Integer Multiplication

Multiplication of Positive Integers

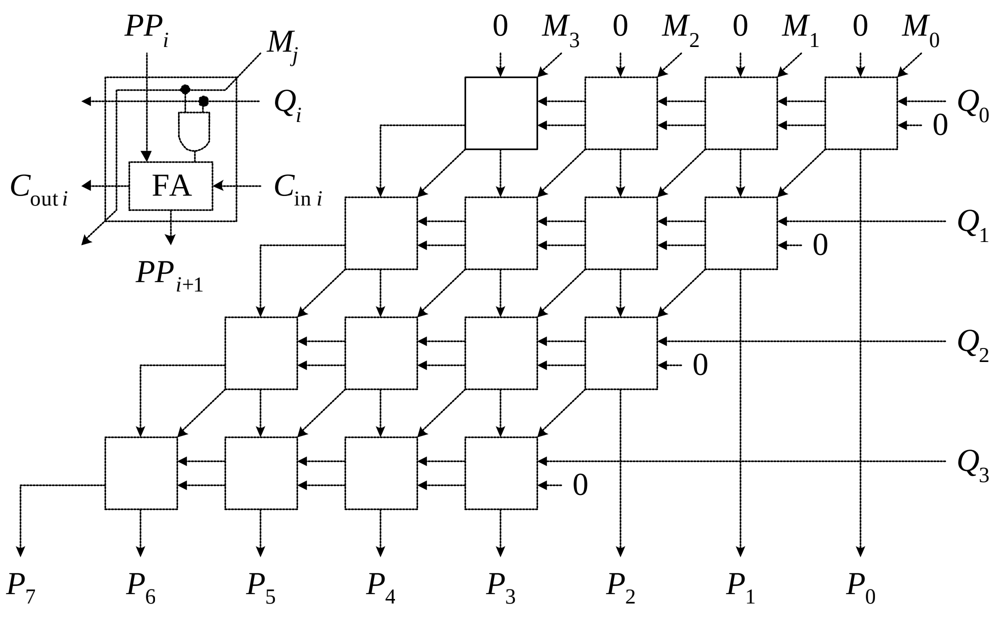
Paper-And-Pencil Method

- ❖ In a positional number system, multiplication is performed by multiplying the multiplicand M by each digit of the multiplier Q. These are then weighted and added.

$$
\begin{array}{cccccccccc}
 & & & & 1 & 1 & 0 & 1 & & M \\
 & & & \times & 1 & 0 & 1 & 1 & & Q \\
\hline
 & & & & 1 & 1 & 0 & 1 & & \\
 & & & 1 & 1 & 0 & 1 & & & \\
 & & 0 & 0 & 0 & 0 & & & & \\
 & 1 & 1 & 0 & 1 & & & & & \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & & P
\end{array}
$$

  - ➢ The length of the product p is the sum of the lengths of the multiplicand m and the multiplier q.
  - ➢ The addend exists only, and is a copy of the multiplicand, if the multiplier bit is 1.
- ❖ The integer multiplier can be implemented using full-adders and AND gates. – *Parallel Multiplier*
  - ➢ Since addition is performed using 2 operands at a time, *partial products* can be obtained while the addends are being generated.

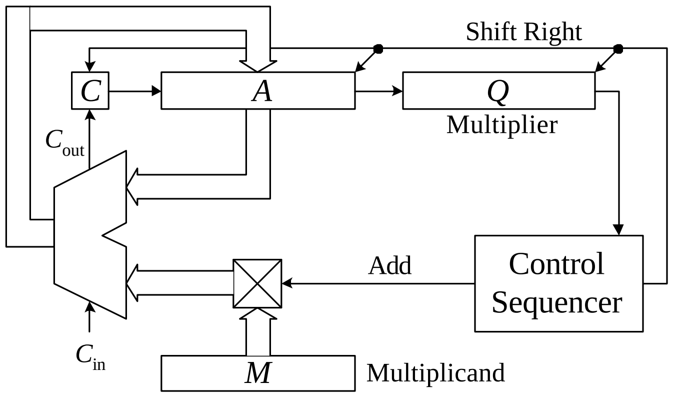
$$
\begin{array}{ccccccccccc}
 & & & & & 1 & 1 & 0 & 1 & & M \\
 & & & & \times & 1 & 0 & 1 & 1 & & Q \\
\hline
 & & & & & 0 & 0 & 0 & 0 & & PP_0 \\
 & & & & + & 1 & 1 & 0 & 1 & & \\
\hline
 & & & & & 1 & 1 & 0 & 1 & & PP_1 \\
 & & & + & 1 & 1 & 0 & 1 & & & \\
\hline
 & & & 1 & 0 & 0 & 1 & 1 & 1 & & PP_2 \\
 & & + & 0 & 0 & 0 & 0 & & & & \\
\hline
 & & & 1 & 0 & 0 & 1 & 1 & 1 & & PP_3 \\
 & + & 1 & 1 & 0 & 1 & & & & & \\
\hline
 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & & P
\end{array}
$$



- Impractical for a dedicated circuit.
  - Requires $m \times q$ blocks with 1 full-adder and 1 AND gate each.
  - Very long propagation delay.

Serial Integer Multiplier

- The circuit size may be reduced by using the existing adder of the ALU. Additionally requires
  - 2 shift registers *A* and *Q* to hold the product
  - 1 register *M* to hold the multiplicand
  - 1 binary cell *C* to hold the carry-out of the partial product
  - A control sequencer



- Process
  - Initialization
    - Clear *A*.
    - *M* gets multiplicand.
    - *Q* gets multiplier.
  - Loop for each bit of multiplier
    - If $Q_0 = 1$ then Add.
    - Shift right *CAQ*.
  - Result contained in register combination *AQ*.

Comments

- Size of *A* at least equal to *M*, not necessarily but typically equal to *Q*.
- Speed is a little slower (due to extra shifting time).
- Size is a lot smaller.

***Example:* 13´ 11**

	M	1101		
	C	*A*	*Q*	
Init	0	0000	1011	
Bit 0	0	1101		Add
		0110	1101	Shift
Bit 1	1	0011		Add
		1001	1110	Shift
Bit 2		0100	1111	Shift
Bit 3	1	0001		Add
		1000	1111	Shift

Signed Integer Multiplication

- ❖ Basically magnitude multiplication + sign adjustment
- ❖ Result is size of magnitudes + 1 for sign bit. Computers typically use just the sum of the sizes.
- ❖ Need to extend the sign bits.

***Example:* –13´ 11**

$$\begin{array}{cccccccccccc}
 & & & & & 1 & 0 & 0 & 1 & 1 & & M \\
 & & & & \times & 0 & 1 & 0 & 1 & 1 & & Q \\
\hline
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & & \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & & & & & \\
\hline
1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & & P
\end{array}$$

- ❖ If multiplier is negative, need to add negative (2's complement) of multiplicand, sign-extended and shifted to sign-bit position. Note: msb of multiplier is actually the sign bit, *i.e.*, a 1 means a negative multiplier.

***Example:* 11´ –13**

$$\begin{array}{cccccccccccc}
 & & & & & 0 & 1 & 0 & 1 & 1 & & M \\
 & & & & \times & 1 & 0 & 0 & 1 & 1 & & Q \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & & \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & \\
1 & 1 & 0 & 1 & 0 & 1 & & & & & & s.adj \\
\hline
1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & & P
\end{array}$$

***Example:* –11´ –13**

$$\begin{array}{cccccccccccc}
 & & & & & 1 & 0 & 1 & 0 & 1 & & M \\
 & & & & \times & 1 & 0 & 0 & 1 & 1 & & Q \\
\hline
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & & \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & \\
0 & 0 & 1 & 0 & 1 & 1 & & & & & & s.adj \\
\hline
0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & & P
\end{array}$$

Comments

- ➢ Different last step for negative multiplier. Leads to difficulty in circuit design.

Booth's Algorithm

- ❖ Treats positive and negative multipliers uniformly.
- ❖ Rewrites multiplier in terms of sums and differences.
  - ➢ Convert code according to next bit at right
    - ▪ 0 to 1 $\Rightarrow$ +1
    - ▪ 1 to 0 $\Rightarrow$ –1
    - ▪ Otherwise, 0
  - ➢ Right of lsb is "nothing", *i.e.*, equal to 0. ☺

$$\begin{array}{rcccccc}
13 & = & 0 & 1 & 1 & 0 & 1 \\
 & \Rightarrow & +1 & 0 & -1 & +1 & -1 \\
 & = & \multicolumn{5}{l}{+2^4 - 2^2 + 2^1 - 2^0} \\
 & & & & & & \\
-13 & = & 1 & 0 & 0 & 1 & 1 \\
 & \Rightarrow & -1 & 0 & +1 & 0 & -1 \\
 & = & \multicolumn{5}{l}{-2^4 + 2^2 - 2^0}
\end{array}$$

❖ Multiplication is performed via add (if +1) and subtract (if –1).

***Example:* 11´ 13**

$$\begin{array}{cccccccccc|c}
 & & & & & 0 & 1 & 0 & 1 & 1 & M \\
 & & & & & 1 & 0 & 1 & 0 & 1 & -M \\
 & & & & \times & +1 & 0 & -1 & +1 & -1 & Q \\
\hline
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & \text{Sub} \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & & \text{Add} \\
1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & & & \text{Sub} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & & & & & \text{Add} \\
\hline
0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & P
\end{array}$$

***Example:* 11´ –13**

$$\begin{array}{cccccccccc|c}
 & & & & & 0 & 1 & 0 & 1 & 1 & M \\
 & & & & & 1 & 0 & 1 & 0 & 1 & -M \\
 & & & & \times & -1 & 0 & +1 & 0 & -1 & Q \\
\hline
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & \text{Sub} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & & & \text{Add} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & & & & & \text{Sub} \\
\hline
1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & P
\end{array}$$

Fast Integer Multiplication

❖ Bit-Pair Recording – reduces to half the number of summands

❖ Carry-Save Addition – reduces time of addition in parallel multipliers

Bit-Pair Recording

❖ The number of summands is reduced by pairing multiplier bits and ensuring left bit is 0 (*Extended Booth Representation*), *i.e.*,

$$\begin{array}{cccccc}
0 & 0 & \Rightarrow & 0 & 0 \\
0 & +1 & \Rightarrow & 0 & +1 \\
0 & -1 & \Rightarrow & 0 & -1 \\
+1 & 0 & \Rightarrow & 0 & +2 \\
+1 & -1 & \Rightarrow & 0 & +1 \\
-1 & 0 & \Rightarrow & 0 & -2 \\
-1 & +1 & \Rightarrow & 0 & -1
\end{array}$$

- ➢ Multiplication is either by:
  - 1 (copy) or 2 (copy shift)
  - positive (*M*) or negative (–*M*)

***Example:* 11´9**

$$\begin{array}{rcrrrrr}
9 & = & 0 & 1 & 0 & 0 & 1 \\
 & & +1 & -1 & 0 & +1 & -1 \\
 & & +1 & & -2 & & +1
\end{array}$$

$$\begin{array}{cccccccccccl}
 & & & & & & 0 & 1 & 0 & 1 & 1 & M \\
 & & & & & & 1 & 0 & 1 & 0 & 1 & -M \\
 & & & & & \times & +1 & & -2 & & +1 & Q \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & & PP_0 \\
1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & & & & PP_1 \\
0 & 0 & 1 & 0 & 1 & 1 & & & & & & PP_2 \\
\hline
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & & P
\end{array}$$

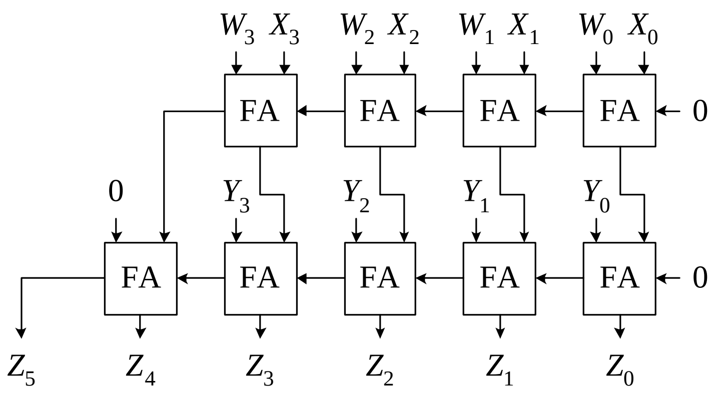
***Example:* 11´–9**

$$\begin{array}{rcrrrrr}
-9 & = & 1 & 0 & 1 & 1 & 1 \\
 & & -1 & +1 & 0 & 0 & -1 \\
 & & -1 & & +2 & & -1
\end{array}$$

$$\begin{array}{cccccccccccl}
 & & & & & & 0 & 1 & 0 & 1 & 1 & M \\
 & & & & & & 1 & 0 & 1 & 0 & 1 & -M \\
 & & & & & \times & -1 & & +2 & & -1 & Q \\
\hline
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & & PP_0 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & & & & PP_1 \\
1 & 1 & 0 & 1 & 0 & 1 & & & & & & PP_2 \\
\hline
1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & & P
\end{array}$$

Carry-Save Addition

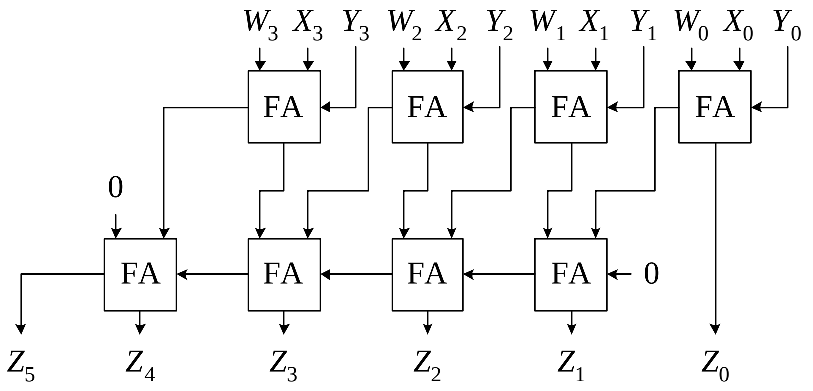
- The parallel multiplier can be improved by improving the adder portion.
- Consider the addition of 3 numbers: $W + X + Y = Z$
- Using standard ripple-carry adders



W_3 X_3 W_2 X_2 W_1 X_1 W_0 X_0
FA
0
Y_3 Y_2 Y_1 Y_0
Z_5 Z_4 Z_3 Z_2 Z_1 Z_0

  - A carry-lookahead circuit can improve the upper level. The lower level cannot benefit from a carry-lookahead circuit due to the large diversity in the delays.

- This process can be improved by using *carry-save addition*.



W_3 X_3 Y_3 W_2 X_2 Y_2 W_1 X_1 Y_1 W_0 X_0 Y_0
FA
0
Z_5 Z_4 Z_3 Z_2 Z_1 Z_0

  - The carry-in inputs are used for the third input and carry-outs are saved for the next level.
  - The inputs to both levels arrive at close intervals, hence, the second level may be improved with the use of a carry-lookahead circuit.
- Considering these, a parallel multiplier may be implemented using a tree structure by grouping summands by threes.

Integer Division

- Unlike subtraction, division cannot easily be performed using multiplication since it is difficult to take the reciprocal of a number.
- Division is performed by repeated subtraction.
  - Loop for each digit of dividend from highest
    - Determine largest multiplier for divisor
    - Subtract
    - Bring down next digit

Comments

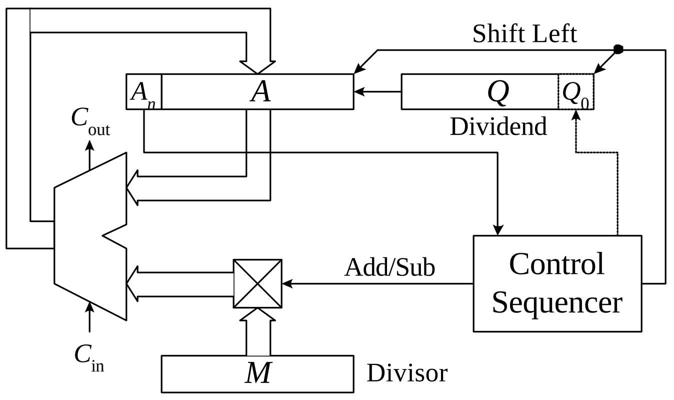
- The size of the quotient is the difference of the sizes (counted from leftmost 1) of the dividend and the divisor.
- There is no simple algorithm to implement signed division. Typically,
  - Both dividend and divisor are converted to positive,
  - Unsigned division is performed, and
  - Sign is adjusted accordingly.
- Another problem is on how to treat the remainder of a signed division, *i.e.*, whether or not the remainder should be negative if the quotient is negative.

Restoring (Unsigned) Division

- In *restoring division*, the multiplier to the divisor is determined by first subtracting the divisor. If the result is negative, the divisor is restored (added back).
  - Initialization
    - Clear A. Requires 1 extra bit for A to be used as a sign bit.
    - Q gets dividend.
    - M gets divisor.
  - Loop for each bit of the dividend Q
    - Shift AQ to the left.
    - Subtract ($A \leftarrow A - M$)

- If negative ($A_n = 1$), restore ($A \leftarrow A + M$) and reset Q_0 ($\leftarrow 0$)
- Else set Q_0 ($\leftarrow 1$)

➢ Quotient in Q while remainder in A.



***Example:* 11 ¸ 3**

$-M$	11101		
M	00011		
	A	Q	
Init	00000	1011	
Bit 0	00001	011–	Shift
	11110		Sub
	00001	0110	Restore
Bit 1	00010	110–	Shift
	11111		Sub
	00010	1100	Restore
Bit 2	00101	100–	Shift
	00010		Sub
		1001	No Restore
Bit 3	00101	001–	Shift
	00010		Sub
		0011	No Restore
Rem$\rightarrow$	00010	0011	$\leftarrow$Quotient

Non-Restoring (Unsigned) Division

❖ The non-restoring division defers restoration to reduce operations.
  - ➢ Addition is performed instead of subtraction in the next step.
  - ➢ This requires a final restoration, when last operation yielded a negative remainder.

❖ The same set-up is used.
  - ➢ Initialization
    - Clear A
    - Q gets dividend.
    - M gets divisor.
  - ➢ Loop for each bit of the dividend Q
    - If A (previous result) is negative ($A_n = 1$)
      - Shift AQ to the left.
      - Add ($A \leftarrow A + M$)

- Else
    - Shift *AQ* to the left.
    - Subtract ($A \leftarrow A - M$)
- $Q_0 \leftarrow \overline{A_n}$

➢ Perform restoration if last result in *A* is negative, *i.e.*, if $A_n = 1$ then $A \leftarrow A + M$
➢ Quotient in *Q* while remainder in *A*.

***Example:* 11 ¸ 3**

$-M$	11101		
M	00011		
	A	Q	
Init	00000	1011	
Bit 0	00001	011–	Shift
	11110		Sub
		0110	Clear Q_0
Bit 1	11100	110–	Shift
	11111		Add
		1100	Clear Q_0
Bit 2	11111	100–	Shift
	00010		Add
		1001	Set Q_0
Bit 3	00101	001–	Shift
	00010		Sub
		0011	Set Q_0
Rem$\rightarrow$	00010	0011	$\leftarrow$Quotient

***Example:* 13 ¸ 3**

$-M$	11101		
M	00011		
	A	Q	
Init	00000	1101	
Bit 0	00001	101–	Shift
	11110		Sub
		1010	Clear Q_0
Bit 1	11101	010–	Shift
	00000		Add
		0101	Set Q_0
Bit 2	00000	101–	Shift
	11101		Sub
		1010	Clear Q_0
Bit 3	11011	010–	Shift
	11110		Add
		0100	Set Q_0
	00001		Restore (Add)
Rem$\rightarrow$	00001	0100	$\leftarrow$Quotient

Floating-Point Arithmetic

- ❖ Real numbers are difficult to represent. This involves a certain amount of approximation with knowledge of a few significant digits at the start of the number, *e.g.*,

$$p \approx 3.1415926536$$
$$c \approx 299{,}792{,}458$$

- ❖ One approach is to use *fixed-point representation* where the number is given according to a fixed number of decimal places (and precision).
  - ➢ Conceals some of what we know for small numbers.
  - ➢ Demands more than we know for large numbers.
- ❖ Scientists and engineers use scientific notation where a number is expressed as

$$M \times 10^{E}$$

e.g.,

$$c \approx 2.998 \times 10^{8}$$
$$\text{Avogadro's Number} \approx 6.0247 \times 10^{23}$$
$$\text{Planck's Constant} \approx 6.6254 \times 10^{-27}$$

  - ➢ The *mantissa M* is a signed number and is said to be *normalized* if $1 \le |M| < 10$.
  - ➢ The power of 10 is always an integer and is called the *exponent E*.

Floating-Point Representation

- ❖ Computers represent real numbers using scientific notation in base 2. ⇒ *floating-point*

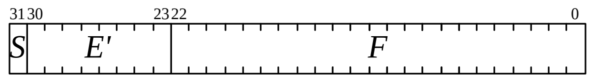
$$(-1)^{S} \times M \times 2^{E}$$

  - ➢ The mantissa is also typically *normalized* ($1 \le |M| < 2$) and can be represented as 1 plus a fraction, *i.e.*,

$$|M| = 1 + F \text{ or } 1.F$$

- ❖ Common schemes used include the IEEE-754 (Institute of Electrical and Electronics Engineers Standard 754).

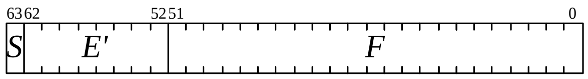
IEEE Single Precision Representation



$$(-1)^{S} \times 1.F \times 2^{E'-127}$$

- ❖ Uses 32 bits: 1-bit sign, 23-bit fraction, 8-bit exponent.
  - ➢ The fraction is normalized. The smallest fraction is at 2^{-23} or approximately 1.2×10^{-7} for 7-digit precision.
  - ➢ For ease in comparing numbers, the exponent is biased by 127 to represent negative values (*excess-127*).
  - ➢ Due to some exceptions, the approximate effective range is $\pm 1.17 \times 10^{-38}$ to $\pm 3.40 \times 10^{38}$.

IEEE Double Precision Representation



$$(-1)^S \times 1.F \times 2^{E'-1023}$$

❖ Uses 64 bits: 1-bit sign, 52-bit fraction, 11-bit exponent.
- ➢ The fraction is normalized. The smallest fraction is at 2^{-52} or approximately 2.2×10^{-16} for 16-digit precision.
- ➢ The exponent is biased by 1023 (*excess-1023*).
- ➢ Due to some exceptions, the approximate effective range is $\pm2.23\times10^{\pm308}$ to $\pm1.80\times10^{\pm308}$.

IEEE-754 Exceptions

Single Precision *E'*		**Double Precision *E'***		**Fraction *F***	**Value**
255	FFH	2047	7FFH	0	±Infinity
255	FFH	2047	7FFH	≠0	NaN, overflow, error etc.
0	00H	0	000H	0	0
0	00H	0	000H	≠0	Denormalized

Examples: 9.7578125 to IEEE Floating-Point Notation

- ➢ Conversion to IEEE Single-Precision
  - ▪ Determine sign bit ⇒ S = 0 (positive)
  - ▪ Convert to binary ⇒ 1001.1100001_2
  - ▪ Shift dot ⇒ $1.0011100001_2\times2^3$
  - ▪ Determine F ⇒ 00111000010000000000000 (Extended to 23 bits) or 1C2000H
  - ▪ Determine E' ⇒ $E + 127 = 130 = 10000010_2$ = 82H
  - ▪ Combine ⇒ 0 10000010 00111000010000000000000

    or grouping by 4's ⇒ 0100 0001 0001 1100 0010 0000 0000 0000 = 411C 2000H
- ➢ Conversion to IEEE Double-Precision
  - ▪ Differs mainly in E' ⇒ $E + 1023 = 1026 = 10000000010_2$ = 402H
  - ▪ Combining gives ⇒ 0 10000000010 0011100001000000000…0

    or ⇒ 0100 0000 0010 0011 1000 0100 0000 00…0

    ⇒ 4023 8400 0000 0000H
- ➢ NB. It is actually easier to work in hexadecimal. ☺

Guard Bits and Truncation

❖ *Guard bits* are extra bits in the fraction retained during arithmetic operations to increase accuracy.

❖ *Truncation* refers to the removal of the guard bits to fit into the representation.

❖ 3 Common Truncation Methods: Chopping, von Neumann Method, Round-Off

Chopping

- ➢ Guard bits are dropped regardless of the value.
- ➢ Biased error: 0 to +1 of lsb

Von Neumann Method

- ➢ Drop guard bits if all are 0, else set lsb to 1.
- ➢ Unbiased error: –1 to +1 of lsb

Round-Off

- Add 1 to lsb if msb of guard bits is 1, else just chop.
- Unbiased error: –0.5 to +0.5 of lsb

Examples:

Mantissa	Chopping	Von Neumann	Round-off
1.00000	1.000	1.000	1.000
1.0001×	1.000	1.001	1.001
1.0010×	1.001	1.001	1.001
1.0011×	1.001	1.001	1.010

Floating-Point Addition/Subtraction

- Adjust exponent to largest (Align mantissas).
- Add/subtract mantissas.
- Normalize mantissa and truncate

Floating-Point Multiplication/Division

- Add/Subtract exponents
- Multiply/divide mantissas
- Normalize mantissa and truncate

Examples:

$$100_{10} = +1.1001_2 \times 2^6 = \text{42C8 0000H}$$
$$-0.1_{10} = -1.10011001100110011001101_2 \times 2^{-4} = \text{BDCC CCCDH}$$

- 100 + (–0.1)
  - Align mantissa of –0.1 $\Rightarrow -0.0000000001100110011001100110011001101 \times 2^6$
  - Add (Subtract) $\Rightarrow +1.1000111110011001100110011001100110011 \times 2^6$
  - No need to normalize
  - Result $\Rightarrow$ 42C7 CCCDH
- 100 – (–0.1)
  - Align mantissa of –0.1 $\Rightarrow -0.0000000001100110011001100110011001101 \times 2^6$
  - Subtract (Add) $\Rightarrow +1.1001000001100110011001100110011001101 \times 2^6$
  - No need to normalize
  - Result $\Rightarrow$ 42C8 3333H
- 100 × (–0.1)
  - Add exponents $\Rightarrow 6 + (-4) = 2$
  - Multiply mantissa $\Rightarrow 10.10000000000000000000000000000101$
  - Normalize $\Rightarrow 1.010000000000000000000000000000000101 \times 2^3$
  - Adjust sign $\Rightarrow$ 1 (negative)
  - Result $\Rightarrow$ C120 0000H
- 100 ÷ (–0.1)
  - Subtract exponents $\Rightarrow 6 - (-4) = 10$
  - Divide mantissa $\Rightarrow 0.11111001111111111111111111111111110000011000$
  - Normalize $\Rightarrow 1.1111001111111111111111111110000011000 \times 2^9$
  - Adjust sign $\Rightarrow$ 1 (negative)
  - Result $\Rightarrow$ C47A 0000H