UNIT II-Arithmetic Operations

CO Nos.	Course Outcome(s)	Level of learning domain (Based on revised Bloom’s)
CO2	Familiarize with arithmetic algorithms and procedure for implementing them in hardware.	K2

NUMBER SYSTEM

ADDITION AND SUBTRACTION

Addition/ subtraction of signed numbers

x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

At the i^{th} stage:

Input:

c_i is the carry-in

Output:

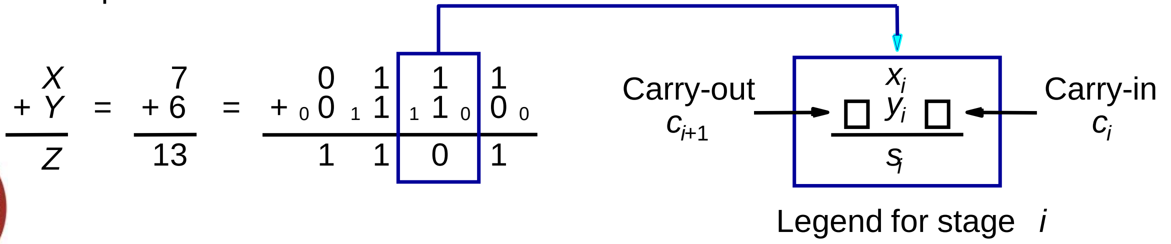
s_i is the sum

c_{i+1} carry-out to $(i+1)^{st}$ state

$$s_i = \bar{x}_i \bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

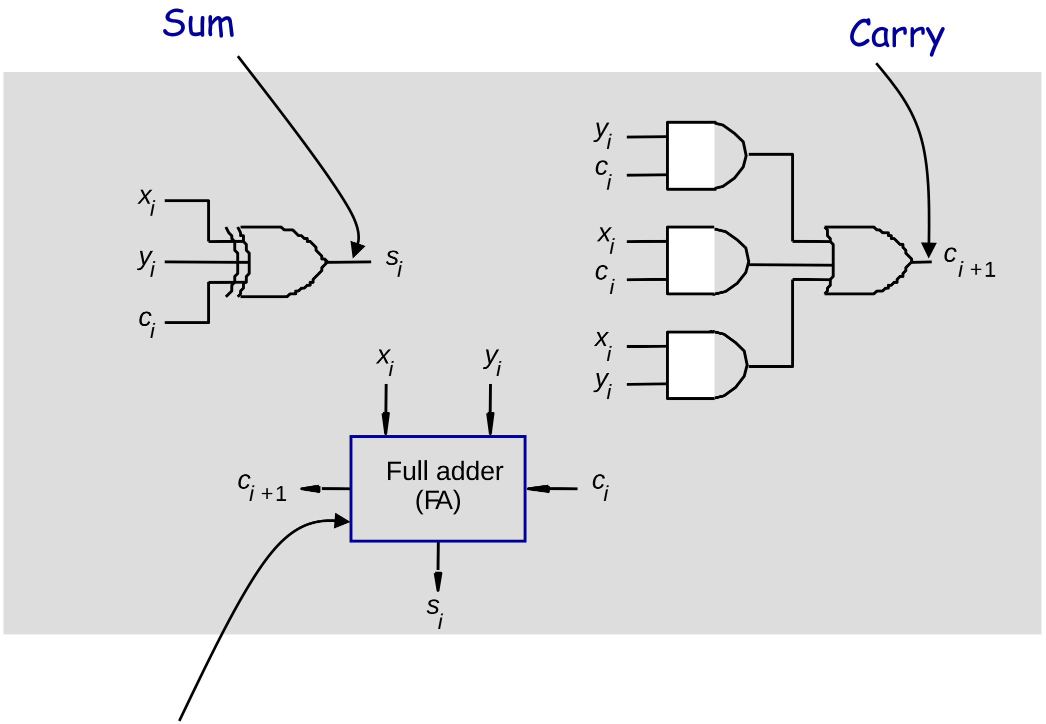
Example:



$$\begin{array}{r} X \\ +\,Y \\ \hline Z \end{array} = \begin{array}{r} 7 \\ +\,6 \\ \hline 13 \end{array} = \begin{array}{rcccc} & 0 & 1 & 1 & 1 \\ + & {}_0\,0 & {}_1\,1 & {}_1\,1 & {}_0\,0\ {}_0 \\ \hline & 1 & 1 & 0 & 1 \end{array}$$

Legend for stage i

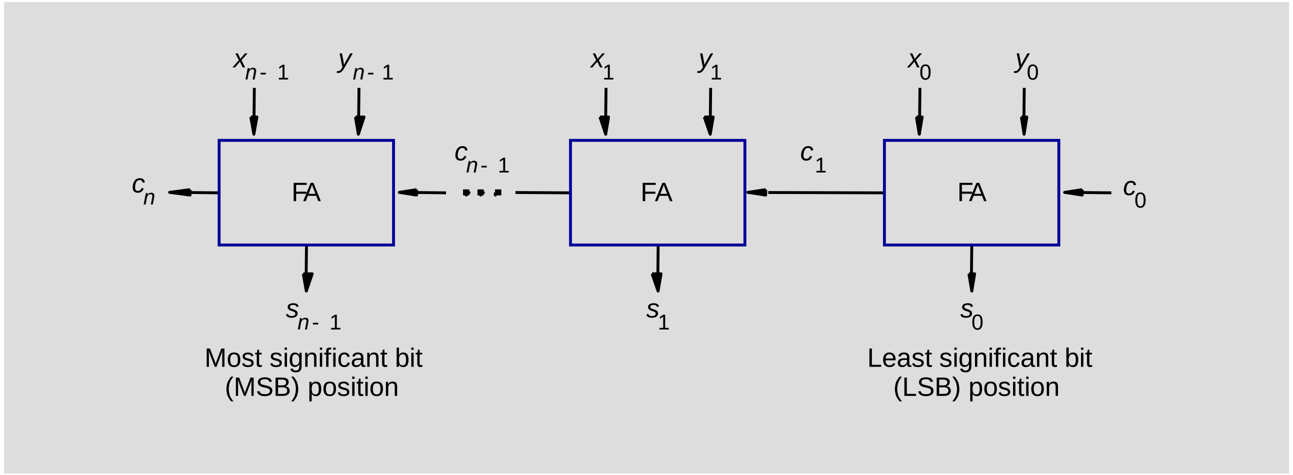
Addition logic for a single stage



Full Adder (FA): Symbol for the complete circuit for a single stage of addition.

n-bit adder

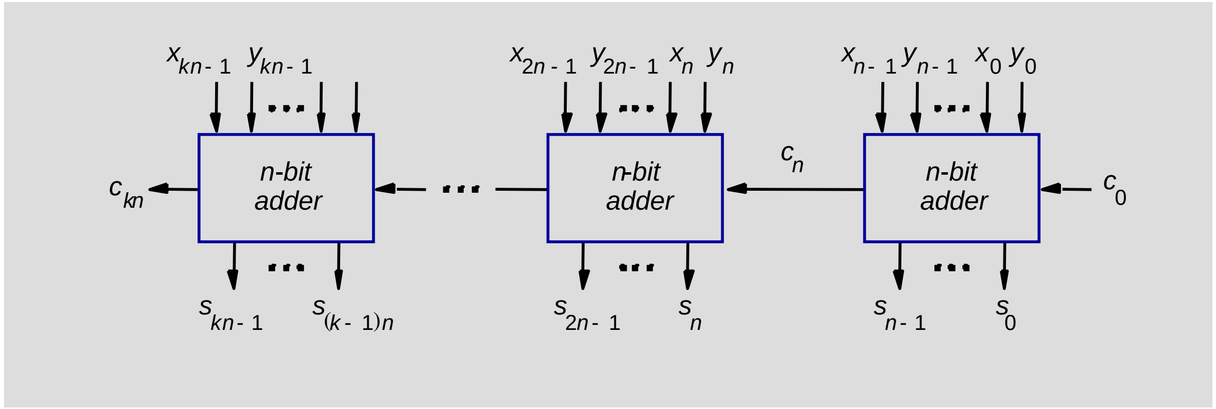
- Cascade *n* full adder (FA) blocks to form a *n*-bit adder.
- Carries propagate or ripple through this cascade, *n*-bit ripple carry adder.



- Carry-in c_0 into the LSB position provides a convenient way to perform subtraction.

K n-bit adder

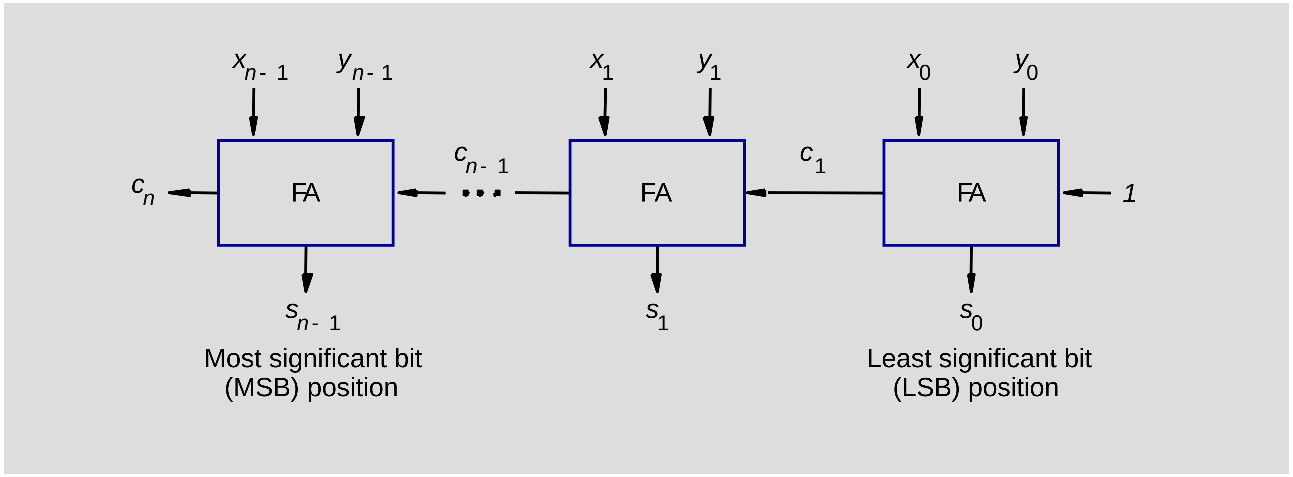
- *K n*-bit numbers can be added by cascading *k n*-bit adders.



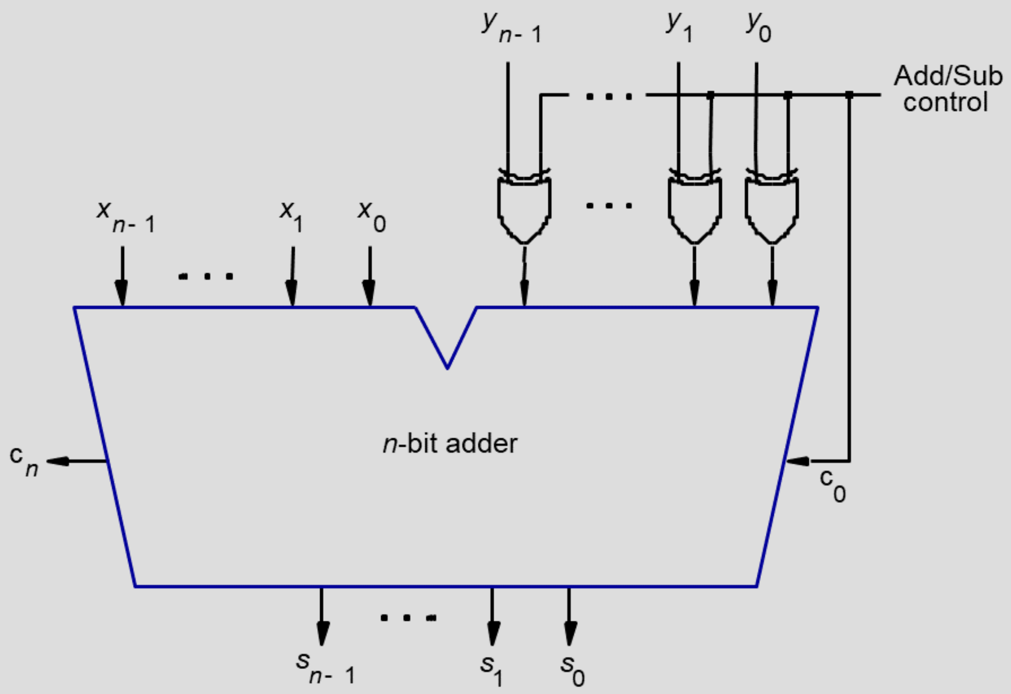
- Each *n*-bit adder forms a block, so this is cascading of blocks.
- Carries ripple or propagate through blocks, Blocked Ripple Carry Adder

n-bit subtractor

- Recall $X - Y$ is equivalent to adding 2’s complement of Y to X.
- 2’s complement is equivalent to 1’s complement + 1.
- $X - Y = X + Y + 1$
- 2’s complement of positive and negative numbers is computed similarly.



n-bit adder/subtractor (contd..)



- Add/sub control = 0, addition.
- Add/sub control = 1, subtraction.

Detecting overflows

- Overflows can only occur when the sign of the two operands is the same.
- Overflow occurs if the sign of the result is different from the sign of the operands.
- Recall that the MSB represents the sign.
  - x_{n-1}, y_{n-1}, s_{n-1} represent the sign of operand *x,* operand *y* and result *s* respectively.
- Circuit to detect overflow can be implemented by the following logic expressions:

$$Overflow = x_{n-1}y_{n-1}\bar{s}_{n-1} + \bar{x}_{n-1}\bar{y}_{n-1}s_{n-1}$$

$$Overflow = c_n \oplus c_{n-1}$$

MULTIPLICATION

Multiplication of unsigned numbers

```
            1  1  0  1        (13)  Multiplicand M
            1  0  1  1        (11)  Multiplier Q
         ------------
            1  1  0  1
         1  1  0  1
      0  0  0  0
   1  1  0  1
-----------------------
1  0  0  0  1  1  1  1        (143)  Product P
```

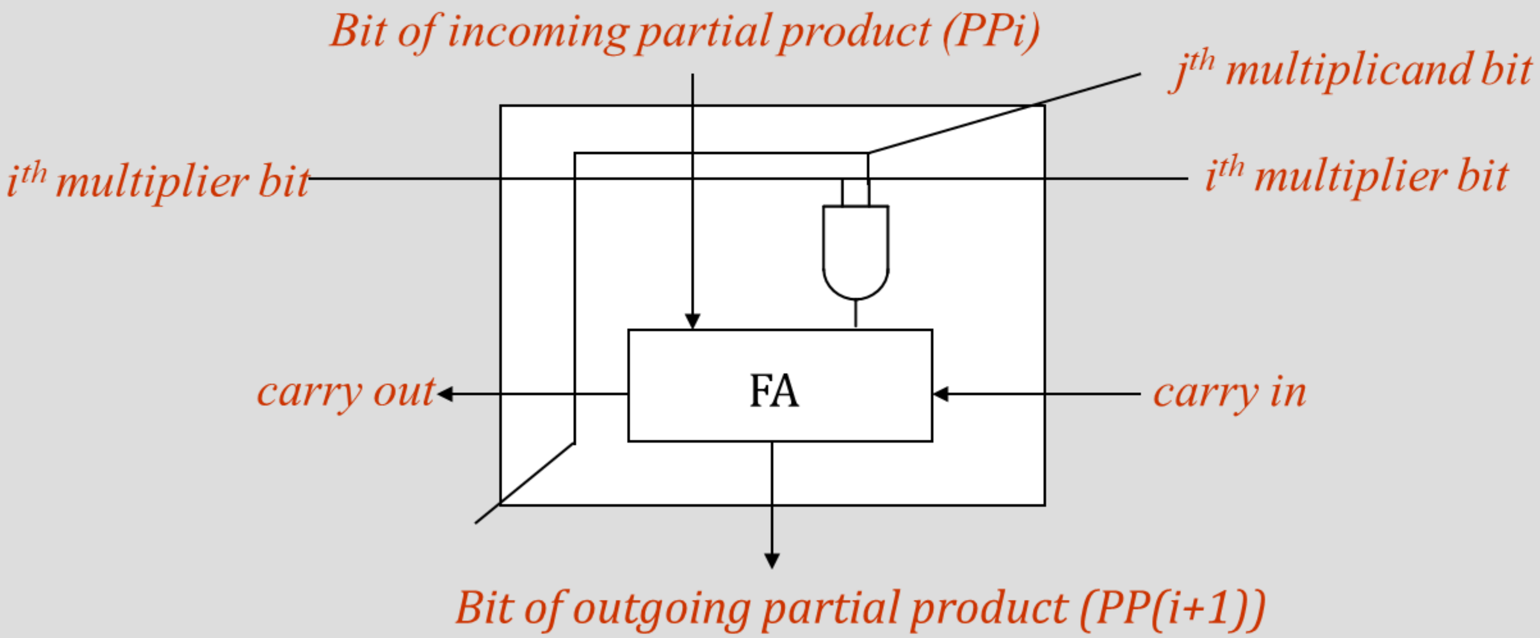
- Product of 2 *n*-bit numbers is at most a *2n*-bit number.
- Unsigned multiplication can be viewed as addition of shifted versions of the multiplicand.

Multiplication of unsigned numbers *(Contd.,)*

- We added the partial products at end.
  - Alternative would be to add the partial products at each stage.
- Rules to implement multiplication are:
  - If the *i*th bit of the multiplier is 1, shift the multiplicand and add the shifted multiplicand to the current value of the partial product.
  - Hand over the partial product to the next stage
  - Value of the partial product at the start stage is 0.

Multiplication of unsigned numbers

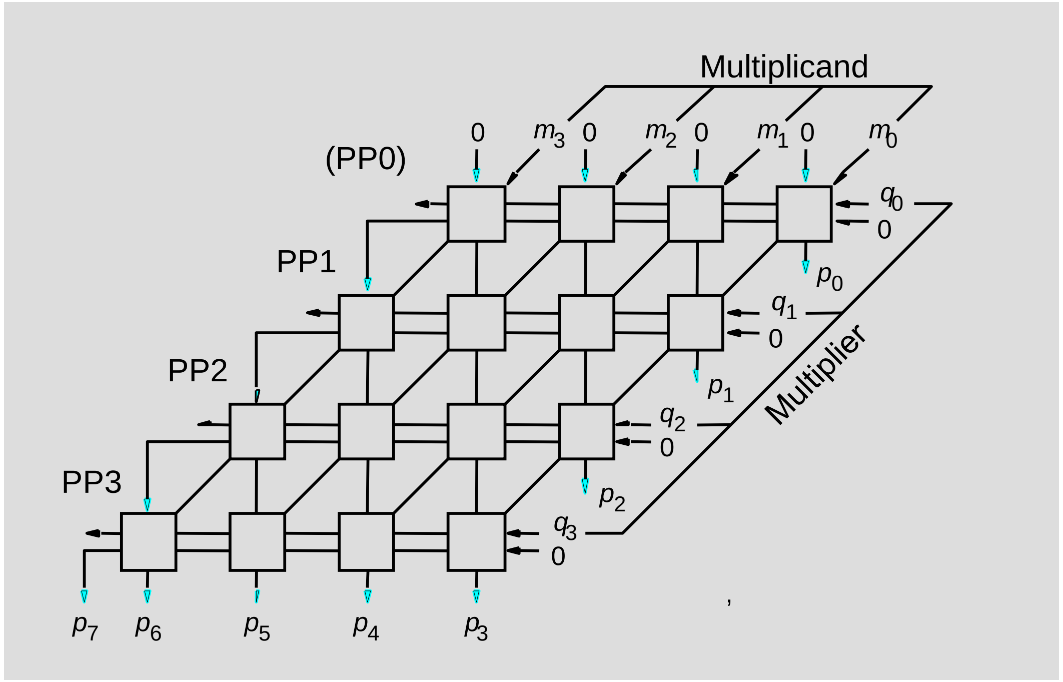
Typical multiplication cell



Bit of incoming partial product (PPi)
jth multiplicand bit
ith multiplier bit
ith multiplier bit
FA
carry out
carry in
Bit of outgoing partial product (PP(i+1))

Combinatorial array multiplier

Combinatorial array multiplier



Product is: $p_7, p_6, ..p_0$

Multiplicand is shifted by displacing it through an array of adders.

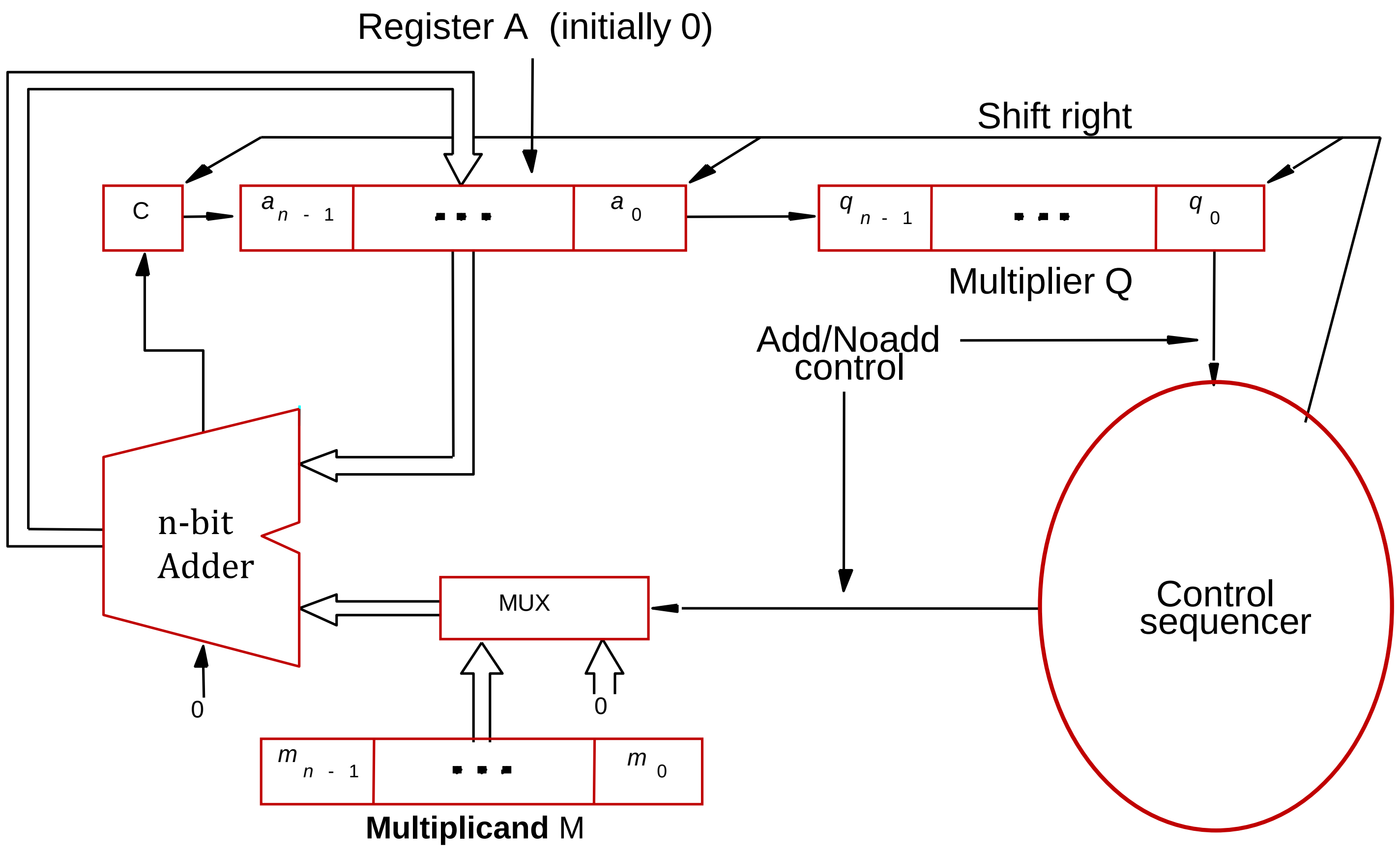
Combinatorial array multiplier *(Contd.,)*

- Combinatorial array multipliers are:
  - Extremely inefficient.
  - Have a high gate count for multiplying numbers of practical size such as 32-bit or 64-bit numbers.
  - Perform only one function, namely, unsigned integer product.
- Improve gate efficiency by using a mixture of combinatorial array techniques and sequential techniques requiring less combinational logic.

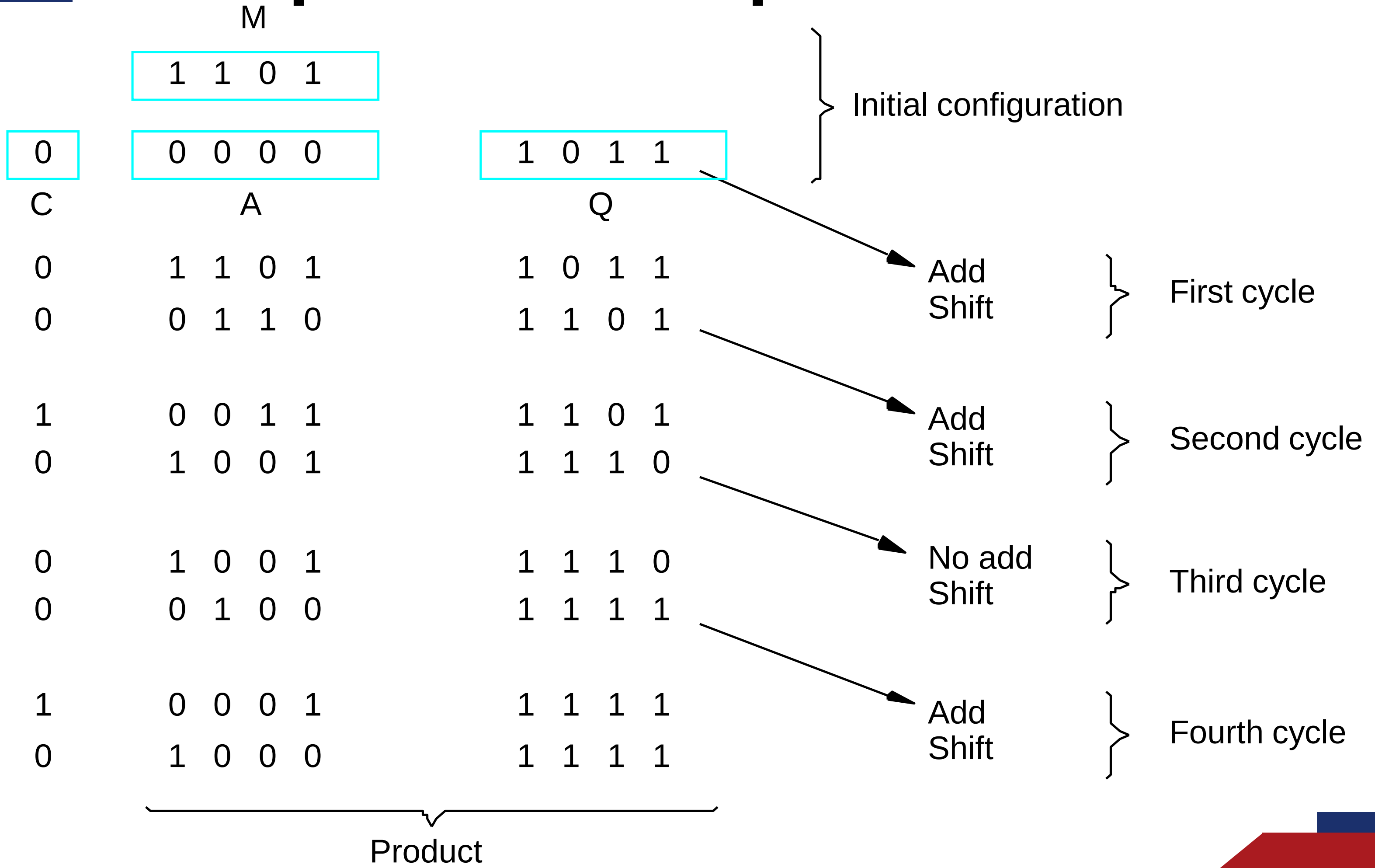
Sequential multiplication

- Recall the rule for generating partial products:
  - If the ith bit of the multiplier is 1, add the appropriately shifted multiplicand to the current partial product.
  - Multiplicand has been shifted left when added to the partial product.
- However, adding a left-shifted multiplicand to an unshifted partial product is equivalent to adding an unshifted multiplicand to a right-shifted partial product.

Sequential Circuit Multiplier



Sequential multiplication *(Contd.,)*



M: 1 1 0 1

C	A	Q		
0	0 0 0 0	1 0 1 1		Initial configuration
0	1 1 0 1	1 0 1 1	Add	First cycle
0	0 1 1 0	1 1 0 1	Shift	First cycle
1	0 0 1 1	1 1 0 1	Add	Second cycle
0	1 0 0 1	1 1 1 0	Shift	Second cycle
0	1 0 0 1	1 1 1 0	No add	Third cycle
0	0 1 0 0	1 1 1 1	Shift	Third cycle
1	0 0 0 1	1 1 1 1	Add	Fourth cycle
0	1 0 0 0	1 1 1 1	Shift	Fourth cycle

Product: 1 0 0 0 1 1 1 1

Signed Multiplication

- Considering 2’s-complement signed operands, what will happen to (-13)×(+11) if following the same method of unsigned multiplication?

```
                    1  0  0  1  1   (-13)
                    0  1  0  1  1   (+11)
                ------------------
 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
 0  0  0  0  0  0
-----------------------------
 1  1  0  1  1  1  0  0  0  1   (-143)
```

Sign extension is shown in blue

Sign extension of negative multiplicand.

Signed Multiplication

- For a negative multiplier, a straightforward solution is to form the 2’s-complement of both the multiplier and the multiplicand and proceed as in the case of a positive multiplier.
- This is possible because complementation of both operands does not change the value or the sign of the product.
- A technique that works equally well for both negative and positive multipliers – Booth algorithm.

BOOTH'S ALGORITHM

Booth’s Algorithm

Points to remember

- When using Booth's Algorithm:
  - You will need twice as many bits in your **product** as you have in your original two **operands**.
  - The **leftmost bit** of your operands (both your multiplicand and multiplier) is a SIGN bit, and cannot be used as part of the value.

To begin

- Decide which operand will be the **multiplier** and which will be the **multiplicand**
- Convert both operands to **two's complement** representation using X bits
  - X must be at least one more bit than is required for the binary representation of the numerically larger operand
- Begin with a product that consists of the multiplier with an additional X leading zero bits

Booth's Algorithm - Example

- There is an example of multiplying **2 x (-5)**
- For our example, let's reverse the operation, and multiply (**-5) x 2**
- The numerically larger operand (5) would require 3 bits to represent in binary (101). So we must use AT LEAST 4 bits to represent the operands, to allow for the sign bit.
- Let's use 5-bit 2's complement:
- -5 is 11011 (multiplier)
- 2 is 00010 (multiplicand)

Beginning Product

- The multiplier is:

  11011

- Add 5 leading zeros to the **multiplier** to get the **beginning product**:

  00000 11011

Step 1 for each pass

- Use the **LSB** (least significant bit) and the **previous LSB** to determine the arithmetic action.
  - If it is the FIRST pass, use **0** as the previous LSB.
- Possible arithmetic actions:
  - **00** → no arithmetic operation
  - **01** → add multiplicand to left half of product
  - **10** → subtract multiplicand from left half of product
  - **11** → no arithmetic operation

Step 2 for each pass

- Perform an **Arithmetic Shift Right (ASR)** on the entire product.
- NOTE: For X-bit operands, Booth's algorithm requires X passes.

- Let's continue with our example of multiplying **(-5) x 2**
- Remember:
  - -5 is 11011 (multiplier)
  - 2 is 00010 (multiplicand)

- And we added 5 leading zeros to the **multiplier** to get the **beginning product**:

  00000 11011
- Initial Product and previous LSB

  00000 11011 0

(Note: Since this is the first pass, we use 0 for the previous LSB)

- Pass 1, Step 1: Examine the last 2 bits

  00000 11011 0

  - The last two bits are 10, so we need to:
    - subtract the **multiplicand** from left half of product
  - Pass 1, Step 1: Arithmetic action

```
(1)  00000        (left half of product)
    -00010        (mulitplicand)
     11110        (uses a phantom borrow)
```

  - Place result into **left half** of product

  11110 11011 0

- Pass 1, Step 2: ASR (arithmetic shift right)

  Before ASR

  11110 11011 0

  After ASR

  11111 01101 1

  (left-most bit was 1, so a 1 was shifted in on the left)

Pass 1 is complete.

Current Product and previous LSB

11111 01101   1

- Pass 2, Step 1: Examine the last 2 bits

  11111 01101   1

  The last two bits are 11, so we do NOT need to perform an arithmetic action -- just proceed to step 2.

- Pass 2, Step 2: ASR (arithmetic shift right)

  Before ASR

  11111 01101   1

  After ASR

  11111 10110   1

(left-most bit was 1, so a 1 was shifted in on the left)

Pass 2 is complete.

Current Product and previous LSB

11111 10110 1

- Pass 3, Step 1: Examine the last 2 bits

11111 10110 1

The last two bits are 01, so we need to:

add the **multiplicand** to the left half of the product

- Pass 3, Step 1: Arithmetic action

```
(1) 11111        (left half of product)
   +00010        (mulitplicand)
    00001        (drop the leftmost carry)
```

Place result into **left half** of product

00001 10110 1

- Pass 3, Step 2: ASR (arithmetic shift right)

  Before ASR

  00001 10110 1

  After ASR

  00000 11011 0

  (left-most bit was 0, so a 0 was shifted in on the left)

Pass 3 is complete.

Current Product and previous LSB

00000 11011 0

- Pass 4, Step 1: Examine the last 2 bits

  00000 11011 0

  The last two bits are 10, so we need to:

  subtract the **multiplicand** from the left half of the product

- Pass 4, Step 1: Arithmetic action

```
(1) 00000      (left half of product)
   -00010      (multiplicand)
    11110  (uses a phantom borrow)
```

Place result into **left half** of product

11110 11011 0

Current Product and previous LSB

11111 01101 1

- Pass 5, Step 1: Examine the last 2 bits

  11111 01101 1

  The last two bits are 11, so we do NOT need to perform an arithmetic action -- just proceed to step 2.

- Pass 5, Step 2: ASR (arithmetic shift right)

  Before ASR

  11111 01101 1

  After ASR

  11111 10110 1

  (left-most bit was 1, so a 1 was shifted in on the left)

Pass 5 is complete.

Final Product

- We have completed 5 passes on the 5-bit operands, so we are done.
- Dropping the previous LSB, the resulting **final product** is:

11111 10110

Verification

- To confirm we have the correct answer, convert the 2's complement **final product** back to decimal.

Final product: **11111 10110**

Decimal value: **-10**

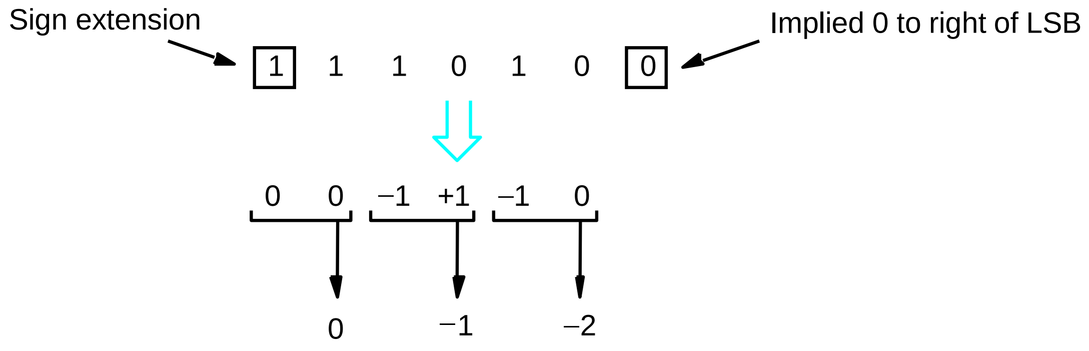
which is the CORRECT product of:

(-5) x 2

FAST MULTIPLICATION –
Bit-Pair Recoding of Multipliers

Bit-Pair Recoding of Multipliers

- Bit-pair recoding halves the maximum number of summands (versions of the multiplicand).



Example of bit-pair recoding derived from Booth recoding

Bit-Pair Recoding of Multipliers (Contd.,)

Multiplier bit-pair		Multiplier bit on the right	Multiplicand selected at position i
$i+1$	i	$i-1$	
0	0	0	0 X M
0	0	1	+ 1 X M
0	1	0	+ 1 X M
0	1	1	+ 2 X M
1	0	0	− 2 X M
1	0	1	− 1 X M
1	1	0	− 1 X M
1	1	1	0 X M

Table of multiplicand selection decisions

Bit-Pair Recoding of Multipliers (Contd.,)

```
      0  1  1  0  1   (+13)
 ´    1  1  0  1  0   (-6)
 ---------------------
```

$\Rightarrow$

```
                    0  1  1  0  1
                    0 -1 +1 -1  0
                   ---------------
 0  0  0  0  0  0  0  0  0  0
 1  1  1  1  1  0  0  1  1
 0  0  0  0  1  1  0  1
 1  1  1  0  0  1  1
 0  0  0  0  0  0
 -----------------------------
 1  1  1  0  1  1  0  0  1  0   (-78)
```

$\Downarrow$

```
                    0  1  1  0  1
                    0    -1    -2
                   ---------------
 1  1  1  1  1  0  0  1  1  0
 1  1  1  1  0  0  1  1
 0  0  0  0  0  0
 -----------------------------
 1  1  1  0  1  1  0  0  1  0
```

INTEGER DIVISION

Manual Division

$$\begin{array}{r r}
 & 21 \\ \cline{2-2}
13 \,\big) & 274 \\
 & 26 \\ \cline{2-2}
 & 14 \\
 & 13 \\ \cline{2-2}
 & 1
\end{array}
\qquad\qquad
\begin{array}{r l}
 & 10101 \\ \cline{2-2}
1101 \,\big) & 100010010 \\
 & 1101 \\ \cline{2-2}
 & 10000 \\
 & 1101 \\ \cline{2-2}
 & 1110 \\
 & 1101 \\ \cline{2-2}
 & 1
\end{array}$$

Longhand Division Steps

- Position the divisor appropriately with respect to the dividend and performs a subtraction.
- If the remainder is zero or positive, a quotient bit of 1 is determined, the remainder is extended by another bit of the dividend, the divisor is repositioned, and another subtraction is performed.
- If the remainder is negative, a quotient bit of 0 is determined, the dividend is restored by adding back the divisor, and the divisor is repositioned for another subtraction.

Circuit Arrangement

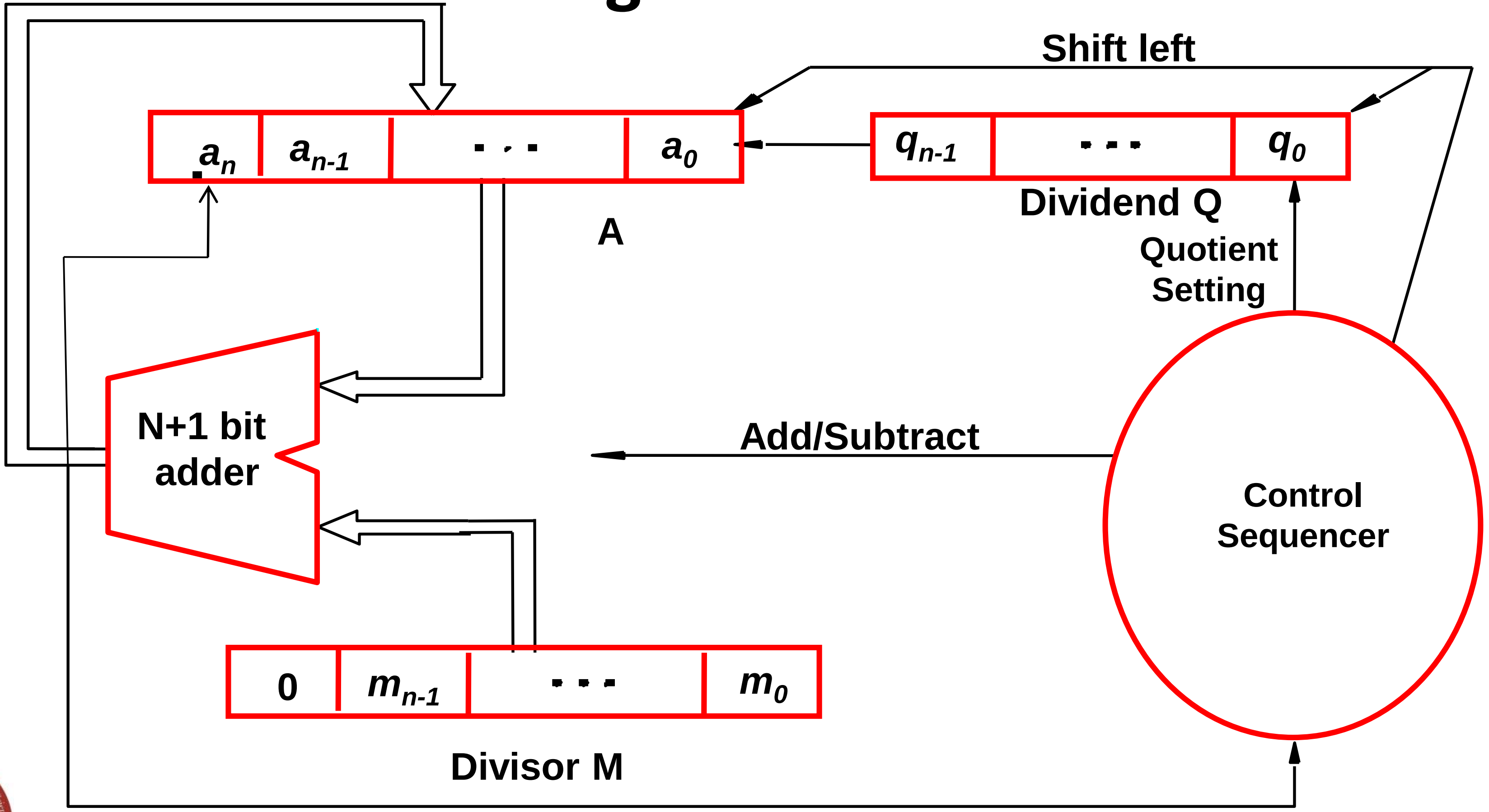


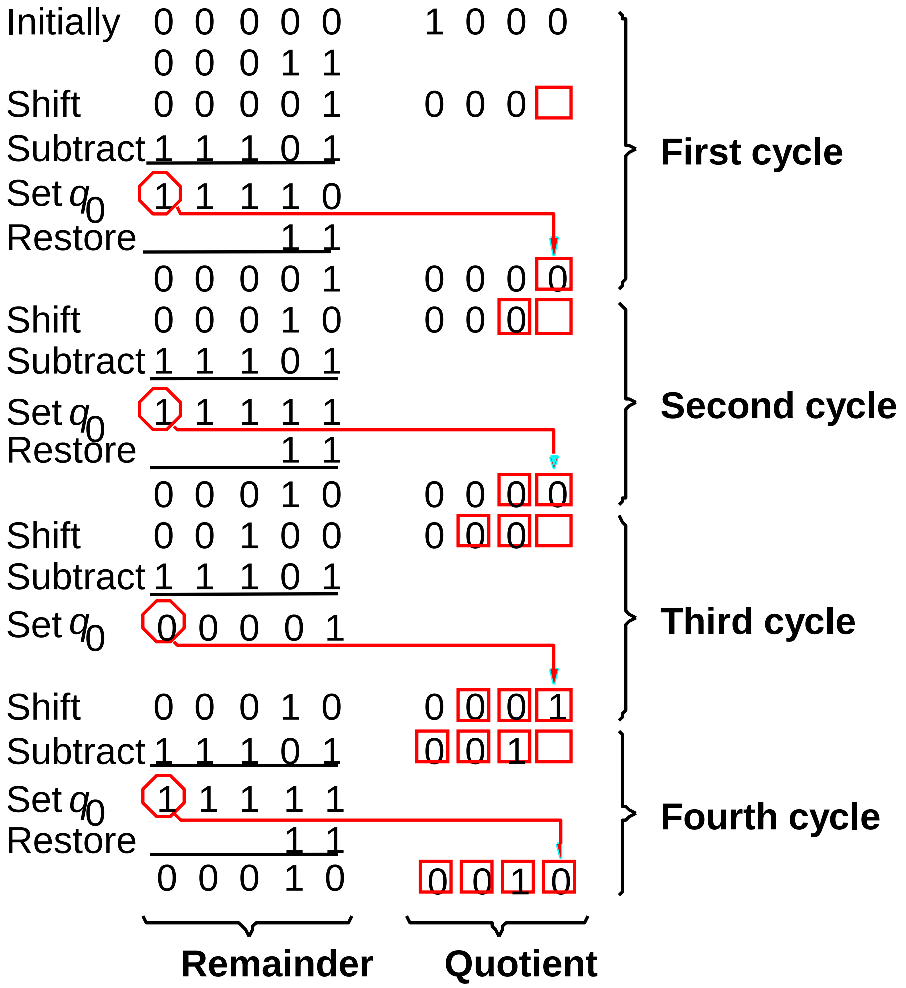
Figure. Circuit arrangement for binary division.

Restoring Division

- Step1: Shift A and Q left one binary position
- Step2: Subtract M from A, and place the answer back in A
- Step3: If the sign of A is 1, set q_0 to 0 and add M back to A (restore A); otherwise, set q_0 to 1
- Repeat Step1 to Step3 for *n* times

A restoring-division example.

```
          1 0
     ___________
1 1 ) 1 0 0 0
        1 1
       -----
          1 0
```

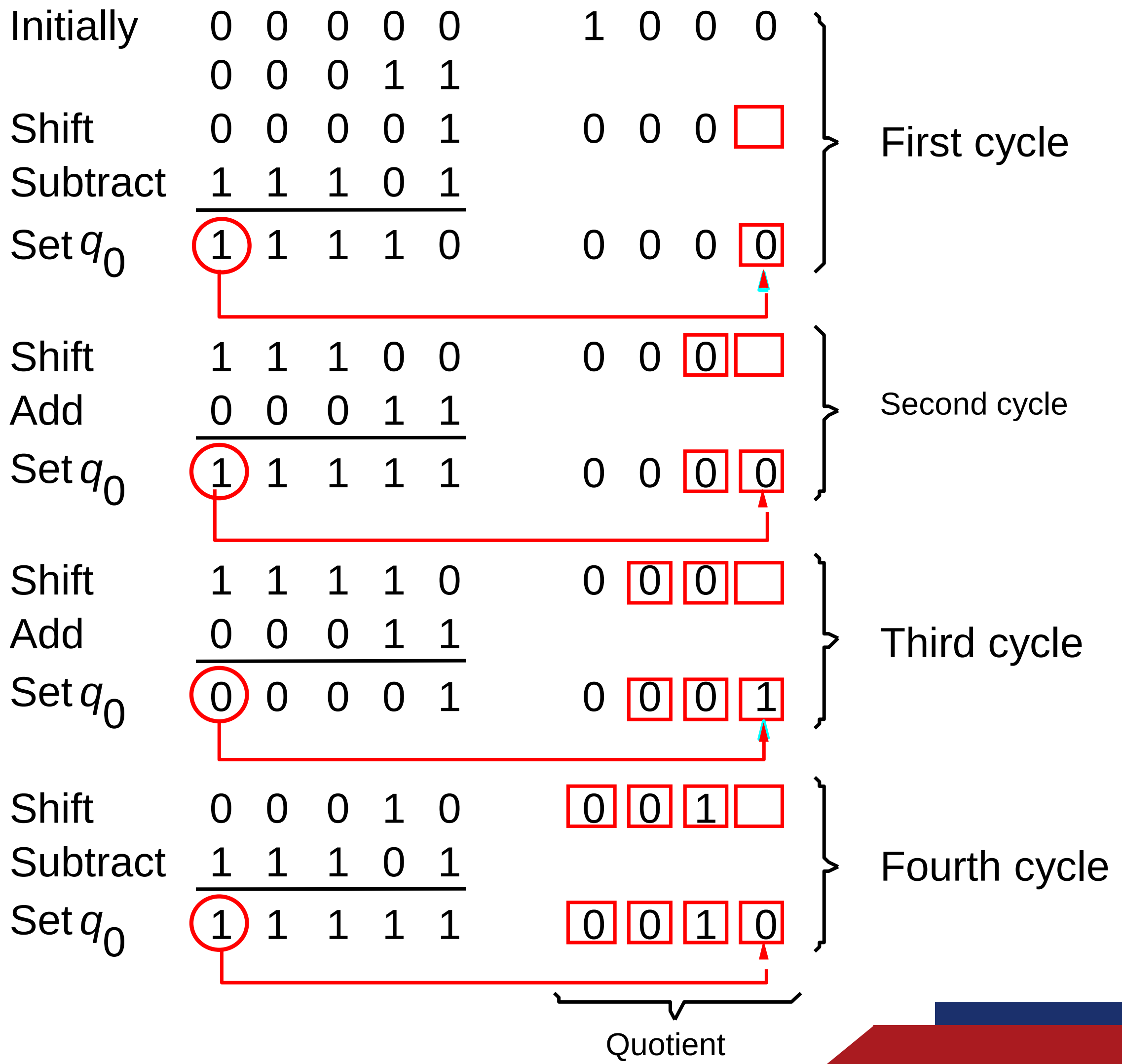


```
Initially  0 0 0 0 0    1 0 0 0
           0 0 0 1 1
Shift      0 0 0 0 1    0 0 0 □
Subtract   1 1 1 0 1
Set q0     1 1 1 1 0
Restore          1 1
           0 0 0 0 1    0 0 0 0      First cycle
Shift      0 0 0 1 0    0 0 0 □
Subtract   1 1 1 0 1
Set q0     1 1 1 1 1
Restore          1 1
           0 0 0 1 0    0 0 0 0      Second cycle
Shift      0 0 1 0 0    0 0 0 □
Subtract   1 1 1 0 1
Set q0     0 0 0 0 1                 Third cycle
Shift      0 0 0 1 0    0 0 0 1
Subtract   1 1 1 0 1    0 0 1 □
Set q0     1 1 1 1 1
Restore          1 1
           0 0 0 1 0    0 0 1 0      Fourth cycle
           Remainder    Quotient
```

Nonrestoring Division

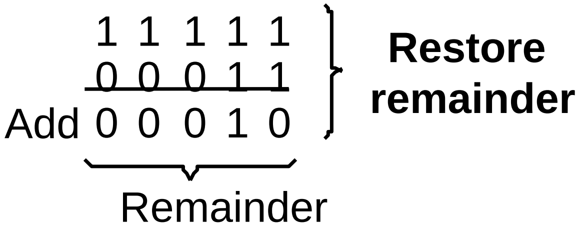
- Step 1: (Repeat *n* times)
  - If the sign of A is 0, shift A and Q left one bit position and subtract M from A; otherwise, shift A and Q left and add M to A.
  - Now, if the sign of A is 0, set q_0 to 1; otherwise, set q_0 to 0.
- Step2: If the sign of A is 1, add M to A

A nonrestoring-division example.



Initially	0	0	0	0	0	1	0	0	0	
	0	0	0	1	1					
Shift	0	0	0	0	1	0	0	0	□	First cycle
Subtract	1	1	1	0	1					
Set q_0	1	1	1	1	0	0	0	0	0	
Shift	1	1	1	0	0	0	0	0	□	Second cycle
Add	0	0	0	1	1					
Set q_0	1	1	1	1	1	0	0	0	0	
Shift	1	1	1	1	0	0	0	0	□	Third cycle
Add	0	0	0	1	1					
Set q_0	0	0	0	0	1	0	0	0	1	
Shift	0	0	0	1	0	0	0	1	□	Fourth cycle
Subtract	1	1	1	0	1					
Set q_0	1	1	1	1	1	0	0	1	0	

Quotient



	1	1	1	1	1	Restore remainder
	0	0	0	1	1	
Add	0	0	0	1	0	

Remainder

Floating-Point Numbers and Operations

Fractions

If *b* is a binary vector, then we have seen that it can be interpreted as an unsigned integer by:

$$\boxed{V(b) = b_{31}.2^{31} + b_{30}.2^{30} + b_{n-3}.2^{29} + ... + b_1.2^1 + b_0.2^0}$$

This vector has an implicit binary point to its immediate right:

$b_{31}b_{30}b_{29}....................b_1b_0.$ implicit binary point

Suppose if the binary vector is interpreted with the implicit binary point is just left of the sign bit:

implicit binary point $.b_{31}b_{30}b_{29}....................b_1b_0$

The value of *b* is then given by:

$$\boxed{V(b) = b_{31}.2^{-1} + b_{30}.2^{-2} + b_{29}.2^{-3} + ... + b_1.2^{-31} + b_0.2^{-32}}$$

Range of Fractions

The value of the unsigned binary fraction is:

$$\boxed{V(b) = b_{31}.2^{-1} + b_{30}.2^{-2} + b_{29}.2^{-3} + \dots + b_1.2^{-31} + b_0.2^{-32}}$$

The range of the numbers represented in this format is:

$$\boxed{0 \le V(b) \le 1 - 2^{-32} \approx 0.9999999998}$$

In general for a *n*-bit binary fraction (a number with an assumed binary point at the immediate left of the vector), then the range of values is:

$$\boxed{0 \le V(b) \le 1 - 2^{-n}}$$

Scientific notation

- Previous representations have a fixed point. Either the point is to the immediate right or it is to the immediate left. This is called Fixed point representation.
- Fixed point representation suffers from a drawback that the representation can only represent a finite range (and quite small) range of numbers.

A more convenient representation is the scientific representation, where the numbers are represented in the form:

$$\boxed{x = m_1.m_2m_3m_4 \times b^{\pm e}}$$

Components of these numbers are:

Mantissa (m), implied base (b), and exponent (e)

Scientific Digits

A number such as the following is said to have 7 significant digits

$$x = \pm 0.m_1 m_2 m_3 m_4 m_5 m_6 m_7 \times b^{\pm e}$$

Fractions in the range 0.0 to 0.9999999 need about 24 bits of precision (in binary). For example the binary fraction with 24 1's:

111111111111111111111111 = 0.9999999404

Not every real number between 0 and 0.9999999404 can be represented by a 24-bit fractional number.

The smallest non-zero number that can be represented is:

000000000000000000000001 = 5.96046×10^{-8}

Every other non-zero number is constructed in increments of this value.

Sign and exponent digits

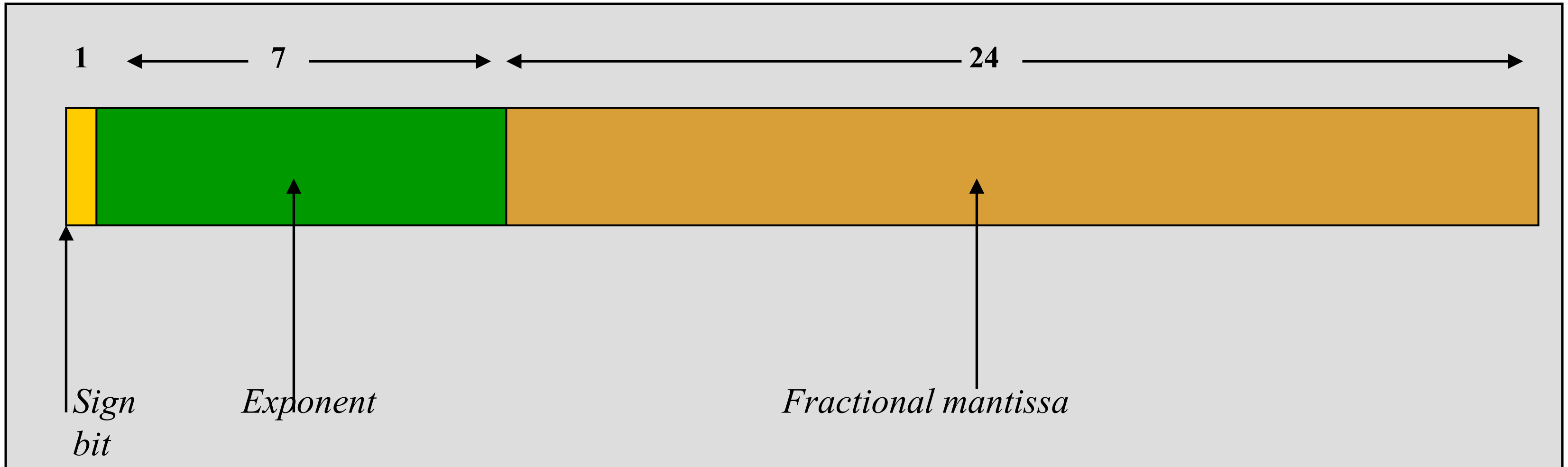
- In a 32-bit number, suppose we allocate 24 bits to represent a fractional mantissa.
- Assume that the mantissa is represented in sign and magnitude format, and we have allocated one bit to represent the sign.
- We allocate 7 bits to represent the exponent, and assume that the exponent is represented as a 2's complement integer.
- There are no bits allocated to represent the base, we assume that the base is implied for now, that is the base is 2.
- Since a 7-bit 2's complement number can represent values in the range -64 to 63, the range of numbers that can be represented is:

$$\boxed{0.0000001 \times 2^{-64} \quad <= \ |x| <= \ 0.9999999 \times 2^{63}}$$

- In decimal representation this range is:

$$\boxed{0.5421 \times 10^{-20} \quad <= \ |x| <= \ 9.2237 \times 10^{18}}$$

A sample representation



•24-bit mantissa with an implied binary point to the immediate left
•7-bit exponent in 2's complement form, and implied base is 2.

Normalization

Consider the number: $\boxed{x = 0.0004056781 \times 10^{12}}$

If the number is to be represented using only 7 significant mantissa digits, the representation ignoring rounding is: $\boxed{x = 0.0004056 \times 10^{12}}$

If the number is shifted so that as many significant digits are brought into 7 available slots: $\boxed{x = 0.4056781 \times 10^{9} = 0.0004056 \times 10^{12}}$

Exponent of *x* was decreased by 1 for every left shift of *x.*

A number which is brought into a form so that all of the available mantissa digits are optimally used (this is different from all occupied which may not hold), is called a normalized number.

Same methodology holds in the case of binary mantissas

$\boxed{0001101000(10110) \times 2^{8} = 1101000101(10) \times 2^{5}}$

Normalization *(Contd.)*

- A floating point number is in normalized form if the most significant 1 in the mantissa is in the most significant bit of the mantissa.
- All normalized floating point numbers in this system will be of the form:

$$\boxed{0.1xxxxx.......xx}$$

- Range of numbers representable in this system, if every number must be normalized is:

$$\boxed{0.5 \times 2^{-64} <= |x| < 1 \times 2^{63}}$$

Normalization, overflow and underflow

The procedure for normalizing a floating point number is:

```
Do (until MSB of mantissa = = 1)
    Shift the mantissa left (or right)
    Decrement (increment) the exponent by 1
end do
```

Applying the normalization procedure to: *.000111001110....0010 x* 2^{-62}

gives: *.111001110........ x* 2^{-65}

But we cannot represent an exponent of –65, in trying to normalize the number we have underflowed our representation.

Applying the normalization procedure to: *1.00111000............x* 2^{63}

gives: 0.100111..............x 2^{64}

This overflows the representation.

Changing the implied base

- So far we have assumed an implied base of 2, that is our floating point numbers are of the form: $\boxed{x = m\ 2^e}$

- If we choose an implied base of 16, then:

$$\boxed{x = m\ 16^e}$$

Then:

$$\boxed{y = (m.16)\ .16^{e-1}\ (m.2^4)\ .16^{e-1} = m\ .\ 16^e = x}$$

- Thus, every four left shifts of a binary mantissa results in a decrease of 1 in a base 16 exponent.
- Normalization in this case means shifting the mantissa until there is a 1 in the first four bits of the mantissa.

Excess notation

- Rather than representing an exponent in 2’s complement form, it turns out to be more beneficial to represent the exponent in excess notation.
- If 7 bits are allocated to the exponent, exponents can be represented in the range of -64 to +63, that is:

$$\boxed{-64 <= e <= 63}$$

- Exponent can also be represented using the following coding called as excess-64:

$$\boxed{E' = E_{true} + 64}$$

- In general, excess-p coding is represented as:

$$\boxed{E' = E_{true} + p}$$

True exponent of -64 is represented as 0
0 is represented as 64
63 is represented as 127

This enables efficient comparison of the relative sizes of two floating point numbers.

IEEE notation

IEEE Floating Point notation is the standard representation in use. There are two representations:

- Single precision.
- Double precision.

Both have an implied base of 2.

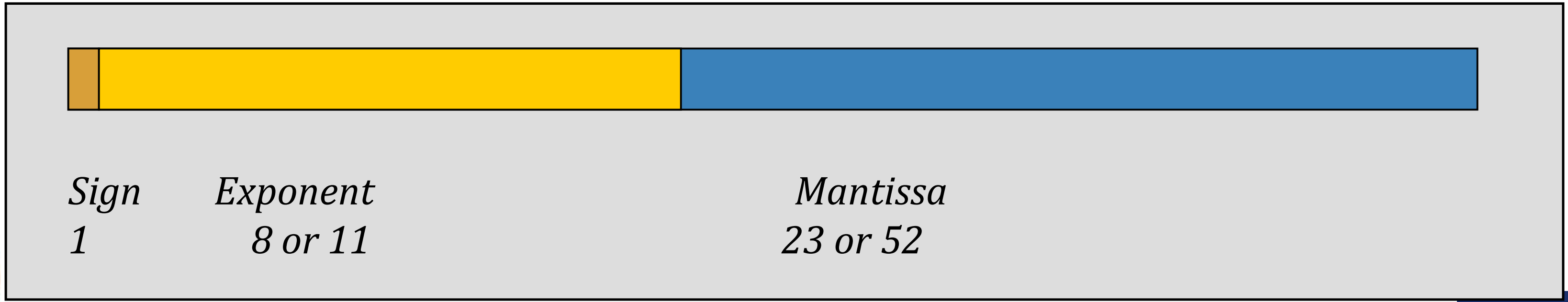
Single precision:

- 32 bits (23-bit mantissa, 8-bit exponent in excess-127 representation)

Double precision:

- 64 bits (52-bit mantissa, 11-bit exponent in excess-1023 representation)

Fractional mantissa, with an implied binary point at immediate left.



Peculiarities of IEEE notation

- Floating point numbers have to be represented in a normalized form to maximize the use of available mantissa digits.
- In a base-2 representation, this implies that the MSB of the mantissa is always equal to 1.
- If every number is normalized, then the MSB of the mantissa is always 1. We can do away without storing the MSB.
- IEEE notation assumes that all numbers are normalized so that the MSB of the mantissa is a 1 and does not store this bit.
- So the real MSB of a number in the IEEE notation is either a 0 or a 1.
- The values of the numbers represented in the IEEE single precision notation are of the form:

$$\boxed{(+,-)\ 1.M \times 2^{(E - 127)}}$$

- The hidden 1 forms the integer part of the mantissa.
- Note that excess-127 and excess-1023 (not excess-128 or excess-1024) are used to represent the exponent.

Exponent field

In the IEEE representation, the exponent is in excess-127 (excess-1023) notation. The actual exponents represented are:

-126 <= E <= 127   and   -1022 <= E <= 1023
not
-127 <= E <= 128   and   -1023 <= E <= 1024

This is because the IEEE uses the exponents -127 and 128 (and -1023 and 1024), that is the actual values 0 and 255 to represent special conditions:

- Exact zero
- Infinity

Floating point arithmetic

Addition:

$$\boxed{3.1415 \times 10^8 + 1.19 \times 10^6 = 3.1415 \times 10^8 + 0.0119 \times 10^8 = 3.1534 \times 10^8}$$

Multiplication:

$$\boxed{3.1415 \times 10^8 \times 1.19 \times 10^6 = (3.1415 \times 1.19) \times 10^{(8+6)}}$$

Division:

$$\boxed{3.1415 \times 10^8 / 1.19 \times 10^6 = (3.1415 / 1.19) \times 10^{(8-6)}}$$

Biased exponent problem:
If a true exponent e is represented in excess-p notation, that is as e+p.
Then consider what happens under multiplication:

$$\boxed{a.\ 10^{(x+p)} * b.\ 10^{(y+p)} = (a.b).\ 10^{(x+p+y+p)} = (a.b).\ 10^{(x+y+2p)}}$$

Representing the result in excess-p notation implies that the exponent should be *x+y+p*. Instead it is *x+y+2p*.
Biases should be handled in floating point arithmetic.

Floating point arithmetic: ADD/SUB rule

- Choose the number with the smaller exponent.
- Shift its mantissa right until the exponents of both the numbers are equal.
- Add or subtract the mantissas.
- Determine the sign of the result.
- Normalize the result if necessary and truncate/round to the number of mantissa bits.

Note: This does not consider the possibility of overflow/underflow.

Floating point arithmetic: MUL rule

- Add the exponents.
- Subtract the bias.
- Multiply the mantissas and determine the sign of the result.
- Normalize the result (if necessary).
- Truncate/round the mantissa of the result.

Floating point arithmetic: DIV rule

- Subtract the exponents
- Add the bias.
- Divide the mantissas and determine the sign of the result.
- Normalize the result if necessary.
- Truncate/round the mantissa of the result.

Note: Multiplication and division does not require alignment of the mantissas the way addition and subtraction does.

Guard bits

- While adding two floating point numbers with 24-bit mantissas, we shift the mantissa of the number with the smaller exponent to the right until the two exponents are equalized.
- This implies that mantissa bits may be lost during the right shift (that is, bits of precision may be shifted out of the mantissa being shifted).
- To prevent this, floating point operations are implemented by keeping guard bits, that is, extra bits of precision at the least significant end of the mantissa.
- The arithmetic on the mantissas is performed with these extra bits of precision.
- After an arithmetic operation, the guarded mantissas are:

  - Normalized (if necessary)

  - Converted back by a process called truncation/rounding to a 24-bit mantissa.

Truncation/ Rounding

Straight chopping:

The guard bits (excess bits of precision) are dropped.

Von Neumann rounding:

If the guard bits are all 0, they are dropped.
However, if any bit of the guard bit is a 1, then the LSB of the retained bit is set to 1.

Rounding:

If there is a 1 in the MSB of the guard bit then a 1 is added to the LSB of the retained bits.

Rounding

- Rounding is evidently the most accurate truncation method.
- However,
  - Rounding requires an addition operation.
  - Rounding may require a renormalization, if the addition operation de-normalizes the truncated number.

0.111111100000 rounds to 0.111111 + 0.000001
=1.000000 which must be renormalized to 0.100000

- IEEE uses the rounding method.