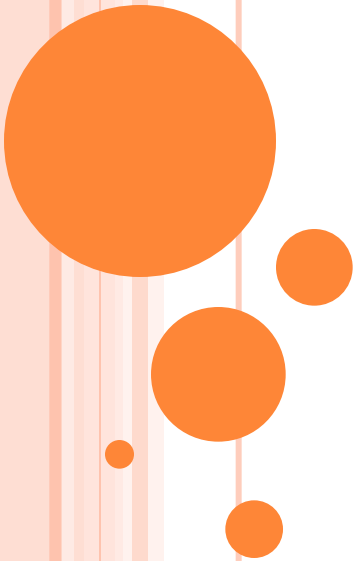
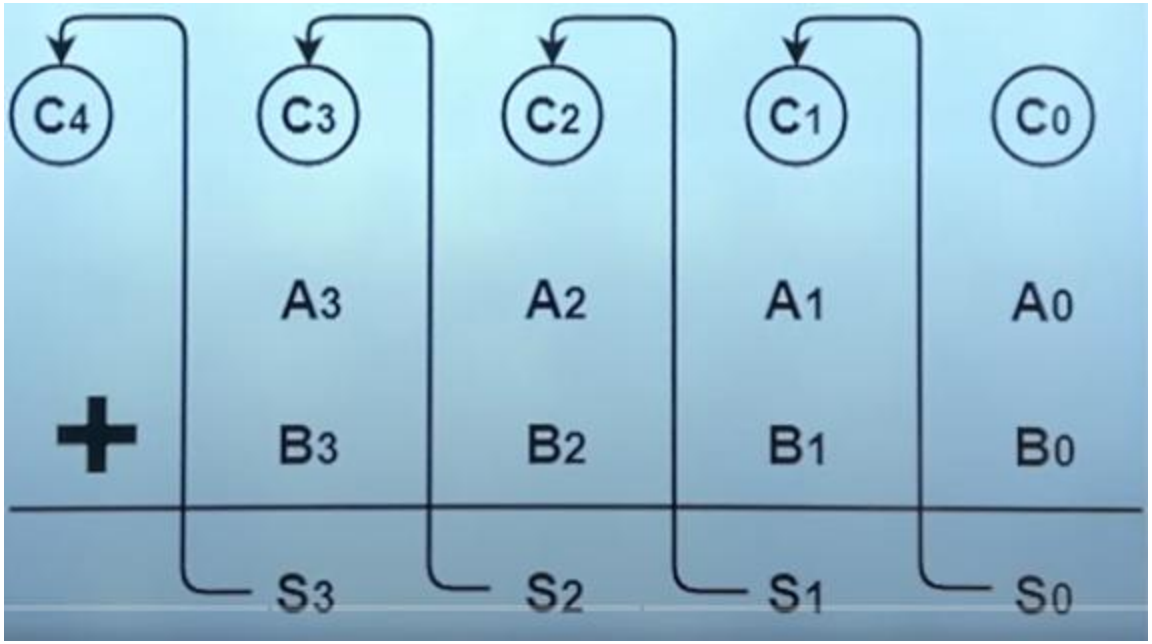
COMPUTER ORGANIZATION AND ARCHITECTURE

2. ALU Unit



Full Adder

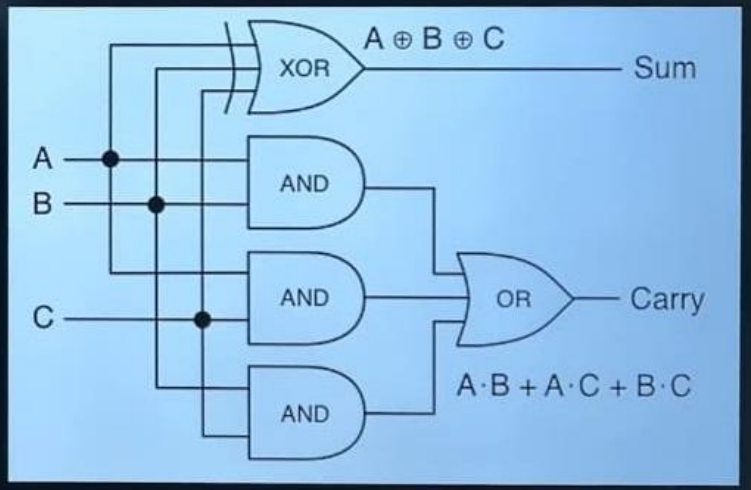
- A Logic circuit which performs the arithmetic sum and it's the part of ALU unit.



	ab	a'b'	a'b	ab	ab'
C_{in}		00	01	11	10
C_{in}'	0	0	2	6	4
C_{in}	1	1	3	7	5

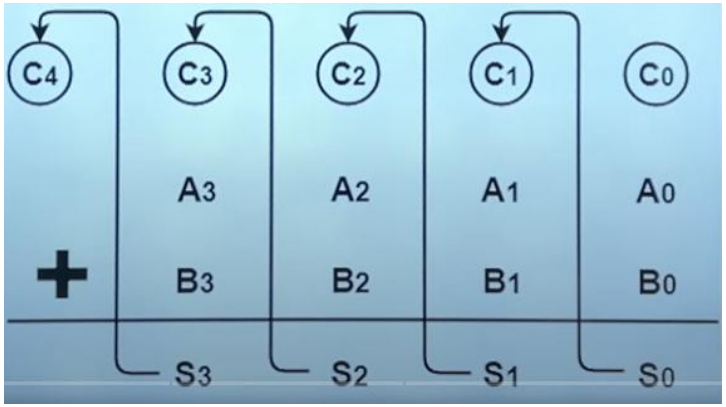
	ab	a'b'	a'b	ab	ab'
C_{in}		00	01	11	10
C_{in}'	0	0	2	6	4
C_{in}	1	1	3	7	5

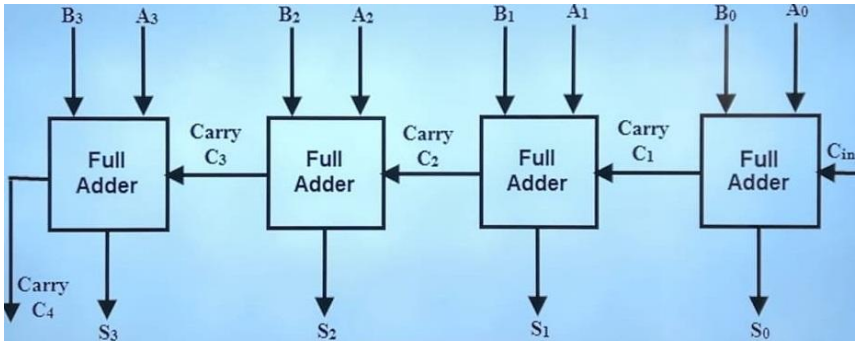
INPUTS			OUTPUTS	
A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

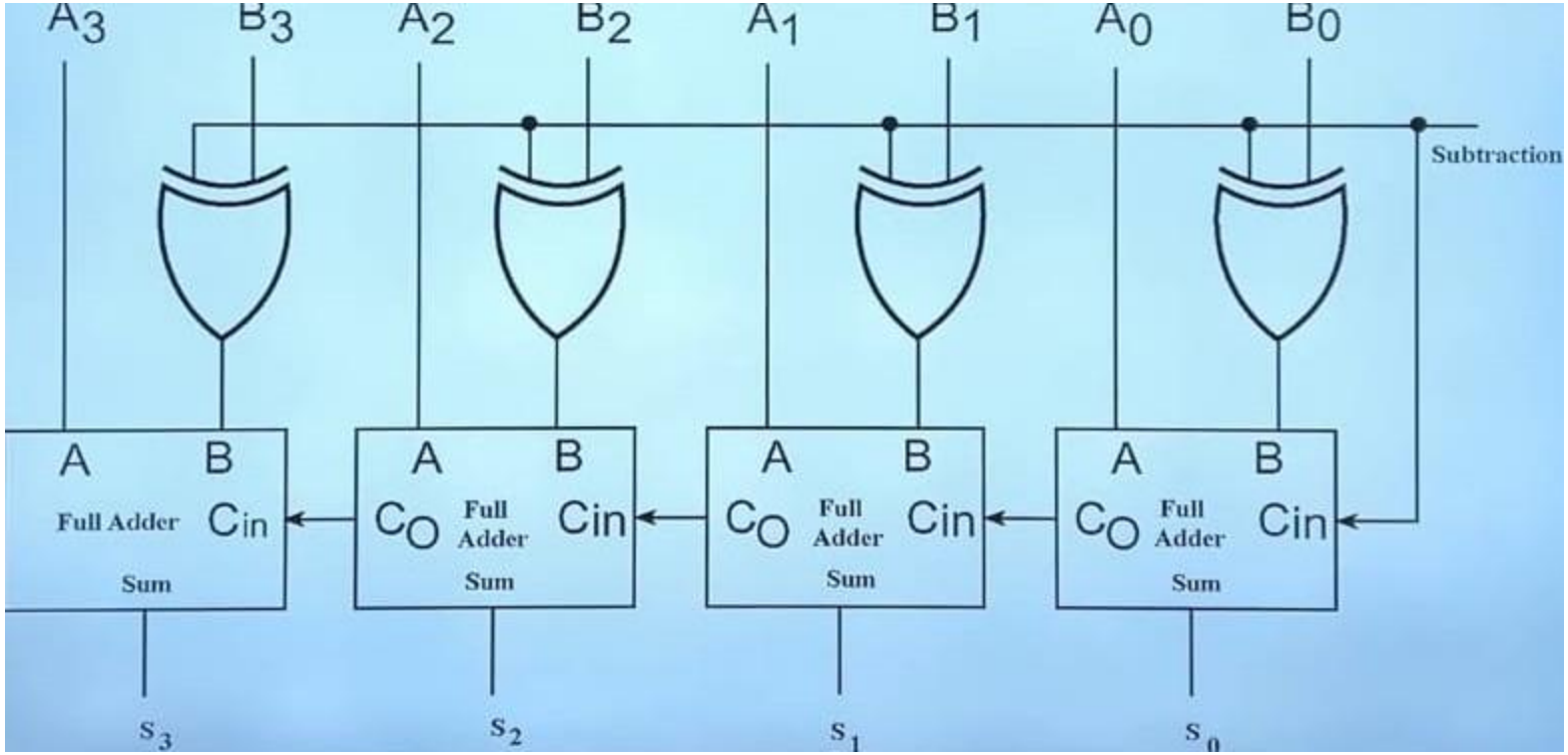


Four bit ripple adder/ Parallel binary adder:

- The subtraction can be done with 2's complement of B and adding it to A.







- M controls the operation, M= 0 is like adder which means B EXOR 0 = B only,

- While M= 1 will behave like; B EXOR 1 = Complement of B . Inputs are complemented and 1 is added through the carry.

Problems:

- Carry Propagation

Solution:

- Look Ahead Carry Generator (Hardware Solution)

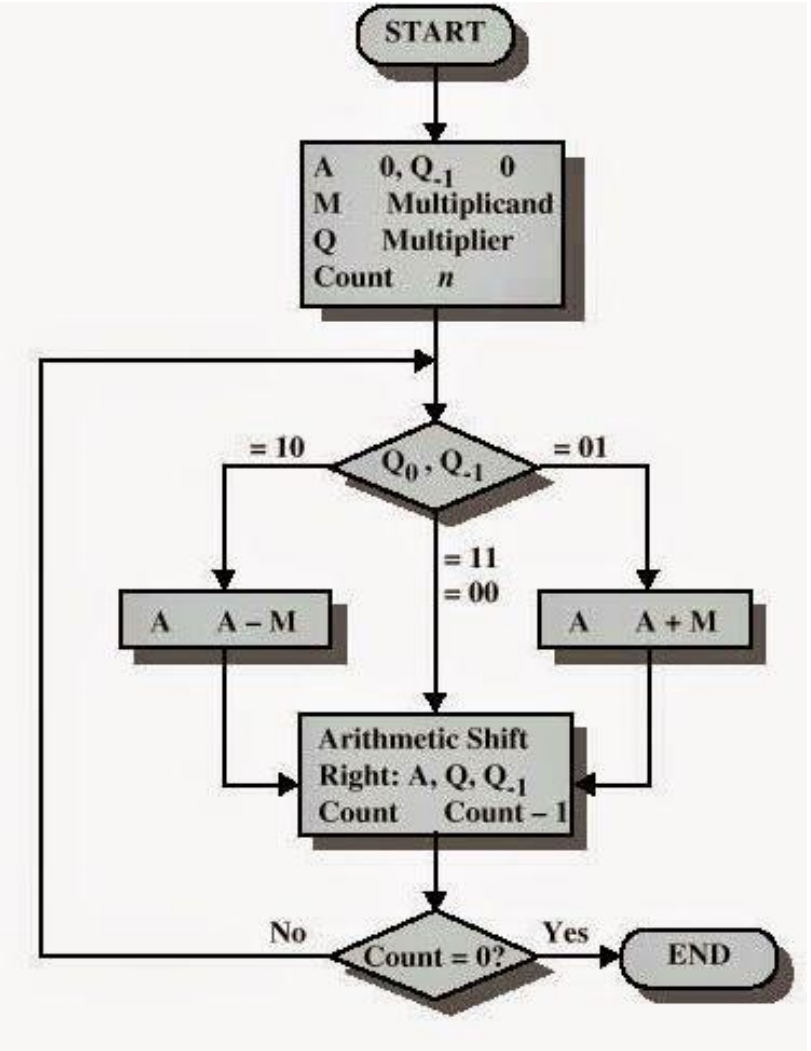
Booth's Algorithms (1950):

- Andrew Donald Booth (1918-2009), was a British electrical engineer, physicist. Was an early developer of magnetic Drum memory for computers. He is known for Booth's multiplication algorithm.
- Booth's algorithm optimizes binary multiplication by reducing the number of addition and subtractions based on the multiplier bits.
- It involves examining the multiplier bits, then either adding, subtracting or leaving the multiplicand unchanged before shifting the partial product.

EXAMPLE:

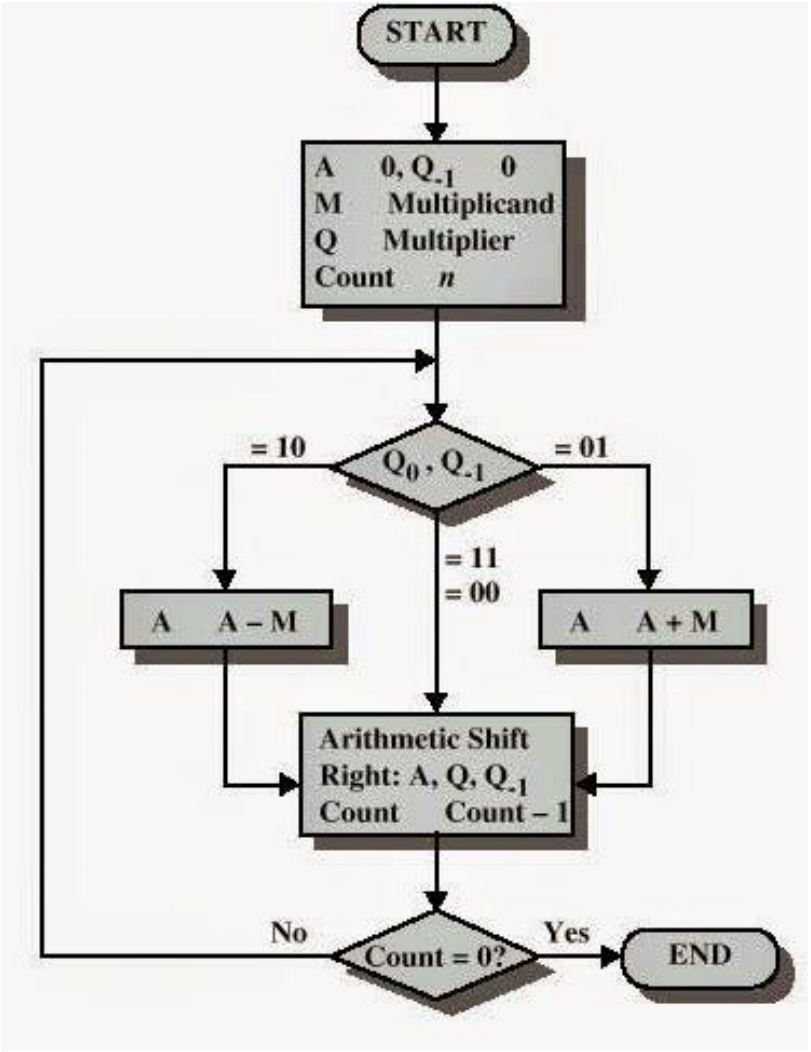
A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial values	
1001	0011	0	0111	$A \leftarrow A - M$	First cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second cycle
0101	0100	1	0111	$A \leftarrow A + M$	Third cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth cycle

Figure 9.13 Example of Booth's Algorithm (7 × 3)

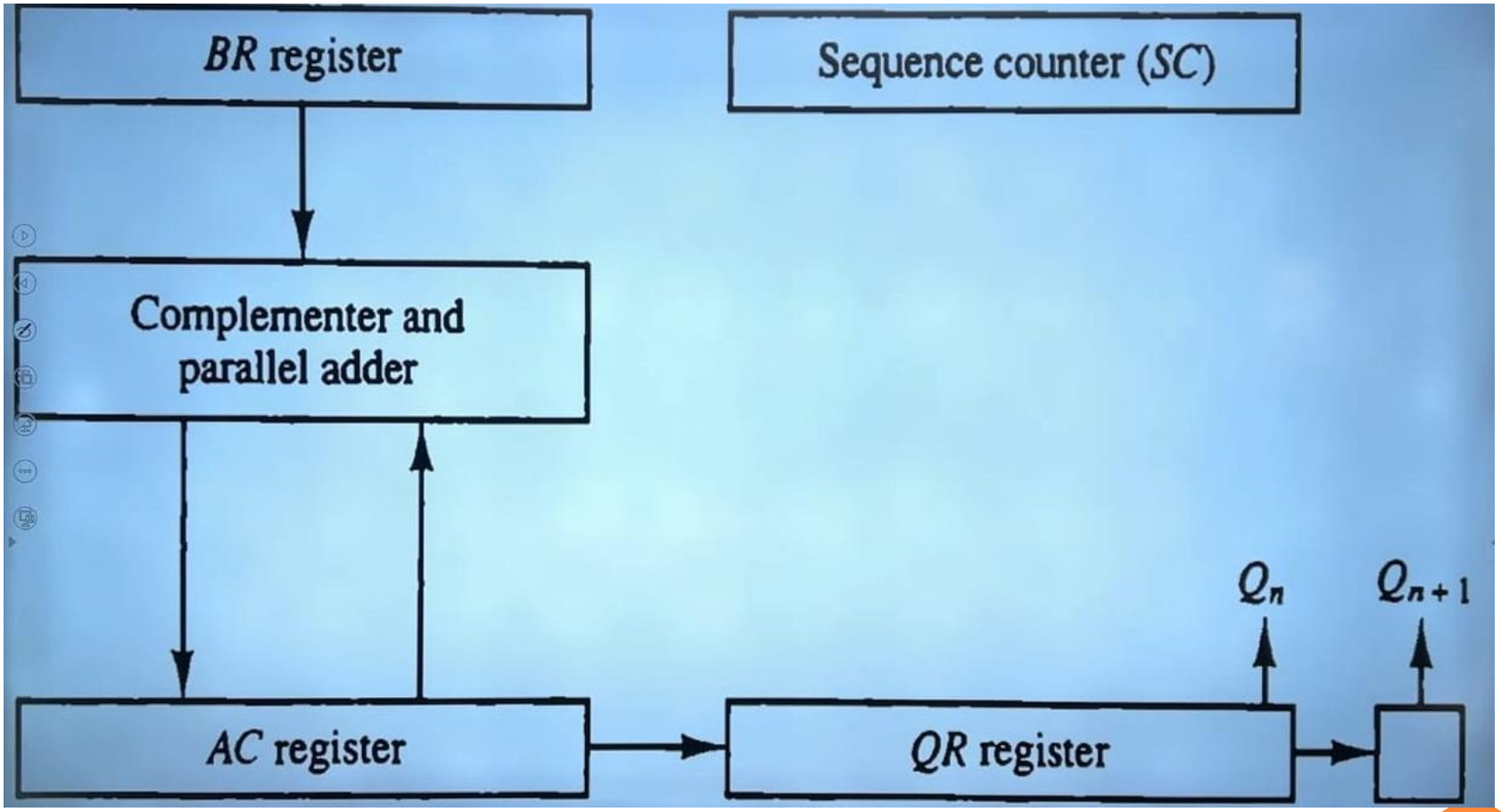


Booth : (7) x (-3)

```
  A       Q         M
          -3         7
0000    1101 0    0111
--------------------
1001    1101 0    0111    A <- (A - M)   1st
1100    1110 1    0111    Shift
--------------------
0011    1110 1    0111    A <- (A + M)   2nd
0001    1111 0    0111    Shift
--------------------
1010    1111 0    0111    A <- (A - M)   3rd
1101    0111 1    0111    Shift
--------------------
                                         4th
1110    1011 1    0111    Shift
--------------------
```



Hardware Implementation of Booth's Algorithm :



Array Multiplier:

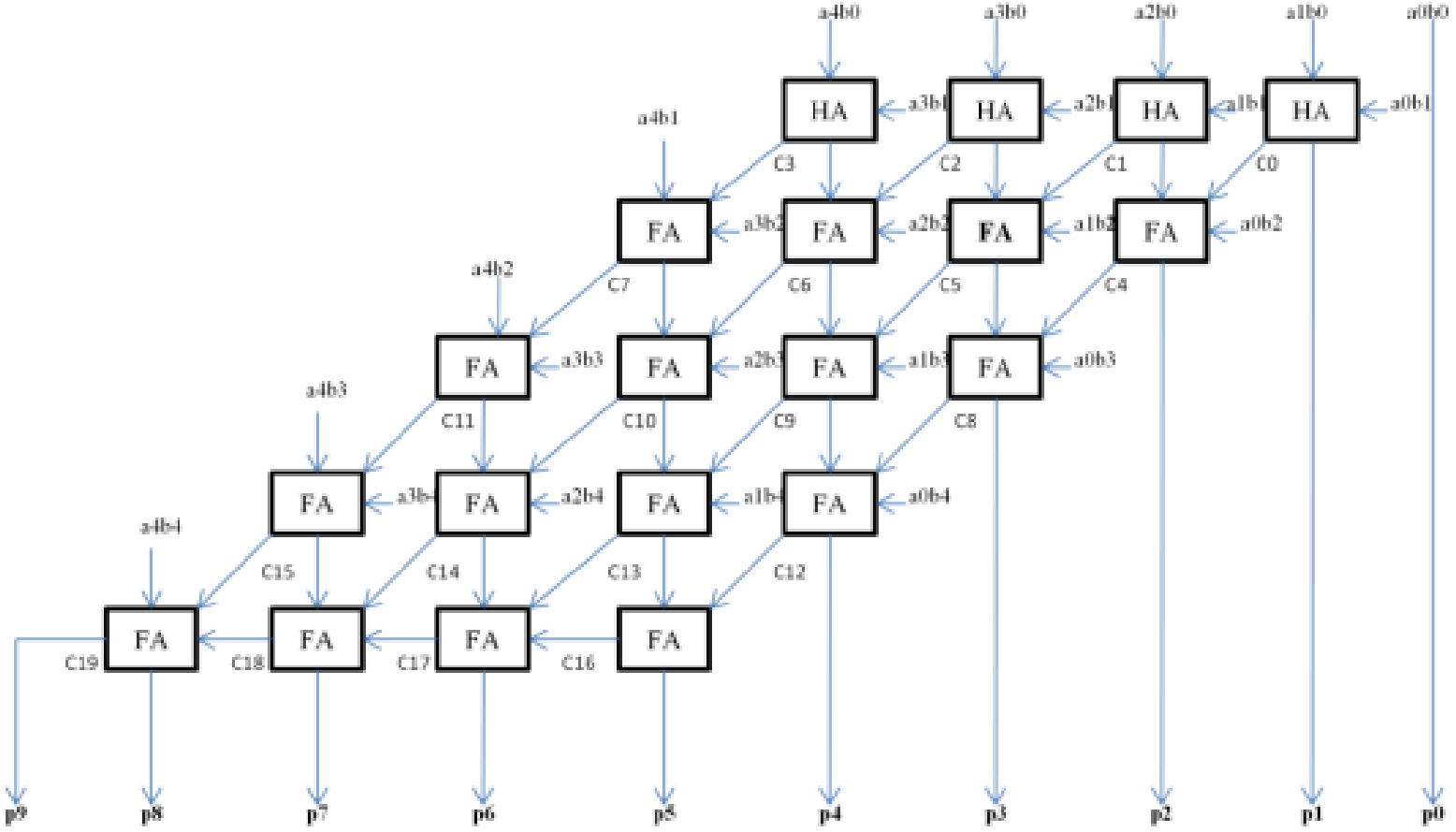
- Uses a grid full of HA's and FA's to perform the simultaneous addition of product terms.
- AND gates are used to form these product terms before they are fed to the adder array
- It forms the product bits all at once, making the operation faster.
- Speed advantage comes at a cost of requiring large number of gates, which became economical with the advent of integrated circuits.

Basic Binary Multiplication:

- 1010*0110

ARCHITECTURE:

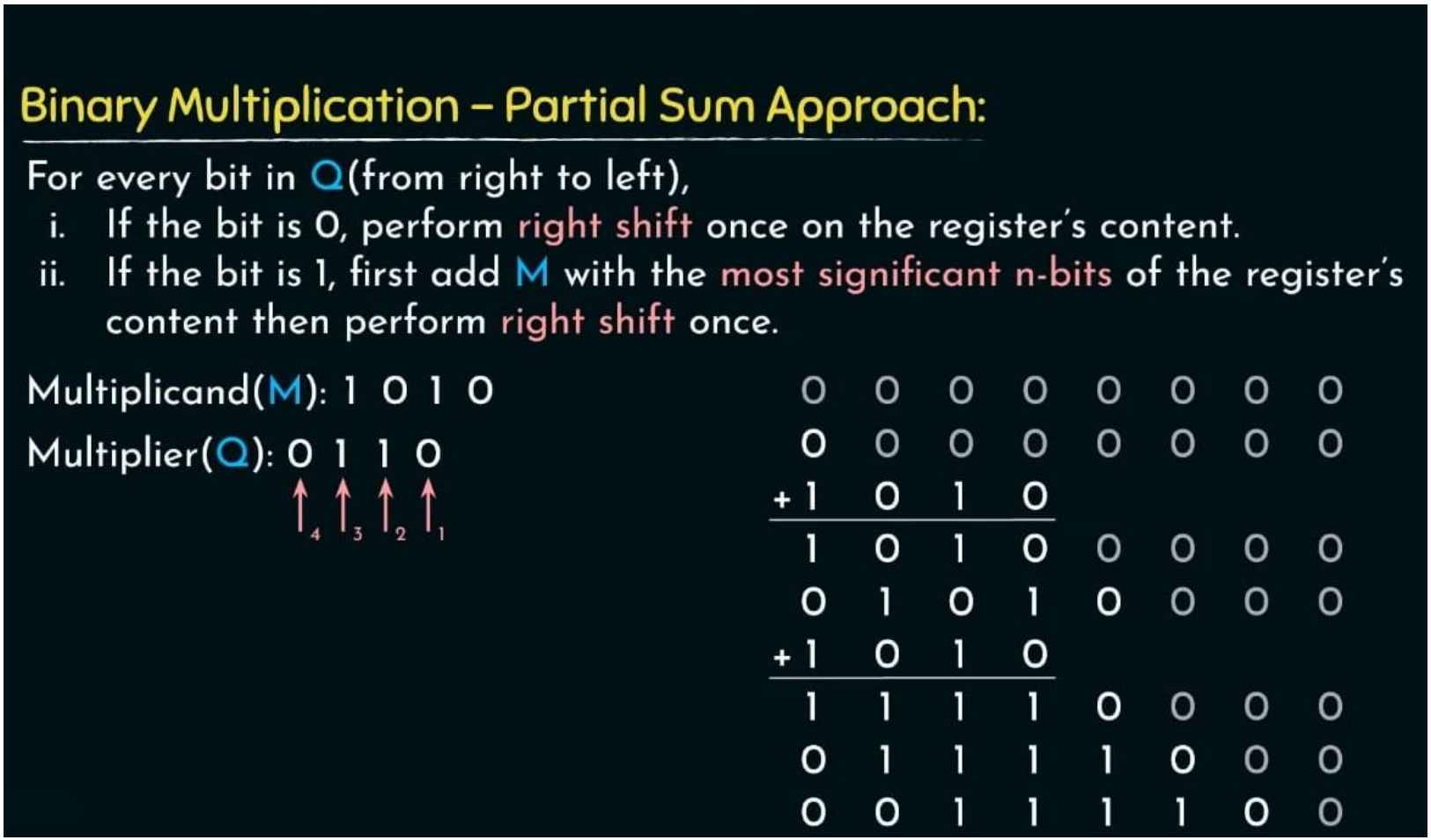
		a_3	a_2	a_1	a_0		
x		b_3	b_2	b_1	b_0		
		p_{30}	p_{20}	p_{10}	p_{00}		
	p_{31}	p_{21}	p_{11}	p_{01}	x		
p_{32}	p_{22}	p_{12}	p_{02}	x	x		
p_{33}	p_{23}	p_{13}	p_{03}	x	x	x	
z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0



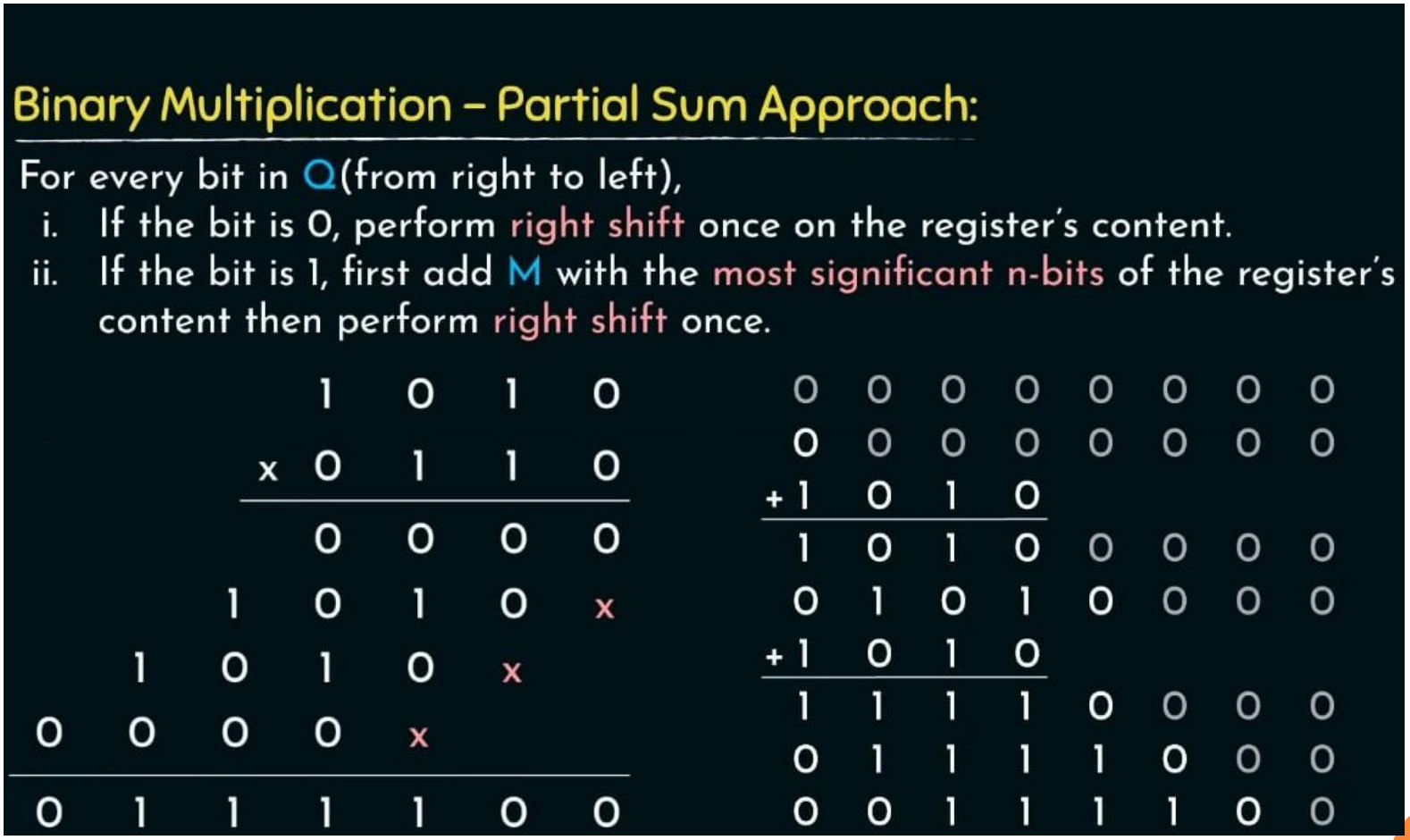
Partial Sum Approach:

- |M|: n-bits; |Q|: n-bits

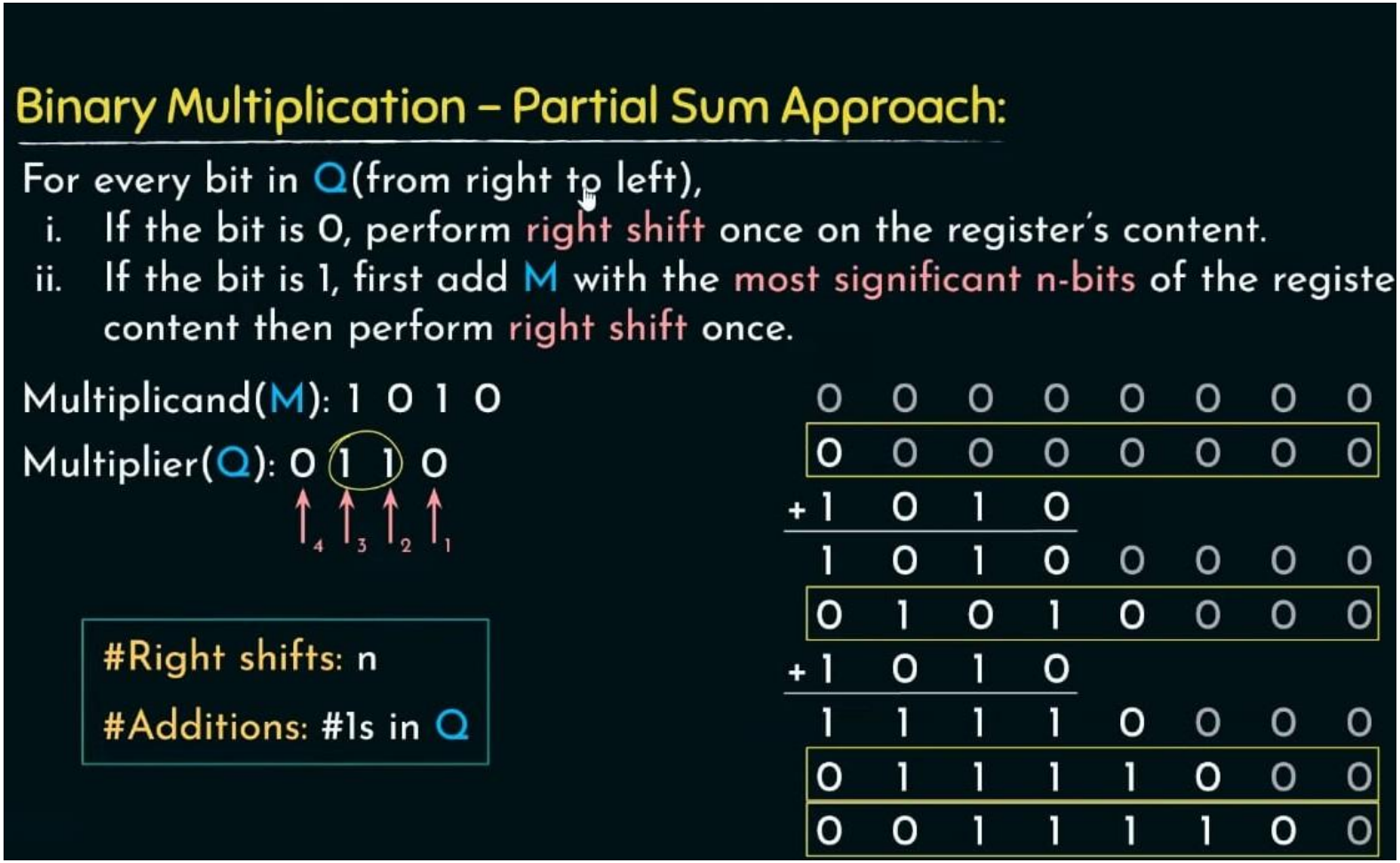
|M*Q|: (n+n)= 2n-bits (in worst case)



COMPARISON:


Binary Multiplication – Partial Sum Approach:
For every bit in Q (from right to left),
i. If the bit is 0, perform right shift once on the register's content.
ii. If the bit is 1, first add M with the most significant n-bits of the register's content then perform right shift once.
1 0 1 0
x 0 1 1 0
0 0 0 0
1 0 1 0 x
1 0 1 0 x
0 0 0 0 x
0 1 1 1 1 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
+ 1 0 1 0
1 0 1 0 0 0 0 0
0 1 0 1 0 0 0 0
+ 1 0 1 0
1 1 1 1 0 0 0 0
0 1 1 1 1 0 0 0
0 0 1 1 1 1 0 0

Analysis of Partial sum approach:



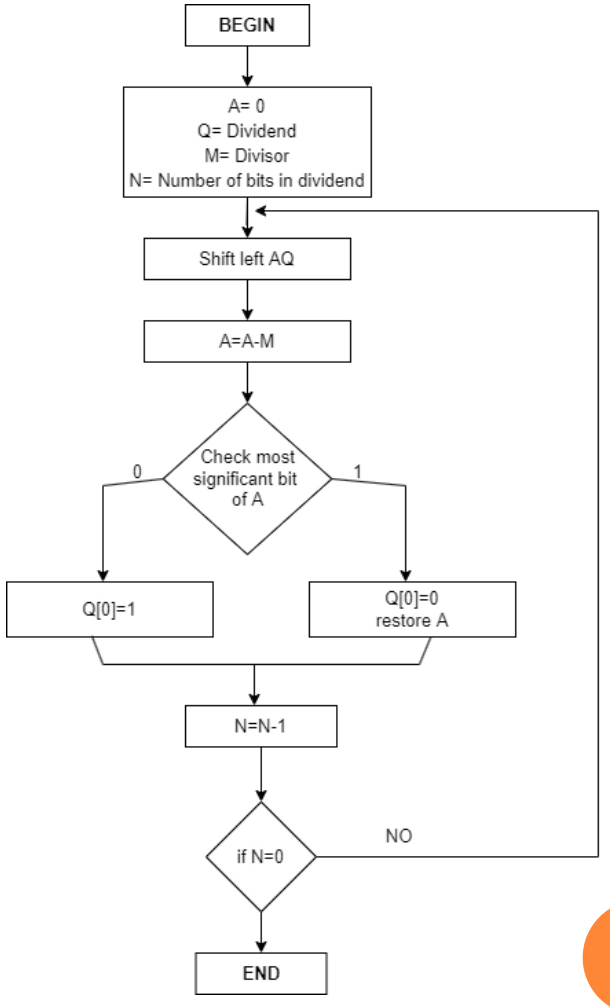
Division:

- Can be grouped into two classes.

1. Slow Algorithms- Restoring Division and Non-Restoring Division Algorithms
2. Fast Algorithms- Division by Series Expansion and Newton-Raphson Division.

Restoring Division Algorithm (11/3):

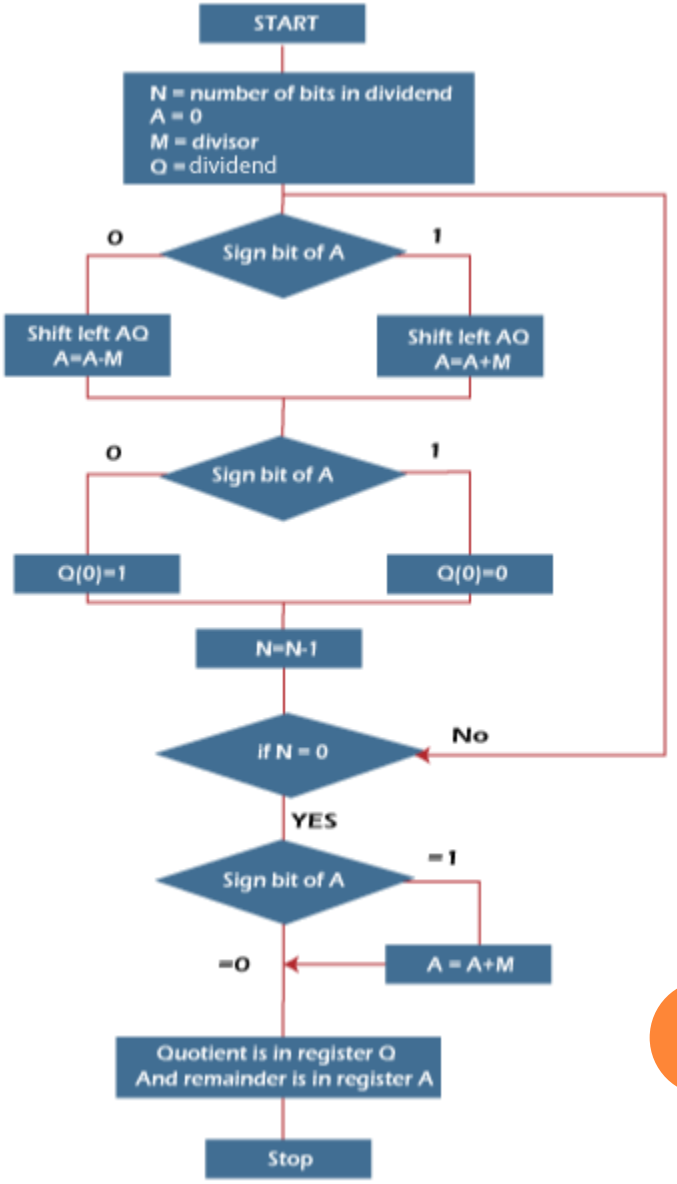
n	M	A	Q	Action/Operation
4	00011	00000	1011	Initialization
		00001	011?	SL AQ
		11110	011?	A = A-M
3	00011	00001	0110	QO ← 0
		00010	110?	SL AQ
		11111	110?	A = A-M
2	00011	00010	1100	QO ← 0
		00101	100?	SL AQ
		00010	100?	A = A-M
1	00011	00010	1001	QO ← 1
		00101	001?	SL AQ
		00010	001?	A = A-M
0	00011	00010	0011	QO ← 1



- Quotient is in register Q while remainder will be in register A

Non-Restoring Division Algorithm (11/3)

n	M	A	Q	Action/Operation
4	00011	00000	1011	Initialization
		00001	011?	SL AQ
		11110	011?	A = A-M
3	00011	11110	0110	Q0 ← 0
		11100	110?	SL AQ
		11111	110?	A = A+M
2	00011	11111	1100	Q0 ← 0
		11111	100?	SL AQ
		00010	100?	A = A+M
1	00011	00010	1001	Q0 ← 1
		00101	001?	SL AQ
		00010	001?	A = A-M
0	00011	00010	0011	Q0 ← 1



Performance comparison of Restoring and Non-Restoring:

Restoring	Non-Restoring
The quotient bits are initially set to zero and then adjusted based on the remainder.	The divisor is subtracted from the dividend, and the result is examined to determine the next quotient bit.
It requires more hardware and more complex control logic compared to non-restoring division.	generally faster than restoring division because it doesn't always correct the quotient towards the correct result.
Always corrects the quotient towards the correct result, which can make it more accurate but slower.	Has simpler hardware requirements and control logic compared to restoring division.
Usually, slower compared to non-restoring division due to the extra correction steps	Requires more steps on average to reach the correct quotient compared to restoring division.

Basic Understanding of Number System:

- Add (-35) and (-31) in binary using 8-bit registers, in signed 1's complement and signed 2's complement.

Floating Point Representation:

- Problem with casual representation methods is that it do not works well if the number is too small or too large, it takes large amount of memory.
- Example: $6.023*10^{23}$
- Floating point representation says that we can use the same concept for binary as well, as we do in decimal format. We can store the mantissa and exponent in separate.
- Range of true exponent is $-2^{k-1}$to $+2^{k-1}-1$ (Ex. For 8 bit -128 to +127). After adding 2^{k-1} , new range goes from 0 to $2^{k}-1$

Example:

- Represent (-21.75) in floating point representation. (s=1, k=7, m=8)

Floating Point Representation (cont.):

- FP represents a special kind of sign magnitude representation.
- Stored in mantissa/exponent form i.e., $m*r^e$

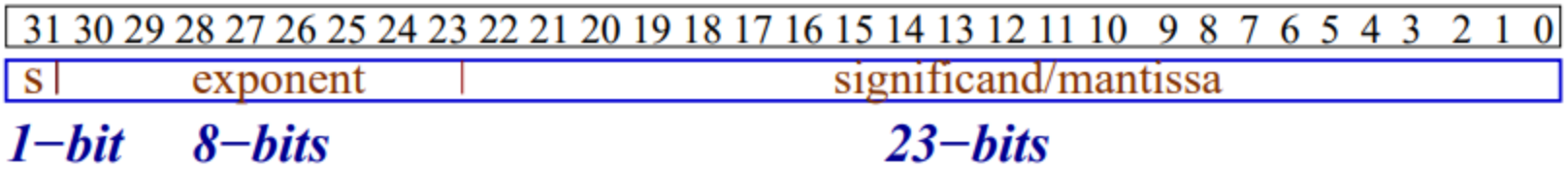
- Mantissa is signed magnitude fraction for most of the cases.
- The exponent is stored in bias form.

Floating-Point Decimal Number

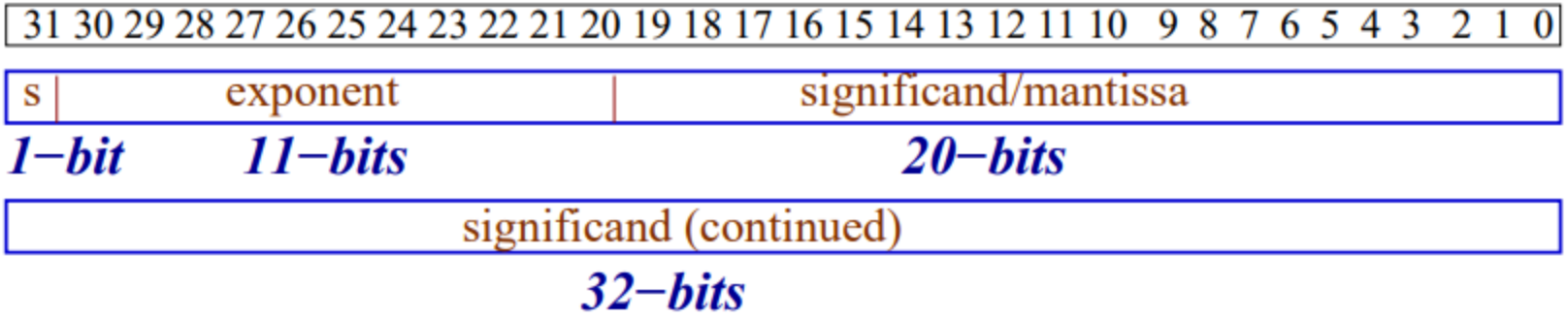
$$
\begin{aligned}
-123456. \times 10^{-1} &= 12345.6 \times 10^{0} \\
&= 1234.56 \times 10^{1} \\
&= 123.456 \times 10^{2} \\
&= 12.3456 \times 10^{3} \\
&= 1.23456 \times 10^{4} (\textit{normalised}) \\
&\approx 0.12345 \times 10^{5} \\
&\approx 0.01234 \times 10^{6}
\end{aligned}
$$

Note

- There are different representations for the same number and there is no fixed position for the decimal point.
- Given a fixed number of digits, there may be a loss of precession.
- Three pieces of information represents a number: sign of the number, the significant value and the signed exponent of 10.



Single Precession (32−bit)



Double Precession (64−bit)

Difference Between Single and Double precision:

Factors	Single Precision	Double Precision
Number of bits	Uses 32 bits to represent a floating-point number.	Uses 64 bits to represent a floating-point number.
Precision	Provides approximately 7 decimal digits of precision	Provides approximately 15-16 decimal digits of precision.
Range	Covers a smaller range of numbers compared to double precision	Covers a wider range of numbers, including larger and smaller values, compared to single precision.
Memory	Consumes less memory	Requires more memory
Accuracy	Offers less accuracy compared to double precision	Provides higher accuracy
Performance	Can be faster to process compared to double precision, especially on hardware that is optimized for single-precision operations.	May require more processing time due to the larger size of the numbers and the higher precision.

Note: The choice between single and double precision depends on the specific requirements of the application, including accuracy, memory usage, and computational performance.

IEEE 754 floating point Standard (1985):

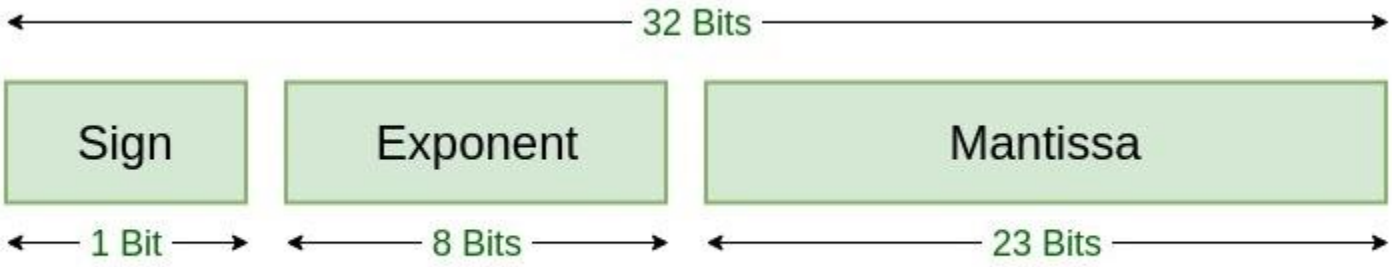
- IEEE standard for floating-point arithmetic (IEEE 754) is a technical standard for floating point arithmetic established in 1985 by IEEE.
- It addresses many problems found in the diverse floating-point implementation that made them difficult to use reliably and portably.
- Most of the hardware floating point units use the IEEE 754 standard.

IEEE 754 floating point Standard (cont):

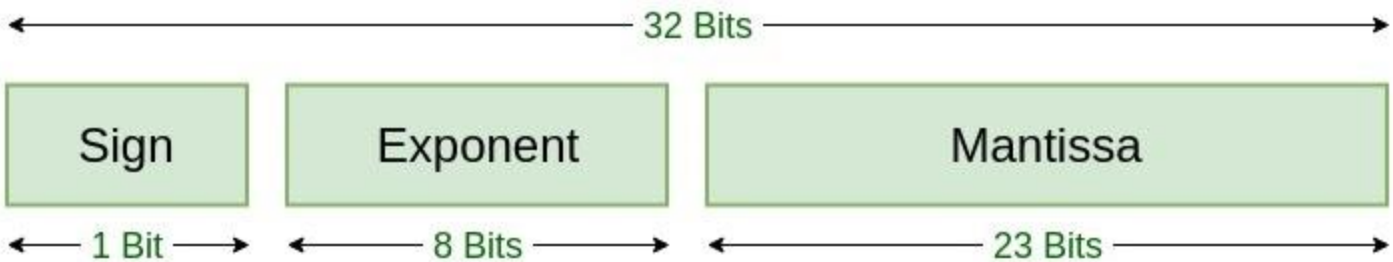
- It gives provision for +-0 and +-∞ (by reserving certain pattern for mantissa and exponent).
- There are number of modes for storage from half precision 16 to very lengthy notation.
- The floating-point number can be stored either in implicit normalize form or in fractional form.
- If the biased exponent bit is K-bits then Bias is $2^{k-1}-1$.
- If certain Mantissa/Exponent pattern does not denote any number. (NAN i.e., not a number)

IEEE 754:

Name	Common Name	Mantissa Bits (M)	Exponent Bits (K)	Exponent Bias ($2^{k-1}-1$)
binary16	Half Precision	10	5	$2^{5-1}-1 = 15$
binary32	Single Precision	23	8	$2^{8-1}-1 = 127$
binary64	Double Precision	52	11	$2^{11-1}-1 = 1023$
binary128	Quadruple Precision	112	15	$2^{15-1}-1 = 16383$
binary256	Octuple Precision	236	19	$2^{19-1}-1 = 262143$



Sign bit S (1)	Exponent E (8)	Mantissa M (23)	Value
0/1	00...0 E = 0	00...0 M = 0	± 0
0/1	11...1 E = 255	00...0 M = 0	± ∞
0/1	1≤ E ≤ 254	M = xx...x	Implicit Normalized No.
0/1	E = 0	M ≠ 0	Fractional form (Un-normalized)
	E = 255	M ≠ 0	NAN



Example 1:

1. Consider the number is in IEEE 754 single precision and bias is 127, identify the number.

S = 1, E = 10000001, F = 11110000000000000000000

Example 2:

1. Given number is in 32-bit single precision.

   00111111001101101000000000000000

Can you tell the number in Decimal.

Single Precession Normalized Number

- The normalized significand is $1.m$ (binary dot). The binary point is before bit-22 and the 1 (one) is not present explicitly.
- The sign bit $s = 1$ for a $-$ve number is zero (0) for a $+$ve number.
- The value of a normalized number is

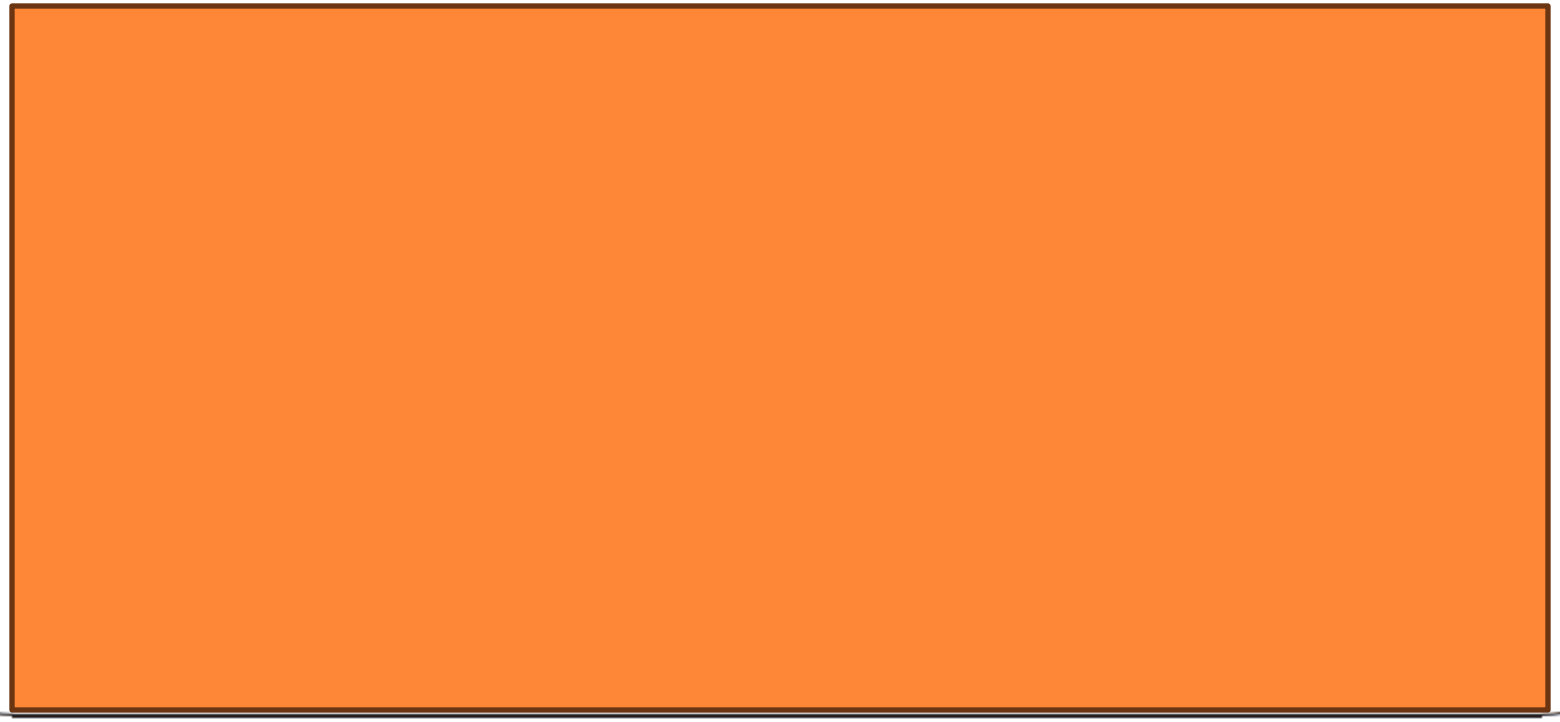
$$(-1)^s \times 1.m \times 2^{e-127}$$

An Example

Consider the following 32-bit pattern

1 1011 0110 011 0000 0000 0000 0000 0000

The value is



An Example

Consider the following 32-bit pattern

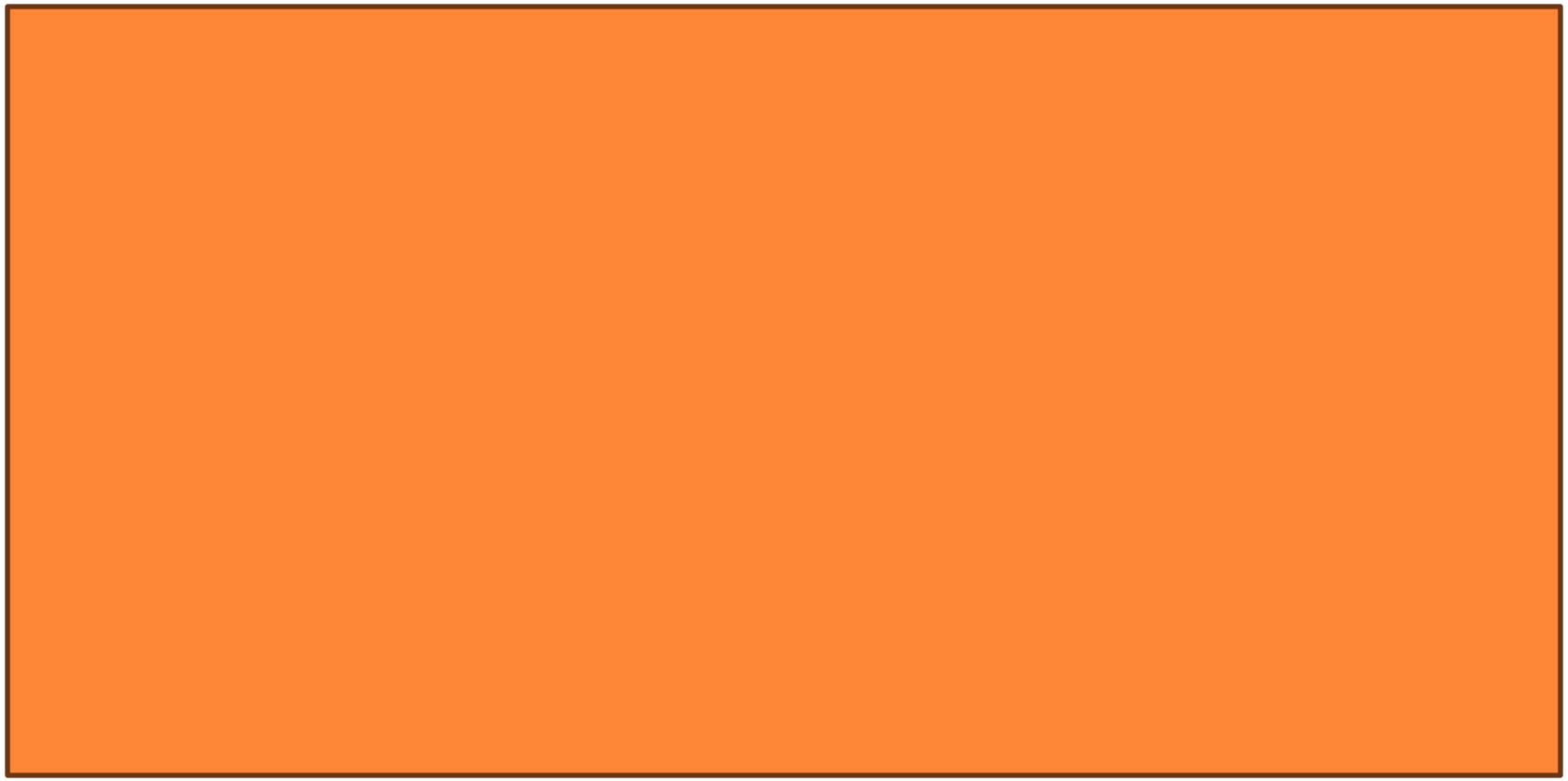
1 1011 0110 011 0000 0000 0000 0000 0000

The value is

$$
\begin{aligned}
& (-1)^{1} \times 2^{10110110-01111111} \times 1.011 \\
= \; & -1.375 \times 2^{55} \\
= \; & -49539595901075456.0 \\
= \; & -4.9539595901075456 \times 10^{16}
\end{aligned}
$$

An Example

Consider the decimal number: $+105.625$. The equivalent binary representation is



An Example

Consider the decimal number: $+105.625$. The equivalent binary representation is

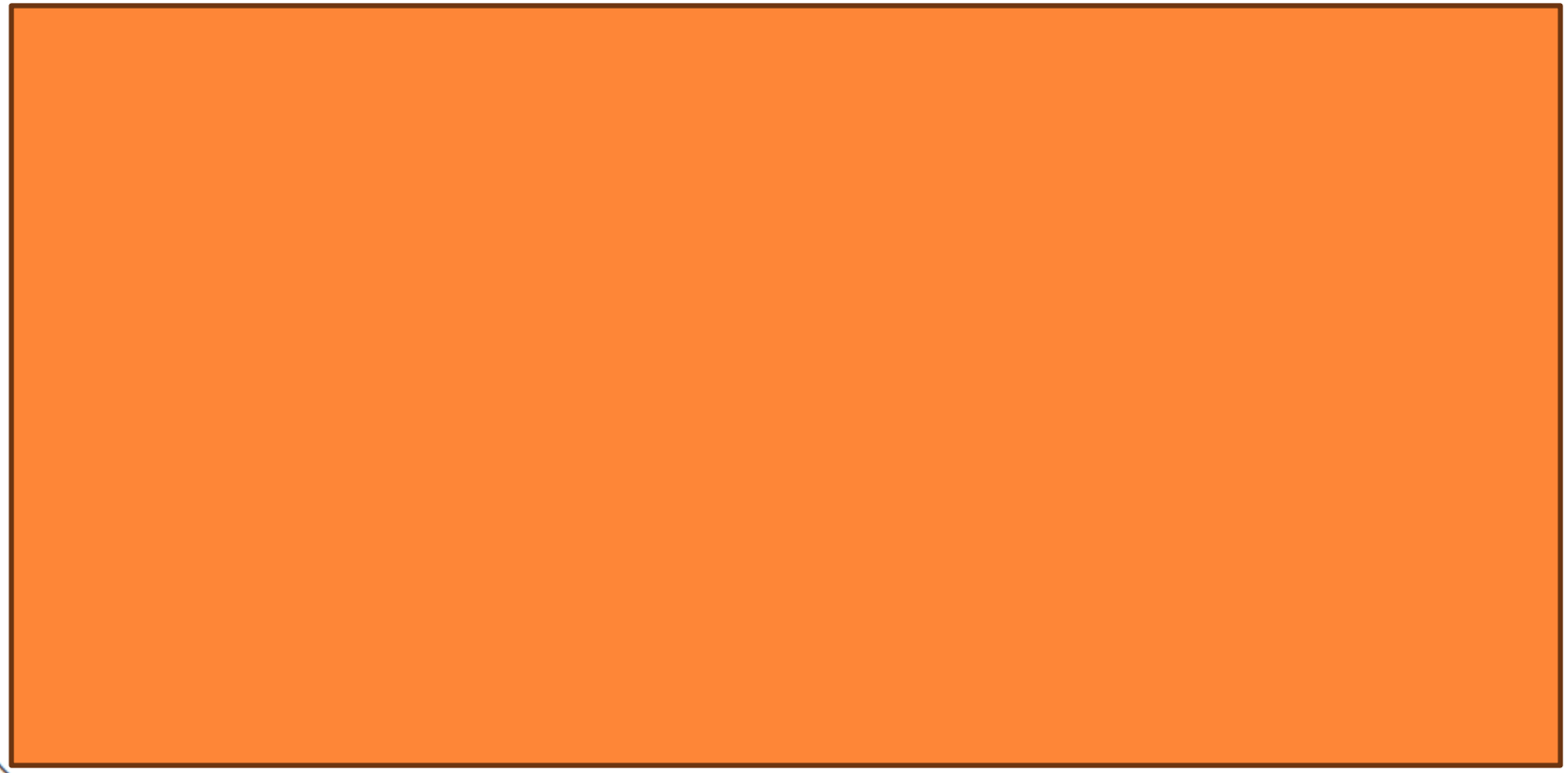
$$
\begin{aligned}
& +1101001.101 \\
= \; & +1.101001101 \times 2^{6} \\
= \; & +1.101001101 \times 2^{133-127} \\
= \; & +1.101001101 \times 2^{10000101-01111111}
\end{aligned}
$$

In IEEE 754 format:

0 1000 0101 101 0011 0100 0000 0000 0000

An Example

Consider the decimal number: $+2.7$. The equivalent binary representation is



An Example

Consider the decimal number: $+2.7$. The equivalent binary representation is

$$
\begin{aligned}
& +10.10\ 1100\ 1100\ 1100\cdots \\
= \ & +1.010\ 1100\ 1100\ 1100\cdots \times 2^{1} \\
= \ & +1.010\ 1100\ 1100\ 1100\cdots \times 2^{128-127} \\
= \ & +1.010\ 1100\ 1100\ 1100\cdots \times 2^{10000000-01111111}
\end{aligned}
$$

In IEEE 754 format (approximate):

$$0\ 1000\ 0000\ 010\ 1100\ 1100\ 1100\ 1100\ 1101$$