VIT-AP UNIVERSITY

Computer Organization & Architecture

ECE 2002

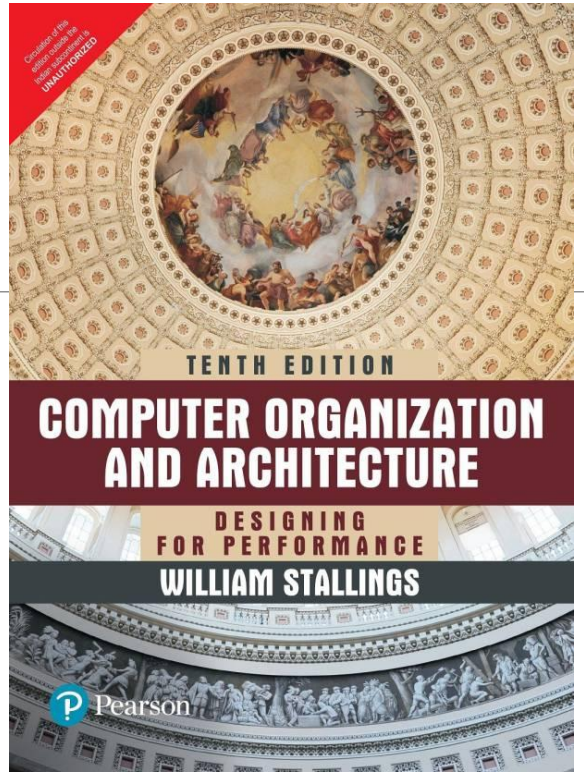
DR. S SARITHA

ASSOCIATE PROFESSOR

SENSE

Text Book:

William Stallings, Computer Organization and Architecture: Designing for Performance, Pearson Education, Tenth Edition,2013



Books:

1. M. Morris Mano, Rajib Mall, Computer System Architecture, Pearson Education Third Edition,2017.
2. Carl Hamacher, Zvonkovranesic, Safwat Zaky , Computer Organization, McGraw Hill, Fifth Edition,2011.

Integer representation

- Arbitrary numbers can be represented with just the digits zero and one, the minus sign, and the period

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

- For computer storage and processing → minus signs and periods cannot be used
- Only binary digits (0 and 1) may be used to represent numbers
- If limited to non-negative integers, the representation is straightforward

An 8-bit word can represent the numbers from 0 to 255, including

$$\begin{aligned} 00000000 &= 0 \\ 00000001 &= 1 \\ 00101001 &= 41 \\ 10000000 &= 128 \\ 11111111 &= 255 \end{aligned}$$

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

Sign-Magnitude representation

- There are several conventions used to represent negative as well as positive integers → all of which involve treating the most significant (leftmost) bit in the word as a sign bit.
- If the sign bit is 0 → the number is positive; if the sign bit is 1 → the number is negative
- The simplest form of representation that employs a sign bit is the sign-magnitude representation
- In an n-bit word, the rightmost bits hold the magnitude of the integer

$$
\begin{aligned}
+\,18 &= 00010010 \\
-\,18 &= 10010010 \quad \text{(sign magnitude)}
\end{aligned}
$$

Sign-Magnitude representation

- General case

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

- DRAWBACKS
- Addition and subtraction requires consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation
- Another drawback is that there are two representations of 0

$$+0_{10} = 00000000$$
$$-0_{10} = 10000000$$

Sign-Magnitude representation (Sample problems):-

(a) −127 (b) +63 (c) −219 (d) −66354 (e) +23655

(f) 1010111000 (g) 0101001110001 (h) 1011101010 (i) 11100011 (j) 00011010

Sign - Magnitude representation (Sample problems):-

(a) −127 (b) +63 (c) −219 (d) −66354 (e) +23655

(f) 10101111000 (g) 0101001110001 (h) 1011101010 (i) 11100011 (j) 00011010

Sol:-

(a) −127 = 1 | 1111111 (S | mag)

(b) +63 = 0 | 111111 (S | mag)

(c) −219 = 1 | 1011011

(d) −66354 = 1 | 1000000110011010

(e) +23655 = 0 | 1011100011100111

(f) 10101111000 → −184

(g) 0101001110001 → +2673

(h) 1011101010 → −938

(i) 1 | 1110011 → −227

(j) 0 | 0011010 → +26

2’s complement representation

- 2’s complement representation also uses the MSB as a sign bit

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.

2’s complement representation

- Consider an n-bit integer, A, in 2’s complement representation
- If A is positive, then the sign bit, a_{n-1} is zero. The remaining bits represent the magnitude of the number in the same fashion as for sign magnitude representation

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

- The number zero is identified as positive and therefore has a 0 sign bit and a magnitude of all 0s.
- The 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).
- Any larger number would require more bits

2’s complement representation

- Now, for a negative number the sign bit, a_{n-1} is one
- The remaining $n - 1$ bits can take on any one of 2^{n-1} values
- Therefore, the range of negative integers that can be represented is from -1 to -2^{n-1}
- The weight of the most significant bit is -2^{n-1}
- This is the convention used in 2’s complement representation, yielding the following expression for negative numbers

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

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

2's complement representation

- For $a_{n-1} = 0$, the term $-2^{n-1}a_{n-1} = 0$ and the equation defines a non-negative integer
- When $a_{n-1} = 1$, the term 2^{n-1} is subtracted from the summation term, 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

Working

- Consider an n-bit sequence of binary digits $a_{n-1}a_{n-2}\ldots a_1a_0$ interpreted as a 2's complement integer A, so that its value is

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

- If A is a positive number, the rule clearly works. Now, if A is negative and we want to construct an m-bit representation, with m > n. Then

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

2’s complement using value box representation

- The nature of 2’s complement representation is a value box, in which the value on the far right in the box is $1\ (2^0)$
- Each succeeding position to the left is double in value, until the leftmost position, which is negated
- The most negative 2’s complement number that can be represented is -2^{n-1} i.e. if any of the bits other than the sign bit is one, it adds a positive amount to the number

-128	64	32	16	8	4	2	1

-128	64	32	16	8	4	2	1
1	0	0	0	0	0	1	1

$-128 \qquad +2 \quad +1 \quad = -125$

Two's complement representation (Decimal to binary)

(1) A = −13

+13 ⇒ 01101

ones comp

10010

Add '1'

```
  10010
+     1
-------
  10011
```

→ two's comp of +13 ⇒ −13

(2) A = −273

+273 ⇒ 0100010001

1011101110

```
  1011101110
+          1
------------
  1011101111
```

→ −273

```
2 | 273
2 | 136 - 1
2 | 68  - 0
2 | 34  - 0
2 | 17  - 0
2 | 8   - 1
2 | 4   - 0
2 | 2   - 0
    1   - 0
```

(3) $A = +186$

$186 \Rightarrow 10111010$

$+186 \Rightarrow 01011010$

(4) $A = -6434$

$6434 \Rightarrow 1100100100010$

$+6434 \Rightarrow 0110010010010$

neg/comp → $1001101101110\,1$

Add 1 → $+\quad 1$

$1001101101111\,0$ → -6434

(5) $A = -2667$

$2667 \Rightarrow 101001101011$

$+2667 \Rightarrow 010100110101\,1$

$+\quad 1$

1010110010100

$+\quad 1$

1010110010101 → -2667

(6) $A = +777$

$+777 \Rightarrow 0110000100\,1$

Binary to Decimal

① 01010001

$2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

$1\times2^6 + 1\times2^4 + 1\times2^0$

$64 + 16 + 1$

$+81$

② $01110111\ 01$

$2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

$1\times2^0 + 1\times2^2 + 1\times2^3 + 1\times2^4 + 1\times2^6 + 1\times2^7 + 1\times2^8$

$1 + 4 + 8 + 16 + 64 + 128 + 256 = +477$

③ $1011001011\ 0$

$1\times2^1 + 1\times2^2 + 1\times2^4 + 1\times2^7 + 1\times2^8 + (1\times -2^{10})$

$2 + 4 + 16 + 128 + 256 - 1024$

$\Rightarrow 406 - 1024$

$\Rightarrow -618$

④ 111100110

$1\times2^1 + 1\times2^2 + 1\times2^6 + 1\times2^7 + 1\times2^8 + 1\times(-2^9)$

$2 + 4 + 64 + 128 + 256 - 512$

$406 - 512$

-106

⑤ 0101000011111111

$1\times2^0 + 1\times2^1 + 1\times2^2 + 1\times2^3 + 1\times2^4 + 1\times2^5$

$+1\times2^6 + 2^7 + 1\times2^{14}$

$\Rightarrow 1+2+4+8+16+32+64+4096+16384 = +20607$

⑥ 0011111011110110

$1\times2^1 + 1\times2^2 + 1\times2^4 + 1\times2^5 + 1\times2^6 + 1\times2^8 + 1\times2^9$

$+1\times2^{10} + 1\times2^{11} + 2^{12} +$

$2+4+16+32+64+256+512+1024+$

$2048 + 4096 = +8054$

⑦ 1001111011101 1

−16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
1	0	0	1	1	1	1	0	1	1	1	0	1	0	1

$\Rightarrow$

$1024 + 512 + 256 + 0 + 64 + 32 + 16 + 0 + 4 + 0 + 1$

$+204 + 0 + 0 - 16384$

$\Rightarrow$ **-12427**

Converting between different bit lengths

- It is sometimes desirable to take an *n*-bit integer and store it in *m* bits, where m > n
- In sign-magnitude notation, this is easily accomplished: simply move the sign bit to the new leftmost position and fill in with zeros.

- For 2's complement negative numbers

+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)

Converting between different bit lengths

- The rule for 2's complement integers is to move the sign bit to the new leftmost position and fill in with copies of the sign bit.
- For positive numbers, fill in with zeros, and for negative numbers, fill in with ones. This is called sign extension.

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

Integer arithmetic: negation

- In sign-magnitude representation, the rule for forming the negation of an integer is simple: invert the sign bit
- In 2's complement notation,
  - Take the Boolean complement of each bit of the integer
  - Treating the result as an unsigned binary integer, add 1

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

Integer arithmetic: Addition

1001 = −7 +0101 = 5 1110 = −2 (a) (−7) + (+5)	1100 = −4 +0100 = 4 10000 = 0 (b) (−4) + (+4)
0011 = 3 +0100 = 4 0111 = 7 (c) (+3) + (+4)	1100 = −4 +1111 = −1 11011 = −5 (d) (−4) + (−1)
0101 = 5 +0100 = 4 1001 = Overflow (e) (+5) + (+4)	1001 = −7 +1010 = −6 10011 = Overflow (f) (−7) + (−6)

Integer arithmetic: Addition

1001 = −7 +0101 = 5 1110 = −2 (a) (−7) + (+5)	1100 = −4 +0100 = 4 10000 = 0 (b) (−4) + (+4)
0011 = 3 +0100 = 4 0111 = 7 (c) (+3) + (+4)	1100 = −4 +1111 = −1 11011 = −5 (d) (−4) + (−1)
0101 = 5 +0100 = 4 1001 = Overflow (e) (+5) + (+4)	1001 = −7 +1010 = −6 10011 = Overflow (f) (−7) + (−6)

Integer arithmetic: Addition

1001 = −7 +0101 = 5 1110 = −2 (a) (−7) + (+5)	1100 = −4 +0100 = 4 10000 = 0 (b) (−4) + (+4)
0011 = 3 +0100 = 4 0111 = 7 (c) (+3) + (+4)	1100 = −4 +1111 = −1 11011 = −5 (d) (−4) + (−1)
0101 = 5 +0100 = 4 1001 = Overflow (e) (+5) + (+4)	1001 = −7 +1010 = −6 10011 = Overflow (f) (−7) + (−6)

Integer arithmetic: subtraction

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

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

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

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

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

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

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

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

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

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

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

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

Integer arithmetic: subtraction

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

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

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

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

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

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

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

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

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

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

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

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

Integer arithmetic: subtraction

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

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

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

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

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

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

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

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

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

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

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

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

10.10 Assume numbers are represented in 8-bit twos complement representation. Show the calculation of the following:

a. $9 + 12$ **b.** $9 - 12$ **c.** $-9 + 12$ **d.** $-9 - 12$

10.11 Find the following differences using twos complement arithmetic:

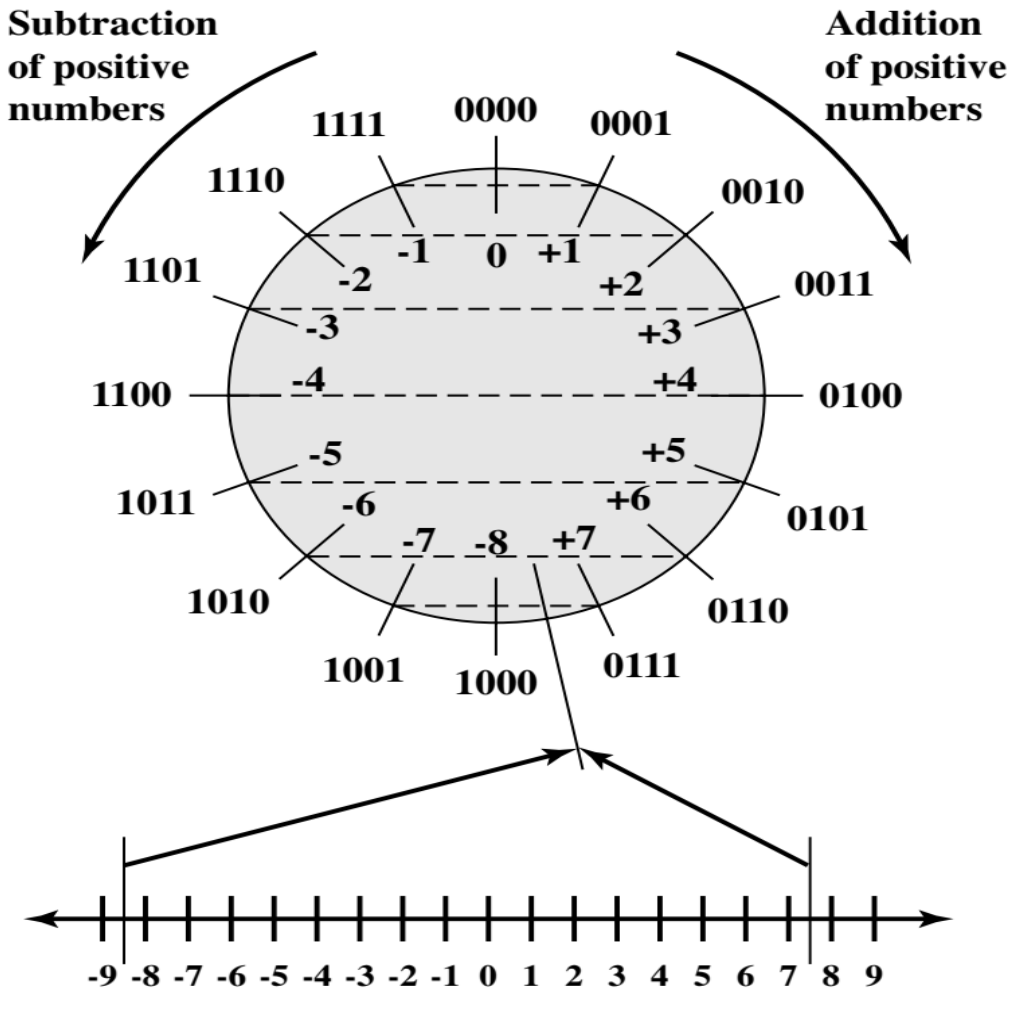
a.
$$\begin{array}{r} 001000 \\ -110111 \end{array}$$

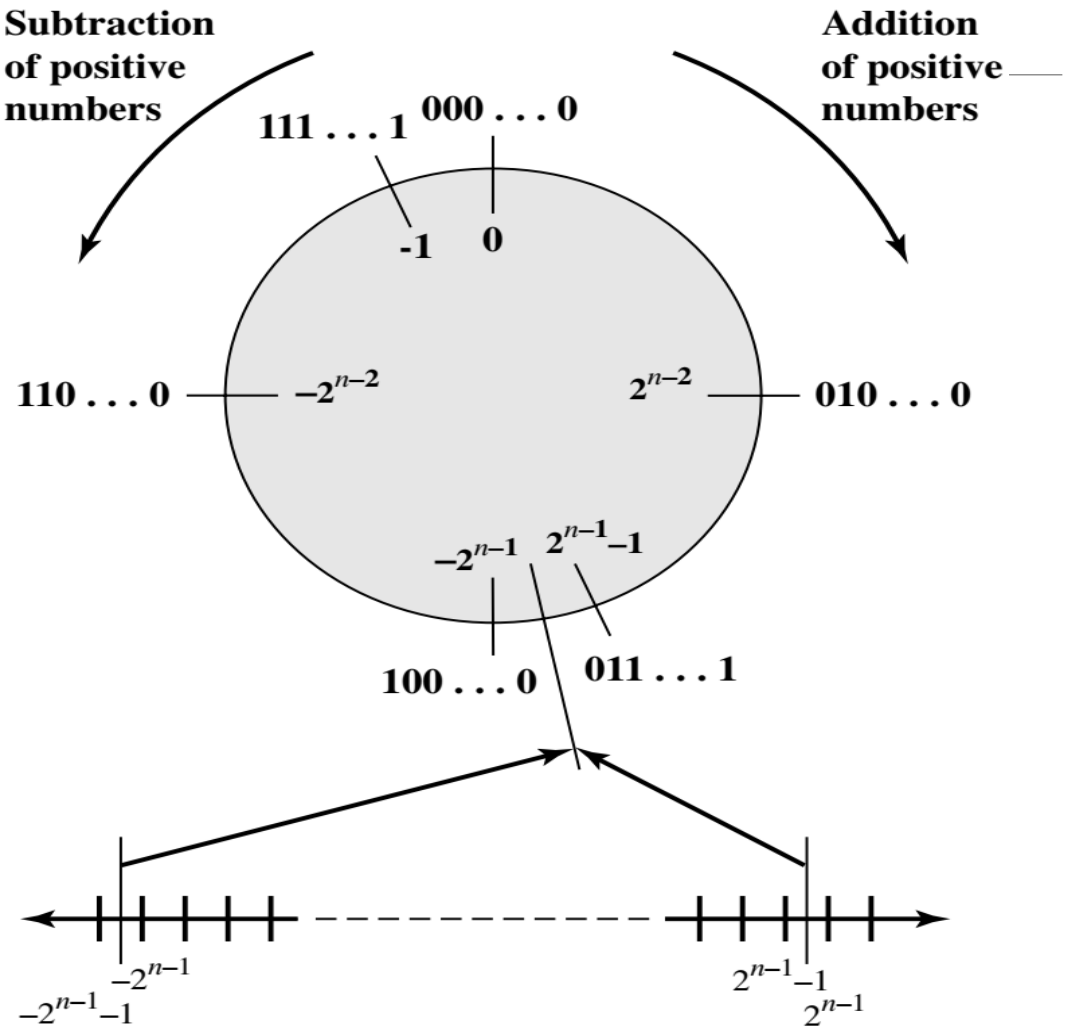
b.
$$\begin{array}{r} 10100101 \\ -\ 011110 \end{array}$$

c.
$$\begin{array}{r} 100100001011 \\ -\ 101010110011 \end{array}$$

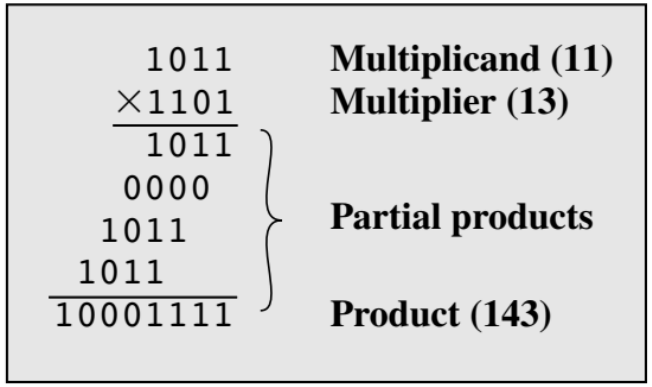
d.
$$\begin{array}{r} 10011011 \\ -\ 10100101 \end{array}$$

Geometric Depiction of Twos Complement Integers





Integer arithmetic: multiplication



- Compared with addition and subtraction, multiplication is a complex operation, whether performed in hardware or software
- A wide variety of algorithms have been used in various computers
- Multiplication of unsigned integers
  - 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
  - The partial products are easily defined. When the multiplier bit is 0, the partial product is 0. When the multiplier bit is 1, the partial product is the multiplicand
  - The total product is produced by summing the partial products. For this operation, each successive partial product is shifted one position to the left relative to the preceding partial product
  - The multiplication of two n-bit binary integers results in a product of up to 2n bits in length (e.g., 11 x 11 = 1001)

Integer arithmetic: multiplication

- Compared with the pencil-and-paper approach, there are several things we can do to make computerized multiplication more efficient
- First, we can perform a running addition on the partial products rather than waiting until the end.
- For each 1 on the multiplier, an add and a shift operation are required; but for each 0, only a shift is required.

Multiplication

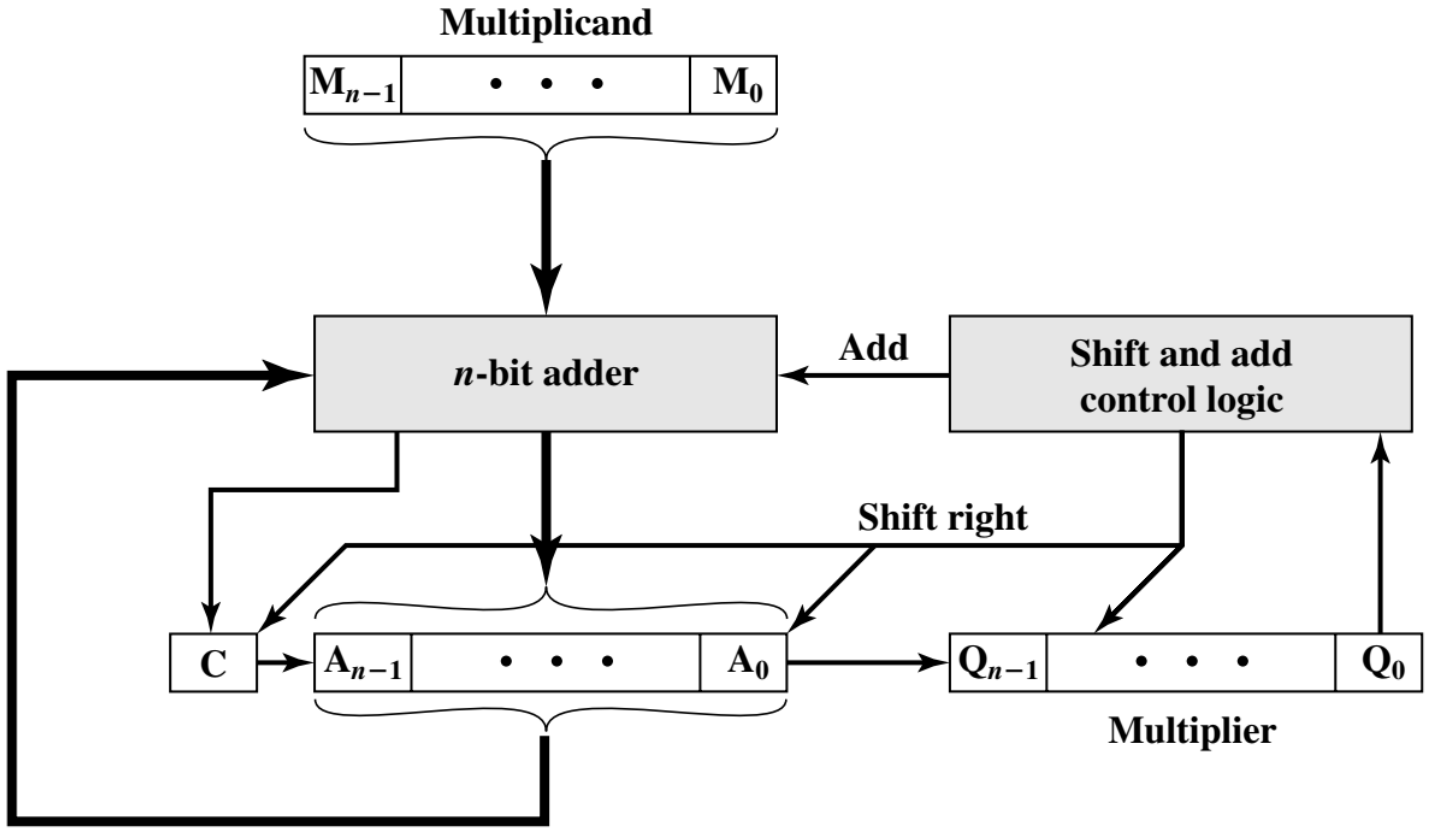
Complex

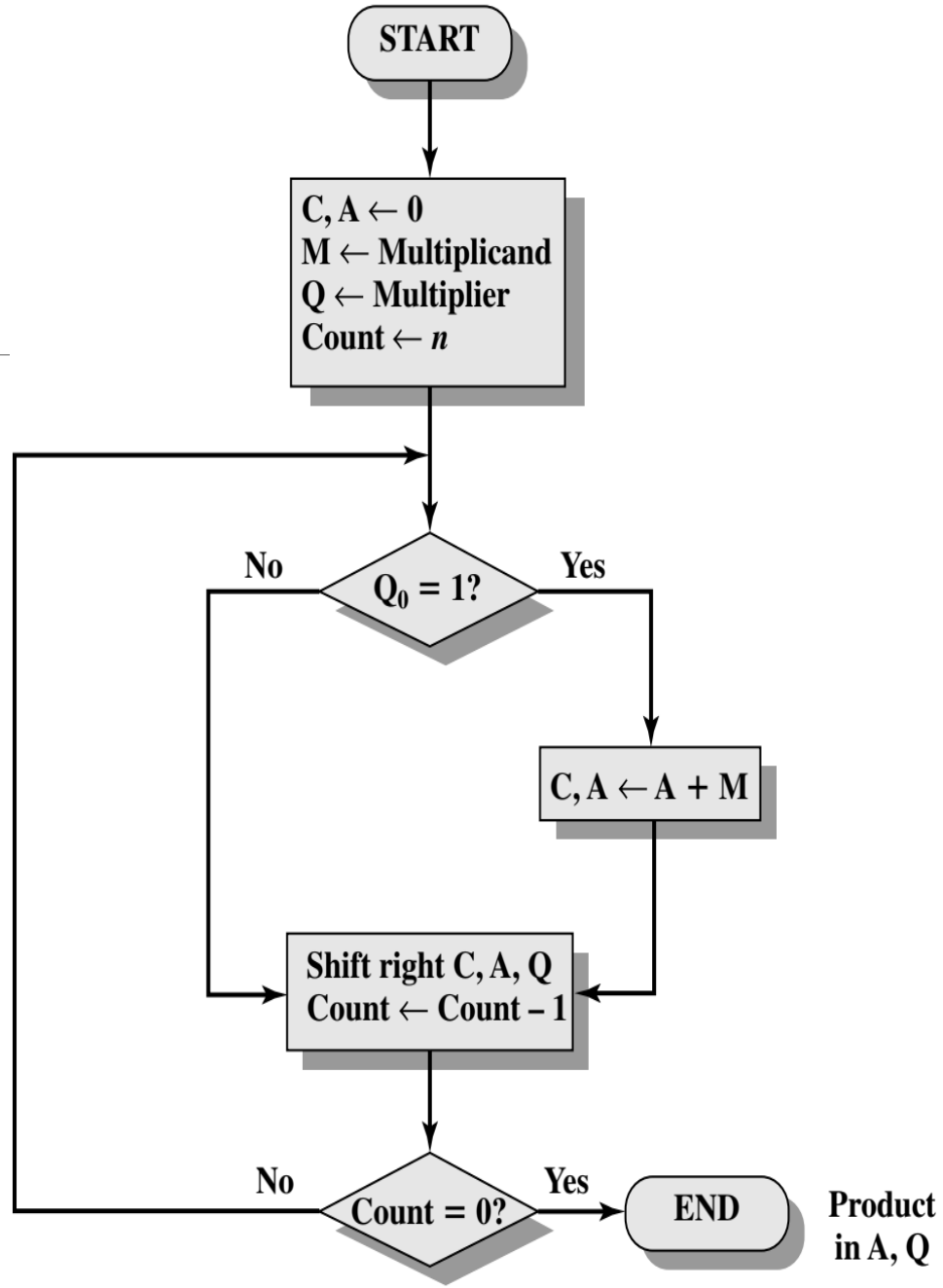
Work out partial product for each digit

Take care with place value (column)

Add partial products

Multiplication





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

Multiplying Negative Numbers

This does not work! Eg. If we multiply 11 (1011) by 13 (1101) we should get 143 (10001111).

If we interpret these as two's complement numbers, we have -5 (1011) times

-3(1101) which equals -113(10001111) which is incorrect.

It will also not work if either the multiplicand or the multiplier is negative.

If 9 and 3 are treated as unsigned integers, the multiplication proceeds simply fig a.

But if 1001 is interpreted as the

twos complement value -7,then each partial product must be a negative twos complement number of 2n(8) bits, as shown in Figure b.

Note: this is accomplished by padding out each partial product to the left with binary 1s

1001 (9)	1001 (−7)
× 0011 (3)	× 0011 (3)
00001001 1001 × 2^0	11111001 (−7) × 2^0 = (−7)
00010010 1001 × 2^1	11110010 (−7) × 2^1 = (−14)
00011011 (27)	11101011 (−21)
(a) Unsigned integers	(b) Twos complement integers

If the multiplier is negative

If the multiplier is negative, straightforward multiplication also will not work.

The reason is that the bits of the multiplier no longer correspond to the shifts or multiplications that must take place.

For example, the 4-bit decimal number -3 is written 1101 in twos complement. If we simply took partial products based on each bit position, we have

instead of

$$1101 \longleftrightarrow -(1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) = -(2^3 + 2^2 + 2^0)$$

$$-(2^1 + 2^0)$$

Solutions to the above issues

Solution 1

- Convert to positive if required
- Multiply as above
- If signs were different, negate answer

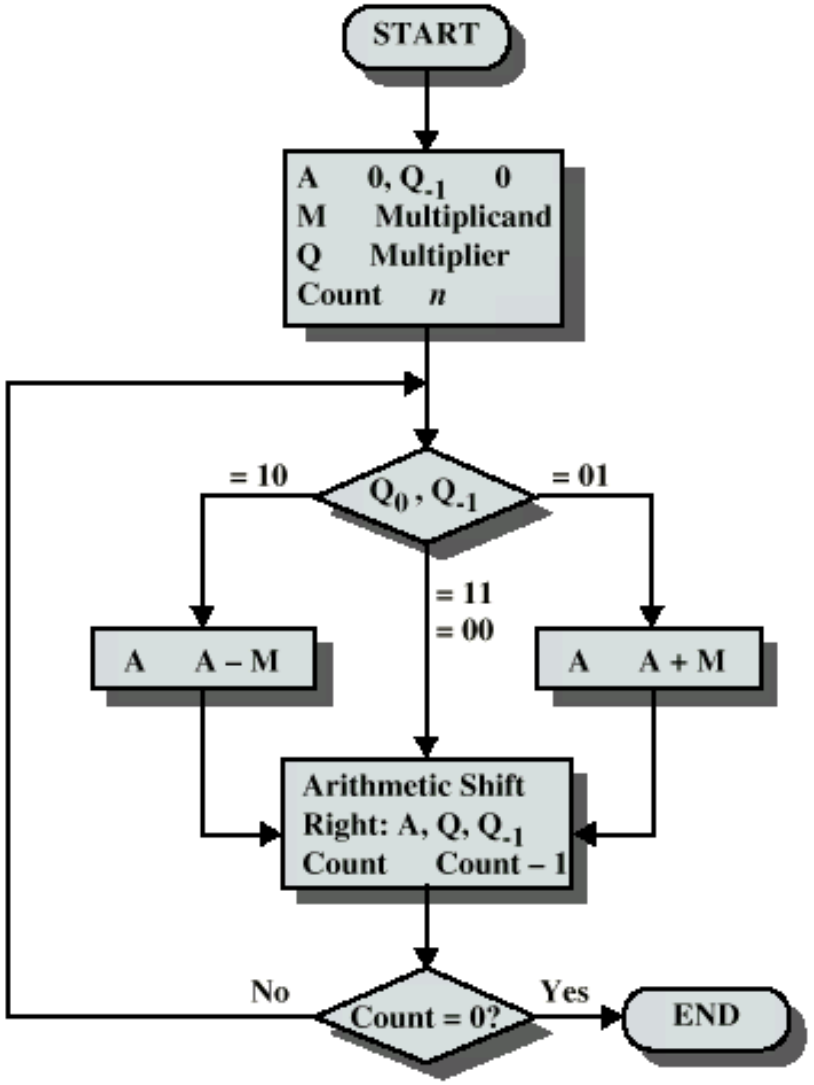
Solution 2

- Booth's algorithm

Booth's Algorithm with example

Example of Booth's Algorithm (7×3)

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial Values	
1001	0011	0	0111	A A - M	First Cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second Cycle
0101	0100	1	0111	A A + M	Third Cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth Cycle



Description of the algorithm

As before, the multiplier and multiplicand are placed in the Q and M registers, respectively.

There is also a 1-bit register placed logically to the right of the least significant bit of the Q register and designated its use is explained shortly.

The results of the multiplication will appear in the A and Q registers.

A and Q_{-1} are initialized to 0. As before, control logic scans the bits of the multiplier one at a time.

Now, as each bit is examined, the bit to its right is also examined.

If the two bits are the same (1–1 or 0–0), then all of the bits of the A, Q, and Q_{-1} registers are shifted to the right by 1 bit. If the two bits differ, then the multiplicand is added to or subtracted from the A register, depending on whether the two bits are 0–1 or 1–0.

Following the addition or subtraction, the right shift occurs. In either case, the right shift is such that the leftmost bit of A, namely A_{n-1}not only is shifted into A_{n-2} but also remains in A_{n-1}

This is required to preserve the sign of the number in A and Q. It is known as an arithmetic shift, because it preserves the sign bit.

Finally, the result is at A and Q registers.

Some more examples

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

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

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

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

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

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

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

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

Note: For Q_n and Q_{n-1}

If 1-1/0-0 the value of A will not change ie. Will remain 0000

If 1-0, A=A-M= 0000-0111=1001

If 0-1, A=A+M= 0000+0111=0111

But for each positive intermediate value, 0s will be appended at left and for negative value, 1s will be appended at the left side.

EXAMPLE: 7X(-3)

A	Q	Q-1	M	
		(-3)	7	
0000	1101	0	0111	
1001	1101	0	0111	A <- (A - M) 1st cycle
1100	1110	1	0111	Shift
0011	1110	1	0111	A <- (A + M) 2nd cycle
0001	1111	0	0111	Shift
1010	1111	0	0111	A <- (A - M) 3rd cycle
1101	0111	1	0111	Shift
1110	1011	1	0111	Shift

EXAMPLE: (-5)X(-7)

OPERATION	AC	QR	Qn+1	SC
	0000	1001	0	4
AC + BR' + 1	0101	1001	0	
ASHR	0010	1100	1	3
AC + BR	1101	1100	1	
ASHR	1110	1110	0	2
ASHR	1111	0111	0	1
AC + BR' + 1	0100	0111	0	
ASHR	0010	0011	1	0

Array multiplication

- Let us consider the multiplicand to be M = (A_3, A_2, A_1, A_0) and the multiplier to be Q = (B_3, B_2, B_1, B_0)

				A3	A2	A1	A0	Inputs
			x	B3	B2	B1	B0	
			C	B0 x A3	B0 x A2	B0 x A1	B0 x A0	
		+	B1 x A3	B1 x A2	B1 x A1	B1 x A0		
		C	sum	sum	sum	sum		
	+	B2 x A3	B2 x A2	B2 x A1	B2 x A0			Internal Signals
	C	sum	sum	sum	sum			
+	B3 x A3	B3 x A2	B3 x A1	B3 x A0				
C	sum	sum	sum	sum				
Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	Outputs

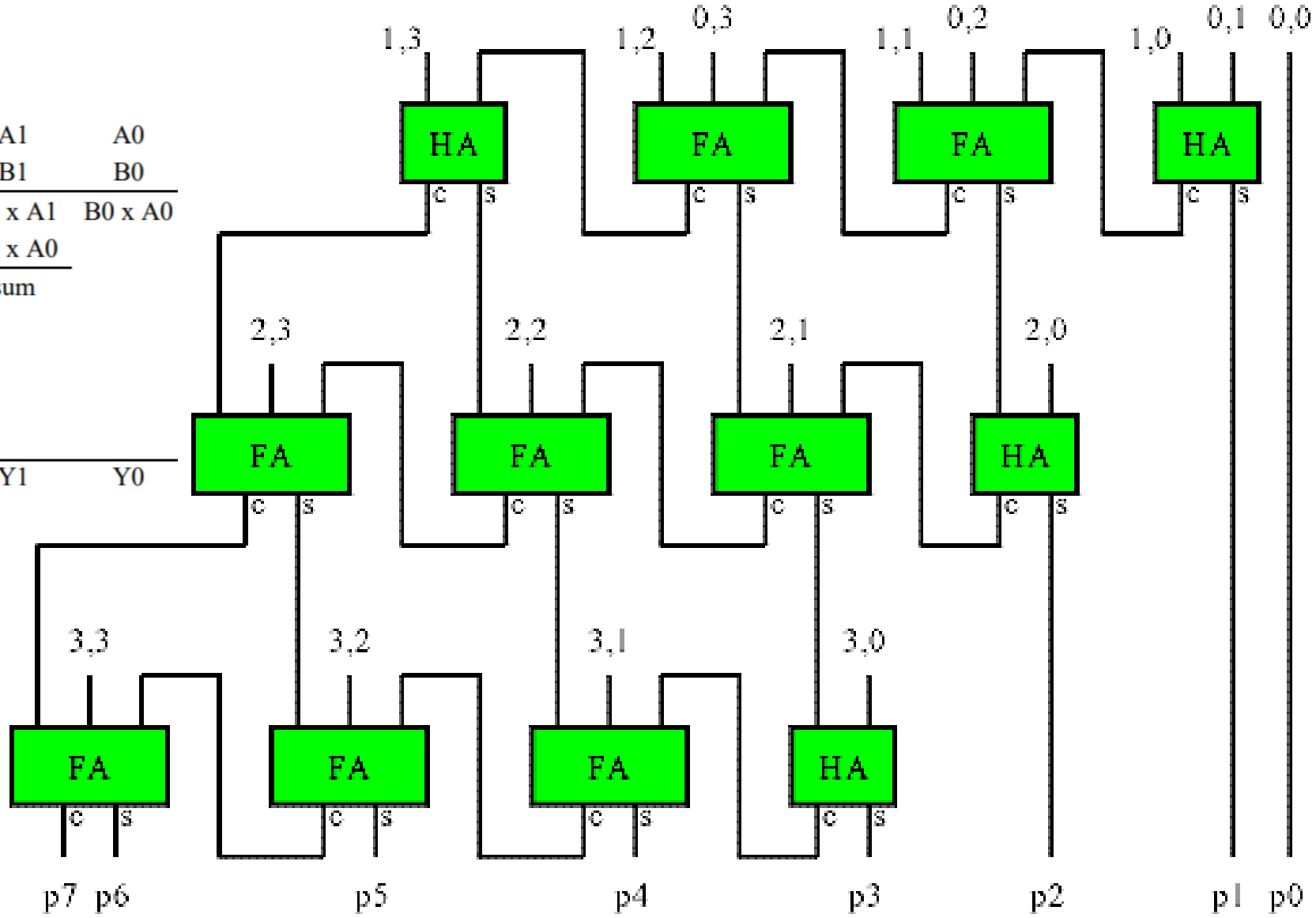
Array multiplication

- Let us consider the multiplicand to be M = (a_3, a_2, a_1, a_0) and the multiplier to be Q = (b_3, b_2, b_1, b_0)

				A3	A2	A1	A0	Inputs
			x	B3	B2	B1	B0	
			C	B0 x A3	B0 x A2	B0 x A1	B0 x A0	
		+	B1 x A3	B1 x A2	B1 x A1	B1 x A0		
		C	sum	sum	sum	sum		
	+	B2 x A3	B2 x A2	B2 x A1	B2 x A0			Internal Signals
	C	sum	sum	sum	sum			
+	B3 x A3	B3 x A2	B3 x A1	B3 x A0				
C	sum	sum	sum	sum				
Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	Outputs

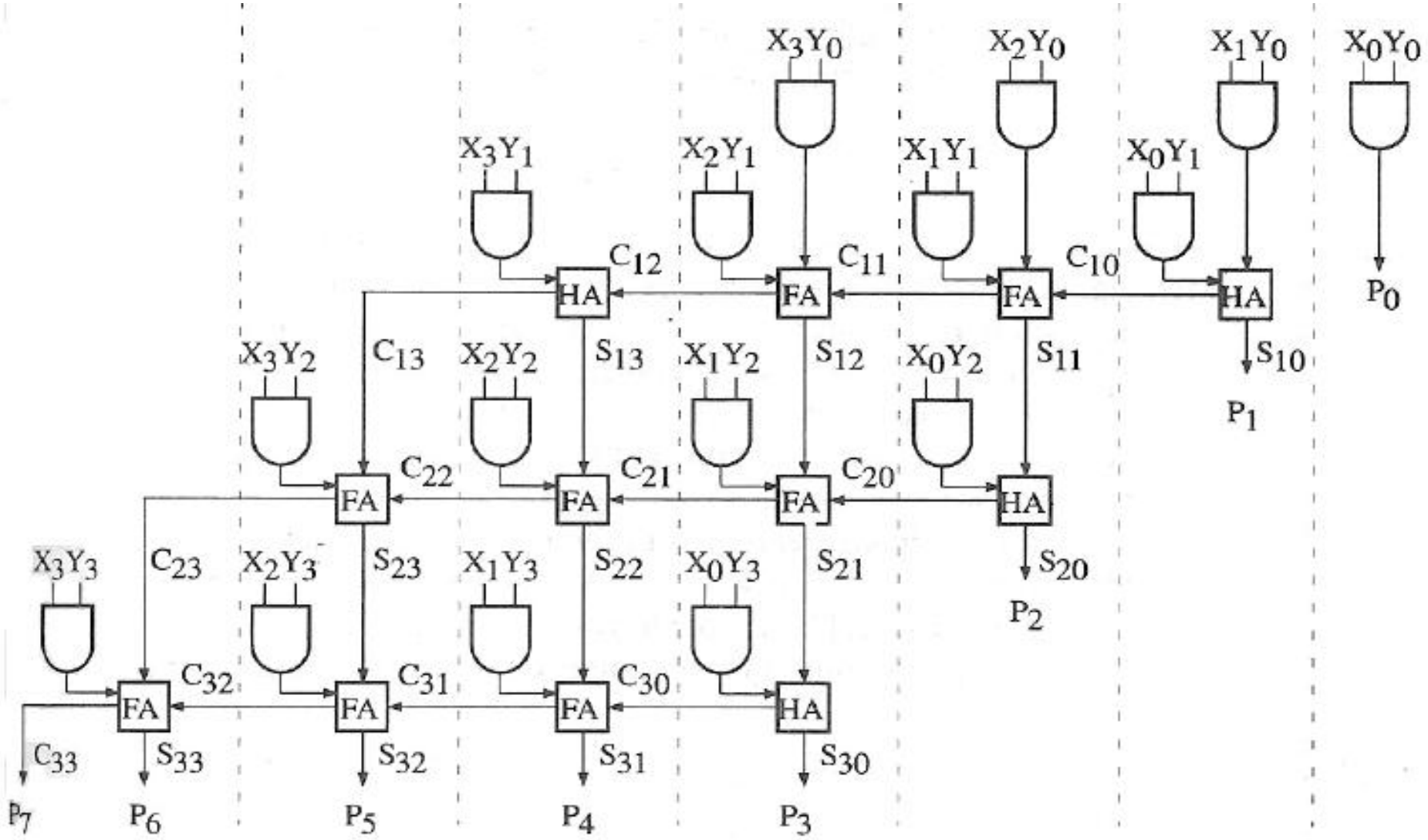
Hardware configuration of 4*4 Array multiplier

				A3	A2	A1	A0
			x	B3	B2	B1	B0
			C	B0 x A3	B0 x A2	B0 x A1	B0 x A0
		+	B1 x A3	B1 x A2	B1 x A1	B1 x A0	
		C	sum	sum	sum	sum	
	+	B2 x A3	B2 x A2	B2 x A1	B2 x A0		
	C	sum	sum	sum	sum		
+	B3 x A3	B3 x A2	B3 x A1	B3 x A0			
C	sum	sum	sum	sum			
Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0



For m-bit*n-bit multiplier we need m*n AND gates, n Half adders, and (m-2)*n Full adders

Hardware configuration of 4*4 Array multiplier



For m-bit*n-bit multiplier we need m*n AND gates, n Half adders, and (m-2)*n Full adders

Signed multiplication using array multiplier

```
                                                p0[7] p0[6] p0[5] p0[4] p0[3] p0[2] p0[1] p0[0]
                                      + p1[7] p1[6] p1[5] p1[4] p1[3] p1[2] p1[1] p1[0] 0
                                + p2[7] p2[6] p2[5] p2[4] p2[3] p2[2] p2[1] p2[0] 0     0
                          + p3[7] p3[6] p3[5] p3[4] p3[3] p3[2] p3[1] p3[0] 0     0     0
                    + p4[7] p4[6] p4[5] p4[4] p4[3] p4[2] p4[1] p4[0] 0     0     0     0
              + p5[7] p5[6] p5[5] p5[4] p5[3] p5[2] p5[1] p5[0] 0     0     0     0     0
        + p6[7] p6[6] p6[5] p6[4] p6[3] p6[2] p6[1] p6[0] 0     0     0     0     0     0
  + p7[7] p7[6] p7[5] p7[4] p7[3] p7[2] p7[1] p7[0] 0     0     0     0     0     0     0
-------------------------------------------------------------------------------------------
P[15] P[14] P[13] P[12] P[11] P[10]  P[9]  P[8]  P[7]  P[6]  P[5]  P[4]  P[3]  P[2]  P[1]  P[0]
```

```
                                                  1  ~p0[7]  p0[6]  p0[5]  p0[4]  p0[3]  p0[2]  p0[1]  p0[0]
                                              ~p1[7] +p1[6] +p1[5] +p1[4] +p1[3] +p1[2] +p1[1] +p1[0]   0
                                          ~p2[7] +p2[6] +p2[5] +p2[4] +p2[3] +p2[2] +p2[1] +p2[0]   0      0
                                      ~p3[7] +p3[6] +p3[5] +p3[4] +p3[3] +p3[2] +p3[1] +p3[0]   0      0      0
                                  ~p4[7] +p4[6] +p4[5] +p4[4] +p4[3] +p4[2] +p4[1] +p4[0]   0      0      0      0
                              ~p5[7] +p5[6] +p5[5] +p5[4] +p5[3] +p5[2] +p5[1] +p5[0]   0      0      0      0      0
                          ~p6[7] +p6[6] +p6[5] +p6[4] +p6[3] +p6[2] +p6[1] +p6[0]   0      0      0      0      0      0
  1  +p7[7] ~p7[6] ~p7[5] ~p7[4] ~p7[3] ~p7[2] ~p7[1] ~p7[0]   0      0      0      0      0      0      0
 ---------------------------------------------------------------------------------------------------------------------
P[15]  P[14]  P[13]  P[12]  P[11]  P[10]   P[9]   P[8]   P[7]   P[6]   P[5]   P[4]   P[3]   P[2]   P[1]  P[0]
```

Division

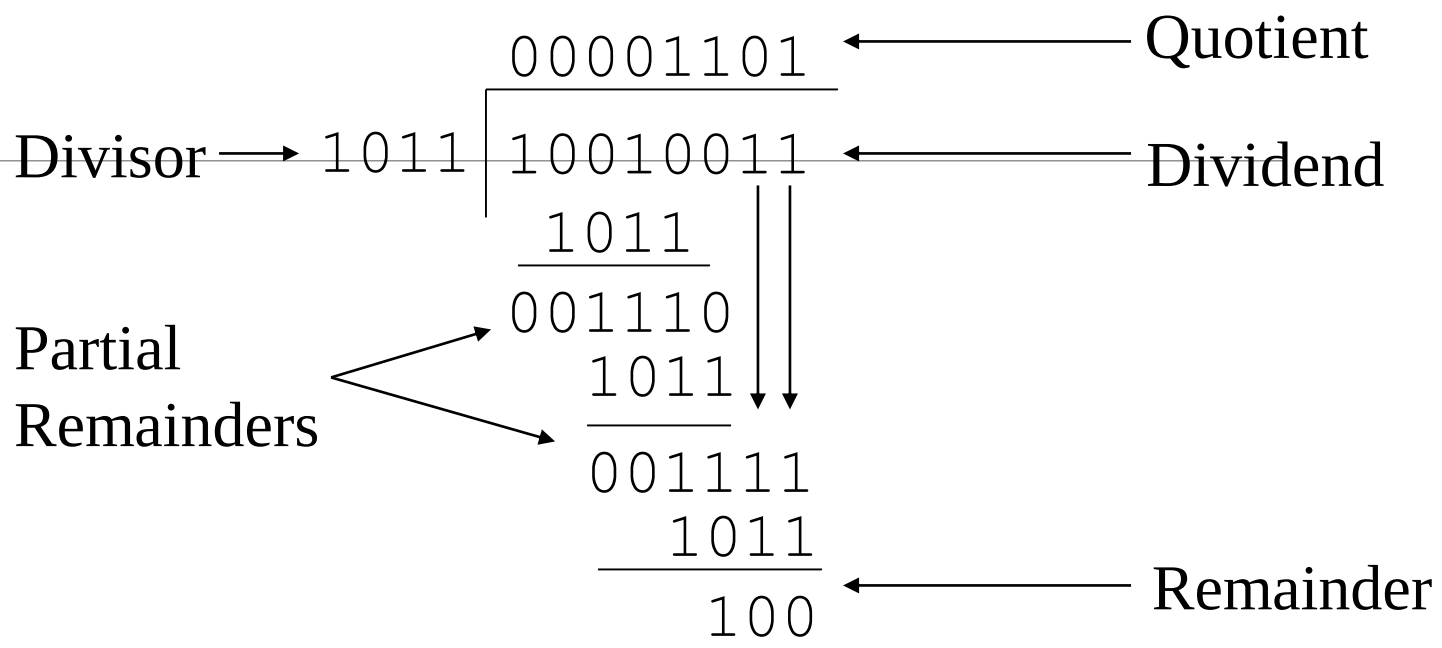
More complex than multiplication

Negative numbers are really bad!

Based on long division

Division of Unsigned Binary Integers

- First, the bits of the dividend are examined from left to right, until the set of bits examined represents a number greater than or equal to the divisor; this is referred to as the divisor being able to divide the number.
- Until this event occurs, 0s are placed in the quotient from left to right.
- When the event occurs, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend. The result is referred to as a partial remainder.
- This process is carried out in cyclic manner. At each cycle, additional bits from the dividend are appended to the partial remainder until the result is greater than or equal to the divisor.
- As before, the divisor is subtracted from this number to produce a new partial remainder.
- The process continues until all the bits of the dividend are exhausted.

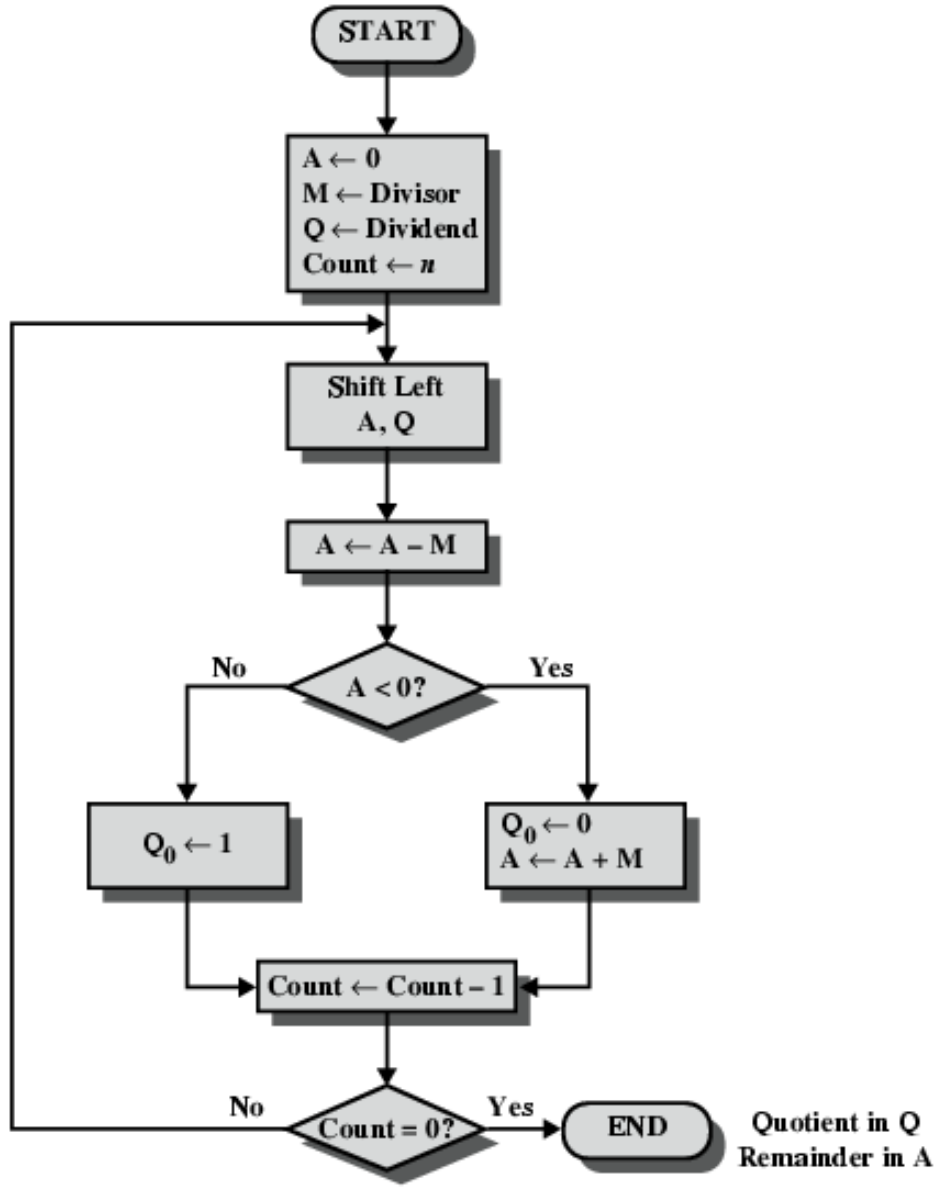


Flowchart for Unsigned Binary Division Restoring Algorithm

Example of Restoring Twos Complement Division (7/3)

M=0011

A	Q	
0000	0111	Initial value
0000	1110	Shift
1101		Use twos complement of 0011 for subtraction
1101		Subtract
0000	1110	Restore, set $Q_0 = 0$
0001	1100	Shift
1101		
1110		Subtract
0001	1100	Restore, set $Q_0 = 0$
0011	1000	Shift
1101		
0000	1001	Subtract, set $Q_0 = 1$
0001	0010	Shift
1101		
1110		Subtract
0001	0010	Restore, set $Q_0 = 0$



Procedure

- In the 1st step, the A and Q registers together are shifted to the left 1 bit.
- M is subtracted from A to determine whether A divides the partial remainder.
- If it does, then Q_0 gets a 1 bit. Otherwise, Q_0 gets a 0 bit and M must be added back to A to **restore the previous value**. The count is then decremented, and the process continues for n steps.
- At the end, the quotient is in the Q register and the remainder is in the A register.

Negative number Division

The procedure is little different, though the restoring algorithm will be adopted.

To deal with negative numbers, we recognize that the dividend is defined by D = Q * V + R

$$
\begin{array}{llllll}
D = 7 & V = 3 & \Rightarrow & Q = 2 & R = 1 \\
D = 7 & V = -3 & \Rightarrow & Q = -2 & R = 1 \\
D = -7 & V = 3 & \Rightarrow & Q = -2 & R = -1 \\
D = -7 & V = -3 & \Rightarrow & Q = 2 & R = -1
\end{array}
$$

Examples with different combinations

The **magnitudes** of Q and R are unaffected by the input signs and that the **signs** of Q and R are easily derivable from the signs of D and V

Sign (R) = Sign (D) whereas Sign (Q) = Sign (D) * Sign(V)

Division of Signed numbers using Restoring Algorithm

1) Let M register hold the divisor, Q register hold the divided.
2) A register should be the signed extension of Q.
3) On completion of the algorithm, Q will get the quotient and A will get the remainder.

Algorithm:

The **number of steps** required is equal to the **number of bits in the Dividend**.

1) At each step, **left shift the dividend by 1 position**.
2) If Sign of A and M is the same then **Subtract the divisor from A** (perform A - M),
   Else **Add M to A**
3) After the operation,
   If **Sign of A remains the same** or the **dividend** (in A and Q) **becomes zero**,
   then the step is said to be **"Successful"**.
   In this case **quotient bit will be "1"** and **Restoration is NOT Required**.
4) If **Sign of A changes**, then the step is said to be **"Unsuccessful"**.
   In this case **quotient bit will be "0".**
   Here Restoration is Performed.
   Hence, the method is called Restoring Division.

Repeat steps 1 to 4 for **all bits** of the Dividend.

- Remember with signed division, we negate the quotient if the signs of the divisor and dividend don't match.
- The remainder and the dividend must have the same signs

Examples

A	Q	M = 0011	
0000	0111	Initial values	
0000	1110	Shift	1
1101		Subtract	
0000	1110	Restore	
0001	1100	Shift	2
1110		Subtract	
0001	1100	Restore	
0011	1000	Shift	3
0000		Subtract	
0000	1001	$Q_0 = 1$	
0001	0010	Shift	4
1110		Subtract	
0001	0010	Restore	

(7) / (3)

A	Q	M = 1101	
0000	0111	Initial values	
0000	1110	Shift	1
1101		Add	
0000	1110	Restore	
0001	1100	Shift	2
1110		Add	
0001	1100	Restore	
0011	1000	Shift	3
0000		Add	
0000	1001	$Q_0 = 1$	
0001	0010	Shift	4
1110		Add	
0001	**0010**	Restore	

(7) / (-3)

A	Q	M = 0011	
1111	1001	Initial values	
1111	0010	Shift	1
0010		Add	
1111	0010	Restore	
1110	0100	Shift	2
0001		Add	
1110	0100	Restore	
1100	1000	Shift	3
1111		Add	
1111	1001	$Q_0 = 1$	
1111	0010	Shift	4
0010		Add	
1111	**0010**	Restore	

(-7) / (3)

A	Q	M = 1101	
1111	1001	Initial values	
1111	0010	Shift	1
0010		Subtract	
1111	0010	Restore	
1110	0100	Shift	2
0001		Subtract	
1110	0100	Restore	
1100	1000	Shift	3
1111		Subtract	
1111	1001	$Q_0 = 1$	
1111	0010	Shift	4
0010		Subtract	
1111	0010	Restore	

(-7) / (-3)

Example: (5) / (3)

5 = 0101 3 = 0011
-5 = 1011 -3 = 1101

	ACCUMULATOR	DIVIDEND	DIVISOR
	A (Sign Extension)	Q (5)	M (3)
Initial Values	0000	0101	0011
Step 1:			
Left-Shift	0000	101_	
Sign (A, M) Same: A - M	+1101		
Sign Changes: Unsuccessful	1101		
Restore	0000	1010	
Step 2:			
Left-Shift	0001	010_	
Sign (A, M) Same: A - M	+1101		
Sign Changes: Unsuccessful	1110		
Restore	0001	0100	
Step 3:			
Left-Shift	0010	100_	
Sign (A, M) Same: A - M	+1101		
Sign Changes: Unsuccessful	1111		
Restore	0010	1000	
Step 4:			
Left-Shift	0101	000_	
Sign (A, M) Same: A - M	+1101		
Sign still Same: Successful	0010		
Restore not required	0010	0001	
	Remainder(2)	Quotient(1)	

Example: (-19) / (7)

19 = 010011 7 = 000111
-19 = 101101 -7 = 111001

	ACCUMULATOR A (Sign Extension)	DIVIDEND Q (-19)	DIVISOR M (7)
Initial Values	111111	101101	000111
Step 1:			
Left-Shift	111111	01101_	
Sign (A, M) Different: A + M	+000111		
Sign Changes: Unsuccessful	000110		
Restore	111111	011010	
Step 2:			
Left-Shift	111110	11010_	
Sign (A, M) Different: A + M	+000111		
Sign Changes: Unsuccessful	000101		
Restore	111110	110100	
Step 3:			
Left-Shift	111101	10100_	
Sign (A, M) Different: A + M	+000111		
Sign Changes: Unsuccessful	000100		
Restore	111101	101000	
Step 4:			
Left-Shift	111011	01000_	
Sign (A, M) Different: A + M	+000111		
Sign Changes: Unsuccessful	000010		
Restore	111011	010000	
Step 5:			
Left-Shift	110110	10000_	
Sign (A, M) Different: A + M	+000111		
Sign still Same: Successful	111101		
Restore not required	111101	100001	
Step 6:			
Left-Shift	111011	00001_	
Sign (A, M) Different: A + M	+000111		
Sign Changes: Unsuccessful	000010		
Restore	111011	000010	
	Remainder (-5)	Quotient (2)	

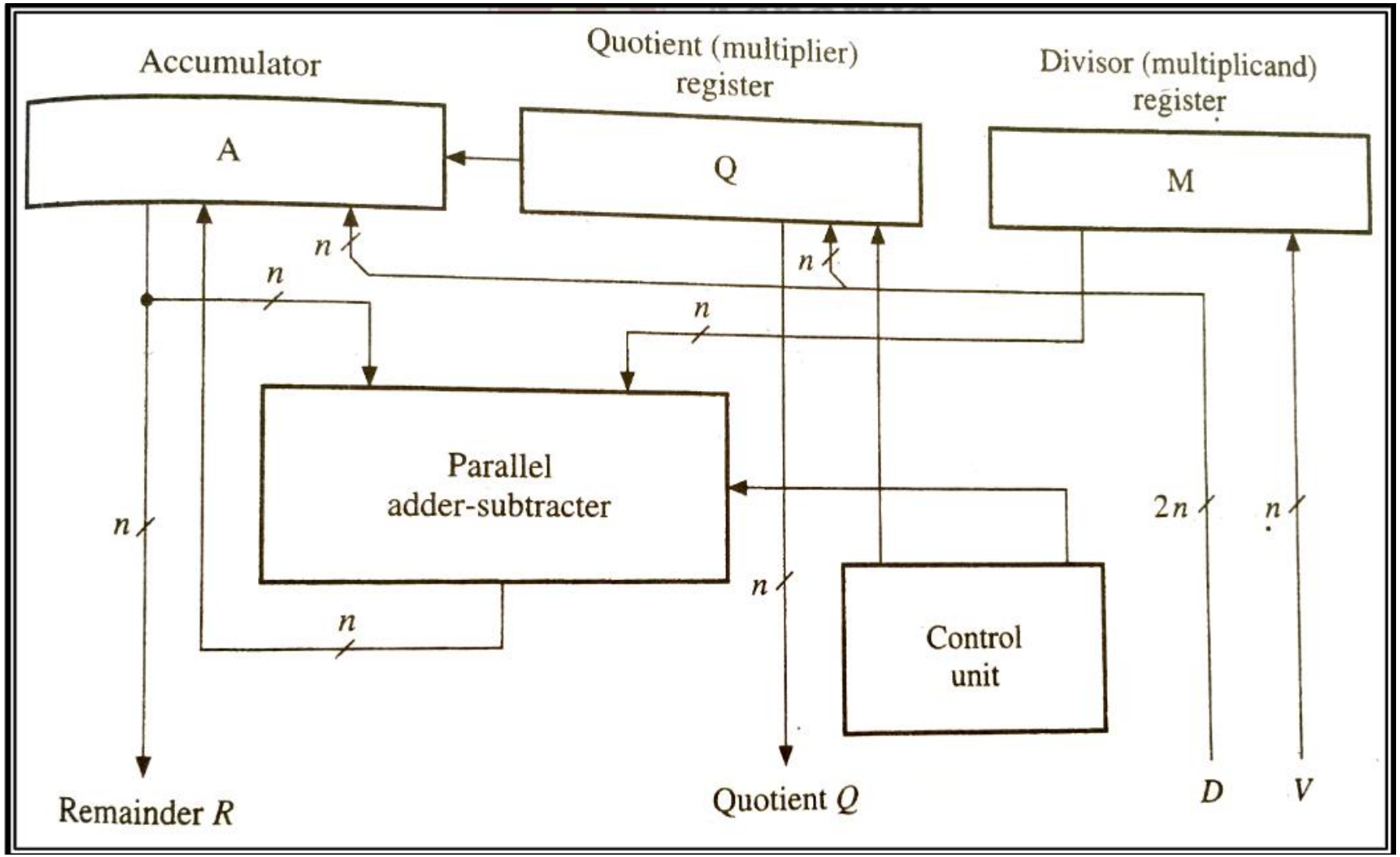
Example: (7) / (5)

Dividend (Q) = 7
Divisor (M) = 5
Accumulator (A) = 0

7 = 0111 5 = 0101
-7 = 1001 -5 = 1011

	ACCUMULATOR A (0)	DIVIDEND Q (7)	DIVISOR M (5)
Initial Values	0000	0111	0101
Step 1:			
Left-Shift	0000	111_	
A – M	+1011		
Unsuccessful (-ve)	1011	1110	
Next Step: "Add"			
Step 2:			
Left-Shift	0111	110_	
A + M	+0101		
Unsuccessful (-ve)	1100	1100	
Next Step: "Add"			
Step 3:			
Left-Shift	1001	100_	
A + M	+0101		
Unsuccessful (-ve)	1110	1000	
Next Step: "Add"			
Step 4:			
Left-Shift	1101	000_	
A + M	+0101		
Successful (+ve)	0010	0001	
	Remainder (2)	Quotient (1)	

configuration of the Restoring Algorithm



Non restoring algorithm

1) Let Q register hold the divided, M register holds the divisor and A register is 0.
2) On completion of the algorithm, Q will get the quotient and A will get the remainder.

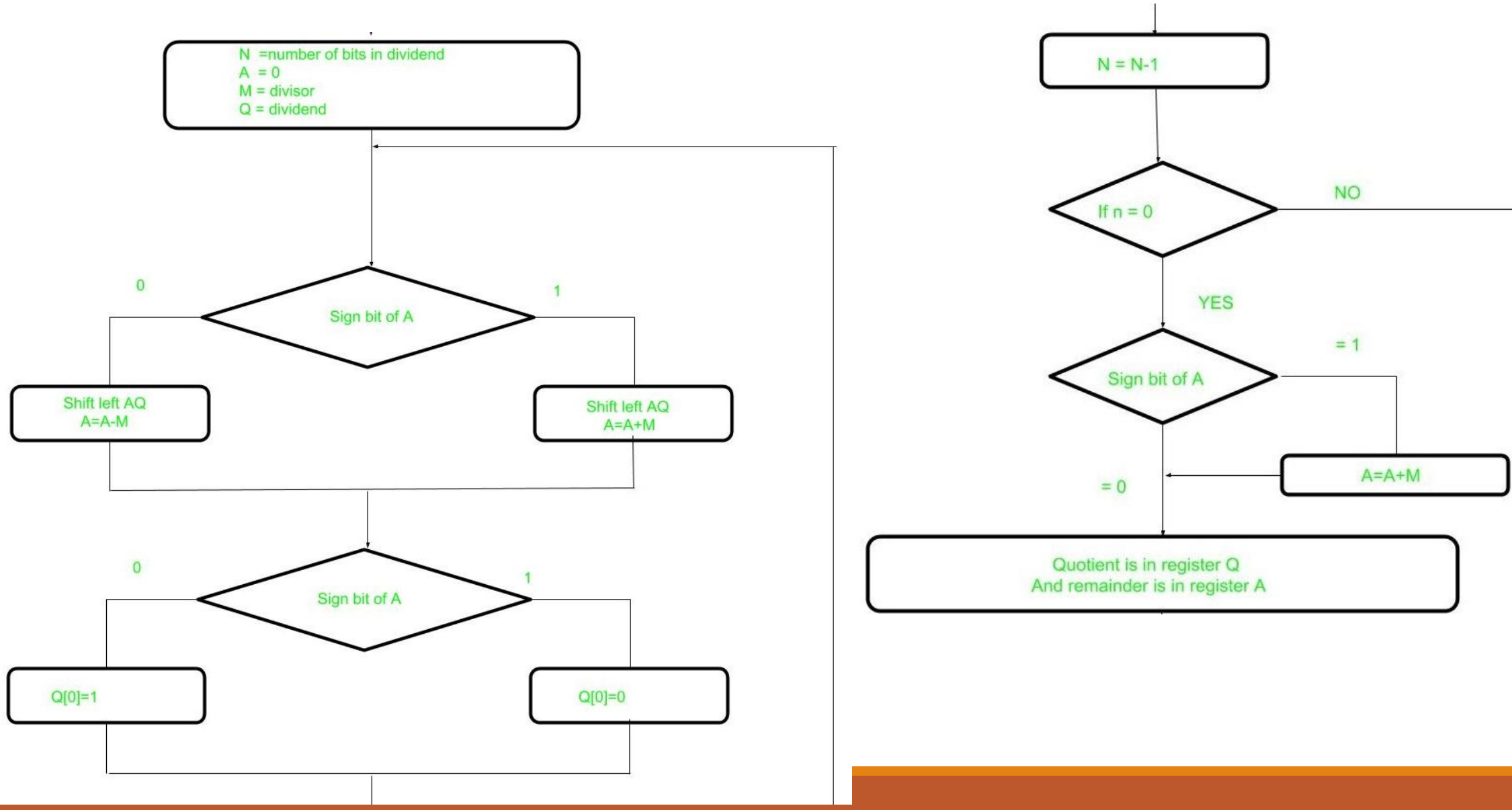
Algorithm:

The **number of steps** required is equal to the **number of bits in the Dividend**.

1) At each step, **left shift the dividend by 1 position**.
2) **Subtract the divisor from A** (perform A - M).
3) If the **result is positive** then the step is said to be **"Successful"**.
   In this case **quotient bit will be "1"** and **Restoration is NOT Required**.
   The **Next Step** will also be **Subtraction**.
4) If the **result is negative** then the step is said to be **"Unsuccessful"**.
   In this case **quotient bit will be "0".**
   Here Restoration is NOT Performed.
   Instead the next step will be ADDITION in place of subtraction.
   As restoration is not performed, the method is called Non-Restoring Division.

Repeat steps 1 to 4 for **all bits** of the Dividend.

Note: In the last step if the A value is negative after the operation then add 'M' to it (A=A+M) to get the true remainder


N =number of bits in dividend
A = 0
M = divisor
Q = dividend
Sign bit of A
0
1
Shift left AQ
A=A-M
Shift left AQ
A=A+M
Sign bit of A
0
1
Q[0]=1
Q[0]=0
N = N-1
If n = 0
NO
YES
Sign bit of A
= 1
A=A+M
= 0
Quotient is in register Q
And remainder is in register A

Example:

Perform Non-Restoring Division for Unsigned Integer

Dividend =11 Divisor =3 -M =11101

- **Step-1:** First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend)
- **Step-2:** Check the sign bit of register A
- **Step-3:** If it is 1 shift left content of AQ and perform A = A+M, otherwise shift left AQ and perform A = A-M (means add 2's complement of M to A and store it to A)
- **Step-4:** Again the sign bit of register A
- **Step-5:** If sign bit is 1 Q[0] become 0 otherwise Q[0] become 1 (Q[0] means least significant bit of register Q)
- **Step-6:** Decrements value of N by 1
- **Step-7:** If N is not equal to zero go to **Step 2** otherwise go to next step
- **Step-8:** If sign bit of A is 1 then perform A = A+M
- **Step-9:** Register Q contain quotient and A contain remainder

N	M	A	Q	ACTION
4	00011	00000	1011	Start
		00001	011_	Left shift AQ
		11110	011_	A=A-M

N	M	A	Q	ACTION
2		11111	1100	Q[0]=0
		11111	100_	Left Shift AQ
		00010	100_	A=A+M

N	M	A	Q	ACTION
3		11110	0110	Q[0]=0
		11100	110_	Left shift AQ
		11111	110_	A=A+M

N	M	A	Q	ACTION
1		00010	1001	Q[0]=1
		00101	001_	Left Shift AQ
		00010	001_	A=A-M

N	M	A	Q	ACTION
0		00010	0011	Q[0]=1

Example: 13/4

n	M	A	Q	Operation
4	00100	00000	1101	initialize
4	00100	00001	101_	shift left AQ
		11101	101_	A=A-M
		11101	1010	Q[0]=0
3	00100	11011	010_	shift left AQ
		11111	010_	A=A+M
		11111	0100	Q[0]=0
2	00100	11110	100_	shift left AQ
		00010	100_	A=A+M
		00010	1001	Q[0]=1
1	00100	00101	001_	shift left AQ
		00001	001_	A=A-M
		00001	0011	Q[0]=1

Example: 7/5

	Accumulator-A(0)	Dividend-Q([redacted])	Status
Initial Values	0000	0111	0101(M)
Step1:Left-Shift	0000	111_	
Operation:A – M	1011	1110	Unsuccessful(-ve) A+M in Next Step
Step2:Left-Shift	0111	110_	
Operation:A + M	1100	1100	Unsuccessful(-ve) A+M in Next Step
Step3:Left-Shift	1001	100_	
Operation:A + M	1110	1000	Unsuccessful(-ve) A+M in Next Step
Step4:Left-Shift	1101	000_	
Operation:A + M	0010	0001	Successful(+ve)
	Remainder(2)	Quotient(1)	

Dividend (A) = 101110, ie 46, and
Divisor (B) = 010111, ie 23.

Action	A	Q	Count
Initial	000 000	101 110	6
A > 0 => SHL (AQ)	000 001	011 10□	
A = A-M	101 010	011 10□	
A < 0 => Q0 = 0	101 010	011 100	5
A < 0 => SHL (AQ)	010 100	111 00□	
A = A+M	101 011	111 00□	
A < 0 => Q0 = 0	101 011	111 000	4
A < 0 => SHL (AQ)	010 111	110 00□	
A = A+M	101 110	110 00□	
A < 0 => Q0 = 0	101 110	110 000	3
A < 0 => SHL (AQ)	011 101	100 00□	
A = A+M	110 100	100 00□	
A < 0 => Q0 = 0	110 100	100 000	2
A < 0 => SHL (AQ)	101 001	000 00□	
A = A+M	000 000	000 00□	
A < 0 => Q0 = 1	000 000	000 001	1
A > 0 => SHL (AQ)	000 000	000 01□	
A = A+M	101 001	000 01□	
A < 0 => Q0 = 1	101 001	000 010	0
Count has reached Zero, So final steps			
A < 0 => A = A+M	000 000	000 010	
	Reminder	Quotient	

Floating Point Numbers

Floating Point Numbers

For example, 37.25 can be analyzed as:

10^1	10^0		10^{-1}		10^{-2}
Tens	Units		Tenths	Hundredths	
3	**7**		**2**	**5**	

37.25 = (3 x 10) + (7 x 1) + (2 x 1/10) + (5 x 1/100)

Binary Equivalence

The binary equivalent of a floating point number can be determined by computing the binary representation for each part separately.

1) For the **whole** part:

   Use subtraction or division method previously learned.

2) For the **fractional** part:

   Use the subtraction or multiplication method (to be shown next)

Fractional Part – Multiplication Method

In the binary representation of a floating point number the column values will be as follows:

…	2^5	2^4	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4} …
…	32	16	8	4	2	1	.	1/2	1/4	1/8	1/16…
…	32	16	8	4	2	1	.	.5	.25	.125	.0625…

Fractional Part – Multiplication Method

Ex 1. Find the binary equivalent of **0.25**

Step 1: Multiply the fraction by 2 until the fractional part becomes 0

.25

x 2

0.5

x 2

1.0

Step 2: Collect the whole parts in forward order. Put them after the radix point

. .5 .25 .125 .0625

. 0 1

Fractional Part – Multiplication Method

Ex 2. Find the binary equivalent of **0.625**

Step 1: Multiply the fraction by 2 until the fractional part becomes 0

.625

x 2

1.25

x 2

0.50

x 2

1.0

Step 2: Collect the whole parts in forward order. Put them after the radix point

. .5 .25 .125 .0625

. 1 0 1

Fractional Part – Subtraction Method

Start with the column values again, as follows:

… 2^0 .	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}…
… 1 .	1/2	1/4	1/8	1/16	1/32	1/64…
… 1 .	.5	.25	.125	.0625	.03125	.015625…

Fractional Part – Subtraction Method

Starting with 0.5, subtract the column values from left to right. Insert a 0 in the column if the value cannot be subtracted or 1 if it can be. Continue until the fraction becomes .0

Ex 1.

.25	.5	.25	.125	.0625
− .25	.0	1		
.0				

Binary Equivalent of FP number

Ex 2. Convert 37.25, using subtraction method.

64 32 16 8 4 2 1 . .5 .25 .125 .0625

2^6 2^5 2^4 2^3 2^2 2^1 2^0 . 2^{-1} 2^{-2} 2^{-3} 2^{-4}

1 0 0 1 . 0 0 1 1

37 .25

- 32 - .25 5

.0

- 4

1

-1

0

$37.25_{10} = 100101.01_2$

Binary Equivalent of FP number

Ex 3. Convert 18.625, using subtraction method.

64	32	16	8	4	2	1	.	.5	.25	.125	.0625
2^6	2^5	2^4	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}
		1	0	0	1	0		1	0	1	

18	.625
- 16	- .5
2	.125
- 2	- .125
0	0

$$18.625_{10} = 10010.101_2$$

Problem storing binary form

Possibility to store the radix point is not available

Large numbers requires much space

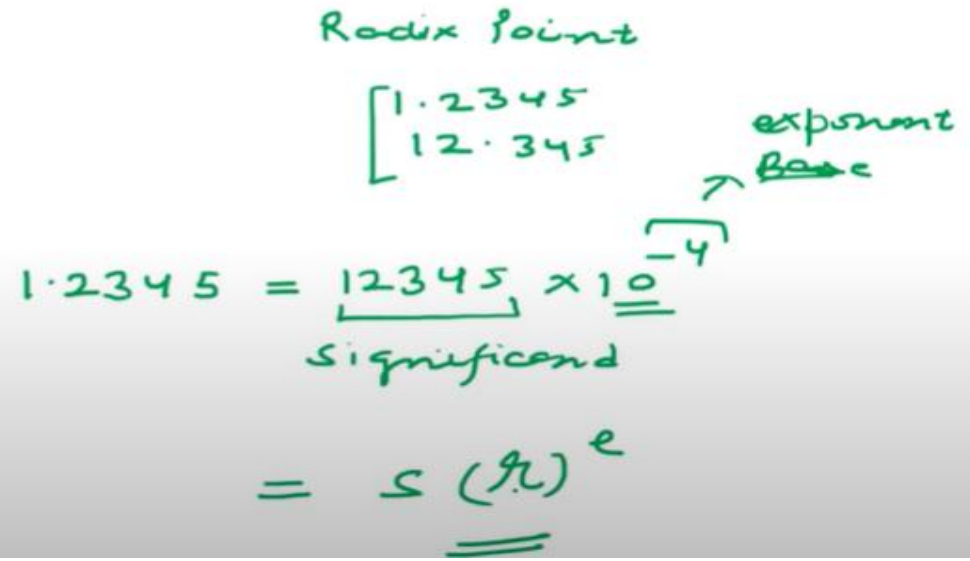
Standards committee came up with a solution to store floating point numbers (that have a decimal point)

Floating point numbers

- A **Floating Point** number **usually has a decimal point**
- This means that 0, 3.14, 6.5, and -125.5 are Floating Point numbers
- Floating Point numbers **represent** a **wide variety of numbers** their precision varies

General representation of Floating Point number:

- A **significand:** contains the number's digits, negative significands represent negative numbers
- An **exponent:** where the decimal (or binary) point is placed relative to the beginning of the significand, negative exponents represent numbers that are very small



- In the decimal system, a decimal point (**radix point**) separates the whole numbers from the fractional part
- Examples:37.25 (whole = 37, fraction = 25/100)

  123.567

  10.12345678

- Various **Floating Point** representations are used in computers
- In year **1985**, **IEEE 754 Standard** for **Floating Point Arithmetic** was established
- Since **1990**, it is **most commonly encountered** representation

The **IEEE Standard** for **Floating-Point Arithmetic (IEEE 754)** is a **technical standard** for floating-point computation which was established in 1985 by the **Institute of Electrical and Electronics Engineers (IEEE)**

IEEE 754 has 3 basic components:

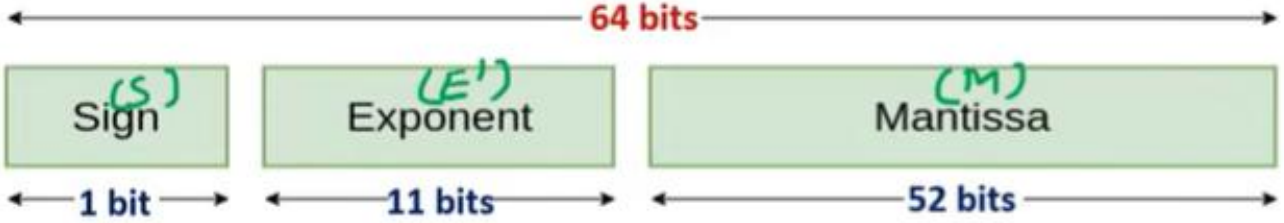
- **Sign of Mantissa (S):**
  0 represents a positive number while 1 represents a negative number

- **Biased/Modified exponent (E'):**
  The exponent field needs to represent both positive and negative exponents. A bias is added to the actual exponent in order to get the stored exponent

- **Normalized Mantissa (M):**
  The mantissa is part of a number in scientific notation or a floating-point number, consisting of its significant digits. Here we have only 2 digits, i.e. O and 1. So a normalized mantissa is one with only one 1 to the left of **The decimal**

Double Precision Floating Point Number

Format:



- 64 bits
- Largest number can be: 9,223,372,036,854,775,807
- Smallest number can be: - 9,223,372,036,854,775,807
- **E' (Modified Exponent) = e + Bias (1023)**

Example: $(1460.125)_{10}$

Solution:- $(1460.125)_{10} = (10110110100.001)_2$

$= 1.0110110100001 * 2^{10}$

- **S = 0**
- **e = 10**
- **M = 0110110100001**

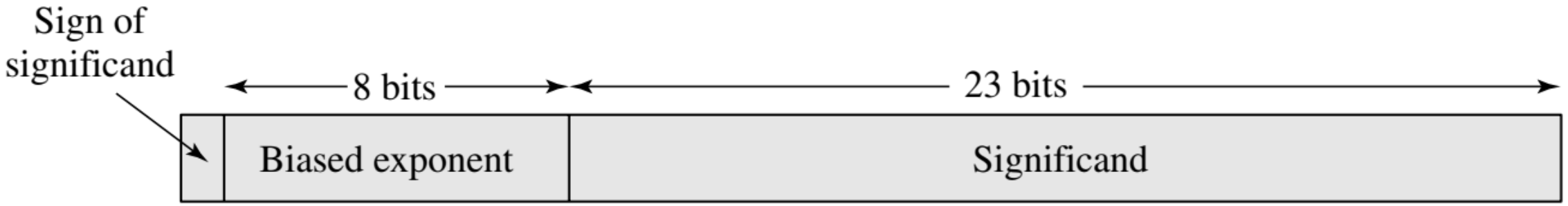
S	E'	M
0	10000001001	0110110100001 - - - - 0

← 52-bits →

$E' = e + 1023 = 10 + 1023$

$= (1033)_{10}$

$= (10000001001)_2$



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

- Sign
- Base = 2
- Exponent = Value of the 8bit exponent – Bias
  (where Bias = $2^{k-1}-1$, k = no. of bits in the exponent)
- Significand

Storing the Binary Form

How to store a radix point?

- All we have are zeros and ones…

Make sure that the radix point is ALWAYS in the same position within the number.

Use the IEEE 32-bit standard

→ the leftmost digit must be a 1

Solution is Normalization

Every binary number, **except the one corresponding to the number zero**, can be normalized by choosing the exponent so that the radix point falls to the right of the leftmost 1 bit.

$37.25_{10} = 100101.01_2 = 1.0010101 \times \mathbf{2^5}$

$7.625_{10} = 111.101_2 = 1.11101 \times \mathbf{2^2}$

$0.3125_{10} = 0.0101_2 = 1.01 \times \mathbf{2^{-2}}$

IEEE Floating Point Representation

The second field of the floating point number will be the **exponent**.

The exponent is stored as an unsigned 8-bit number, RELATIVE to a **bias of 127.**

- Exponent 5 is stored as (127 + 5) or 132
  - 132 = 10000100
- Exponent -5 is stored as (127 + (-5)) or 122
  - 122 = 01111010

Try It Yourself

How would the following exponents be stored (8-bits, 127-biased):

2^{-10}

2^{8}

(Answers on next slide)

Answers

2^{-10}

exponent -10 8-bit

bias +127 value

117 → 01110101

2^{8}

exponent 8 8-bit

bias +127 value

135 → 10000111

IEEE Floating Point Representation

The **significand** or **mantissa** is the set of 0's and 1's to the right of the radix point of the **normalized** (when the digit to the left of the radix point is 1) binary number.

Ex: $1.\mathbf{00101} \times 2^3$

(The mantissa is 00101)

- The mantissa is stored in a 23 bit field, so we add zeros to the right side and store:

0010100000000000000000

Decimal Floating Point to IEEE standard Conversion

Ex 1: Find the IEEE FP representation of 40.15625

Step 1.

Compute the binary equivalent of the whole part and the fractional part. (i.e. convert 40 and .15625 to their binary equivalents)

Decimal Floating Point to IEEE standard Conversion

40			.15625	
- 32	**Result:**		-.12500	**Result:**
8	**101000**		.03125	**.00101**
- 8			-.03125	
0			.0	

So: $40.15625_{10} = 101000.00101_{2}$

Decimal Floating Point to IEEE standard Conversion

Step 2. Normalize the number by moving the decimal point to the right of the leftmost one.

$101000.00101 = 1.0100000101 \times \mathbf{2^5}$

Decimal Floating Point to IEEE standard Conversion

Step 3. Convert the exponent to a biased exponent

$$127 + 5 = 132$$

And convert biased exponent to 8-bit unsigned binary:

$$132_{10} = 10000100_{2}$$

Decimal Floating Point to IEEE standard Conversion

Step 4. Store the results from steps 1-3:

Sign	Exponent (from step 3)	Mantissa (from step 2)
0	**10000100**	**01000001010000000000000**

Decimal Floating Point to IEEE standard Conversion

Ex 2: Find the IEEE FP representation of **–24.75**

Step 1. Compute the binary equivalent of the whole part and the fractional part.

24			.75	
- 16	Result:	- .50	Result:	
8	11000		.25	.11
- 8			- .25	
0			.0	

So: $-24.75_{10} = -11000.11_{2}$

Decimal Floating Point to IEEE standard Conversion

Step 2.

Normalize the number by moving the decimal point to the right of the leftmost one.

$$-11000.11 = -1.100011 \times 2^4$$

Decimal Floating Point to IEEE standard Conversion

Step 3. Convert the exponent to a biased exponent

127 + 4 = 131

==> $131_{10} = 10000011_2$

Step 4. Store the results from steps 1-3

Sign	Exponent	mantissa
1	**10000011**	**1000110..0**

IEEE standard to Decimal Floating Point Conversion

Do the steps in reverse order

In reversing the normalization step move the radix point the number of digits equal to the exponent:

- If exponent is positive, move to the right
- If exponent is negative, move to the left

IEEE standard to Decimal Floating Point Conversion

Ex 1: Convert the following 32-bit binary number to its decimal floating point equivalent:

Sign	Exponent	Mantissa
1	01111101	010..0

IEEE standard to Decimal Floating Point Conversion

Step 1: Extract the biased exponent and unbias it

Biased exponent = 01111101_2 = 125_{10}

Unbiased Exponent: 125 – 127 = -2

IEEE standard to Decimal Floating Point Conversion..

Step 2: Write Normalized number in the form:

$$(-1)^{S}\ 1.\underline{\quad\text{Mantissa}\quad} \times 2^{\text{----}\ \text{Exponent}}$$

For our number:

$$-1.01 \times 2^{-2}$$

IEEE standard to Decimal Floating Point Conversion.

Step 3: Denormalize the binary number from step 2 (i.e. move the decimal and get rid of ($x\ 2^n$) part):

-0.0101_2 (negative exponent – move left)

Step 4: Convert binary number to the FP equivalent (i.e. Add all column values with 1s in them)

$$-0.0101_2 = -(0.25 + 0.0625)$$

$$= -0.3125_{10}$$

IEEE standard to Decimal Floating Point Conversion.

Ex 2: Convert the following 32 bit binary number to its decimal floating point equivalent:

Sign	Exponent	Mantissa
0	10000011	10011000..0

IEEE standard to Decimal Floating Point Conversion..

Step 1: Extract the biased exponent and unbias it

Biased exponent = 1000011_2 = 131_{10}

Unbiased Exponent: 131 – 127 = 4

IEEE standard to Decimal Floating Point Conversion..

Step 2: Write Normalized number in the form:

1 . _____ x 2 ----

Mantissa

Exponent

For our number:

1.10011×2^{4}

IEEE standard to Decimal Floating Point Conversion.

Step 3: Denormalize the binary number from step 2 (i.e. move the decimal and get rid of ($x\ 2^n$) part:

11001.1_2 (positive exponent – move right)

Step 4: Convert binary number to the FP equivalent (i.e. Add all column values with 1s in them)

11001.1= 16 + 8 + 1 +.5

= 25.5_{10}

Examples

Express the following numbers in IEEE 32-bit floating-point format:

a. -5 **b.** -6 **c.** -1.5 **d.** 384 **e.** 1/16 **f.** $-1/32$

The following numbers use the IEEE 32-bit floating-point format. What is the equivalent decimal value?

a. 1 10000011 11000000000000000000000

b. 0 01111110 10100000000000000000000

c. 0 10000000 00000000000000000000000