Computer Arithmetic

Motivation

How can a computer perform arithmetic operations? Any idea?

How can a computer perform arithmetic operations? Any ideas?

- Well, it depends on the type of numbers: integer and floating point;
- Representation of number system is a crucial design issue.

What is the computer component responsible for calculations? Any ideas?

Arithmetic and Logic Unit

What is the computer component responsible for calculations? Any ideas?

Arithmetic Logic Unit

- Component that performs arithmetic and logical operations;
- All other system components are there mainly to:
  - Bring data into the ALU;
  - Process data;
  - Take results back out;

What is the general organization of the ALU? Any idea?

What is the general organization of the ALU? Any idea?

Very generic:

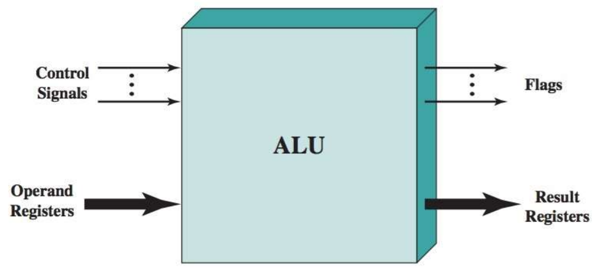


Figure: ALU Inputs and outputs (Source: [Stallings, 2015])

- Operands for arithmetic/logic operations are provided in registers;
- Results of an operation are also stored in registers;
- ALU may also set flags as the result of an operation, *e.g.:*
  - Overflow flag is set to 1:
    - If a result exceeds the length of the register into which it is to be stored.
  - Zero flag is set to 1:
    - If a result produces value zero (JMP.Z, JMP.NZ, etc...)

- Flags are also stored in Flag registers within the processor.
- Processor provides signals that control:
  - ALU operation;
  - Data movement into and out of the ALU.

Arithmetic and Logic Unit

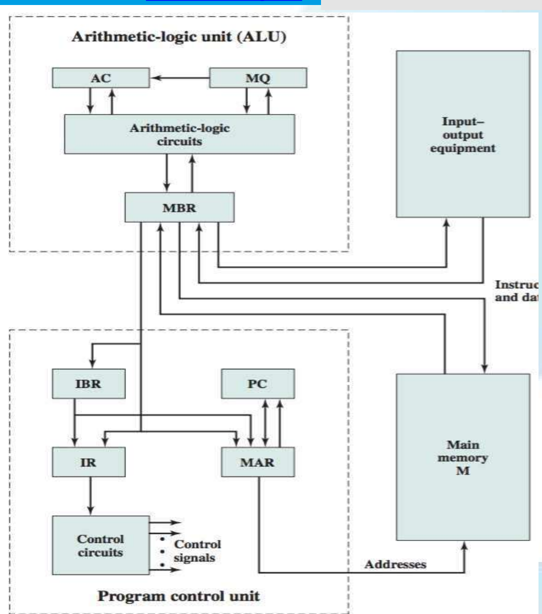


Figure: Expanded Structure of a computer (Source: [Stallings, 2015])

- **Accumulator (AC)** and **Multiplier quotient (MQ):**
  - Employed to hold temporarily operands and results of ALU operations
- **Memory buffer register(MBR):**
  - Contains a word to be stored in memory or sent to the I/O unit, or is used to receive a word from memory or from the I/O unit.
- **Memory address register (MAR):**
  - Specifies the address in memory of the word to be written from or read into the MBR.
- **Instruction register (IR):**
  - Contains the instruction being executed.
- **Instruction buffer register (IBR):**
  - Employed to hold the right-hand instruction from a word in memory.
- **Program counter (PC)**
  - Contains the address of the next instruction pair to be fetched from memory.

Integer representation

An n-bit sequence $a_{n-1}a_{n-2}\cdots a_0$ is an **unsigned** integer A:

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

But what if we have to express negative numbers. Any idea?

Sign-Magnitude Representation

The sign of a number can be represented using the **leftmost bit**:

- If bit is 0, the number is positive;
- If bit is 1, the number is negative;

$+18$	$= 00010010$	
-18	$= 10010010$	(sign magnitude)

$$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 1 \end{cases}$$

The number sign can be represented using the leftmost bit:

- If bit is 0: number is positive;
- If bit is 1: number is negative;

$+18$	$= 00010010$	
-18	$= 10010010$	(sign magnitude)

$$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 1 \end{cases}$$

Can you see any problems with this method?

There are several problems in fact:

- Addition and subtraction operations require:
  - Considering both signs and the magnitudes of each number;
- There are two representations of 0:

$$+0_{10} = 00000000$$
$$-0_{10} = 10000000 \quad \text{(sign magnitude)}$$

  - We need to test for two cases representing zero;
  - This operation is frequently used in computers...
- Because of these drawbacks sign-magnitude representation is rarely used.

So what can we use to represent integers? Any idea?

So what can we use to represent integers? Any idea?

- Two's Complement Representation

Two's Complement Representation

Like the sign magnitude system:

- uses the most significant bit as a signbit;
- Easy to test whether an integer is positive or negative

However, it differs from the use of the sign-magnitude representation:

- in the way that the other bits are interpreted.

Key characteristics of two's complement representation and arithmetic:

Range	-2^{n-1} through $2^{n-1} - 1$
Number of Representations of Zero	One
Negation	Take the Boolean complement of each bit of the corresponding positive number, then add 1 to the resulting bit pattern viewed as an unsigned integer.
Expansion of Bit Length	Add additional bit positions to the left and fill in with the value of the original sign bit.
Overflow Rule	If two numbers with the same sign (both positive or both negative) are added, then overflow occurs if and only if the result has the opposite sign.
Subtraction Rule	To subtract B from A, take the twos complement of B and add it to A.

Figure: Characteristics of twos complement representation and arithmetic (Source: [Stallings, 2015])

Consider an n-bit integer, A, in two's complement (1/3):

- If A is positive, then the sign bit, a_{n-1}, is zero;
- Remaining bits represent number magnitude:

$$A = \sum_{i=0}^{n-2} 2^i a_i \text{ , for } A \geq 0$$

Consider an n-bit integer, A, in twos complement:

What is the maximum positive integer? An idea?

Consider an n-bit integer, A, in two's complement:

What is the maximum positive integer? An idea?

- Zero is identified as positive (zero sign bit) and a magnitude of all 0s;
- Range of positive integers that may be represented is:
  - from 0 (all of the magnitude bits are 0)...
  - through $2^{n-1} - 1$ (all of the magnitude bits are 1).

Consider an n-bit integer, A, in twos complement:

What is the minimum positive integer? An idea?

Consider an n-bit integer, A, in twos complement (3/3):

What is the minimum positive integer? An idea?

- A negative number A has the sign bit, a_{n-1}, set to one:
  - Remaining $n - 1$ bits can take on any one of 2^{n-1} values
  - Therefore, range is from -1 to -2^{n-1};

Ideally: negative numbers should facilitate arithmetic operations:

- Similar to unsigned integer arithmetic;
- In unsigned integer representation:
  - Weight of the most significant bit is $+2^{n-1}$;
- It turns out that with a sign bit desired arithmetic properties are achieved if:
  - Weight of the most significant bit is -2^{n-1};

This is the convention used in twos complement representation:

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

- For $a_{n-1} = 0$, then $-2^{n-1}a_{n-1} = 0$, *i.e.:*
  - Equation defines nonnegative integer;
- For $a_{n-1} = 1$, then the term -2^{n-1} is subtracted from the summation, *i.e.:*
  - yielding a negative integer

Decimal Representation	Sign-Magnitude Representation	Twos Complement Representation
+8	—	—
+7	0111	0111
+6	0110	0110
+5	0101	0101
+4	0100	0100
+3	0011	0011
+2	0010	0010
+1	0001	0001
+0	0000	0000
−0	1000	—
−1	1001	1111
−2	1010	1110
−3	1011	1101
−4	1100	1100
−5	1101	1011
−6	1110	1010
−7	1111	1001
−8	—	1000

Figure: Alternative representations for 4 bit integers (Source: [Stallings, 2015])

Two's complement is a weird representation from the human perspective:

- However:
  - Facilitates addition and subtraction operations;
- For this reason:
  - It is almost universally used as the processor representation for integers;

Example

A useful illustration of twos complement:

−128	64	32	16	8	4	2	1

Figure: An eight-position twos complement value box (Source: [Stallings, 2015])

Example

A useful illustration of twos complement:

−128	64	32	16	8	4	2	1	
1	0	0	0	0	0	1	1	
−128						+2	+1	= −125

Figure: Convert binary 10000011 to decimal (Source: [Stallings, 2015])

Example

A useful illustration of twos complement:

−128	64	32	16	8	4	2	1
1	0	0	0	1	0	0	0

$-120 = -128 \quad +8$

Figure: Convert decimal -120 to binary (Source: [Stallings, 2015])

Range Extension

Sometimes it is desirable to take an *n*-bit integer and store it in *m*-bits:

- where $m > n$
- Easy in **sign-magnitude notation:**
  - move the sign bit to the new leftmost position and fill in with zeros.

+18	=	00010010	(sign magnitude, 8 bits)
+18	=	0000000000010010	(sign magnitude, 16 bits)
−18	=	10010010	(sign magnitude, 8 bits)
−18	=	1000000000010010	(sign magnitude, 16 bits)

But how will range extension work from two's complement perspective? Any idea?

Integer representation Range Extension

But how will range extension work from two's complement perspective? Any idea?

Can you see any potental problems? Any idea?

Range Extension

Sometimes it is desirable to take an n-bit integer and store it in m-bits:

- where $m > n$
- **Same procedure will not work for twos complement:**

$+18$	$=$	00010010	(twos complement, 8 bits)
$+18$	$=$	0000000000010010	(twos complement, 16 bits)
-18	$=$	11101110	(twos complement, 8 bits)
$-32{,}658$	$=$	1000000001101110	(twos complement, 16 bits)

What can we do to solve this problem? Any idea?

Rule for twos complement integers is:

- Move sign bit to leftmost position and fill in with copies of the sign bit, *i.e.:*
  - For positive numbers:
    - Fill in with zeros
  - For negative numbers:
    - Fill in with ones
- Example:

-18	$=$	11101110	(twos complement, 8 bits)
-18	$=$	1111111111101110	(twos complement, 16 bits)

Negation

How can we negate a number in twos complement? Any idea?

How can we negate a number in twos complement? Any ideas?

To negate an integer in two's complement notation:

1. Complement each bit (including sign bit);
2. Add 1;
3. Example:

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

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

What is the negation of zero in twos complement? Any idea?

What is the negation of zero in twos complement? Any idea?

- Consider A = 0. In that case, for an 8-bit representation:

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

- Carry bit out of the most significant bit is ignored;
- The result is that the negation of 0 is 0;

What is the negation of 1 followed by $n-1$ zeros? Any idea?

Integer Arithmetic Negation

What is the negation of 1 followed by $n-1$ zeros? Any idea?

- Negation of the bit pattern of 1 followed by $n-1$ zeros:

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

  - Produces the same number

Why does this happen? Any idea?

What is the negation of 1 followed by $n - 1$ zeros? Any idea?

- Negation of the bit pattern of 1 followed by $n - 1$ zeros:

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

  - Produces the same number

Why does this happen? Any idea?

- Twos complement range: $[-2^{n-1}, 2^{n-1} - 1]$;
- Negating -128 falls out of this range;

Addition

How can we add a number in twos complement? Any idea?

Addition

Addition proceeds as if the two numbers were unsigned integers:

$$\begin{array}{rcr} 1001 & = & -7 \\ +\underline{0101} & = & 5 \\ 1110 & = & -2 \end{array}$$

(a) $(-7) + (+5)$

Addition

Addition proceeds as if the two numbers were unsigned integers:

$$\begin{array}{rcr} 1100 & = & -4 \\ +\underline{0100} & = & 4 \\ \boxed{1}0000 & = & 0 \end{array}$$

(b) $(-4) + (+4)$

- Carry bit beyond (indicated by shading) is ignored;

Addition

Addition proceeds as if the two numbers were unsigned integers:

$$\begin{array}{rcr} 0011 & = & 3 \\ +\underline{0100} & = & 4 \\ 0111 & = & 7 \end{array}$$

(c) (+3) + (+4)

Integer Arithmetic Addition

Addition

Addition proceeds as if the two numbers were unsigned integers:

$$\begin{array}{rcr} 1100 & = & -4 \\ +\underline{1111} & = & -1 \\ \boxed{1}1011 & = & -5 \end{array}$$

(d) (−4) + (−1)

- Carry bit beyond (indicated by shading) is ignored;

Addition

But what happens with this case? Any idea?

$$
\begin{array}{r}
1001 \\
+\underline{1010} \\
\boxed{1}0011
\end{array}
$$

Integer Arithmetic | Addition

Addition

But what happens with this case? Any idea?

$$
\begin{array}{rcl}
1001 & = & -7 \\
+\underline{1010} & = & -6 \\
\boxed{1}0011 & = & \text{Overflow}
\end{array}
$$

$$(f)\,(-7) + (-6)$$

- Two numbers of the same sign produce a different sign...

Addition

Why do you think this happens? Any idea?

Integer Arithmetic Addition

Addition

Why do you think this happens? Any idea?

- **Overflow:**
  - Result is larger that what can be stored with the word;
    - Solution: Increase word size;
  - When overflow occurs:
    - ALU must signal this fact so that no attempt is made to use the result.

Subtraction

How can we subtract a number in twos complement? Any idea?

Subtraction

To subtract the **subtrahend** from the **minuend**:

- Take the twos complement of the subtrahend **(S)** and add it to the minuend **(M)**.
  - $M + (-S)$
- *I.e.:* subtraction is achieved using addition;

```
        0010 =  2
       +1001 = -7
        1011 = -5

(a) M = 2 = 0010
    S = 7 = 0111
   -S =     1001
```

Subtraction is achieved using addition: $M + (-S)$

```
     0101 =  5
    +1110 = –2
    10011 =  3

(b) M = 5 = 0101
    S = 2 = 0010
   –S =     1110
```

- Carry bit beyond (indicated by shading) is ignored;

Subtraction is achieved using addition: $M + (-S)$

```
     1011 = –5
    +1110 = –2
    11001 = –7

(c) M = –5 = 1011
    S =  2 = 0010
   –S =      1110
```

- Carry bit beyond (indicated by shading) is ignored;

Subtraction is achieved using addition: $M + (-S)$

```
     0101 = 5
    +0010 = 2
     0111 = 7

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

Integer Arithmetic Subtraction

Subtraction is achieved using addition: $M + (-S)$

```
     0111 = 7
    +0111 = 7
     1110 = Overflow

(e) M =  7 = 0111
    S = -7 = 1001
   -S =      0111
```

- **Overflow:** two numbers of the same sign produce a different sign;

Subtraction is achieved using addition: $M + (-S)$

```
     1010 = -6
    +1100 = -4
    10110 = Overflow

(f) M = -6 = 1010
    S =  4 = 0100
   -S =      1100
```

- Carry bit beyond (indicated by shading) is ignored;
- **Overflow:** two numbers of the same sign produce a different sign;

Hardware Block Diagram for Adder

So, the question now is:

How can we map these concepts into hardware?

Hardware Block Diagram for Adder

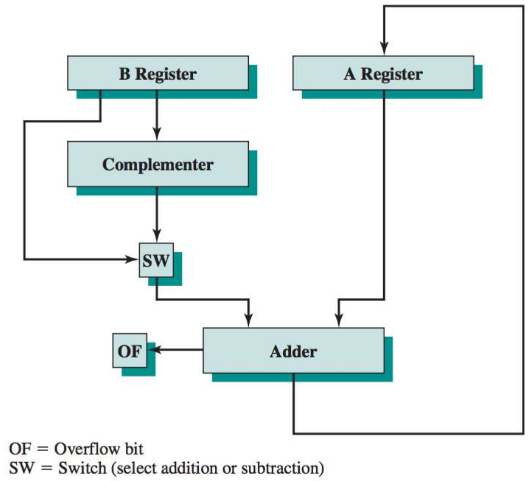


Figure: Block diagram of hardware for addition and subtraction (Source: [Stallings, 2015])

From the previous figure:

- Central element is a binary adder:
  - Presented with two inputs;
  - Produces a sum and an overflow indication;
- Adder treats the two numbers as unsigned integers
- For the **addition** operation:
  - The two numbers are presented in two registers;
  - Result may be stored in one of these registers or in a third;
- For the **subtraction** operation:
  - Subtrahend (B register) is passed through a twos complementer;
- Overflow indication is stored in a 1-bit overflow flag.

Multiplication

Multiplication is a complex operation:

- Compared with addition and subtraction;
- Again lets consider multiplying for the following cases:
  - Two unsigned numbers;
  - Two signed (twos complement) numbers;

Integer Arithmetic Multiplication

Unsigned Integers

How can we perform multiplication? Any idea?

$$\begin{array}{r} 1011 \\ \times 1101 \\ \hline \end{array}$$

How can we perform multiplication? Any idea?

```
   1011     Multiplicand (11)
  ×1101     Multiplier (13)
   1011  }
  0000   }
 1011    }  Partial products
1011     }
10001111    Product (143)
```


Several important observations:

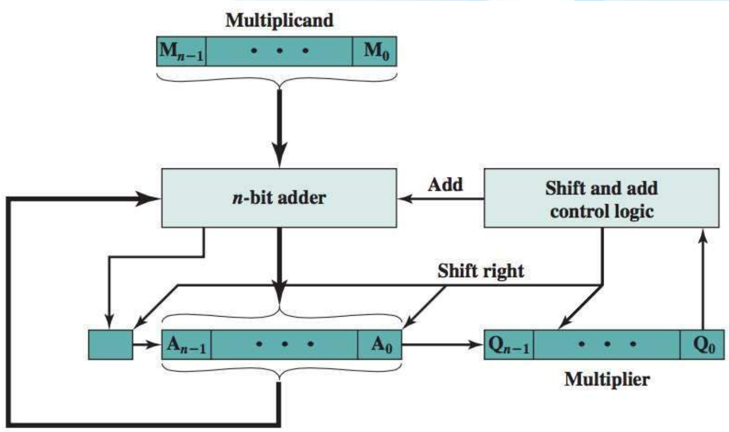
1. Multiplication involves the generation of partial products:
   - One for each digit in the multiplier;
   - These partial products are then summed to produce the final product.
2. The partial products are easily defined.
   - When the multiplier bit is 0, the partial product is 0;
   - When the multiplier is 1, the partial product is the multiplicand;
3. Total product is produced by summing the partial products:
   - each successive partial product is shifted one position to the left relative
4. Multiplication of two n-bit binary integers produces up to $2n$ bits;

How can we translate these concepts into hardware? Any idea?

How can we translate these concepts into hardware? Any idea?

- We can perform a **running addition** on the partial products:
  - Rather than waiting until the end;
  - Eliminates the need for storage of all the partial products;
    - Fewer registers are needed
- We can save some time on the generation of partial products:
  - For each 1 on the multiplier, an add and a shift operation are required
  - For each 0, only a shift is required.

Possible implementation employing these measures:



Integer Arithmetic Multiplication

1. Multiplier and multiplicand are loaded into two registers (**Q** and **M**);
2. A third register, the **A** register, is also needed and is initially set to 0;
3. There is also a 1-bit **C** register, initialized to 0:
   - which holds a potential carry bit resulting from addition.

4 Control logic then reads the bits of the multiplier one at a time:

- If Q_0 is 1, then:
  - the multiplicand is added to the A register...
  - and the result is stored in the A register...
  - with the C bit used for overflow.
  - Then all of the bits of the C, A, and Q registers are shifted to the right one bit:
  - So that the C bit goes into A_{n-1} , A_0 goes into Q_{n-1} and Q_0 is lost.
- If Q_0 is 0, then:
  - Then no addition is performed, just the shift;
  - Process is repeated for each bit of the original multiplier;
  - Resulting 2n-bit product is contained in the A and Q registers

Integer Arithmetic Multiplication

Example

```
    1011      Multiplicand (11)
   ×1101      Multiplier (13)
    1011  )
   0000   |
  1011    }   Partial products
 1011     )
10001111      Product (143)
```

C	A	Q	M

Example

```
    1011      Multiplicand (11)
   ×1101      Multiplier (13)
    1011
   0000
  1011        Partial products
 1011
10001111      Product (143)
```

C	A	Q	M	
0	0000	1101	1011	Initial Values

Example

```
    1011      Multiplicand (11)
   ×1101      Multiplier (13)
    1011
   0000
  1011        Partial products
 1011
10001111      Product (143)
```

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

Example

```
    1011     Multiplicand (11)
   ×1101     Multiplier (13)
    1011
   0000
  1011       Partial products
 1011
10001111     Product (143)
```

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

Integer Arithmetic | Multiplication

Example


```
    1011     Multiplicand (11)
   ×1101     Multiplier (13)
    1011
   0000
  1011       Partial products
 1011
10001111     Product (143)
```


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

Example

```
   1011     Multiplicand (11)
  ×1101     Multiplier (13)
   1011 ┐
  0000  │
 1011   ├  Partial products
1011    ┘
10001111    Product (143)
```

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

Integer Arithmetic | Multiplication

Example

```
   1011     Multiplicand (11)
  ×1101     Multiplier (13)
   1011 ┐
  0000  │
 1011   ├  Partial products
1011    ┘
10001111    Product (143)
```

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

Example

```
    1011      Multiplicand (11)
   ×1101      Multiplier (13)
    1011  ⎫
   0000   ⎬   Partial products
  1011    ⎪
 1011     ⎭
10001111      Product (143)
```

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

Example


```
    1011      Multiplicand (11)
   ×1101      Multiplier (13)
    1011  ⎫
   0000   ⎬   Partial products
  1011    ⎪
 1011     ⎭
10001111      Product (143)
```


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

Or in flowchart form:

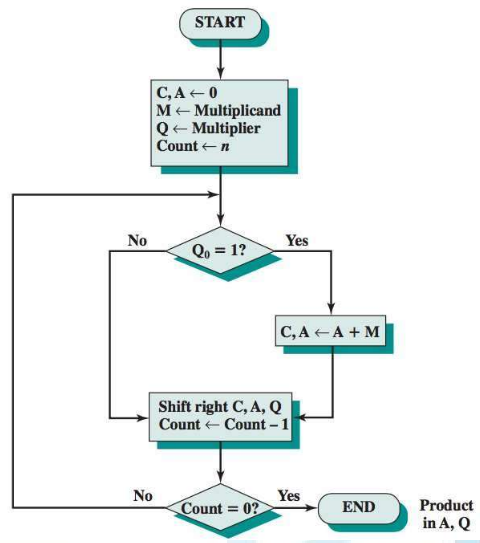


Figure: Flowchart for unsigned binary multiplication (Source: [Stallings, 2015])

Twos complement multiplication

How can we perform multiplication using twos complement? Any idea?

Product of two N-bit numbers requires maximum of $2N$:

- Double operands precision using two's complement;
- For example, take $6 \times -5 = -30$
  - $6_{(10)} = 0110_{(2)} = 00000110_{(2)}$
  - $-5_{(10)} = 1011_{(2)} = 11111011_{(2)}$

```
     00000110   (6)
*    11111011   (-5)
============
          110
         1100
        00000
       110000
      1100000
     11000000
    x10000000
+  xx00000000
============
   xx11100010
```

Figure: (Source: wikipedia)

- Discarding the bits beyond the eighth bit will produce the correct result;

Can you see any problems with this approach? Any idea?

Very inefficient:

- Precision is doubled ahead of time;
- This means that all additions must be double-precision;
- Therefore twice as many partial products are needed...

Can we do better than this?

Can we do better than this?

- Yes we can through Booth's algorithm =)

Booth's Algorithm

Example of Booth's Algorithm for: 7×3

A	Q	Q_{-1}	M

Booth's Algorithm

Example of Booth's Algorithm for: 7 × 3

A	Q	Q_{-1}	M	
0000	0011	0	0111	Initial Values

Integer Arithmetic Multiplication

Booth's Algorithm

Example of Booth's Algorithm for: 7 × 3

A	Q	Q_{-1}	M	
0000	0011	0	0111	Initial Values
1001	0011	0	0111	$A \leftarrow A - M$

Booth's Algorithm

Example of Booth's Algorithm for: 7×3

A	Q	Q_{-1}	M	
0000	0011	0	0111	Initial Values
1001	0011	0	0111	$A \leftarrow A - M$
1100	1001	1	0111	Arithmetic Shift Right

Booth's Algorithm

Example of Booth's Algorithm for: 7×3

A	Q	Q_{-1}	M	
0000	0011	0	0111	Initial Values
1001	0011	0	0111	$A \leftarrow A - M$
1100	1001	1	0111	Arithmetic Shift Right
1110	0100	1	0111	Arithmetic Shift Right

Booth's Algorithm

Example of Booth's Algorithm for: 7 × 3

A	Q	Q_{-1}	M	
0000	0011	0	0111	Initial Values
1001	0011	0	0111	$A \leftarrow A - M$
1100	1001	1	0111	Arithmetic Shift Right
1110	0100	1	0111	Arithmetic Shift Right
0101	0100	1	0111	$A \leftarrow A + M$

Integer Arithmetic Multiplication

Booth's Algorithm

Example of Booth's Algorithm for: 7 × 3

A	Q	Q_{-1}	M	
0000	0011	0	0111	Initial Values
1001	0011	0	0111	$A \leftarrow A - M$
1100	1001	1	0111	Arithmetic Shift Right
1110	0100	1	0111	Arithmetic Shift Right
0101	0100	1	0111	$A \leftarrow A + M$
0010	1010	0	0111	Arithmetic Shift Right

Booth's Algorithm

Example of Booth's Algorithm for: 7 × 3

A	Q	Q_{-1}	M	
0000	0011	0	0111	Initial Values
1001	0011	0	0111	$A \leftarrow A - M$
1100	1001	1	0111	Arithmetic Shift Right
1110	0100	1	0111	Arithmetic Shift Right
0101	0100	1	0111	$A \leftarrow A + M$
0010	1010	0	0111	Arithmetic Shift Right
0001	0101	0	0111	Arithmetic Shift Right

Integer Arithmetic Multiplication

Booth's Algorithm

Example of Booth's Algorithm for: 7 × 3

A	Q	Q_{-1}	M	
0000	0011	0	0111	Initial Values
1001	0011	0	0111	$A \leftarrow A - M$
1100	1001	1	0111	Arithmetic Shift Right
1110	0100	1	0111	Arithmetic Shift Right
0101	0100	1	0111	$A \leftarrow A + M$
0010	1010	0	0111	Arithmetic Shift Right
0001	0101	0	0111	Arithmetic Shift Right

Final result appears in the A and Q registers:

- $A = 0001$
- $Q = 0101$
- $AQ = 00010101_2 = 21_{10}$

1. Multiplier and multiplicand are placed in the Q and M registers;
2. Q_{-1} is a 1-bit register placed logically to the right of Q_0;
3. Results of the multiplication will appear in the A and Q registers;
4. A and Q_{-1} are initialized to 0.

5. Control logic scans the bits of the multiplier one at a time:
   - 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):
     - 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/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.
   - All shifts performed preserve the sign:
     - Arithmetic Right Shift

In flowchart form:

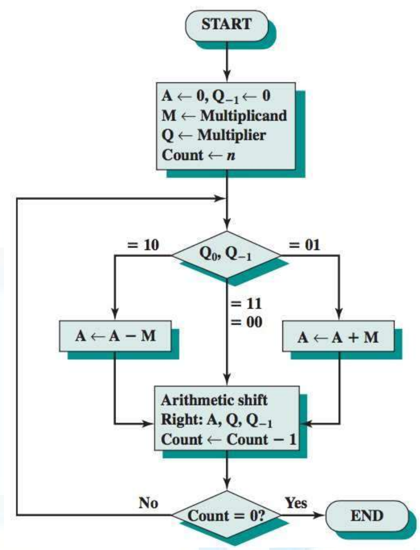


Figure: (Source: [Stallings, 2015])

Exercise

Use Booth's algorithm to multiply 23 (M) by 29 (Q):

A	Q	Q_{-1}	M

Rules:

1 ⟶ 1 = A = A + M, Arithmetic Right Shift

2⟶ 0 = A = A + (-M), Arithmetic Right Shift

0 ⟶ 0 = Arithmetic Right Shift

1 ⟶ 1 = Arithmetic Right Shift

Exercise

Use Booth's algorithm to multiply 22 (M) by 28 (Q):

A	Q	Q_{-1}	M

Rules:

1 ⟶ 1 = A = A + M, Arithmetic Right Shift
2⟶ 0 = A = A + (-M), Arithmetic Right Shift
0 ⟶ 0 = Arithmetic Right Shift
1 ⟶ 1 = Arithmetic Right Shift

Seems complicated?

- That is because it is;
- Focus on performance not on human easiness to understand...
- Can be proved mathematically that works.
- Same concepts could be used to divise a division method.

Now that we have a basic understanding about:

- integer representation;
- integer arithmetic operations;

How can we represent floating-point numbers?

Floating-point representation

We can represent decimal numbers in several ways, *e.g.:*

- $976,000,000,000,000 = 9.76 \times 10^{14}$
- $0.0000000000000976 = 9.76 \times 10^{-14}$

I.e. the decimal point floats to a convenient location:

- We use the exponent of 10 to keep track of the decimal point;
- This way we can represent large and small numbers with only a few digits;

Same approach can be taken with binary numbers, *i.e.:*

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

This number can be stored in a binary word with three fields:

- Sign: plus or minus
- Significand: S
- Exponent: E
- Base: B (binary)

In general:

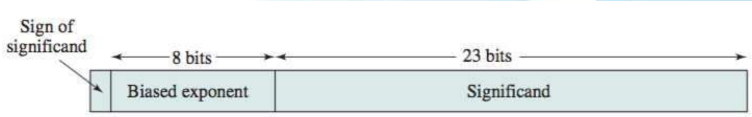


Figure: Typical 32-Bit floating-point format (Source: [Stallings, 2015])

- Leftmost bit stores the sign of the number (0 = positive, 1 = negative)
- Exponent value is stored in the next $k = 8$ bits:
  - Fixed value is subtracted from the field to get the true value.
  - Value equals $2^{k-1} - 1$
  - Range of possible values $[-(2^{k-1} - 1), 2^{k-1}]$
- Significand: right portion of the word;

Example

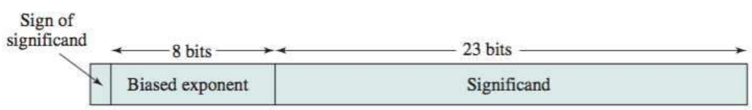


Figure: Typical 32-Bit floating-point format (Source: [Stallings, 2015])

- Example:

  $1.1010001 \times 2^{10100} = 0\ 10010011\ \ 10100010000000000000000$

Biased exponent:

$20_{(10)}$:=	0	0	0	1	0	1	0	0
$+\ 127_{(10)}$:=	0	1	1	1	1	1	1	1
	1	0	0	1	0	0	1	1

What is the decimal form of this number:
0 1001 0011 10100010000000000000000? Any ideas?

What is the decimal form of this number:
0 1001 0011 10100010000000000000000? Any ideas?

- 1.6328125×2^{20}
- $2^{-1} + 2^{-3} + 2^{-7} = 1.6328125$

Example

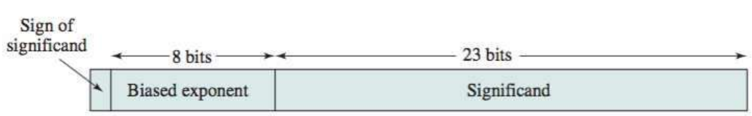


Figure: Typical 32-Bit floating-point format (Source: [Stallings, 2015])

- Example:
  $-1.1010001 \times 2^{10100} = 1\ 10010011\ 10100010000000000000000$

Biased exponent:

$20_{(10)}$:=	0	0	0	1	0	1	0	0
$+ 127_{(10)}$:=	0	1	1	1	1	1	1	1
	1	0	0	1	0	0	1	1

What is the decimal form of this number:
1 1001 0011 10100010000000000000000? Any ideas?

Floating-point representation

What is the decimal form of this number:
1 1001 0011 10100010000000000000000? Any ideas?

- -1.6328125×2^{20}
- $2^{-1} + 2^{-3} + 2^{-7} = 1.6328125$

Example

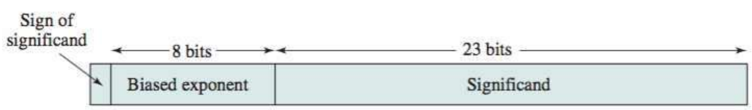


Figure: Typical 32-Bit floating-point format (Source: [Stallings, 2015])

- Example:
  $1.1010001 \times 2^{-10100} = 0\ 01101011\ 10100010000000000000000$

Biased exponent:

$$\begin{array}{rcccccccc} -20_{(10)} := & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ +\,127_{(10)} := & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \hline & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \end{array}$$

What is the decimal form of this number:
0 0110 1011 10100010000000000000000? Any ideas?

What is the decimal form of this number:
0 0110 1011 101000100000000000000000? Any ideas?

- 1.6328125×2^{-20}
- $2^{-1} + 2^{-3} + 2^{-7} = 1.6328125$

Example

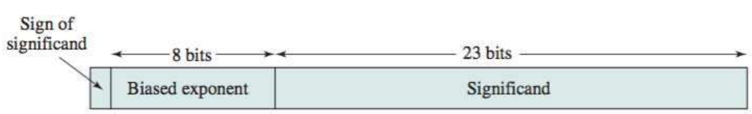


Figure: Typical 32-Bit floating-point format (Source: [Stallings, 2015])

- Example:
  $-1.1010001 \times 2^{-10100} = 1\ 11101101\ 10100010000000000000000$

Biased exponent:

$$\begin{array}{rcccccccc} -20_{(10)} := & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ +\,127_{(10)} := & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \hline & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \end{array}$$

What is the decimal form of this number:
1 11101101 10100010000000000000000? Any ideas?

What is the decimal form of this number:
1 11101101 10100010000000000000000? Any ideas?

- $-1.6328125 \times 2^{-20}$
- $2^{-1} + 2^{-3} + 2^{-7} = 1.6328125$

Figure: Typical 32-Bit floating-point format (Source: [Stallings, 2015])

- **Examples conclusion:**

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

Floating-point Arithmetic

Now that we know how to represent floating-point numbers:

How can we perform floating-point arithmetic? Any ideas?

Some observations:

- **Addition and subtraction** operations:
  - Necessary to ensure that both operands have the same exponent value;
  - May require shifting the radix point to achieve **alignment**;
- **Multiplication and division** are more straightforward.

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

Figure: Floating point numbers and arithmetic operations (Source: [Stallings, 2015])

Example

$X = 0.3 \times 10^2 = 30$

$Y = 0.2 \times 10^3 = 200$

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

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

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

$X \div Y = (0.3 \div 0.2) \times 10^{2-3} = 1.5 \times 10^{-1} = 0.15$

Floating-point Arithmetic

Can you see any type of problems that the operations might produce? Any ideas?

Can you see any type of problems that the operations might produce? Any ideas?

Floating-point operation may produce (1/2):

- **Exponent overflow:** Positive exponent exceeds maximum value;
- **Exponent underflow:** Negative exponent exceeds minimum value;
  - *E.g.:* -200 is less than -127.
  - Number is too small to be represented (reported as 0).

Can you see any type of problems that the operations might produce? Any ideas?

Floating-point operation may produce (2/2):

- **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

So now the question is

How can we perform addition/subtraction using floats? Any ideas?

Addition and Subtraction

In floating-point arithmetic:

- Addition/subtraction more complex than multiplication/division;
- This is because of the need for alignment;
- There are four basic phases of the algorithm for addition and subtraction:
  1. Check for zeros.
  2. Align the significands.
  3. Add or subtract the significands.
  4. Normalize the result.

Lets have a look at each one of these...

Phase 1: Zero check:

- Addition and subtraction are identical except for a sign change:
  - Begin by changing the sign of the subtrahend if it is a subtract operation.
- If either operand is 0, the other is reported as the result.

Phase 2: Significand alignment (1/2):

- Manipulate numbers so that the two exponents are equal, *e.g.:*
  - $(123 \times 10^{0}) + (456 \times 10^{-2})$
  - Digits must first be set into equivalent positions, *i.e.:*
    - 4 of the second number must be aligned with the 3 of the first;
    - The two exponents need to be equal
  - Thus:
    - $(123 \times 10^{0}) + (456 \times 10^{-2}) = (123 \times 10^{0}) + (4.56 \times 10^{0}) = 127.56 \times 10^{0}$

Phase 2: Significand alignment (2/2):

- Alignment may be achieved by:
  - Shifting either the smaller number to the right (increasing its exponent)
  - Shifting the larger number to the left
  - Either operation may result in the loss of digits
  - Smaller number is usually shifted:
    - Since any digits lost are of small significance.
- In general terms:
  - Repeatedly shift the significand right 1 digit
  - and increment the exponent until the two exponents are equal.

Phase 3: Addition

- Significands are added together;
- Because the signs may differ: result may be 0;
- There is also the possibility of significand overflow, if so:
  - Significand of the result is shifted right and the exponent is incremented;
  - Exponent overflow could occur as a result;
    - Operation is halted.

Phase 4: Normalization

- Shift significand digits left until most significant digit is nonzero;
- Each shift causes:
  - a decrement of the exponent and...
  - ...thus could cause an exponent underflow.

Floating-point Arithmetic | Addition and Subtraction

In flowchart form:

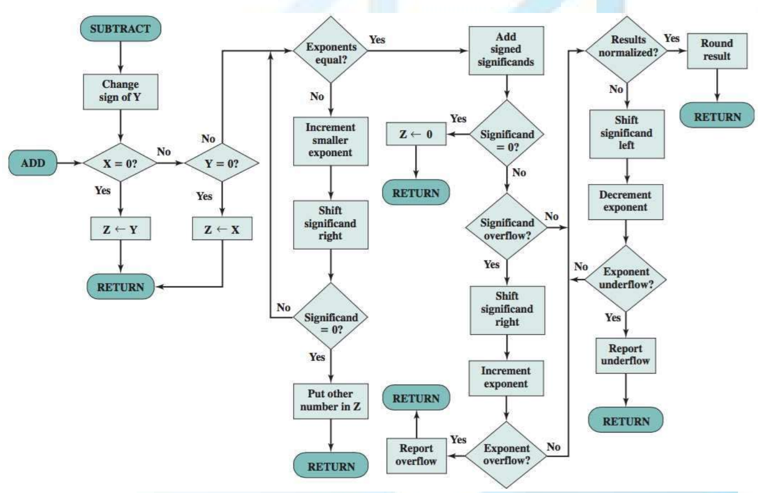


Figure: Floating Point Addition And Subtraction (Source: [Stallings, 2015])

So now the question is

How can we perform multiplication using floats? Any ideas?

Floating-point Arithmetic Floating-Point Multiplication

Floating-Point Multiplication

1. If either operand is 0: 0 is reported as the result;
2. Add the exponents;
3. Multiply the significands:
   - Similarly to twos complement multiplication.
4. Result is normalized.

In flowchart form:

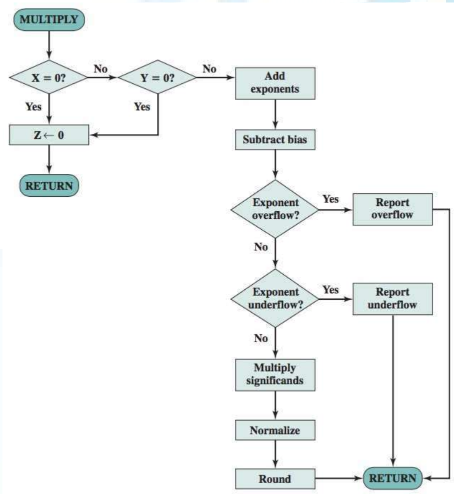


Figure: Floating Point Multiplication (Source: [Stallings, 2015])

So now the question is

How can we perform division using floats? Any ideas?

Floating-Point Division

1. Test for 0:
   - If the divisor is 0: report error;
   - Dividend is 0: results in 0.
2. Divisor exponent is subtracted from the dividend exponent;
3. Divide the significands;
4. Result is normalized;

In flowchart form:

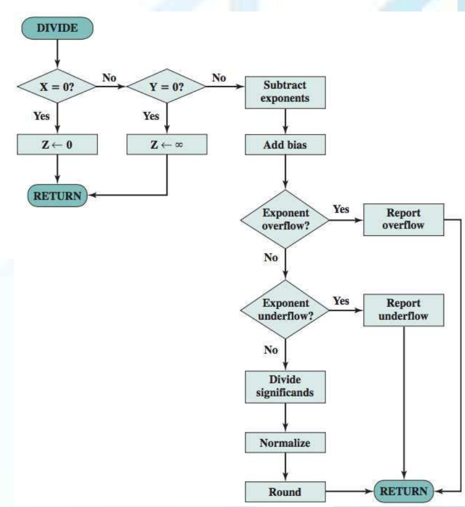


Figure: Floating Point Division (Source: [Stallings, 2015])