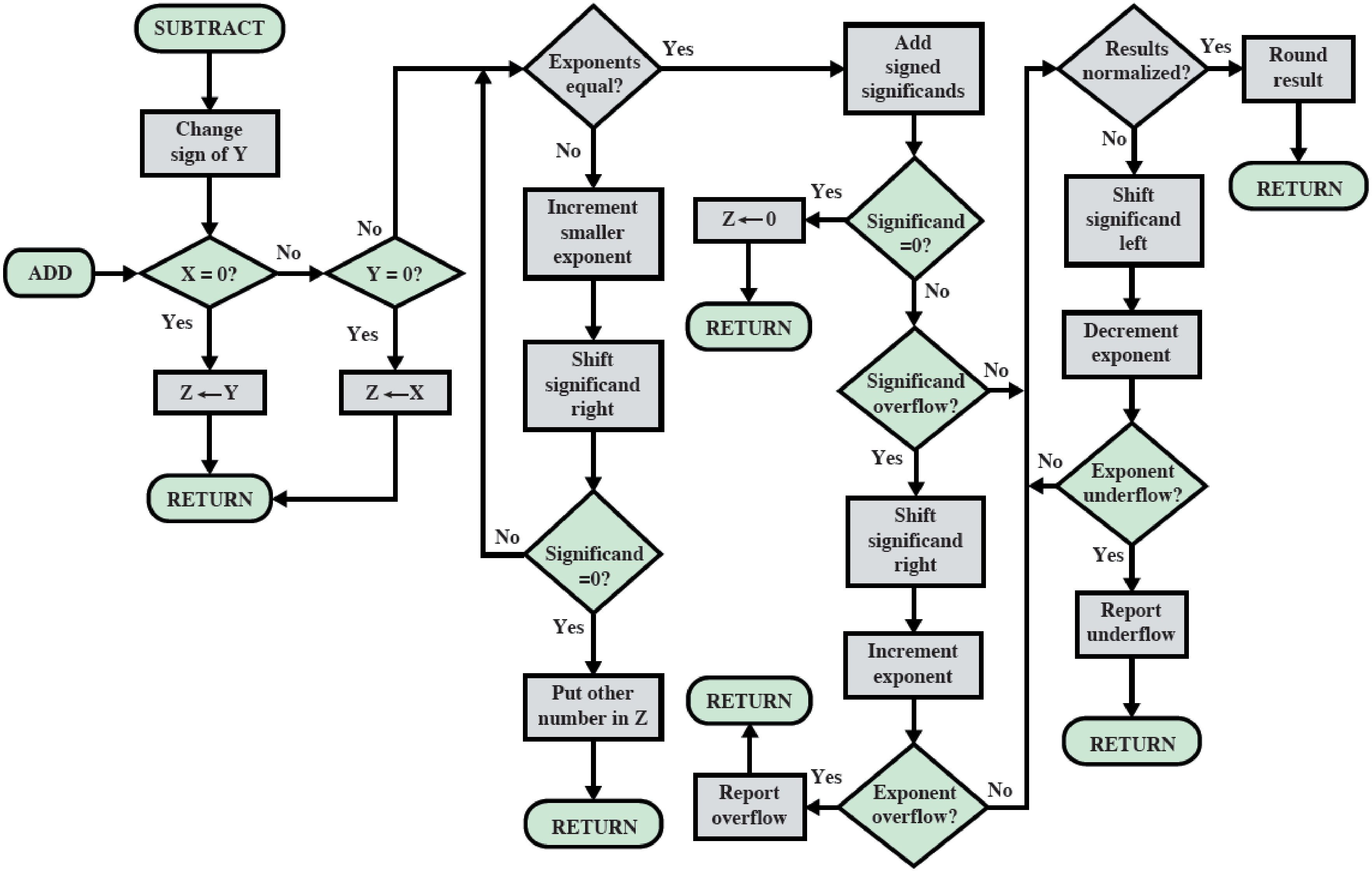
$$\frac{X}{Y} = \left(\frac{0.3}{0.2}\right) \cdot 10^{2-3} = 1.5 \cdot 10^{-1} = 0.15$$

Floating Point Arithmetic

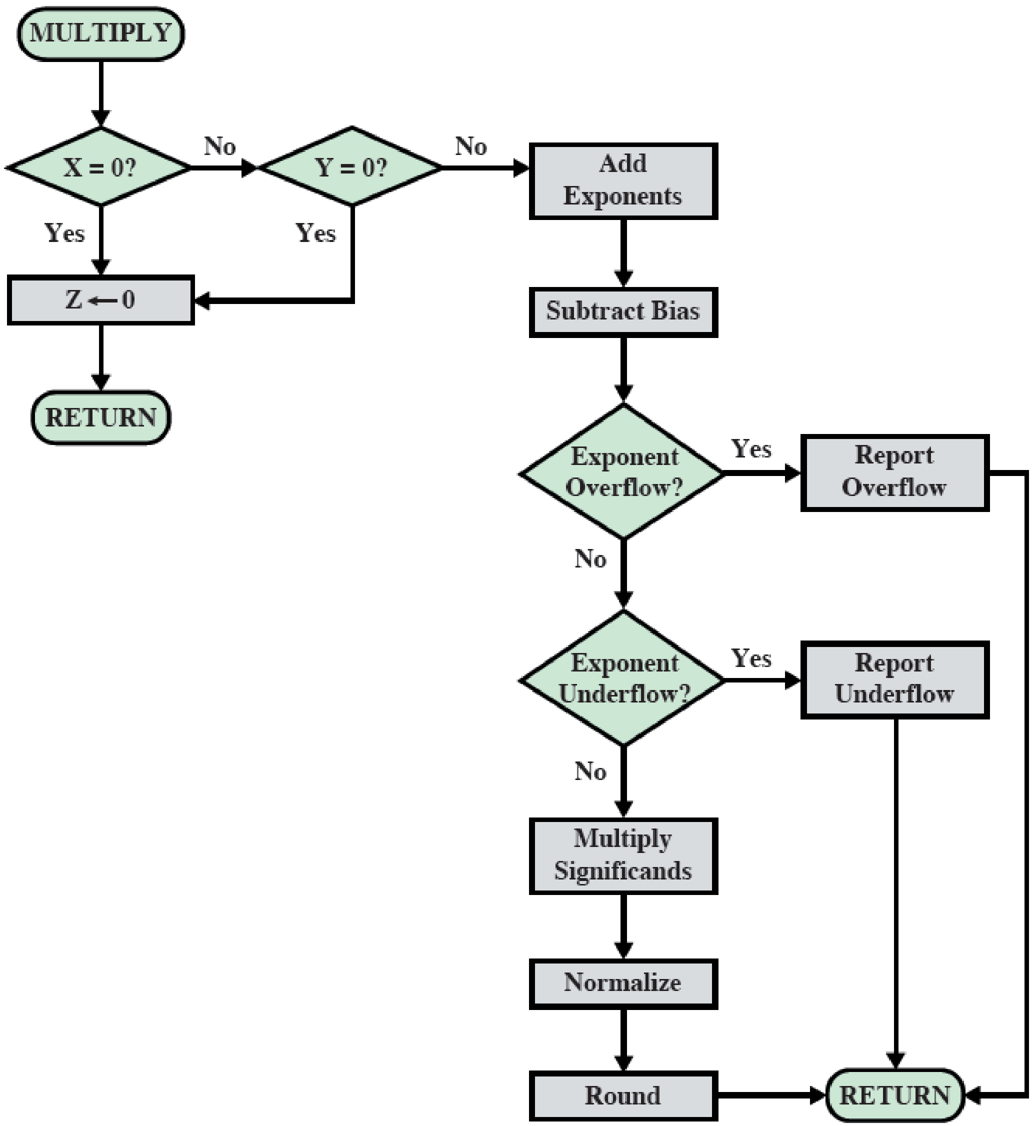
- In floating-point arithmetic, addition and subtraction are more complex than multiplication and division. This is because of the need for alignment.

- The four basic steps are:
  - 1. Check for zeros.
  - 2. Align the significands.
  - 3. Add or subtract the significands.
  - 4. Normalize the result.

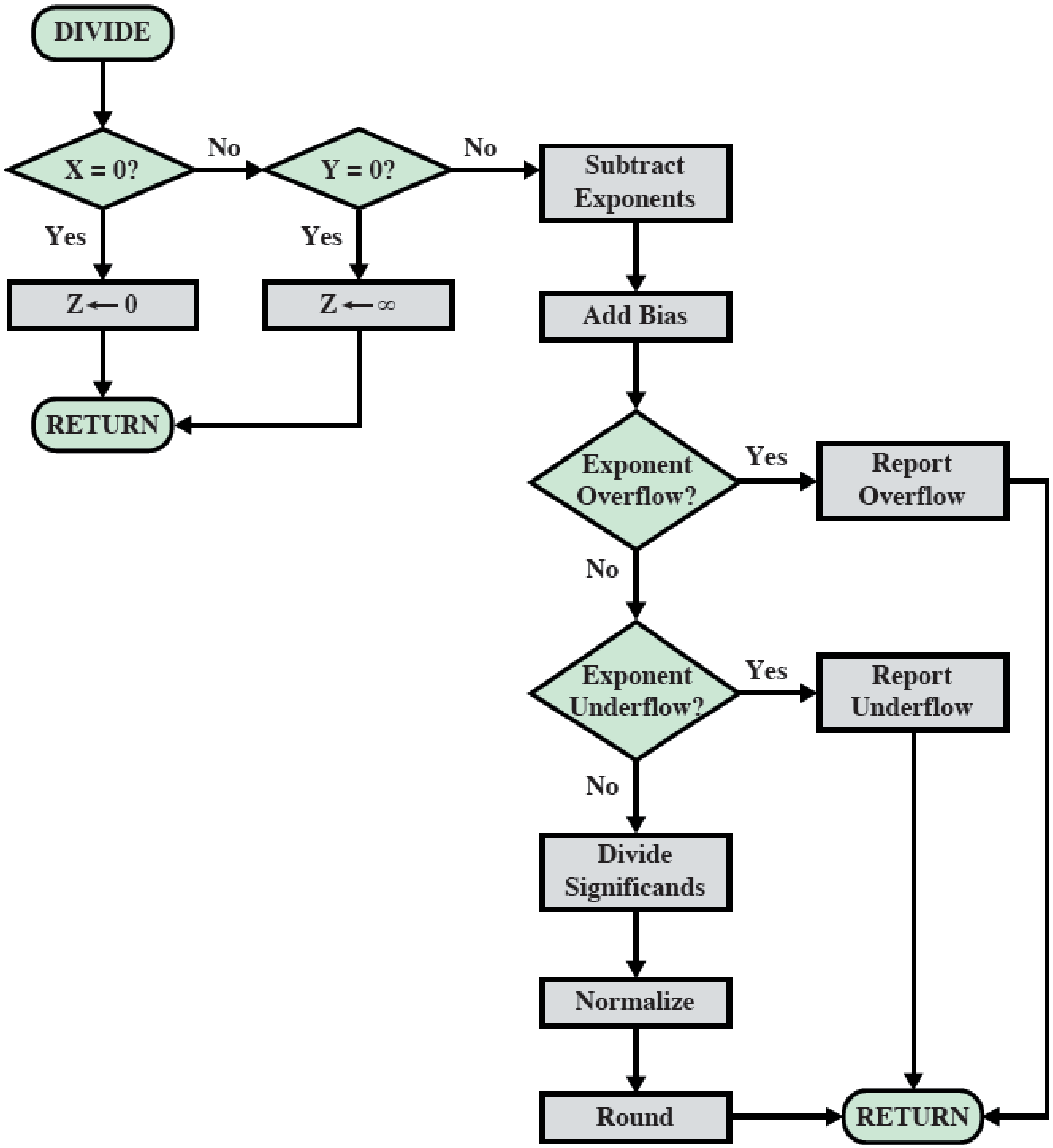
Flowchart Addition



Flowchart Multiplication



Flowchart Division



Precision: Guard Bits

- Consider the subtraction of very close numbers:

```
 x = 1.000…00 x 2^1
-y = 1.111…11 x 2^0
 x = 1.000…00 x 2^1
-y = 0.111…11 x 2^1 Align Significand
 z = 0.000…01 x 2^1
   = 1.000…00 x 2^-22 Normalize
```

Precision: Guard Bits

- Consider the subtraction of very close numbers:

```
 x = 1.000…00 x 2^1
-y = 1.111…11 x 2^0
 x = 1.000…00 x 2^1
-y = 0.111…11 x 2^1 Align Significand
 z = 0.000…01 x 2^1
   = 1.000…00 x 2^-22 Normalize
```

- Guard Bits are used to pad out the right side of the Significand

```
 x = 1.000…00 0000 x 2^1
-y = 1.111…11 0000 x 2^0
 x = 1.000…00 0000 x 2^1
-y = 0.111…11 1000 x 2^1 Align Significand
 z = 0.000…00 1000 x 2^1
   = 1.000…00 0000 x 2^-23 Normalize
```

- The same calculation leads to different results by factor 2!

Round to Nearest

- The extra bits are used to decide (assume we have 5):
  - Extra bits > 10000: Round up (add one to the significand)
  - Extra bits < 10000: Round down (truncate the extra bits)

- What is with the special case 10000?
  - Always round up/down?
    - This introduces a small but cumulative bias over time
  - Randomly decide?
    - This would prevent the bias, but does not produce predictable, reproduceable results
  - Round to even numbers (IEEE):
    - Rounded up if LSB is 1
    - Rounded down (truncate) if LSB is 0

IEEE Standard for Binary Floating-Point Arithmetic
Infinity

- Is treated as the limiting case of real arithmetic, with the infinity values given the following interpretation:

$$-\infty < (\text{every finite number}) < +\infty$$

- Any operation involving infinity yields the expected results

$$\begin{aligned} 5 + (+\infty) &= +\infty \\ 5 - (+\infty) &= -\infty \\ 5 + (-\infty) &= -\infty \\ 5 - (-\infty) &= +\infty \\ 5 \cdot (+\infty) &= +\infty \end{aligned}$$

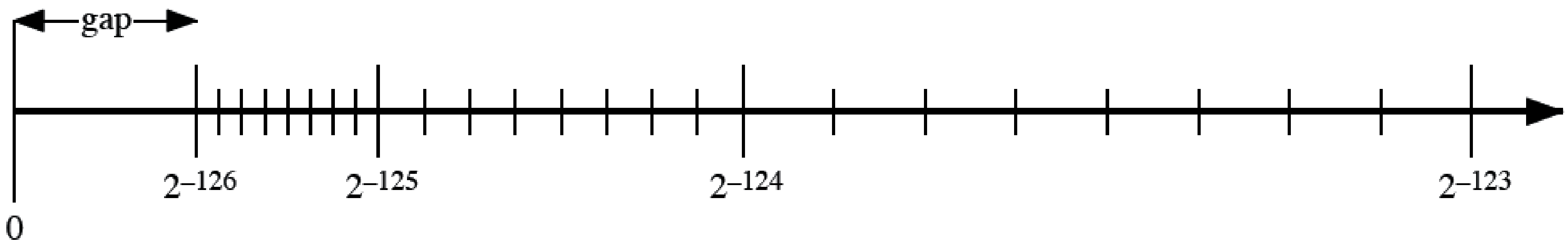
$$\begin{aligned} 5 \div (+\infty) &= +0 \\ (+\infty) + (+\infty) &= +\infty \\ (-\infty) + (-\infty) &= -\infty \\ (-\infty) - (+\infty) &= -\infty \\ (+\infty) - (-\infty) &= +\infty \end{aligned}$$

IEEE Standard for Binary Floating-Point Arithmetic
Quiet and Signaling NaNs

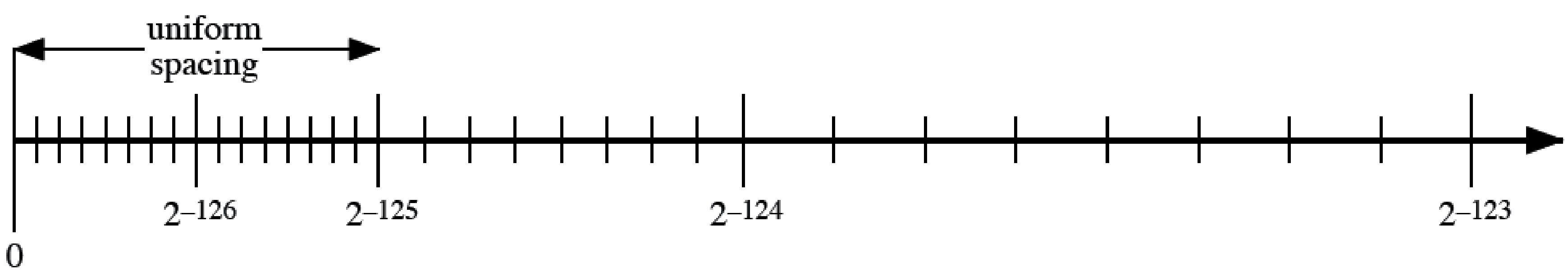
- Signaling NaN signals an invalid operation exception whenever it appears as an operand
- Quiet NaN propagates through almost every arithmetic operation without signaling an exception
- Division by zero (± 0) produces infinity ($\pm\infty$)

Operation	Quiet NaN Produced by
Any	Any operation on a signaling NaN
Add or subtract	Magnitude subtraction of infinities: $(+\infty) + (-\infty)$ $(-\infty) + (+\infty)$ $(+\infty) - (+\infty)$ $(-\infty) - (-\infty)$
Multiply	$0 \cdot \infty$
Division	$^0/_0$ or $^\infty/_\infty$
Remainder	x REM 0 or ∞ REM y
Square root	$\sqrt{x}$ with $x < 0$

IEEE Standard for Binary Floating-Point Arithmetic
Subnormal Numbers



(a) 32-bit format without subnormal numbers



(b) 32-bit format with subnormal numbers