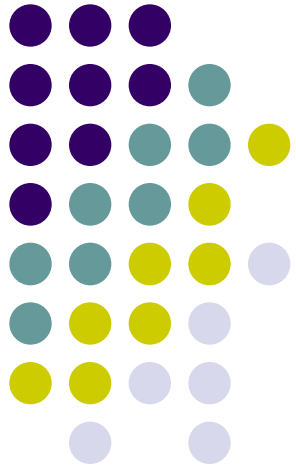
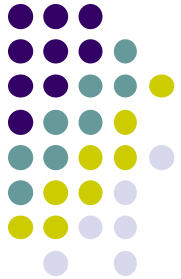
Chapter 9
Arithmetic

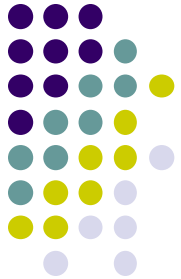


Contents



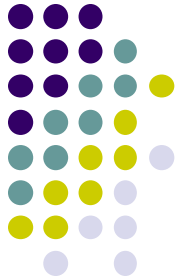
- Integer Representation (1.4)
- 9.1 Addition and Subtraction of Signed Numbers
- 9.2 Design of Fast Adders
- 9.3 Multiplication of Unsigned Numbers
- 9.4 Multiplication of Signed Numbers
- 9.6 Integer Division
- 9.7 Floating-Point Numbers and Operation
- Summary

Integer Representation



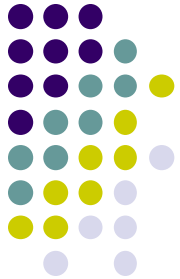
- Unsigned Representation
  - Implies that the representation does not have a separate bit to represent the sign bit of the number.
  - Note that numbers can be either positive or negative in some unsigned representation.
- Unsigned Integer
  - Unsigned Non-Negative Representation
  - Unsigned Two's Complement Representation

Integer Representation



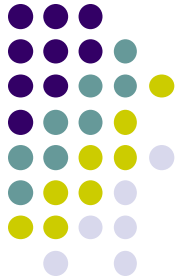
- Unsigned Non-Negative Representation
  - Treats every number as either zero or a positive value.
  - This representation can only represent non-negative numbers.
  - A n-bit number can have a value ranging from 0 to 2^n-1.
  - Example: An 8-bit word could be used to represent the numbers from 0 to 255
    - 10000000 = 128
    - 11111111 = 255

Integer Representation



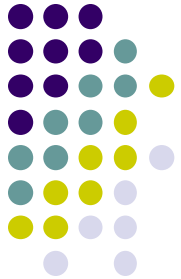
- Unsigned Two's Complement Representation
  - This representation can represent both positive and negative numbers.
  - A positive number (or zero) is represented exactly the same as in non-negative representation.
  - A negative number is represented as the bitwise complement of its absolute value +1.

Integer Representation



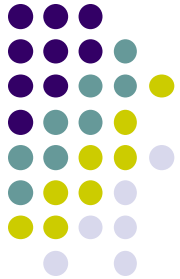
- Unsigned Two's Complement Representation
  - Bitwise Complement (One's Complement)
    - Take the Boolean complement of each bit of the number. That is, set each 1 to 0 and each 0 to 1.
  - Example: -5_{10}
    - $5_{10} = 0101_2$
    - Bitwise complement of 0101 is 1010
    - 1010 + 1 = 1011

Integer Representation



- Unsigned Two's Complement Representation
  - Positive numbers (and zero) have a 0 as the leading bit and negative numbers have a 1 as the most significant bit (MSB).
  - A n-bit number can have a value ranging from -2^{n-1} to $2^{n-1}-1$.

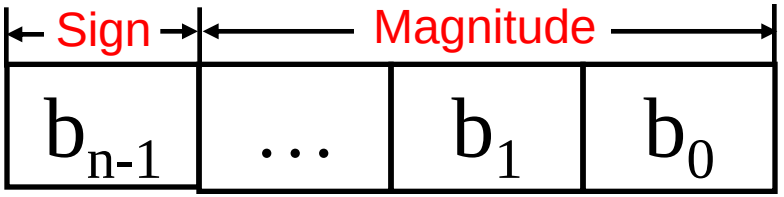
Integer Representation



- Signed Integer
  - Signed-Magnitude Representation
  - Signed Two's Complement Representation
  - Signed One's Complement Representation
- Signed-Magnitude Representation
  - Rule
    - The sign part is a 1-bit value that is 0 for positive numbers, and 1 for negative numbers.
    - The magnitude part is an (n-1)-bit value that holds the absolute value of the number in the same format of unsigned non-negative numbers .

Integer Representation

- Signed-Magnitude Representation
  - B = $b_{n-1}\ldots b_1 b_0$
    - b_{n-1} = 0, B is a positive number.
    - b_{n-1} = 1, B is a negative number.
  - Example
    - $+18_{10}$ = 00010010
    - -18_{10} = 10010010
    - $+0_{10}$ = 00000000
    - -0_{10} = 10000000



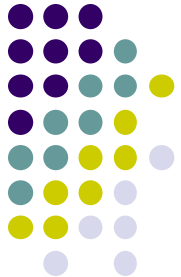
Integer Representation

- Signed-Magnitude Representation
  - In general, if an n-bit sequence of binary digits $b_{n-1}\ldots b_1 b_0$ is interpreted as an signed integer B, its value is

$$V(B) = \begin{cases} \sum_{i=0}^{n-2} 2^i b_i & \text{if } b_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i b_i & \text{if } b_{n-1} = 1 \end{cases}$$

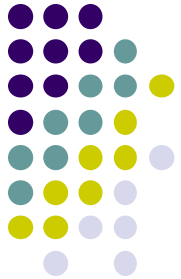
$$-(2^{n-1}-1) \leqslant V(B) \leqslant 2^{n-1}-1$$

Integer Representation



- Signed-Magnitude Representation
  - Drawbacks
    - Addition and subtraction require a consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation.
    - There are two representations of 0.
      - ＋0 = 00000000
      - －0 = 10000000

Integer Representation



- Signed Two's Complement Representation
  - Rule
    - The sign part is a 1-bit value that is 0 for positive numbers, and 1 for negative numbers.
    - The magnitude part is an (n-1)-bit value that is equivalent to a unsigned two's complement number.
  - Example
    - +18 = 00010010
    - - 18 = 11101110
    - + 0 = 00000000
    - - 0 = 00000000

Integer Representation

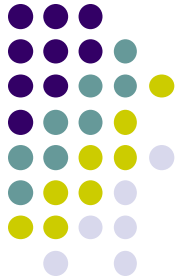
- Signed Two's Complement Representation
  - The general case B= $b_{n-1}\ldots b_1 b_0$

$$V(B) = \begin{cases} \sum_{i=0}^{n-2} 2^i b_i & B \geqslant 0 \\ -2^{n-1} b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i & B < 0 \end{cases}$$

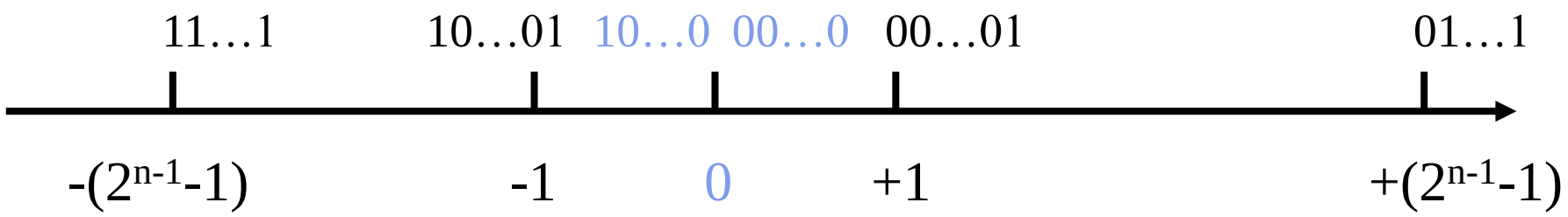
$$= -2^{n-1} b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i \quad \text{(for both positive and negative numbers)}$$

$$-2^{n-1} \leqslant V(B) \leqslant 2^{n-1} - 1$$

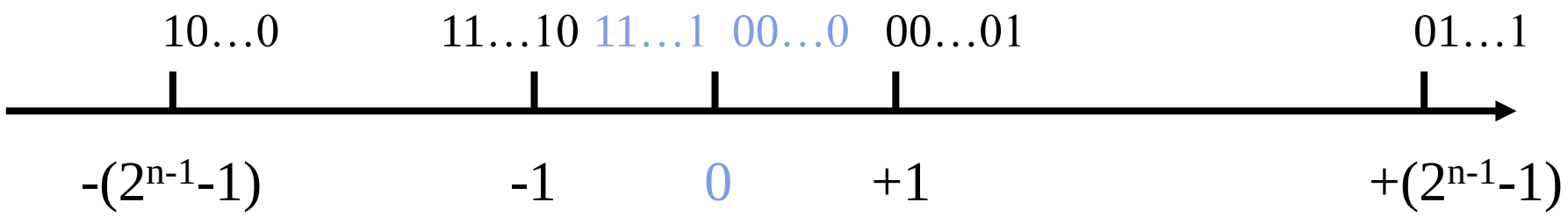
Integer Representation



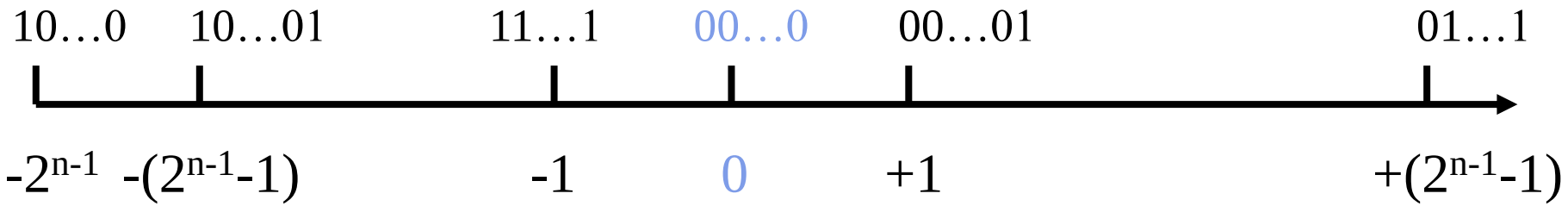
- Signed One's Complement Representation
  - Rule
    - The sign part is a 1-bit value that is 0 for positive numbers, and 1 for negative numbers.
    - The magnitude part is an (n-1)-bit value that complementing each bit of the absolute of the integer.
  - Example
    - +18 = 00010010
    - - 18 = 11101101
    - + 0 = 00000000
    - - 0 = 11111111



signed-magnitude

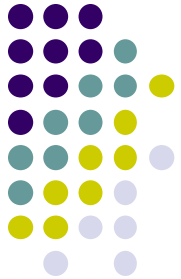


signed one's complement



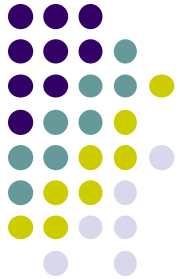
signed two's complement

Integer Representation



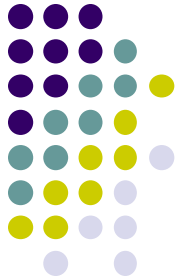
- Conclusions
  - In all three systems, the leftmost bit is 0 for positive numbers and 1 for negative numbers.
  - Positive values have identical representations in all systems, but negative values have different representations.
  - The 2’s-complement of a number is obtained by adding 1 to the 1’s-complement of that number.
  - The Signed-Magnitude system is the simplest representation, but it is also the most awkward for addition and subtraction operations.
  - The 2’s-complement system is the most efficient method for performing addition and subtraction operations.

Integer Representation



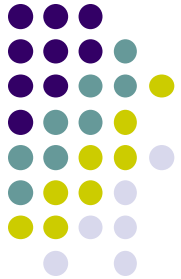
- Converting between Different Bit Lengths
  - Signed-Magnitude Numbers
    - Move the sign bit to the new left-most position and fill in with zeros.
    - Example
      - +18 = 00010010 (8 bits)
      - +18 = 0000000000010010 (16 bits)
      - - 18 = 10010010 (8 bits)
      - - 18 = 1000000000010010 (16 bits)

Integer Representation



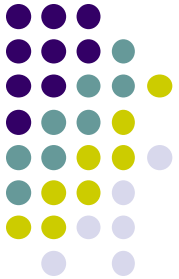
- Converting between Different Bit Lengths
  - Signed Two's Complement Numbers
    - Example
      - +18 = 00010010 (8 bits)
      - +18 = 0000000000010010 (16 bits)
      - - 18 = 11101110 (8 bits)
      - - 32568 = 1000000001101110 (16 bits)

Integer Representation



- Converting between Different Bit Lengths
  - Signed Two's Complement Numbers
    - Sign Extension
      - Move the sign bit to the new left-most 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.
      - Example
        - - 18 = 11101110 (8 bits)
        - - 18 = 1111111111101110 (16 bits)

9.1 Addition and Subtraction of Signed Numbers

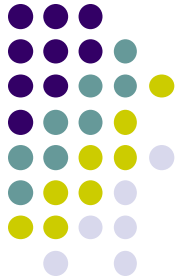


- Addition of 1-bit Positive Numbers

0	1	0	1
+ 0	+ 0	+ 1	+ 1
0	1	1	1 0

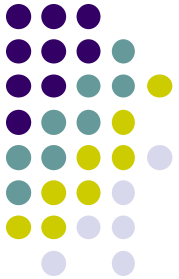
Carry-out

9.1 Addition and Subtraction of Signed Numbers



- Addition of n-bit Signed Numbers
  - Rule (In Signed Two's Complement Form)
    - To add two numbers, add their n-bit representations, treating the sign bit as the most significant bit (MSB), ignoring the carry-out signal from the MSB position.
    - The sum will be algebraically correct value in the two's-complement representation as long as the answer is in the range - 2^{n-1} through $+2^{n-1} -1$.

9.1 Addition and Subtraction of Signed Numbers



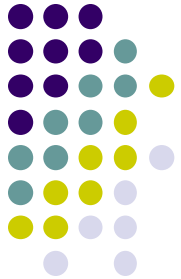
- Addition of n-bit Signed Numbers
  - Example
    - P14 Figure 1.6 (a)-(d)
    - (+5)+(+4)

$$\begin{array}{r} 0\ 1\ 0\ 1 \\ +0\ 1\ 0\ 0 \\ \hline 1\ 0\ 0\ 1 \end{array} \longleftarrow \textbf{overflow}$$

    - (-7)+(-6)

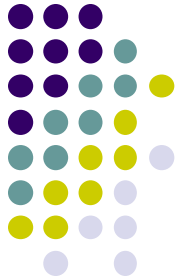
$$\begin{array}{r} 1\ 0\ 0\ 1 \\ +1\ 0\ 1\ 0 \\ \hline 1\ 0\ 0\ 1\ 1 \end{array} \longleftarrow \textbf{overflow}$$

9.1 Addition and Subtraction of Signed Numbers



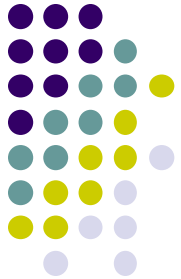
- Addition of n-bit Signed Numbers
  - Arithmetic Overflow
    - The result of an arithmetic operation is outside the representable range.
    - If two numbers are added, and they have the same sign, then overflow occurs if and only if the result has the opposite sign to both summands.
    - The carry-out signal from the sign-bit position is not a sufficient indicator of overflow when adding signed numbers. Overflow can occur whether or not there is a carry-out.

9.1 Addition and Subtraction of Signed Numbers



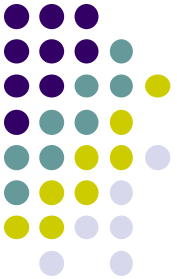
- Subtraction of n-bit Signed Numbers
  - Rule (In Signed Two's Complement Form)
    - To subtract two numbers X and Y, that is, to perform X-Y, form the 2's-complement of Y and then add it to X, as in addition rule.
    - The result will be the algebraically correct value in the two's-complement representation system if the answer is in the range - 2^{n-1} through $+2^{n-1}$ -1.
    - Essence of subtraction rule X-Y= X+(-Y)

9.1 Addition and Subtraction of Signed Numbers



- Subtraction of n-bit Signed Numbers
  - Two's Complement Operation (Negation)
    - Take the Boolean complement of each bit of the integer (including the sign bit). That is, set each 1 to 0 and each 0 to 1.
    - Treating the result as an unsigned binary integer, add 1.
  - Example
    - P14 Figure 1.6 (e)-(j)

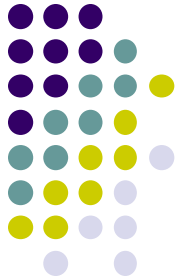
9.1 Addition and Subtraction of Signed Numbers



- Subtraction of n-bit Signed Numbers
  - Example: M = 7, S = -7, M – S = ?
    - M = 7 = 0111
    - S = -7 = 1001
    - -S = 0111

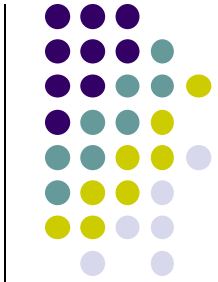
$$\begin{array}{r} 0\ 1\ 1\ 1 \\ +0\ 1\ 1\ 1 \\ \hline 1\ 1\ 1\ 0 \end{array} \longleftarrow \text{overflow}$$

9.1 Addition and Subtraction of Signed Numbers



- Subtraction of n-bit Signed Numbers
  - Conclusions
    - The examples in Figure 1.6 (P14) show that two, n-bit, signed numbers can be added using n-bit binary addition, treating the sign bit the same as the other bit.
    - In other words, a logic circuit that is designed to add unsigned binary numbers can also be used to add signed numbers in 2's-complement.

9.1 Addition and Subtraction of Signed Numbers

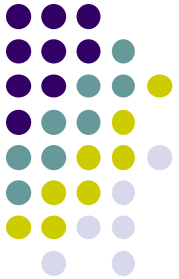


- 1-bit Full Adder
  - Example: X=7, Y=6, X+Y= ?

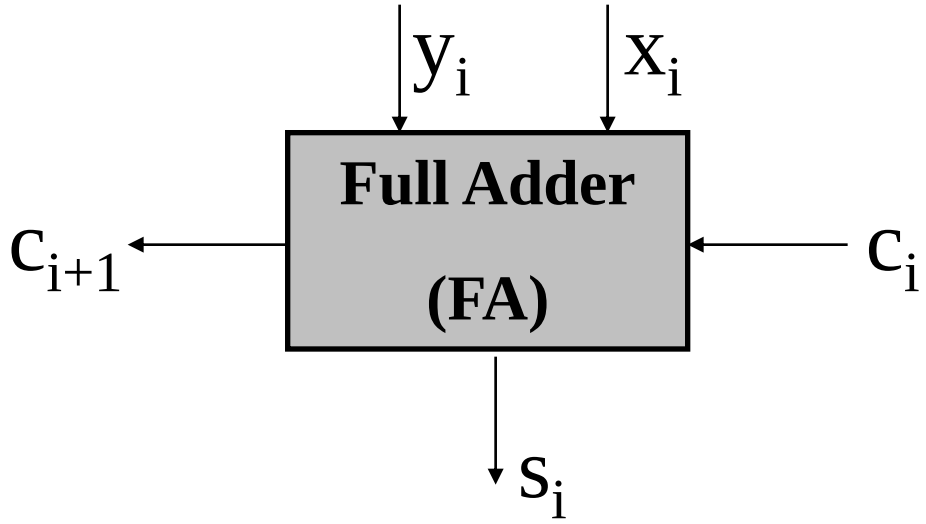
$$\begin{array}{rcccc} X & 7 & 0\ 1 & 1 & 1 \\ +\,Y & +\,6 & +\ {}_{0}0\ {}_{1}1\ {}_{1}1\ {}_{0}0\ {}_{0} & & \\ \hline Z & 13 & 1\ 1 & 0 & 1 \end{array}$$

$$\begin{array}{ccc} & x_i & \\ c_{i+1} & y_i & c_i \\ \hline & s_i & \end{array}$$

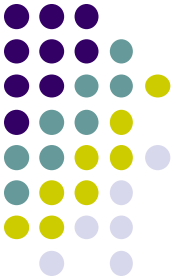
9.1 Addition and Subtraction of Signed Numbers



- 1-bit Full Adder
  - Full Adder
    - A full adder circuit takes three bits of input, and produces a two-bit output consisting of a sum and a carry out.



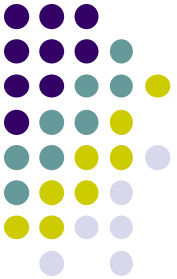
9.1 Addition and Subtraction of Signed Numbers



- 1-bit Full Adder
  - Logic Truth Table

x_i	y_i	c_i	s_i	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

9.1 Addition and Subtraction of Signed Numbers

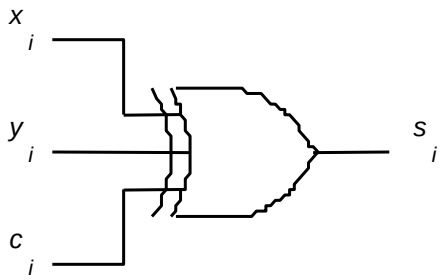


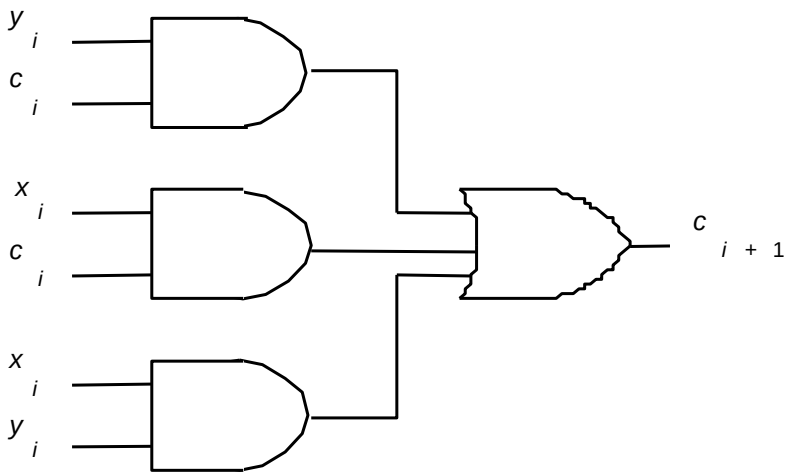
- 1-bit Full Adder
  - Logic Expressions

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

$$= x_i \oplus y_i \oplus c_i$$

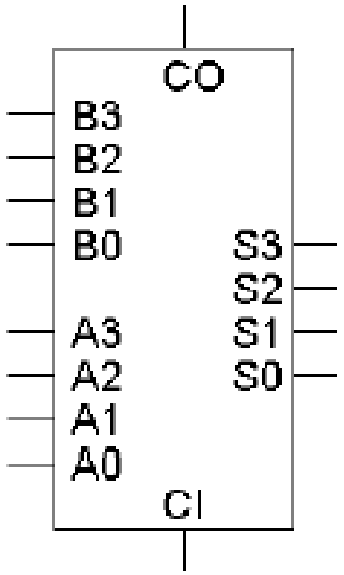
$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$



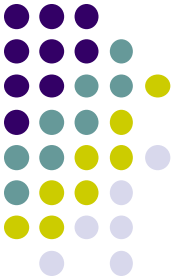


9.1 Addition and Subtraction of Signed Numbers

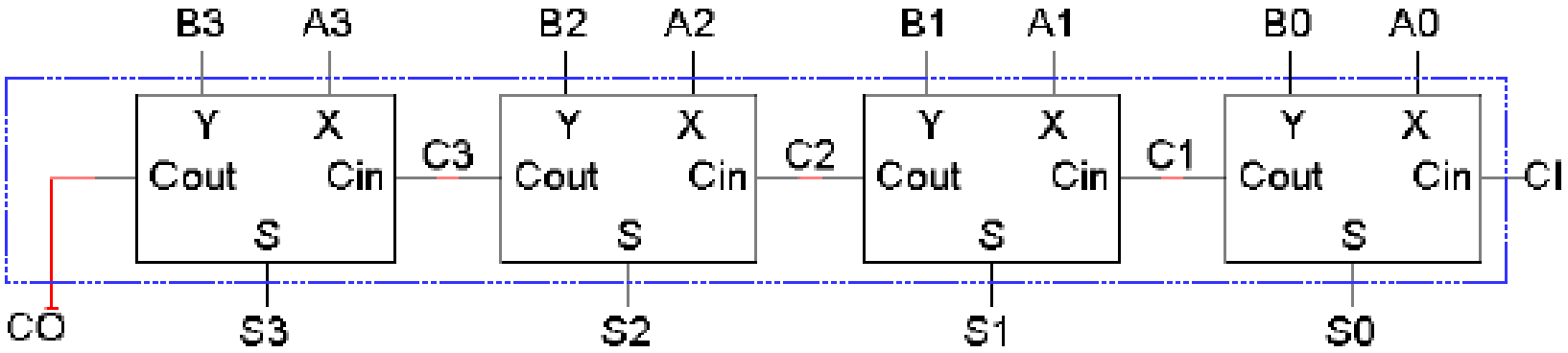
- A 4-bit Adder
  - Four full adders together make a 4-bit adder.
  - There are nine total inputs:
    - Two 4-bit numbers, A3 A2 A1 A0 and B3 B2 B1 B0
    - An initial carry in, CI
  - The five outputs are:
    - A 4-bit sum, S3 S2 S1 S0
    - A carry out, CO



9.1 Addition and Subtraction of Signed Numbers

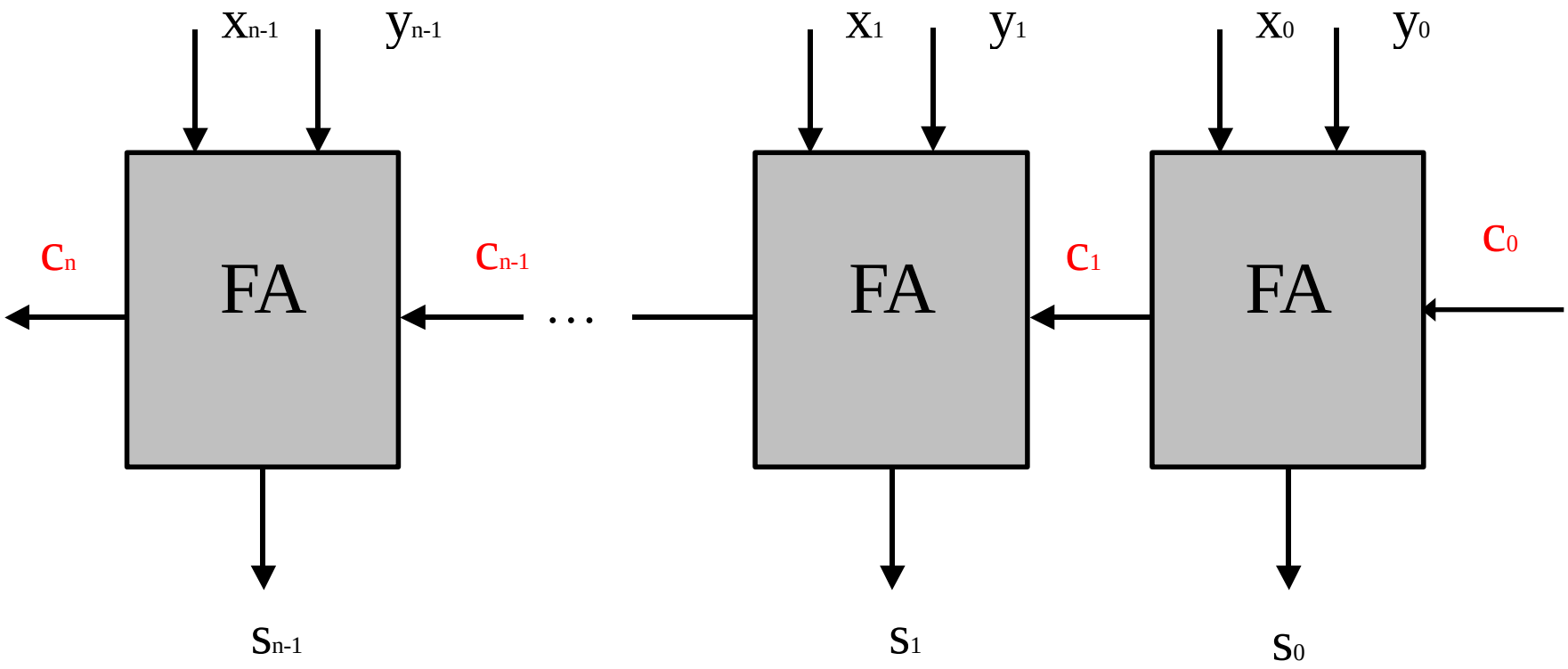


- A 4-bit Adder



9.1 Addition and Subtraction of Signed Numbers

- n-bit Ripple-Carry Adder



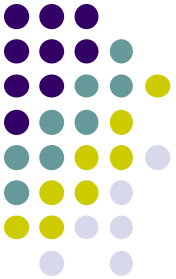
Most significant bit

(MSB) position

Least significant bit

(LSB) position

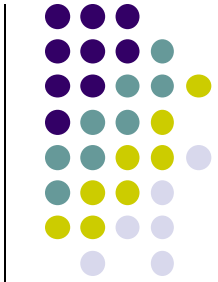
9.1 Addition and Subtraction of Signed Numbers



- n-bit Ripple-Carry Adder
  - Overflow Detect

x_{n-1}	y_{n-1}	s_{n-1}	Overflow
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

9.1 Addition and Subtraction of Signed Numbers



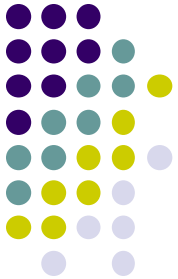
- n-bit Ripple-Carry Adder
  - Overflow Detect
    - Overflow = $x_{n-1}y_{n-1}\bar{s}_{n-1} + \bar{x}_{n-1}\bar{y}_{n-1}s_{n-1}$
    - Or Overflow = $c_n \oplus c_{n-1}$
  - Example
    - (+5)+(+4)

$$\begin{array}{r} 0\ 1\ 0\ 1 \\ +0\ 1\ 0\ 0 \\ \hline 1\ 0\ 0\ 1 \end{array} \longleftarrow \textbf{overflow}$$

(-7)+(-6)

$$\begin{array}{r} 1\ 0\ 0\ 1 \\ +1\ 0\ 1\ 0 \\ \hline \underline{1}\ 0\ 0\ 1\ 1 \end{array} \longleftarrow \textbf{overflow}$$

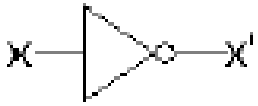
9.1 Addition and Subtraction of Signed Numbers

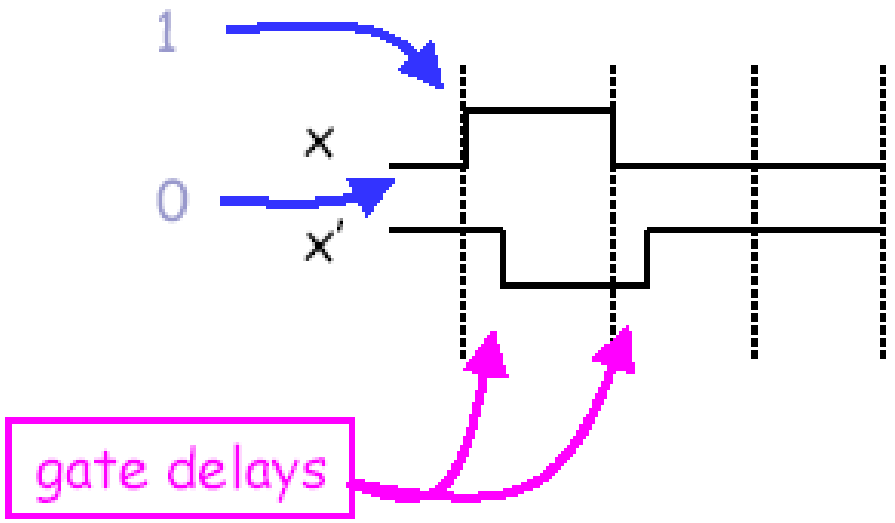


- Gate Delays
  - Every gate takes some small fraction of a second between the time inputs are presented and the time the correct answer appears on the outputs. This little fraction of a second is called a gate delay.
  - There are actually detailed ways of calculating gate delays that can get quite complicated, but for this class, let's just assume that there's some small constant delay that's the same for all gates.

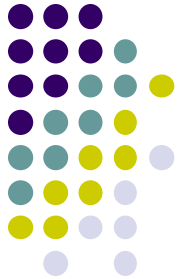
9.1 Addition and Subtraction of Signed Numbers

- Gate Delays
  - We can use a timing diagram to show gate delays graphically.





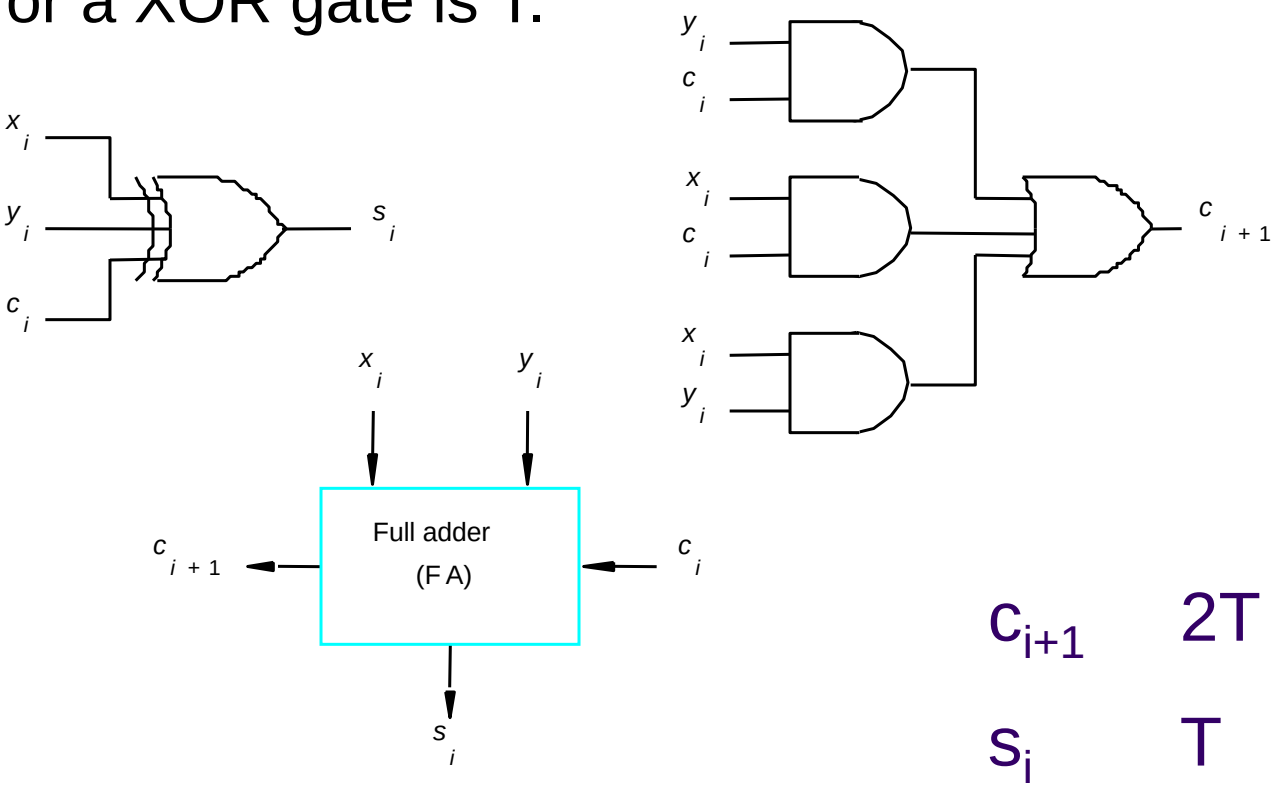
9.1 Addition and Subtraction of Signed Numbers



- Propagation Delays in the Ripple Carry Adder
  - The delay through a network of logic gates depends on the *integrated circuit electronic technology* used in fabricating the network and on *the number of gates* in the paths from inputs to outputs.
  - The delay through any combinational logic network constructed from gates in a particular technology is determined by *adding up the number of logic-gate delays along the longest signal propagation path* through the network.
  - In the case of the n-bit ripple-carry adder, the longest path is from inputs x_0,y_0 and c_0 at the LSB position to outputs c_n and s_{n-1} at the MSB position.

9.1 Addition and Subtraction of Signed Numbers

- Propagation Delays in the Ripple Carry Adder
  - Assume that the gate delay of an AND gate or an OR gate or a XOR gate is T.



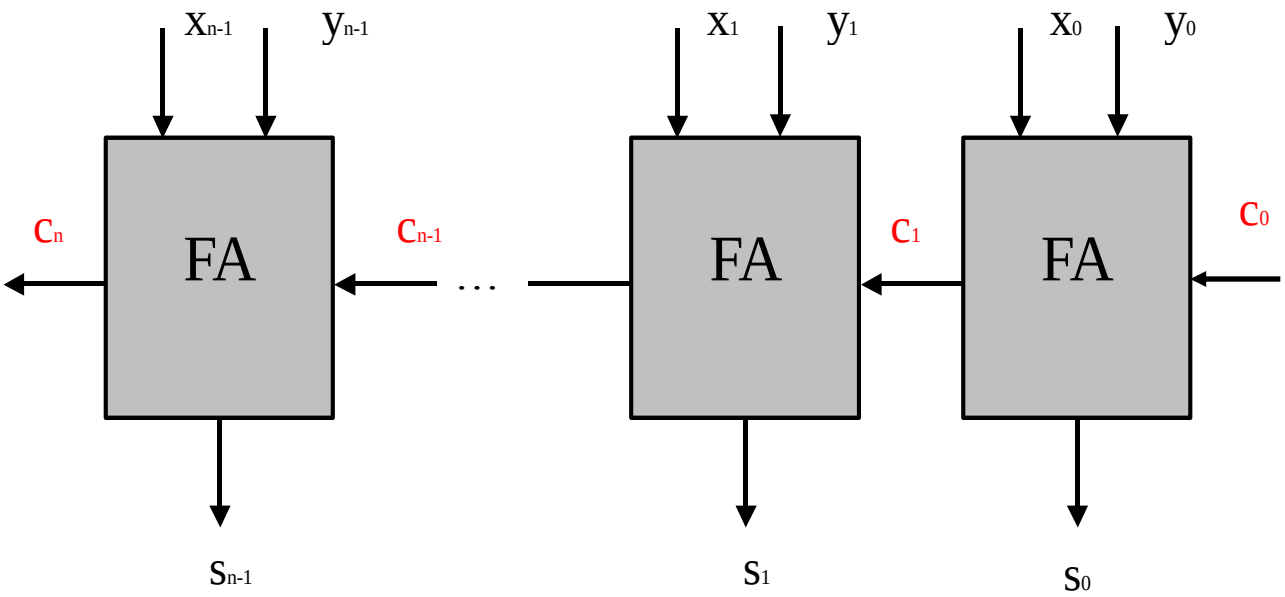
(a) Logic for a single stage

c_{i+1} 2T

s_i T

9.1 Addition and Subtraction of Signed Numbers

- Propagation Delays in the Ripple Carry Adder



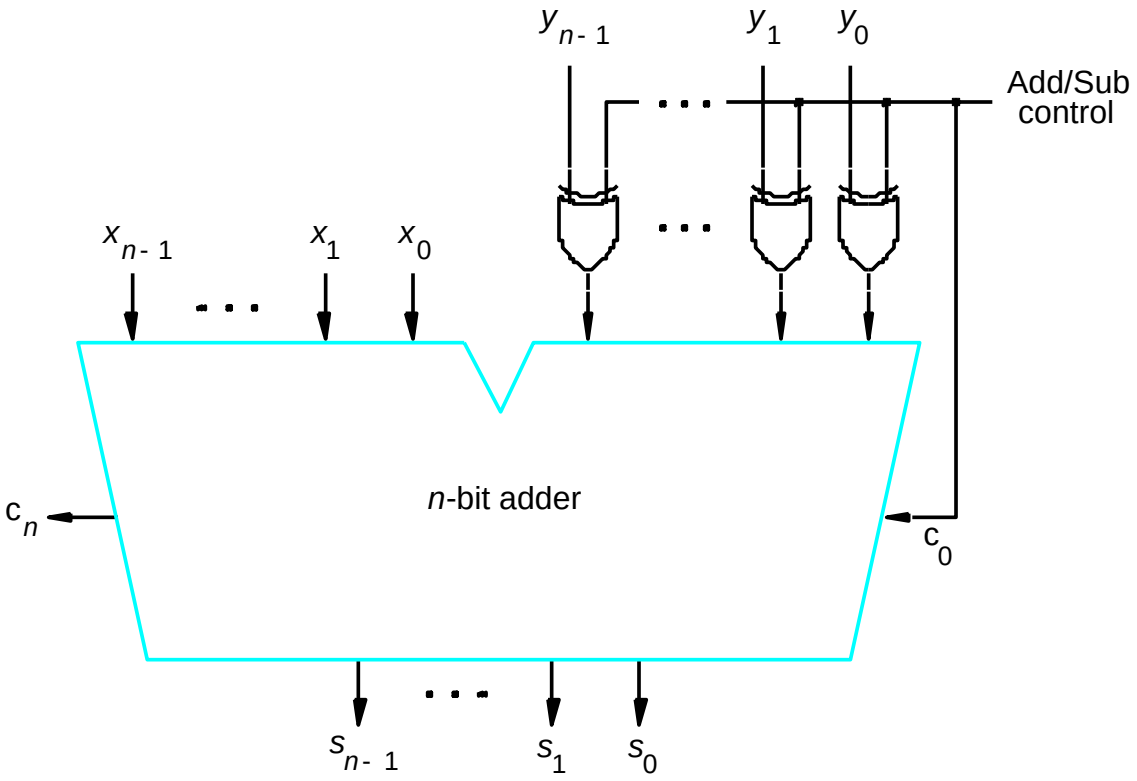
c_{n-1} 2(n-1)T

s_{n-1} 2(n-1)T+T = (2n-1)T

c_n 2(n-1)T + 2T = (2n)T

9.1 Addition and Subtraction of Signed Numbers

- Addition/Subtraction Logic Unit



$$0 \oplus A = A$$

$$1 \oplus A = \overline{A}$$

Control = 0 $c_0 = 0$

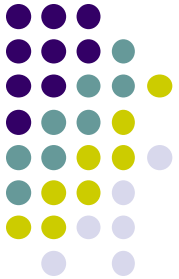
Perform Addition

Control = 1 $c_0 = 1$

Perform Subtraction

Figure 9.3. Binary addition-subtraction logic network

9.1 Addition and Subtraction of Signed Numbers



- Addition/Subtraction Logic Unit
  - An XOR gate can be added to detect the overflow.
  - All sum bits are available in 2n gate delays, including the delay through the XOR gate on the Y input.

9.2 Design of Fast Adders

- Drawback of n-bit ripple-carry adder
  - It may have too much delay in developing its outputs, s_0 through s_{n-1} and c_n.
    - s_{n-1} (2n-1)T
    - c_n (2n)T
  - Solutions
    - Use the fastest possible electronic technology in implementing the ripple-carry logic design or variations of it.
    - Use an augmented logic gate network structure.

9.2 Design of Fast Adders

- Carry-Lookahead Addition

$$s_i = x_i \oplus y_i \oplus c_i$$

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

$$= x_i y_i + (x_i + y_i) c_i$$

$G_i = x_i y_i$ (generate function for stage i)

$P_i = x_i + y_i$ (propagate function for stage i)

$$c_{i+1} = G_i + P_i c_i$$

9.2 Design of Fast Adders

- Carry-Lookahead Addition

$$P_i = x_i + y_i \longrightarrow P_i = x_i \oplus y_i$$

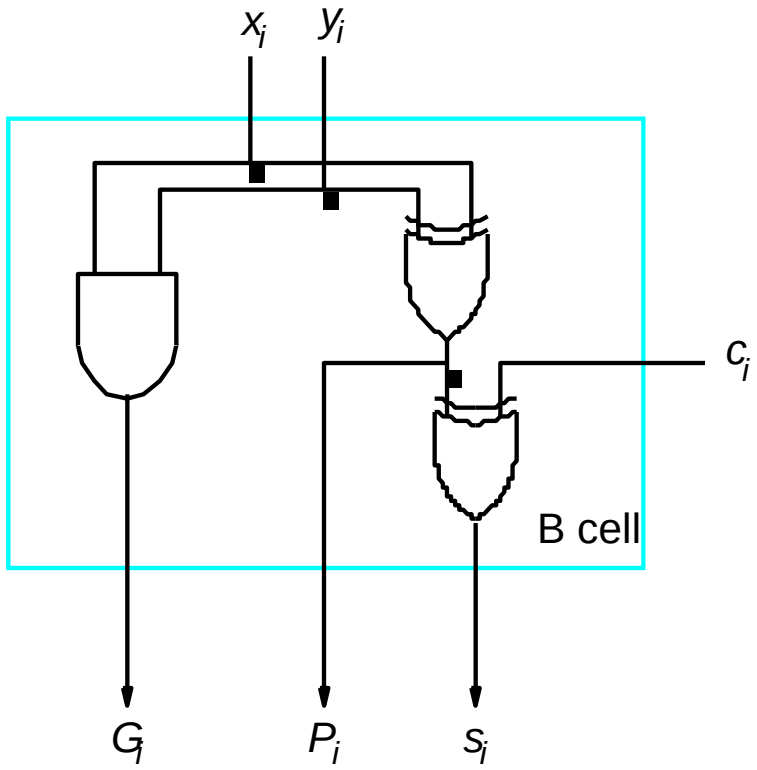
x_i	y_i	x_i+y_i	$x_i \oplus y_i$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	0

if $x_i = 1$ and $y_i = 1$

$$c_{i+1} = G_i + P_i c_i = 1 + P_i c_i = 1$$

9.2 Design of Fast Adders

- Carry-Lookahead Addition
  - A simpler circuit of bit stage



(a) Bit-stage cell

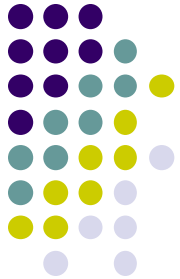
9.2 Design of Fast Adders

- Carry-Lookahead Addition

$$s_i = x_i \oplus y_i \oplus c_i$$

$$\begin{aligned} c_{i+1} &= G_i + P_i c_i \\ &= G_i + P_i(G_{i-1} + P_{i-1} c_{i-1}) \\ &= G_i + P_i G_{i-1} + P_i P_{i-1} c_{i-1} \\ &\cdots \\ &= G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \ldots + P_i P_{i-1} \ldots P_1 G_0 + P_i P_{i-1} \ldots P_0 c_0 \end{aligned}$$

9.2 Design of Fast Adders



- Carry-Lookahead Addition
  - Delays Analysis (n-bit Adder)
    - All carries can be obtained 3 gate delays after the input signal X, Y and c_0 are applied.
    - All sum bits are available 4 gate delays after the input signal X, Y and c_0 are applied.
  - CLA Summary
    - Provide supplemental logic circuits that form a carry signal into each adder stage
      - To accelerate the propagation of carries
      - Carry propagation becomes concurrent instead of sequential

9.2 Design of Fast Adders

- 4-bit Carry-Lookahead Adder

$$s_i = x_i \oplus y_i \oplus c_i \qquad i = 0,1,2,3$$

$$c_1 = G_0 + P_0 c_0$$

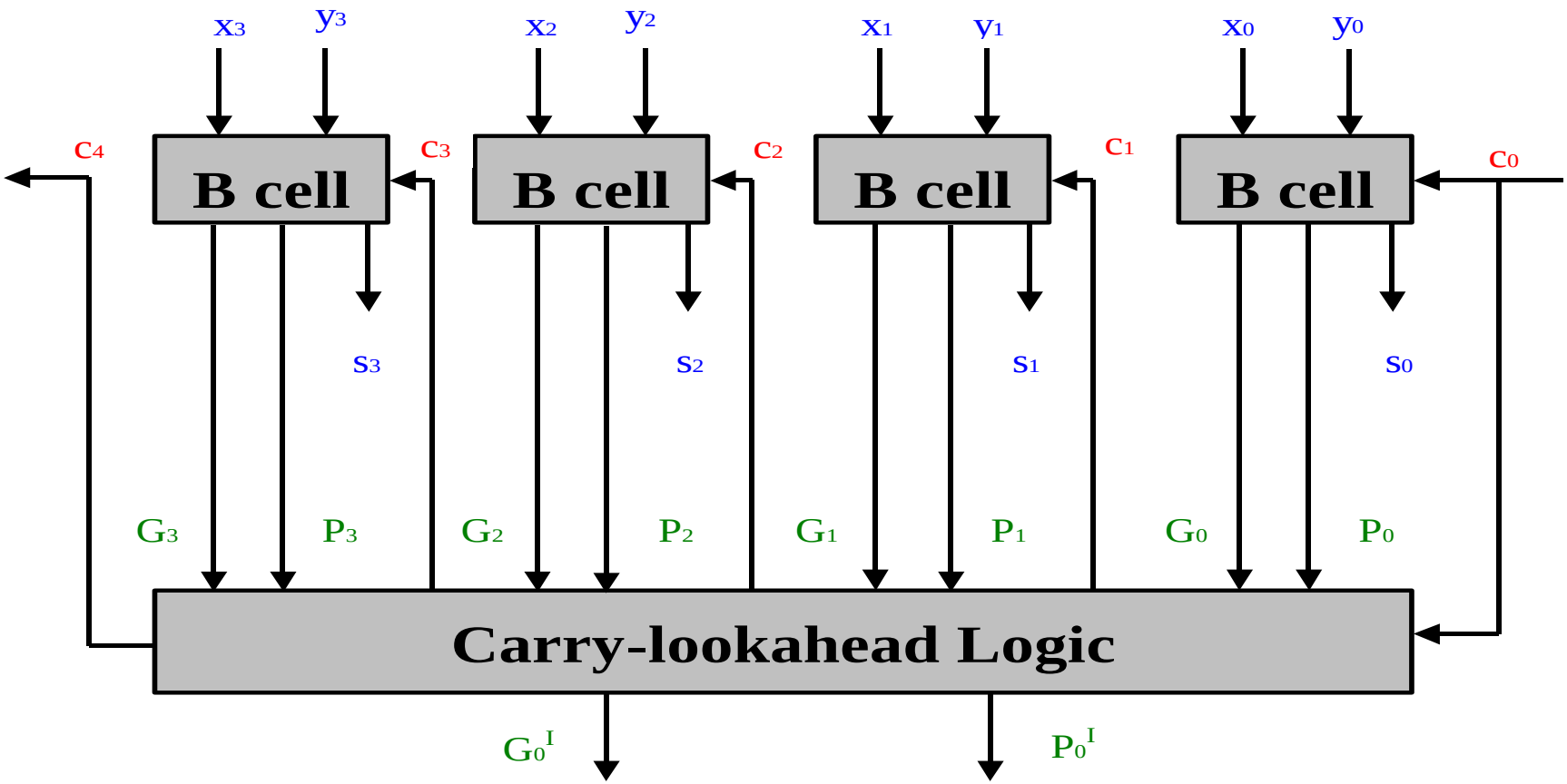
$$c_2 = G_1 + P_1 G_0 + P_1 P_0 c_0$$

$$c_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$$

$$c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$

9.2 Design of Fast Adders

- 4-bit Carry-Lookahead Adder



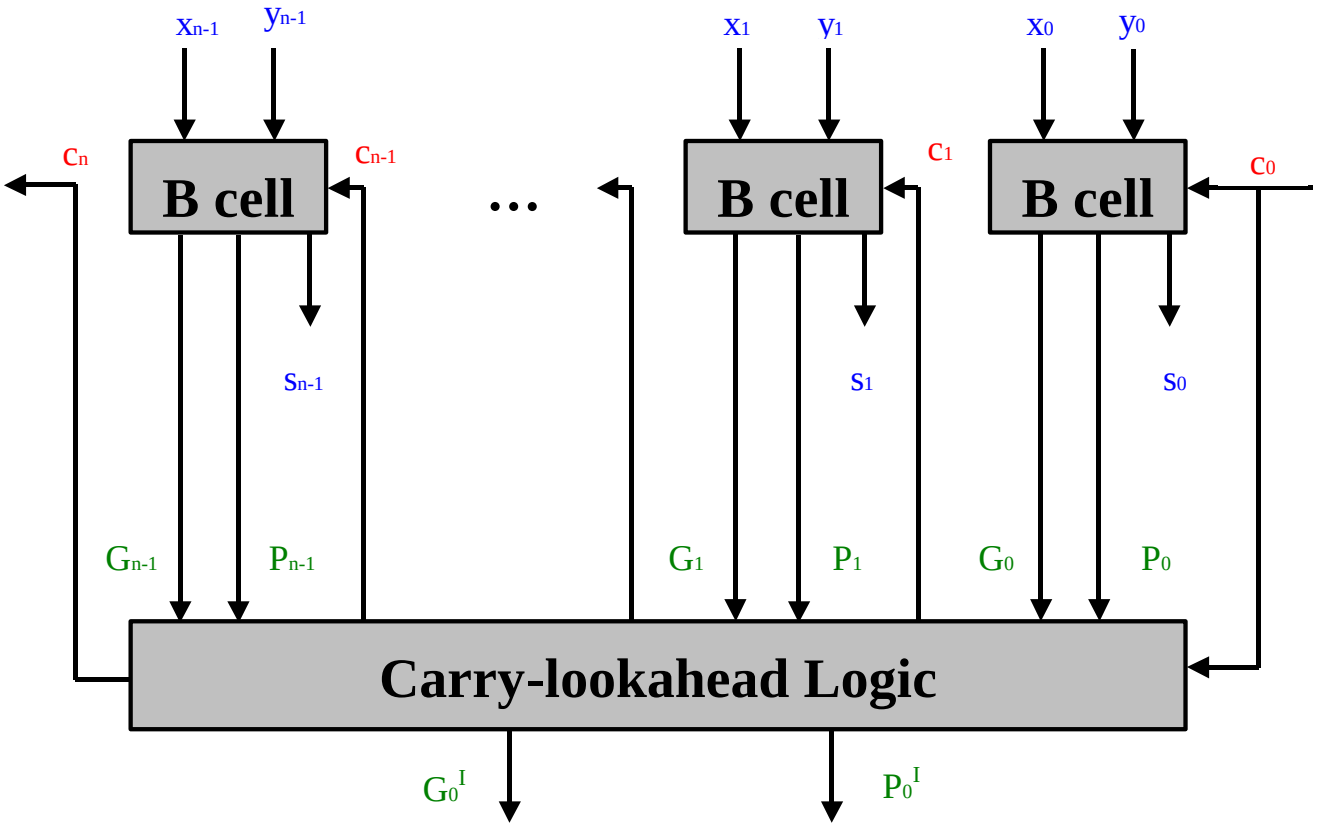
4-bit carry-lookahead adder

9.2 Design of Fast Adders

- 4-bit Carry-Lookahead Adder
  - Delays Comparison
    - 4-bit carry-lookahead adder
      - c_1 c_2 c_3 c_4 3T
      - s_1 s_2 s_3 4T
    - 4-bit ripple carry adder
      - c_4 8T
      - s_3 7T

9.2 Design of Fast Adders

- Build Longer Carry-Lookahead Adders
  - The 4-bit carry-lookahead adder design cannot be directly extended to longer operand sizes.



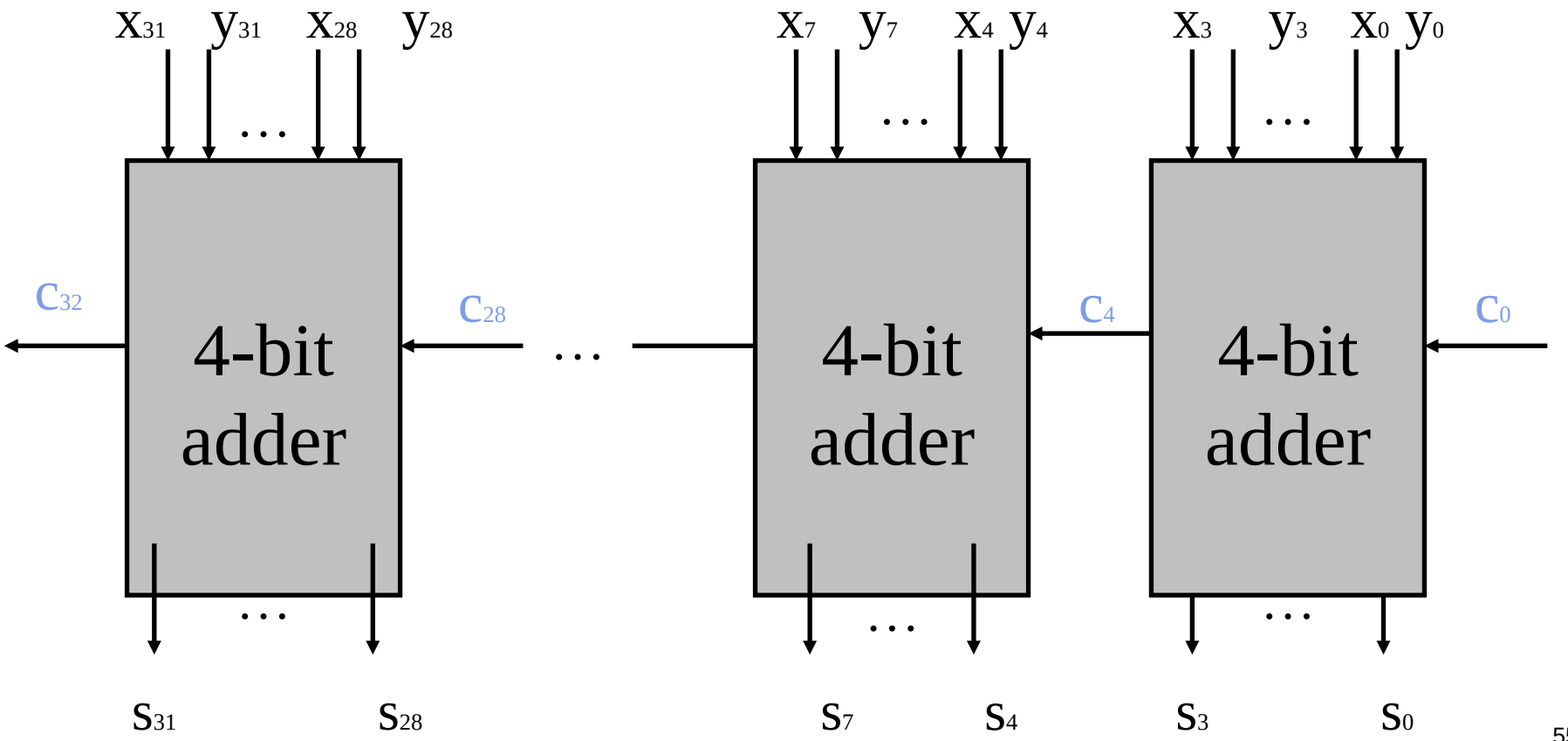
9.2 Design of Fast Adders

- Build Longer Carry-Lookahead Adders
  - Fan-In Constraints
    - Fan-in: The number of inputs to a logic gate is called its fan-in.
    - Fan-out: The number of gate inputs that the output of a logic gate drives is called its fan-out.
    - Practical circuits do not allow large fan-in and fan-out because they both have an adverse effect on the propagation delay and hence the speed of the circuit.
    - Example: Fan-in of the last AND gate and the OR gate is i+2 in generating c_{i+1}.

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \ldots + P_i P_{i-1} \ldots P_1 G_0 + P_i P_{i-1} \ldots P_0 c_0$$

9.2 Design of Fast Adders

- Build 32-bit Adders with 4-bit Adders
  - Cascade of 8 4-bit Carry-Lookahead Adders



9.2 Design of Fast Adders

- Build 32-bit Adders with 4-bit Adders
  - Cascade of 8 4-bit Carry-Lookahead Adders
    - Calculate the delays in generating sum bits s_{28}, s_{29}, s_{30},s_{31}, and c_{32} in the high-order 4-bit adder.

c_4 3T c_8 3T+2T=5T

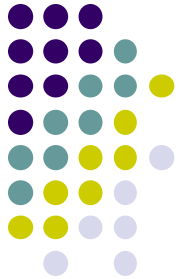
c_{12} 5T+2T …

c_{28} (6×2)T+3T=15T

c_{29} c_{30} c_{31} c_{32} 15T+2T=17T

s_{28} s_{29} s_{30} s_{31} 17T+T=18T

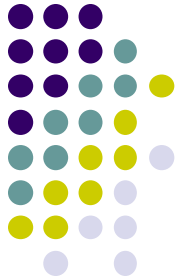
9.3 Multiplication of Unsigned Numbers



- Manual Multiplication Algorithm (unsigned & positive signed)
  - Example: M = 1101, Q = 1011, calculate P = M × Q

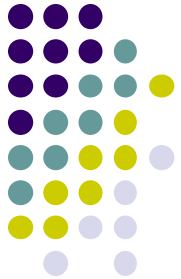
```
      1 1 0 1        Multiplicand
    × 1 0 1 1        Multiplier
  -------------
      1 1 0 1
    1 1 0 1          Partial Products
  0 0 0 0
1 1 0 1
-----------------
1 0 0 0 1 1 1 1      Product
```

9.3 Multiplication of Unsigned Numbers



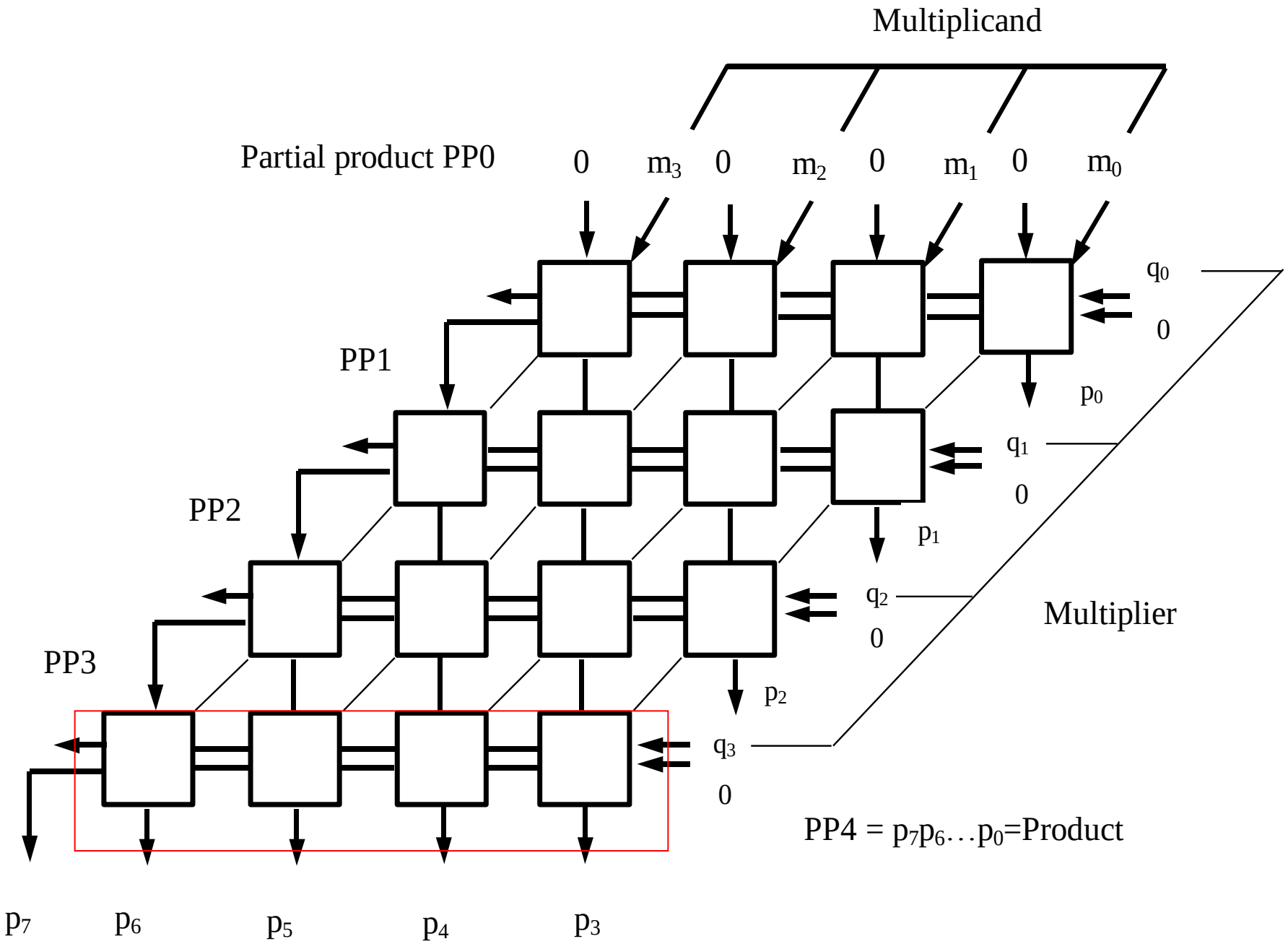
- Manual Multiplication Algorithm (unsigned & positive signed)
  - Multiplication of the multiplicand by one bit of the multiplier
    - If the multiplier bit is 1, the multiplicand is entered in the appropriate position to be added to the partial product. If the multiplier bit is 0, then 0s are entered.
  - Note
    - The multiplication product of two n-bit binary integers results in a product of up to 2n bits in length.

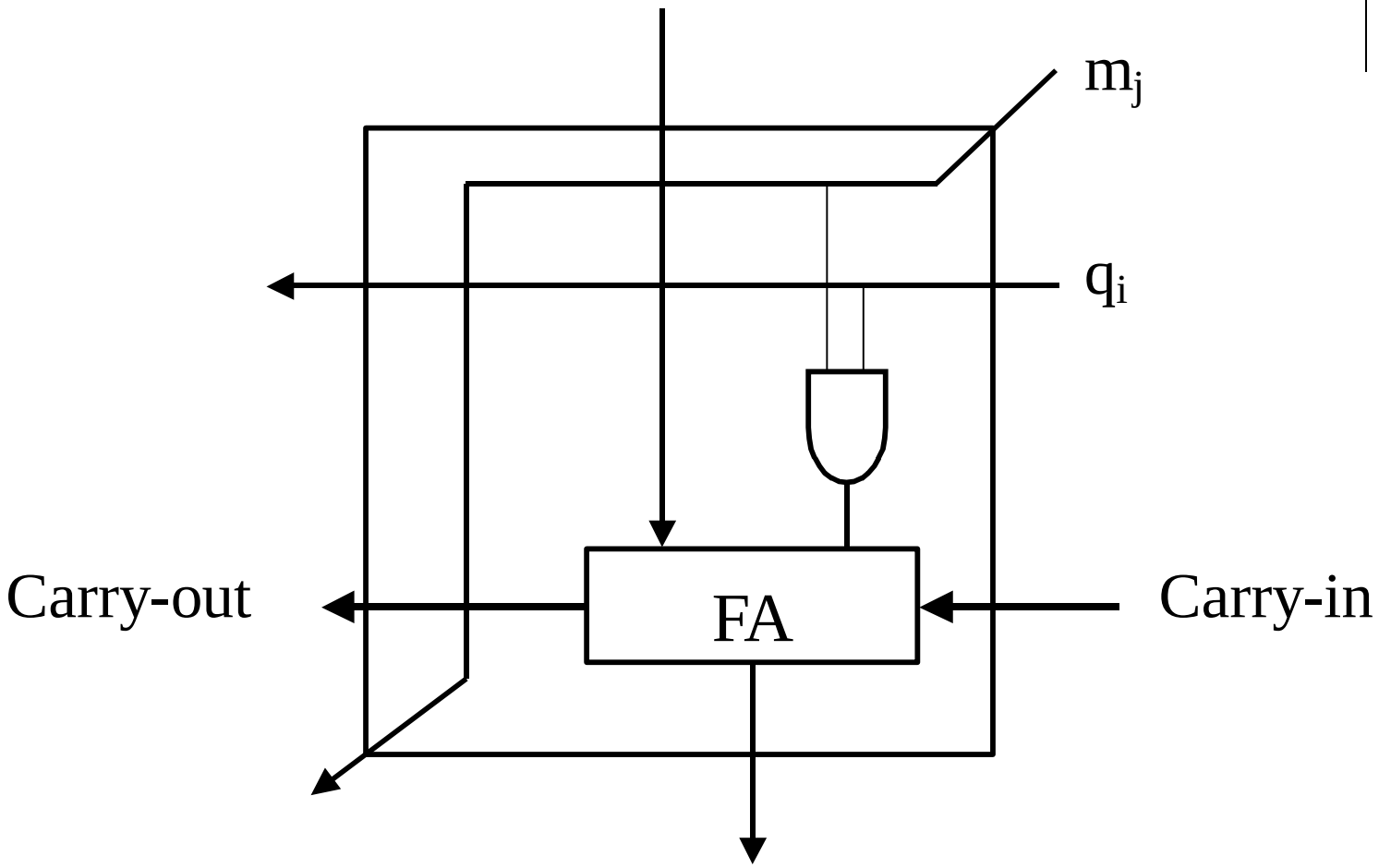
9.3 Multiplication of Unsigned Numbers



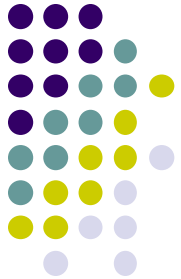
- Array Multiplication
  - $M = m_3m_2m_1m_0$ $\quad Q = q_3q_2q_1q_0$
  - $P = M \times Q = p_7p_6p_5p_4p_3p_2p_1p_0$

$$
\begin{array}{cccccccc}
 & & m_3 & m_2 & m_1 & m_0 & & \\
 & \times & q_3 & q_2 & q_1 & q_0 & & \\
\hline
 & & m_3q_0 & m_2q_0 & m_1q_0 & m_0q_0 & & \\
 & m_3q_1 & m_2q_1 & m_1q_1 & m_0q_1 & & & \\
m_3q_2 & m_2q_2 & m_1q_2 & m_0q_2 & & & & \\
m_3q_3 & m_2q_3 & m_1q_3 & m_0q_3 & & & & \\
\hline
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$


Multiplicand
Partial product PP0
0 m3 0 m2 0 m1 0 m0
PP1
PP2
PP3
q0
0
p0
q1
0
p1
q2
0
p2
q3
0
Multiplier
PP4 = p7p6…p0=Product
p7 p6 p5 p4 p3


Bit of incoming partial product (PPi)
m_j
q_i
Carry-out
FA
Carry-in
Bit of outgoing partial product [PP(i+1)]

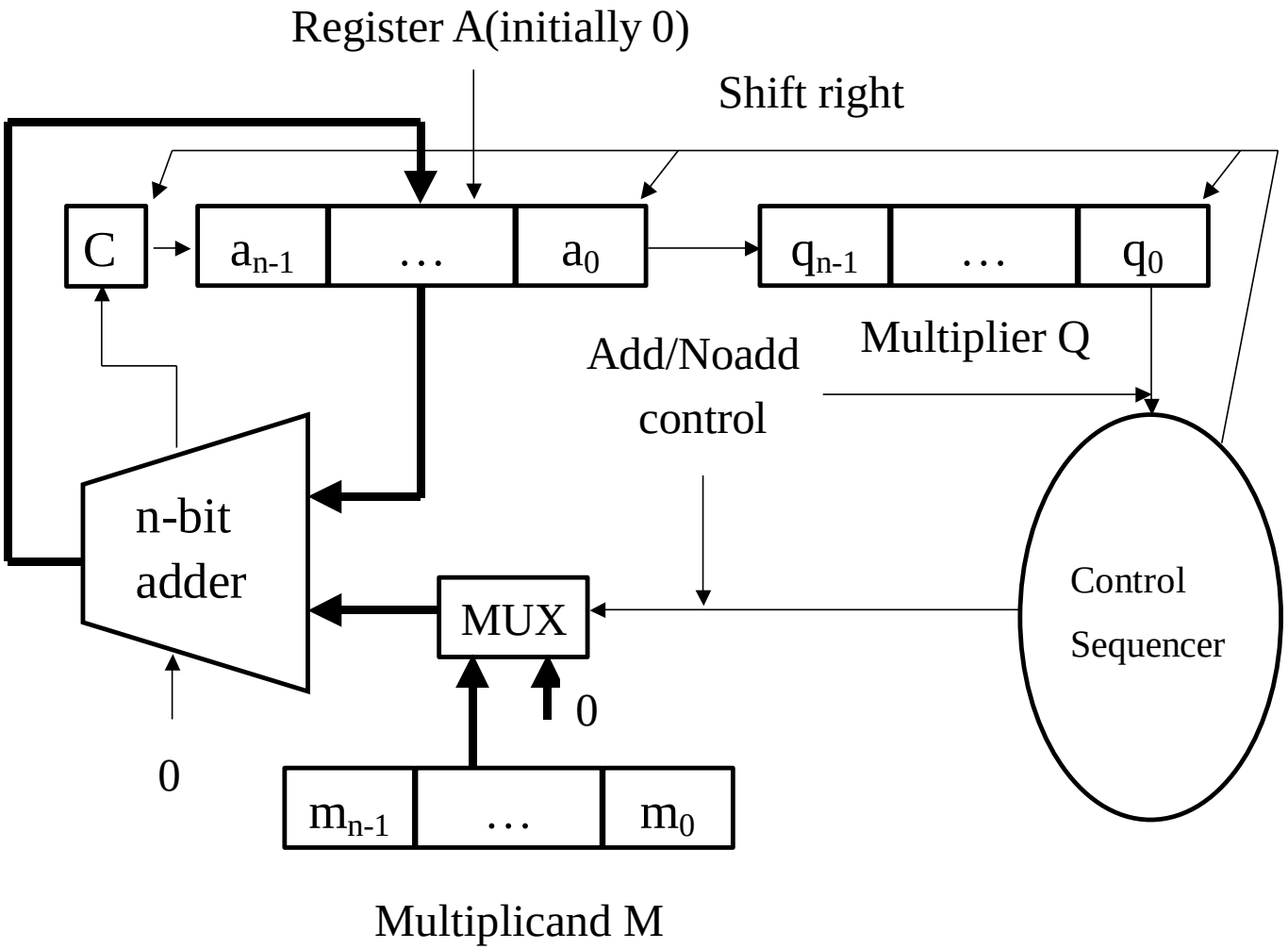
9.3 Multiplication of Unsigned Numbers

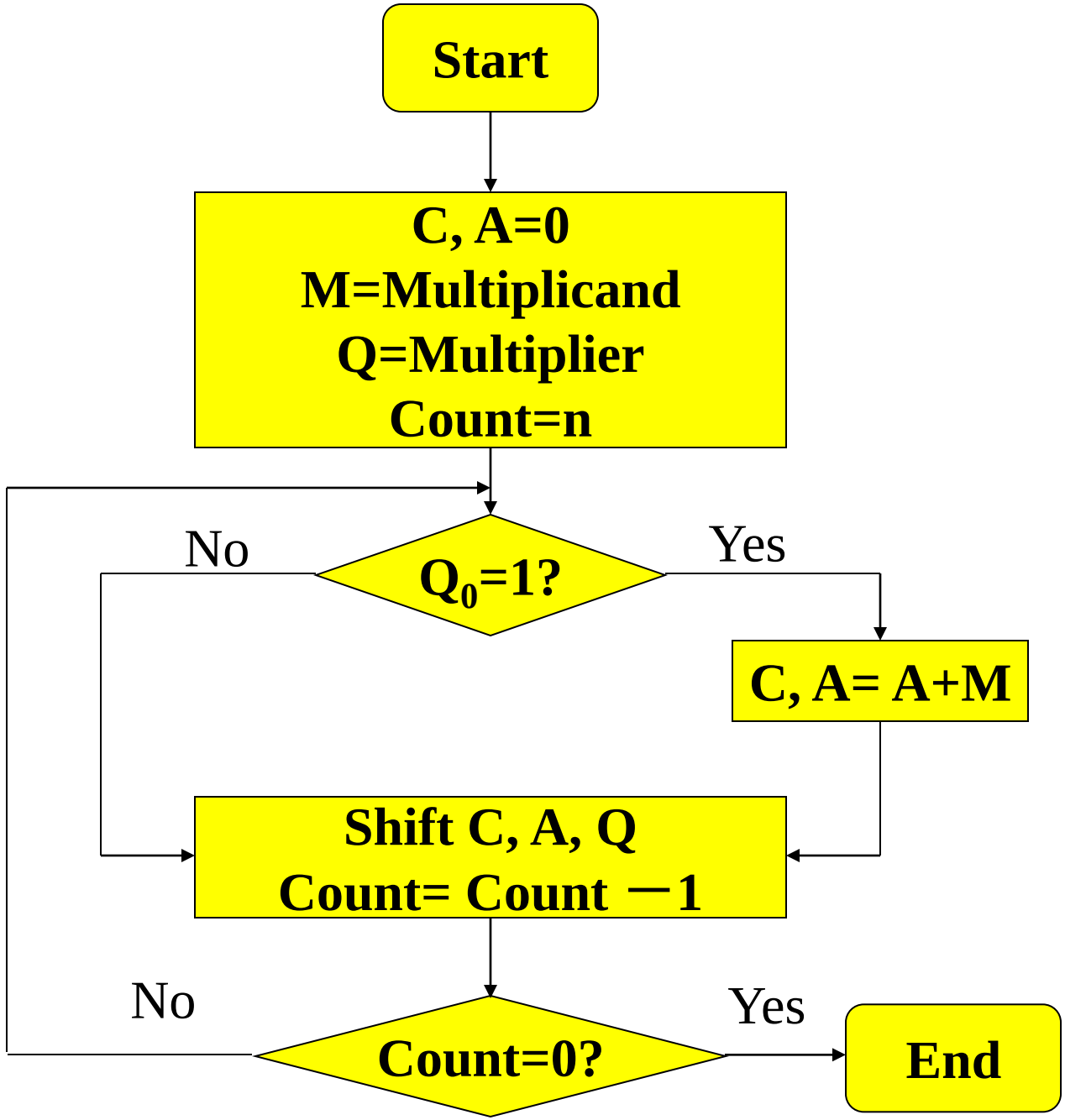


- Array Multiplication
  - Q: When calculating P = M × Q (M and Q are all n bit unsigned binary numbers) , how many FAs and AND gates do we need (using n × n array multiplier)?
  - A:
    - FAs: n(n-1)
    - AND gates: n^2
    - n=32, 992 FAs, 1024 AND gates
    - n=64, 4032 FAs, 4096 AND gates

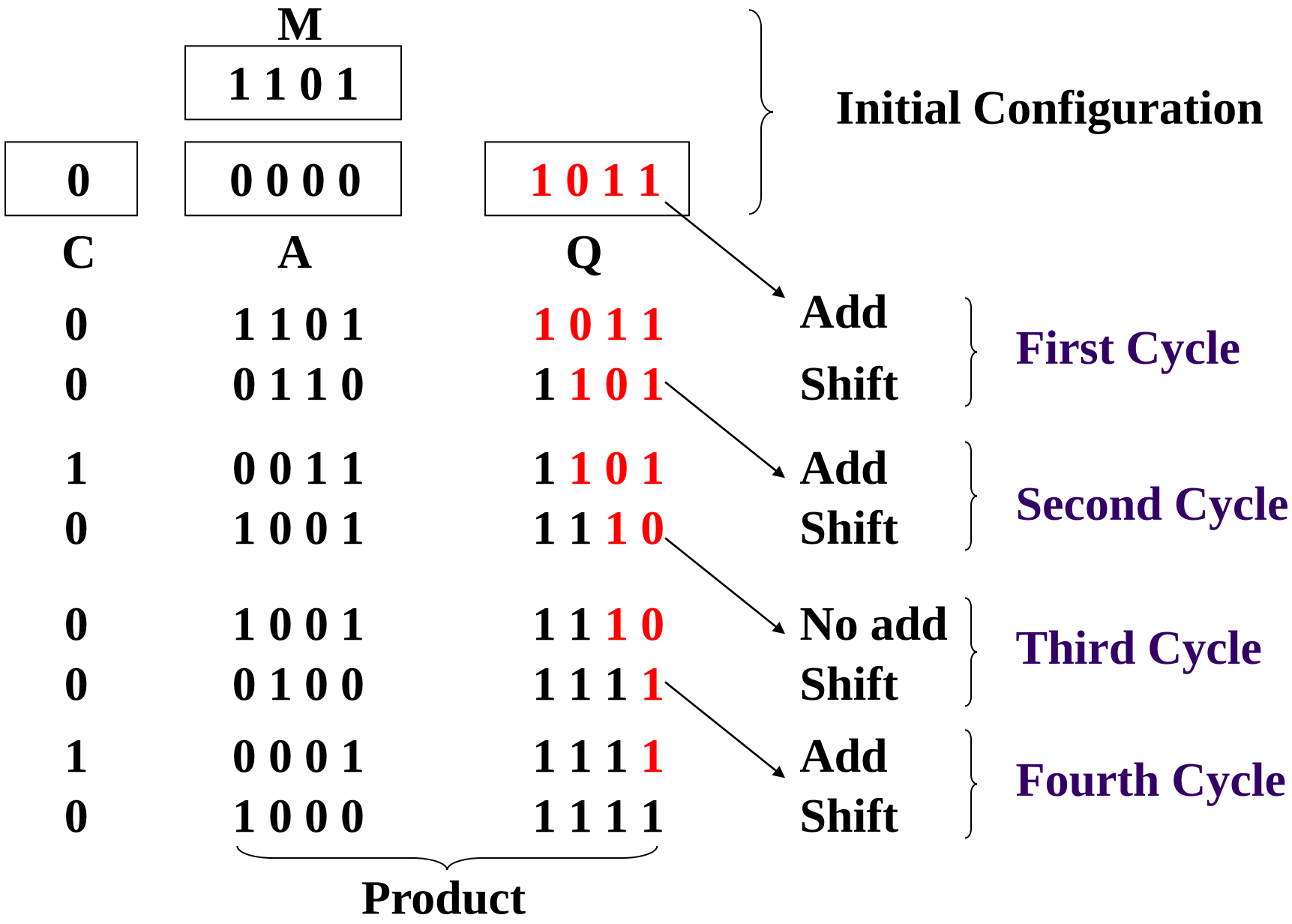
9.3 Multiplication of Unsigned Numbers

- Sequential Multiplication

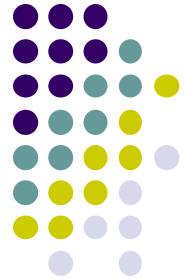



Start
C, A=0
M=Multiplicand
Q=Multiplier
Count=n
No
Q_0=1?
Yes
C, A= A+M
Shift C, A, Q
Count= Count －1
No
Count=0?
Yes
End

Example: M= 1101, Q= 1011, calculate P= M×Q

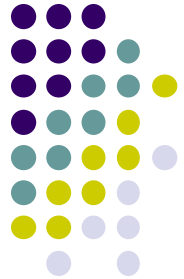


9.4 Multiplication of Signed Numbers



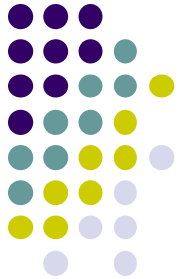
- How do we handle signed numbers?
  - Convert to unsigned, multiply, then convert back to signed
    - Works but is slow and a bit cumbersome
  - Directly handle the multiplication with the signed numbers
    - Booth algorithm recodes the bits to allow for such a computation.
    - Works with 2's complement numbers directly

9.4 Multiplication of Signed Numbers



- If the operand is negative, how should we do multiplication straightly?
  - Straightforward multiplication will not work in the case of a positive multiplier and a negative multiplicand.
    - Example: M = 1001, Q = 0011, calculate P = M×Q
      - P = 00011011
      - If interpret M, Q, and P as unsigned non-negative numbers, then M = 9, Q = 3, P = 27
      - If interpret M, Q, and P as signed 2's complement numbers, then M = -7, Q = 3, P = 27

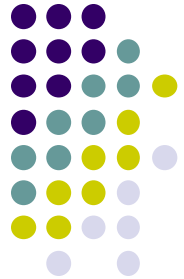
9.4 Multiplication of Signed Numbers



- If the operand is negative, how should we do multiplication straightly?
  - Case1: a positive multiplier and a negative multiplicand
    - Example: M= -13, Q= 11, calculate P= M×Q

$$
\begin{array}{r}
\mathbf{1\ 0\ 0\ 1\ 1} \\
\times\ \mathbf{0\ 1\ 0\ 1\ 1} \\
\hline
\mathbf{1\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1} \\
\mathbf{1\ 1\ 1\ 1\ 0\ 0\ 1\ 1} \\
\mathbf{0\ 0\ 0\ 0\ 0\ 0\ 0\ 0} \\
\mathbf{1\ 1\ 0\ 0\ 1\ 1} \\
\mathbf{0\ 0\ 0\ 0\ 0} \\
\hline
\mathbf{1\ 1\ 0\ 1\ 1\ 1\ 0\ 0\ 0\ 1}
\end{array}
$$

9.4 Multiplication of Signed Numbers



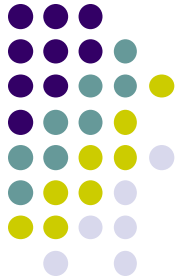
- If the operand is negative, how should we do multiplication straightly?
  - Case2: a negative multiplier and a negative multiplicand
    - Example: M= 1101, Q= 1011, calculate P= M×Q
      - Straightforward multiplication, P = 10001111
      - If interpret M, Q, and P as unsigned non-negative numbers, then M = 13, Q = 11, P = 143
      - Interpret M, Q, and P as signed 2's complement numbers, then M = -3, Q = -5, P = -113

9.4 Multiplication of Signed Numbers

- If the operand is negative, how should we do multiplication straightly?
  - Case2: a negative multiplier and a negative multiplicand
    - Example: M= -13, Q= -11, calculate P= M×Q
      - M = 10011, Q = 10101
      - 2's complement of M (-M)
        - 01101
      - 2's complement of Q (-Q)
        - 01011

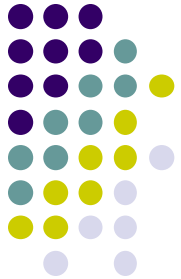
$$
\begin{array}{r}
0\ 1\ 1\ 0\ 1 \\
\times\ 0\ 1\ 0\ 1\ 1 \\
\hline
0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 1 \\
0\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 1 \\
0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
0\ 0\ 0\ 1\ 1\ 0\ 1 \\
0\ 0\ 0\ 0\ 0\ 0 \\
\hline
0\ 0\ 1\ 0\ 0\ 0\ 1\ 1\ 1\ 1 \quad = 143
\end{array}
$$

9.4 Multiplication of Signed Numbers



- Booth Algorithm
  - Reducing number of partial products
  - Fewer partial products generated for groups of consecutive 0's and 1's
    - Group of consecutive 0's in multiplier – no new partial product – only shift partial product right one bit position for every 0
    - Group of m consecutive 1's in multiplier – less than m partial products generated
    - Example: 0011110 = 0100000 – 0000010 (decimal notation: 30 = 32 – 2)

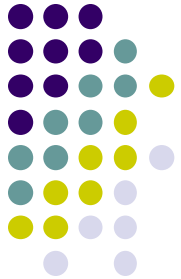
9.4 Multiplication of Signed Numbers



- Booth Algorithm
  - Recoding of a Multiplier

Multiplier		Version of Multiplicand Selected by bit i
Bit i	Bit i-1	
0	0	0×M
0	1	+1×M
1	0	−1×M
1	1	0×M

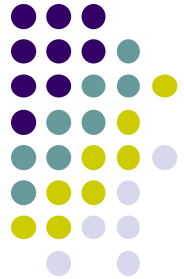
9.4 Multiplication of Signed Numbers



- Booth Algorithm
  - Recoding of a Multiplier
    - Example
      - 0 0 1 0 1 1 0 0 1 1 1 0 1 0 1 1 0 0
      - 0+1 -1+1 0-1 0 +1 0 0 -1+1 -1+1 0 -1 0 0
    - Example

$$
\begin{array}{r}
0\ 1\ 1\ 0\ 1\ (+13) \\
\times\ 1\ 1\ 0\ 1\ 0\ (-6) \\
\hline
\end{array}
$$

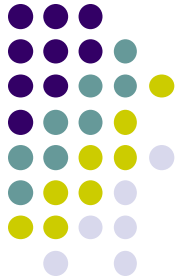
9.4 Multiplication of Signed Numbers



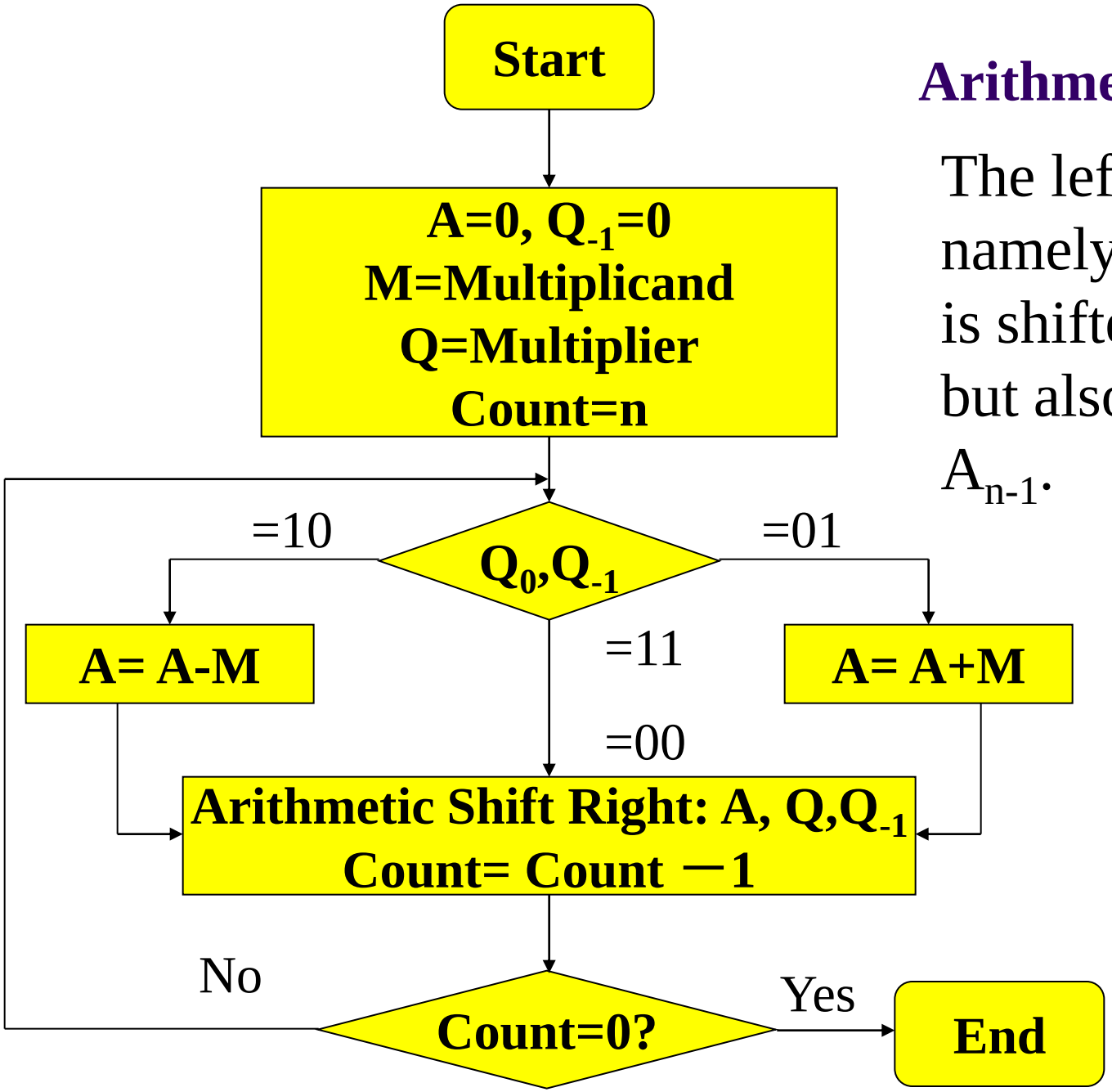
- Booth Algorithm
  - Recoding of a Multiplier
    - Example

$$
\begin{array}{cccccccccc}
 & & & & 0 & 1 & 1 & 0 & 1 & \\
 & & & & 0 & -1 & +1 & -1 & 0 & \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & & \\
1 & 1 & 1 & 0 & 0 & 1 & 1 & & & \\
0 & 0 & 0 & 0 & 0 & 0 & & & & \\
\hline
1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \ (-78)
\end{array}
$$

9.4 Multiplication of Signed Numbers



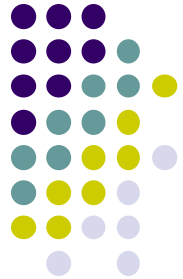
- Booth Algorithm
  - Hardware Organization
    - As before, the multiplicand and multiplier are placed in the M and Q registers, respectively.
    - There is a 1-bit register placed logically to the right of the least significant bit (Q_0) of the Q register and designated Q_{-1}.
    - The results of the multiplication will appear in the A and Q registers.
    - 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.



Arithmetic Shift Right:

The left-most of A, namely A_{n-1}, not only is shifted into A_{n-2}, but also remains in A_{n-1}.

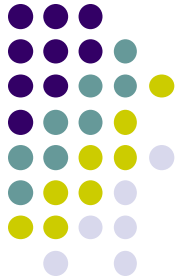
9.4 Multiplication of Signed Numbers



- Booth Algorithm
  - Example

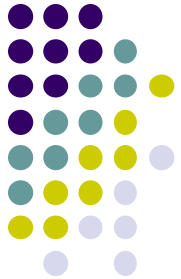
	M		
	0 1 1 1		
0 0 0 0	0 0 1 1	0	Initial Configuration
A	Q	Q_{-1}	
1 0 0 1	0 0 1 1	0	A=A-M
1 1 0 0	1 0 0 1	1	Shift — First Cycle
1 1 1 0	0 1 0 0	1	Shift — Second Cycle
0 1 0 1	0 1 0 0	1	A=A+M
0 0 1 0	1 0 1 0	0	Shift — Third Cycle
0 0 0 1	0 1 0 1	0	Shift — Fourth Cycle

9.4 Multiplication of Signed Numbers



- Booth Algorithm
  - Advantages
    - Handle both positive and negative multipliers uniformly.
    - Achieve some efficiency in the number of additions required when the multiplier has a few large blocks of 1s.
    - On average, the speed of doing multiplication with the Booth algorithm is the same as with the normal algorithm.

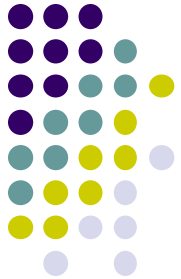
9.4 Multiplication of Signed Numbers



- Booth Algorithm
  - Advantages

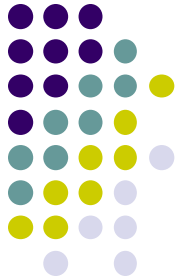
Good	0	0	0	0	1	1	1	1	1	0	0	0	0	1	1	1
multiplier	0	0	0	+1	0	0	0	0	–1	0	0	0	+1	0	0	–1
Ordinary	1	1	0	0	0	1	0	1	1	0	1	1	1	1	0	0
multiplier	0	–1	0	0	+1	–1	+1	0	– 1	+1	0	0	0	–1	0	0
Worst-case	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
multiplier	+1	–1	+1	–1	+1	–1	+1	–1	+1	–1	+1	–1	+1	–1	+1	–1

9.4 Multiplication of Signed Numbers



- Multiplication Summary
  - Multiplication involves 2 basic operations: generation of partial products + their accumulation
  - 2 ways to speed up
    - Reducing number of partial products and/or
    - Accelerating accumulation
  - 3 types of high-speed multipliers
    - Sequential multiplier - generates partial products sequentially and adds each newly generated product to previously accumulated partial product

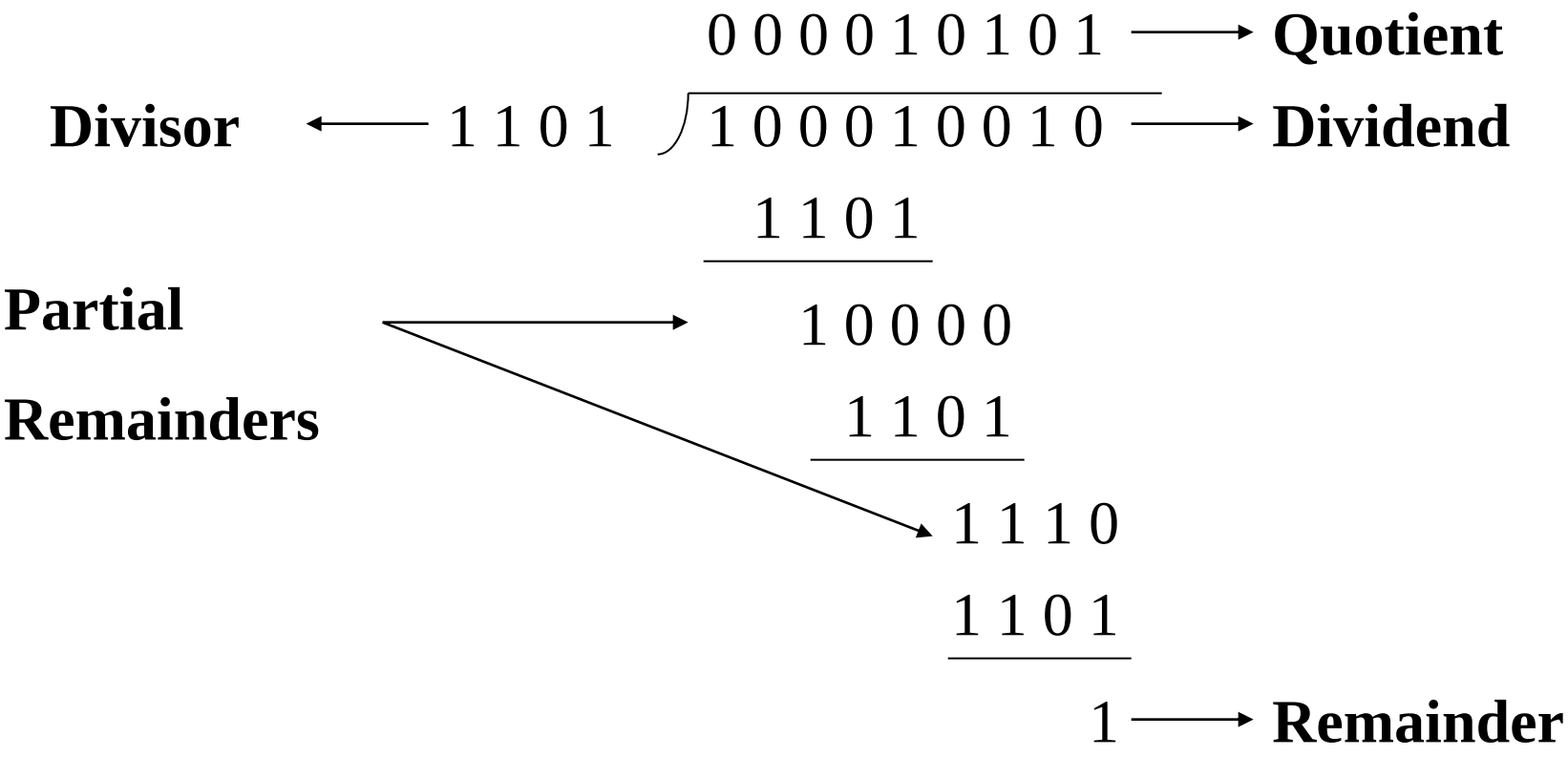
9.4 Multiplication of Signed Numbers



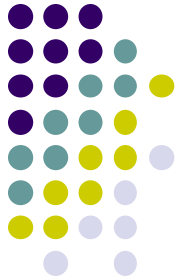
- Multiplication Summary
  - 3 types of high-speed multipliers
    - Parallel multiplier - generates partial products in parallel, accumulates using a fast multi-operand adder
    - Array multiplier - array of identical cells generating new partial products; accumulating them simultaneously
      - No separate circuits for generation and accumulation
      - Reduced execution time but increased hardware complexity

9.6 Integer Division

- Manual Division of Positive Integers
  - Example



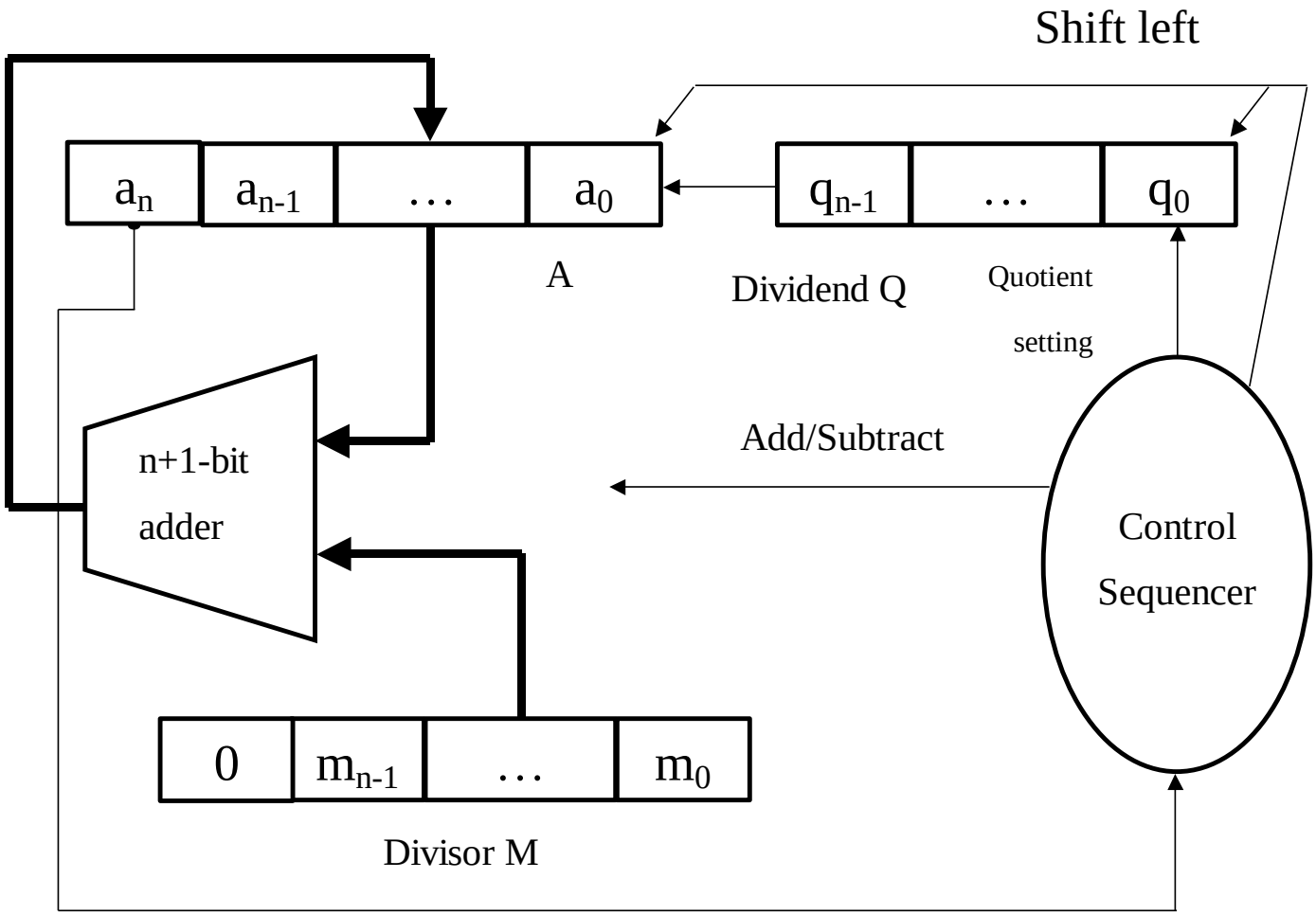
9.6 Integer Division

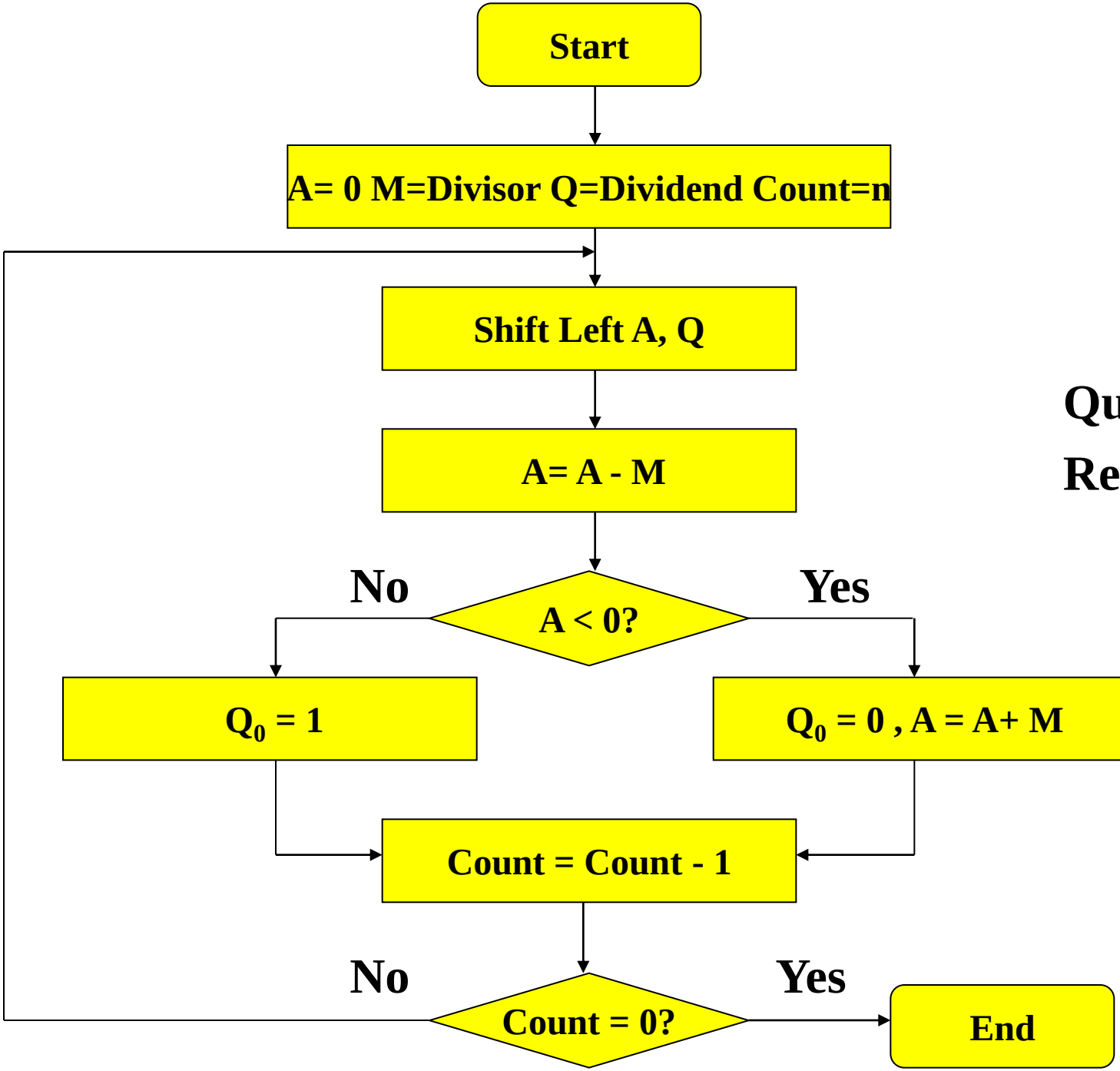


- Manual Division of Positive Integers
  - Process Description
    - The bits of the dividend are examined from left to right, until the set of bits examined represents a number greater than or equal to the divisor. Until this event occurs, 0s are placed in the quotient from left to right.
    - If the event occurs, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend.
    - From this point on, the division follows a cyclic pattern. 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. The divisor is subtracted from this number to produce a new partial remainder.
    - Continue until all the bits of the dividend are exhausted.

9.6 Integer Division

- Restoring Division
  - Hardware





Quotient in Q

Remainder in A

Example

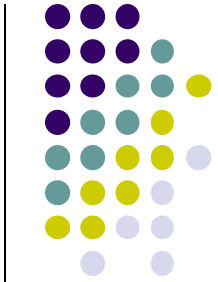
$$\begin{array}{r} 1\ 0 \\ 1\ 1\ \overline{)\ 1\ 0\ 0\ 0} \\ \underline{1\ 1} \\ 1\ 0 \end{array}$$

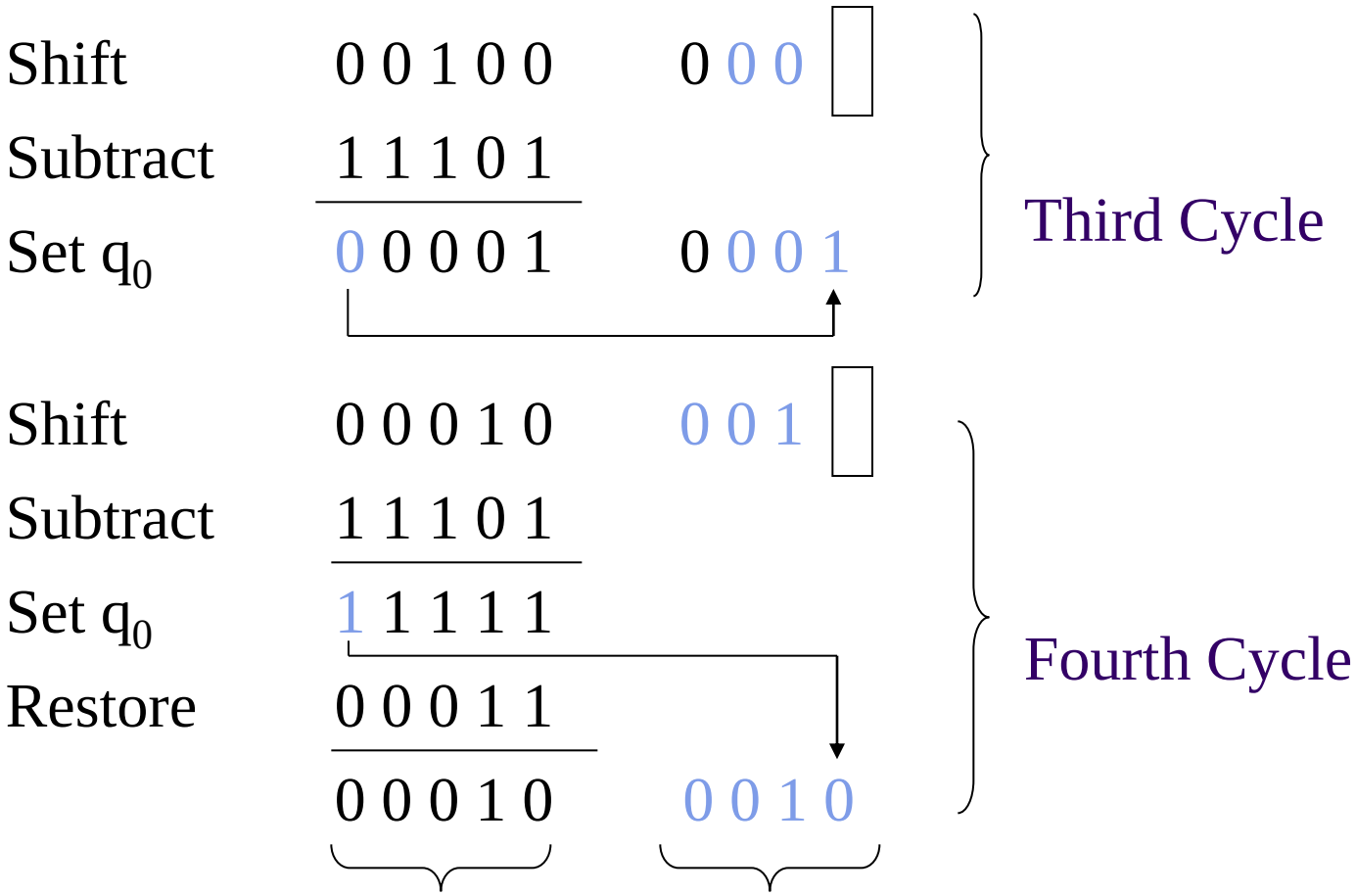
	M	
Initially	0 0 0 1 1	
	0 0 0 0 0	1 0 0 0
	A	**Q**
Shift	0 0 0 0 1	0 0 0 ☐
Subtract	1 1 1 0 1	
Set q_0	1 1 1 1 0	
Restore	0 0 0 1 1	
	0 0 0 0 1	0 0 0 0

First Cycle

Shift	0 0 0 1 0	0 0 0 ☐
Subtract	1 1 1 0 1	
Set q_0	1 1 1 1 1	
Restore	0 0 0 1 1	
	0 0 0 1 0	0 0 0 0

Second Cycle



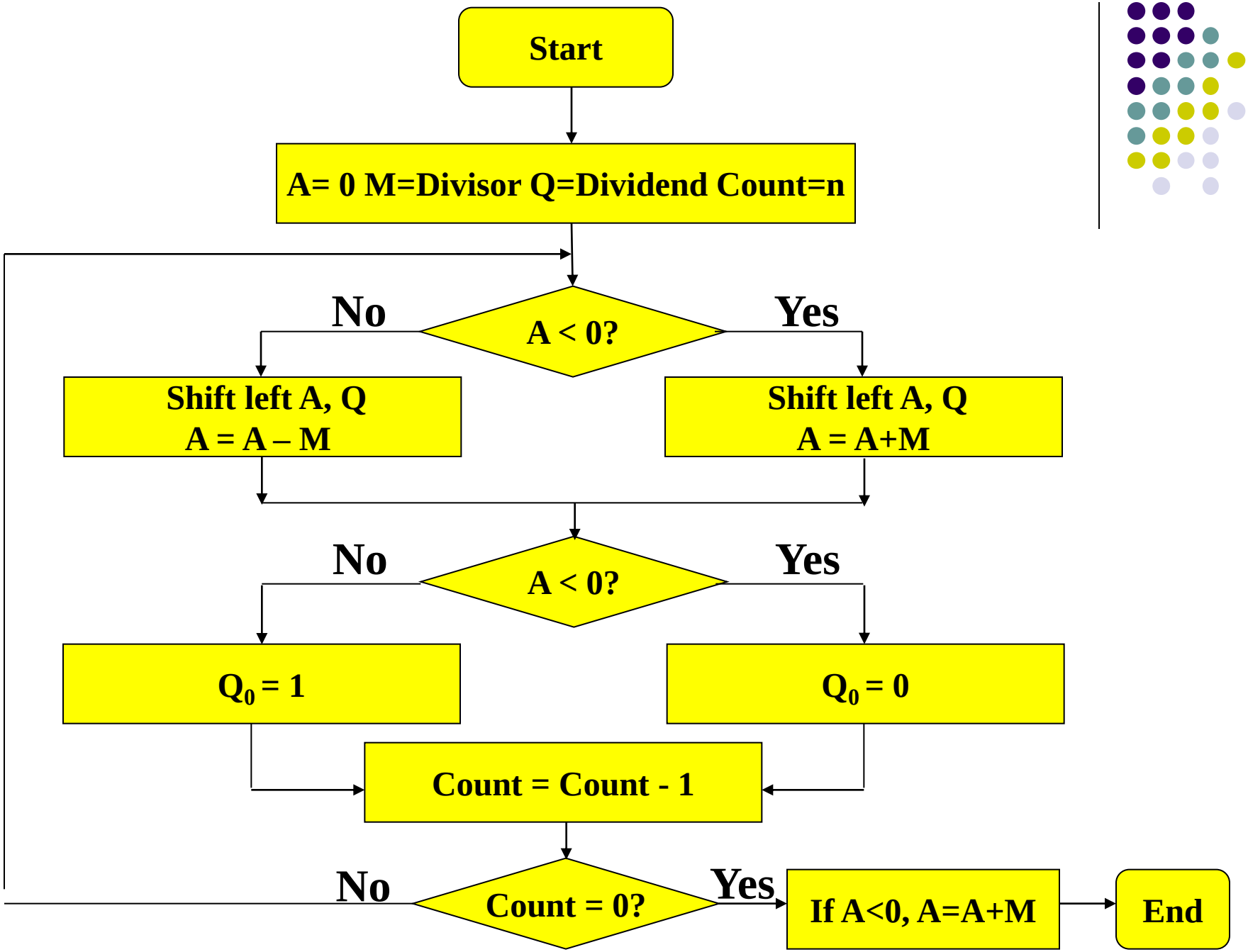

Shift
0 0 1 0 0
0 0 0
Subtract
1 1 1 0 1
Set q0
0 0 0 0 1
0 0 0 1
Third Cycle
Shift
0 0 0 1 0
0 0 1
Subtract
1 1 1 0 1
Set q0
1 1 1 1 1
Restore
0 0 0 1 1
0 0 0 1 0
0 0 1 0
Fourth Cycle
Remainder
Quotient

9.6 Integer Division

- Non-Restoring Division
  - In restoring division
    - (i-1) step, R_{i-1}
    - i step, $R_i' = 2R_{i-1} - M$
      - $R_i' > 0$, $Q_0 = 1$, $R_i = R_i'$
      - $R_i' < 0$, $Q_0 = 0$, $R_i = R_i' + M$
    - (i + 1) step, $R_{i+1}' = 2R_i - M$
      - $R_i' > 0$, $R_i = R_i'$, $R_{i+1}' = 2R_i' - M \longrightarrow A = 2A - M$
      - $R_i' < 0$, $R_i = R_i' + M$, $R_{i+1}' = 2(R_i' + M) - M = 2R_i' + M$

$$\downarrow$$

$$A = 2A + M$$


Start
A= 0 M=Divisor Q=Dividend Count=n
No
A < 0?
Yes
Shift left A, Q
A = A – M
Shift left A, Q
A = A+M
No
A < 0?
Yes
Q0 = 1
Q0 = 0
Count = Count - 1
No
Count = 0?
Yes
If A<0, A=A+M
End

Example

$$
\begin{array}{r}
1\ 0 \\
1\ 1 \overline{)\ 1\ 0\ 0\ 0} \\
\quad 1\ 1 \\
\hline
\quad 1\ 0
\end{array}
$$

	M	
Initially	0 0 0 1 1	
	0 0 0 0 0	1 0 0 0
	A	**Q**
Shift	0 0 0 0 1	0 0 0 ☐
Subtract	1 1 1 0 1	
Set q_0	1 1 1 1 0	0 0 0 0

First Cycle

Shift	1 1 1 0 0	0 0 0 ☐
Add	0 0 0 1 1	
Set q_0	1 1 1 1 1	0 0 0 0

Second Cycle

Shift 1 1 1 1 0 | 0 0 0 [ ]

Add 0 0 0 1 1

Set q_0 0 0 0 0 1 | 0 0 0 1

Third Cycle

Shift 0 0 0 1 0 | 0 0 1 [ ]

Subtract 1 1 1 0 1

Set q_0 1 1 1 1 1 | 0 0 1 0

Fourth Cycle

Quotient

Add 1 1 1 1 1

0 0 0 1 1

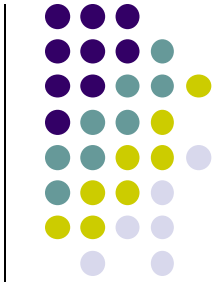
0 0 0 1 0

Restoring

Remainder

Remainder

9.6 Integer Division

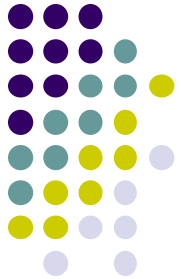


- Non-Restoring Division
  - Note
    - There are no simple algorithms for directly performing division on signed operands that are comparable to the algorithms for signed multiplication.
    - In division, the operands can be processed to transform them into positive values.
    - After using one of the algorithms (restoring division or nonrestoring division), the results are transformed to the correct signed values, as necessary.

9.7 Floating-Point Numbers and Operations

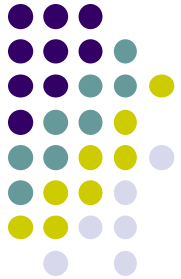
- What can be represented in n bits?
  - Unsigned: $0 \sim 2^n-1$
  - 2's complement: $-2^{n-1} \sim 2^{n-1}-1$
  - But, what about?
    - Very large number?
      - 9,349,398,989,787,762,244,859,087,678
    - Very small number?
      - 0.00000000000000000000045691
    - Fractional values? 0.35
    - Mixed numbers? 10.28
    - Irrationals(无理数)?π

9.7 Floating-Point Numbers and Operations



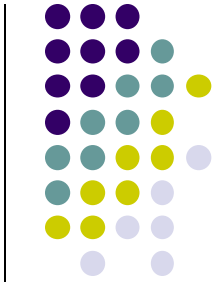
- Number Format
  - According to whether the position of binary point is fixed, there are two number formats:
    - Fixed-point numbers
      - E.g., integers, have an implied binary point at the right end of them.
    - Floating-point numbers

9.7 Floating-Point Numbers and Operations

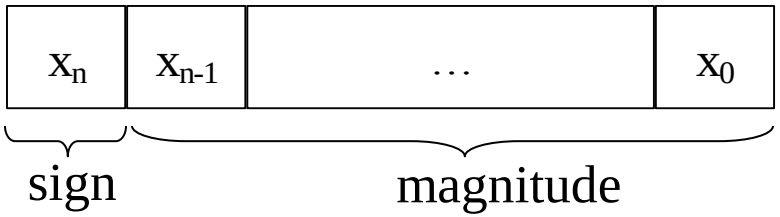


- Fixed-point Representation
  - A fixed-point notation is any number in which the number of bits to the right of the binary point does not change.
    - Unsigned integers: With no bits to the right of the binary point.
    - Signed integers: With no bits to the right of the binary point.
    - Signed fractions: The binary point is to the right of the sign bit.

9.7 Floating-Point Numbers and Operations

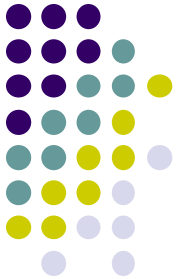


- Fixed-point Representation
  - Let X = $x_n \ldots x_0$ be a fixed-point number



    - Interpret X as a pure fraction
      - The binary point is between x_n and x_{n-1}.
      - Range: $-1 \le V(X) \le 1 - 2^{-n}$
    - Interpret X as an integer
      - The binary point is to the right of x_0.
      - Range: $-2^n \le V(X) \le 2^n - 1$

9.7 Floating-Point Numbers and Operations

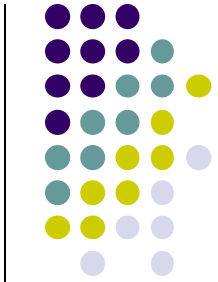


- Fixed-point Representation
  - Limitation
    - Very large integers can not be represented, nor can very small fractions.
    - Example: Consider the range of values representable in a 32-bit, signed, fixed-point format.
      - Interpreted as integers $-2^{31} \le V(X) \le 2^{31}-1$

        $V(X) \in [-2.15\times10^{9},0],[0,+2.15\times10^{9}]$
      - Interpreted as fractions $-1 \le V(X) \le 1-2^{-31}$

        $V(X) \in [-1,-4.55\times10^{-10}],[+4.55\times10^{-10},+1]$

9.7 Floating-Point Numbers and Operations

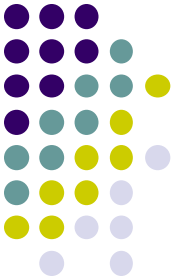


- Fixed-point Representation
  - Limitation
    - Example: Consider the range of values representable in a 32-bit, signed, fixed-point format.
      - In scientific calculations
        - Avogadro's constant

          $6.0247 \times 10^{23}\ \text{mol}^{-1} = 0.60247 \times 10^{24}\ \text{mol}^{-1}$
        - Planck's constant

          $6.6254 \times 10^{-27}\ \text{erg.s} = 0.66254 \times 10^{-26}\ \text{erg.s}$

9.7 Floating-Point Numbers and Operations

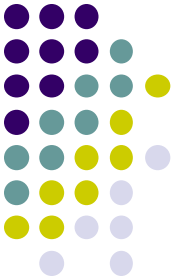


- Floating-point Representation
  - The position of the binary point is variable and is automatically adjusted as computation proceeds.
  - The position of the binary point must be given explicitly in the floating-point representation.
  - Exponential Notation
    - The following are equivalent representations of 1,234

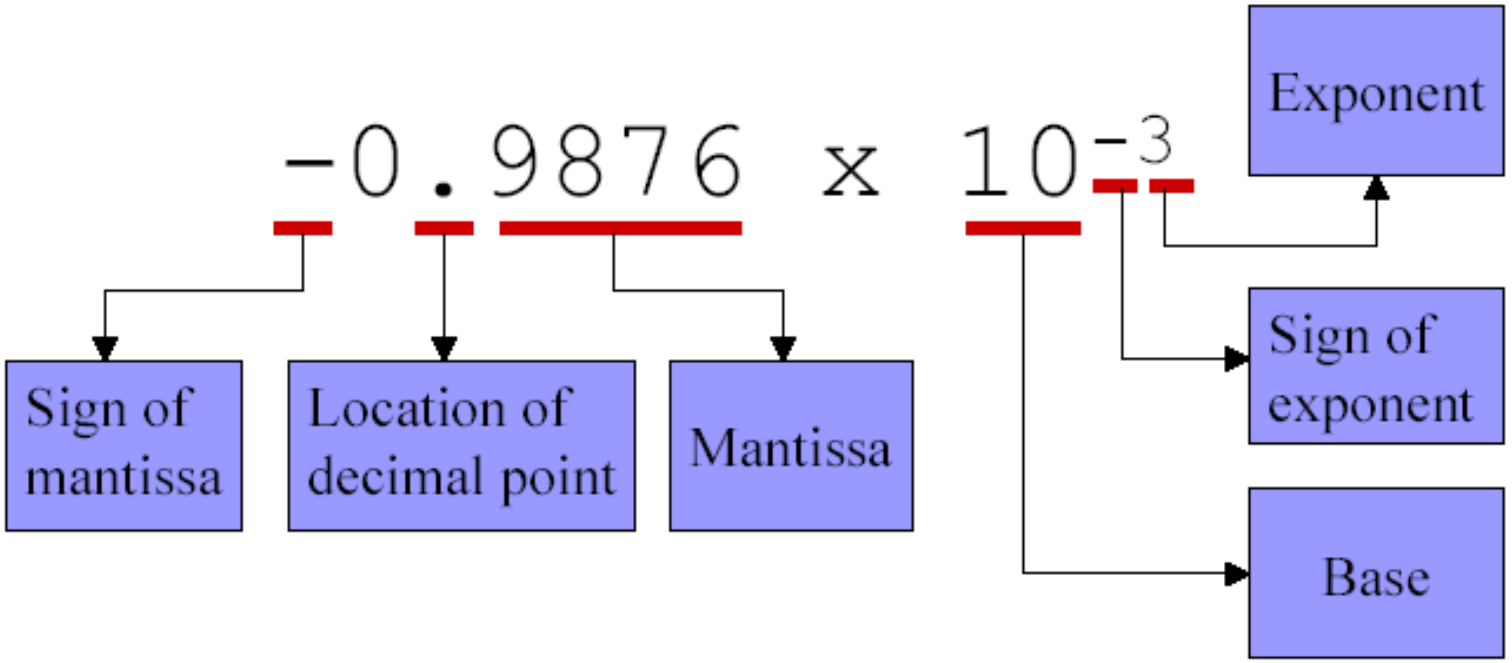
$$
\begin{aligned}
123{,}400.0 &\times 10^{-2} \\
12{,}340.0 &\times 10^{-1} \\
1{,}234.0 &\times 10^{0} \\
123.4 &\times 10^{1} \\
12.34 &\times 10^{2} \\
1.234 &\times 10^{3} \\
0.1234 &\times 10^{4}
\end{aligned}
$$

The representations differ in that the decimal place – the "point" -- "floats" to the left or right (with the appropriate adjustment in the exponent).

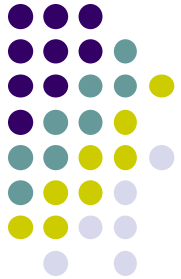
9.7 Floating-Point Numbers and Operations



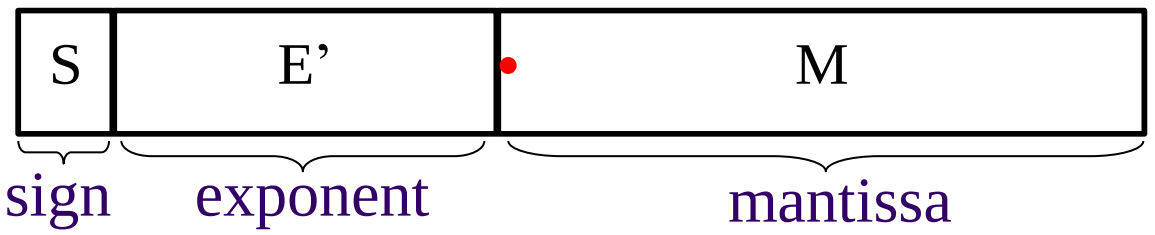
- Floating-point Representation
  - Parts of a floating-point number



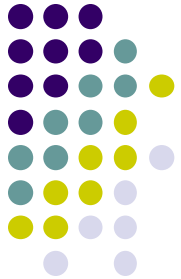
9.7 Floating-Point Numbers and Operations



- Floating-point Numbers in Computers
  - Numerical Form
    - $(-1)^S M 2^E$
      - Sign bit S determines whether number is negative or positive
      - Mantissa M, a fraction in sign-magnitude or 2's complement representation, containing the significant digits
      - Exponent E, in 2's complement or biased notation, the power of base
  - Encoding



9.7 Floating-Point Numbers and Operations

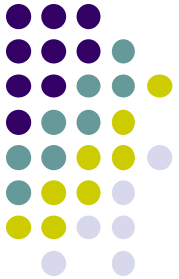


- Floating-point Numbers in Computers
  - Excess or Biased Notation
    - A negative exponent in 2's complement looks like a large exponent.
    - A fixed value is subtracted from the exponent field to get the true exponent.
    - $E' = E + (2^{k-1} - 1)$
      - E is the actual exponent
      - k is the number of bits in the exponent.
    - Note
      - When the bits of a biased representation are treated as unsigned integers, the relative magnitudes of the numbers do not change.

Decimal Representation	Biased Representation	Two's complement representation
+8	1111	_
+7	1110	0111
+6	1101	0110
+5	1100	0101
+4	1011	0100
+3	1010	0011
+2	1001	0010
+1	1000	0001

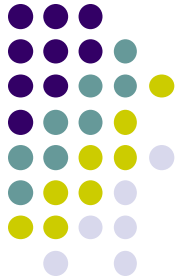
±0	0111	0000
- 1	0110	1111
- 2	0101	1110
- 3	0100	1101
- 4	0011	1100
- 5	0010	1011
- 6	0001	1010
- 7	0000	1001
- 8	_	1000

9.7 Floating-Point Numbers and Operations



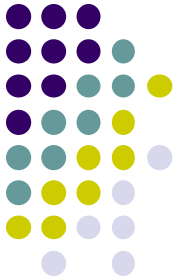
- Floating-point Numbers in Computers
  - Normalization
    - By convention, the number which decimal point is placed to the right of the first (nonzero) significant digit is called to be normalized.
      - 1.0×10^{-9} √(a normalized scientific notation)
      - 0.1×10^{-10} ×
    - In normalized binary, the most significant bit of the mantissa is always equal to 1.
      - $\pm$ 0.1bbb…b $\times$ 2^E (b is either 0 or 1)
      - Example
        - 0.0110×2^6 ×
        - 0.110×2^5 √

9.7 Floating-Point Numbers and Operations

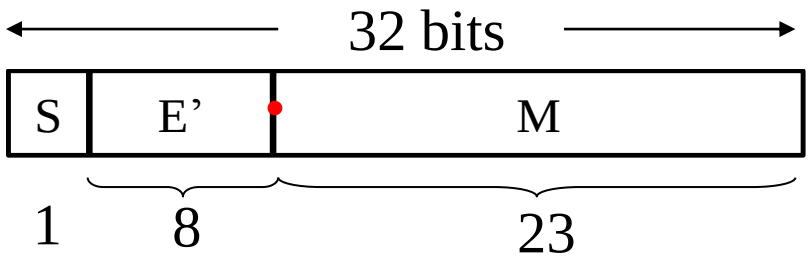


- IEEE 754 Standard
  - IEEE: Institute of Electrical and Electronics Engineers
  - Most common standard for representing floating point numbers.
  - Established in 1985 as uniform standard for floating point arithmetic
  - This standard was developed to facilitate the portability of programs from one processor to another and encourage the development of sophisticated, numerically oriented programs.
  - Supported by all major CPUs

9.7 Floating-Point Numbers and Operations

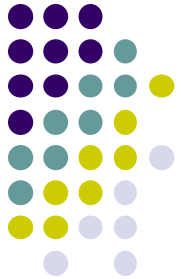


- IEEE 754 Standard
  - Single Precision Format (32 bits)

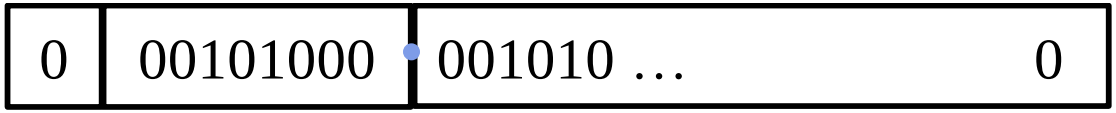


    - S, sign of the number: 0 signifies +, 1 signifies -.
    - E' (the excess-127 format), exponent of the scale factor
      - Note that the value actually stored in the exponent field is an unsigned integer.
      - Assume that E is the signed exponent, then E'= E+127, $0 \le E' \le 2^8 - 1 = 255$ (E'=0 and E'=255 are used to represent special values, $1 \le E' \le 254 \longrightarrow -126 \le E \le 127$)

9.7 Floating-Point Numbers and Operations

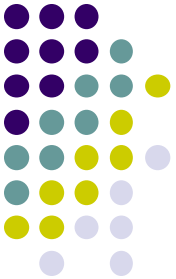


- IEEE 754 Standard
  - Single Precision Format (32 bits)
    - M, mantissa (the sign-magnitude format)
      - Note that the most significant bit of mantissa is not explicitly represented. It is assumed to be to the immediate left of the binary point.
      - Value $= \pm 1.M \times 2^{E' - 127}$
      - Example

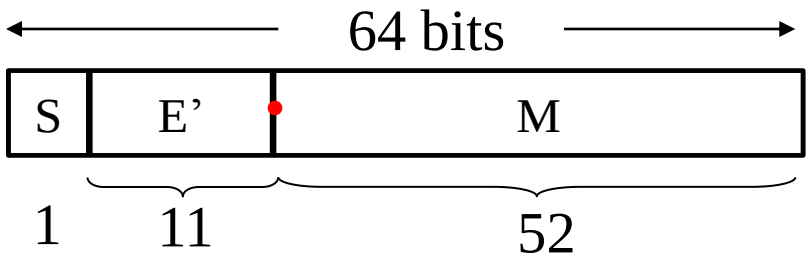


Value = $1.001010\ldots0 \times 2^{-87}$

9.7 Floating-Point Numbers and Operations

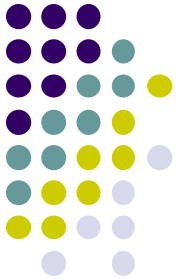


- IEEE 754 Standard
  - Double Precision Format (64 bits)



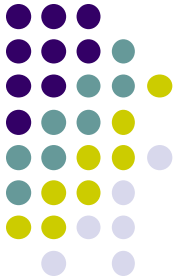
    - S, sign of the number: 0 signifies +, 1 signifies -.
    - E' (the excess-1023 format), exponent of the scale factor
      - Assume that E is the signed exponent, then E'= E+1023, $0 \le E' \le 2^{11} - 1 = 2047$ (E'=0 and E'=2047 are used to represent special values, $1 \le E' \le 2046 \longrightarrow -1022 \le E \le 1023$)
    - M, mantissa (the sign-magnitude format)
      - Value $= \pm 1.M \times 2^{E' - 1023}$

9.7 Floating-Point Numbers and Operations



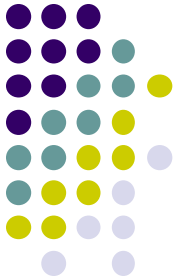
- IEEE 754 Standard
  - A computer must provide at least single-precision representation to conform to the IEEE standard.
  - Double-precision representation is optional.
  - Extended single-precision (more than 32 bits) /Extended double-precision (more than 64 bits)
    - Help to reduce the size of the accumulated round-off error in a sequence of calculations.
    - Enhance the accuracy of evaluation of elementary functions such as sine, cosine, and so on.
  - Trade-off between “accuracy” and “range”
    - Increasing the size of **mantissa** enhances **accuracy**.
    - Increasing the size of **exponent** increases the **range**.

9.7 Floating-Point Numbers and Operations



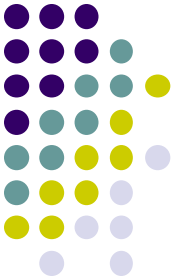
- IEEE 754 Standard
  - Special Values
    - Zero
      - S = 0/1, E' = 0, M = 0 Value = ±0
    - Infinity
      - Operation that overflows
      - E.g., 1.0/0.0 = + infinity
      - S = 0/1, E' = 255 or 2047, M = 0 Value = ± infinity
    - NaN (Not a Number)
      - A NaN is the result of performing an invalid operation such as 0/0 or taking the square root of a negative number.
      - Also, compiler may assign the value NaN to uninitialized variables.
      - S = 0/1, E' = 255 or 2047, M ≠ 0 Value = NaN

9.7 Floating-Point Numbers and Operations



- IEEE 754 Standard
  - Special Values
    - Denormal Numbers
      - There is no implied 1 to the left of the binary point.
      - Numbers very close to 0.0
      - Lose precision as get smaller
      - “Gradual underflow”
      - S = 0, E’ = 0, M ≠ 0
        - Value = $\pm$ 0.M $\times 2^{-126}$
        - Value = $\pm$ 0.M $\times 2^{-1022}$

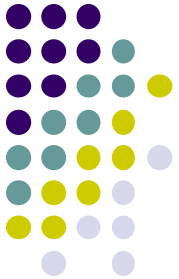
9.7 Floating-Point Numbers and Operations



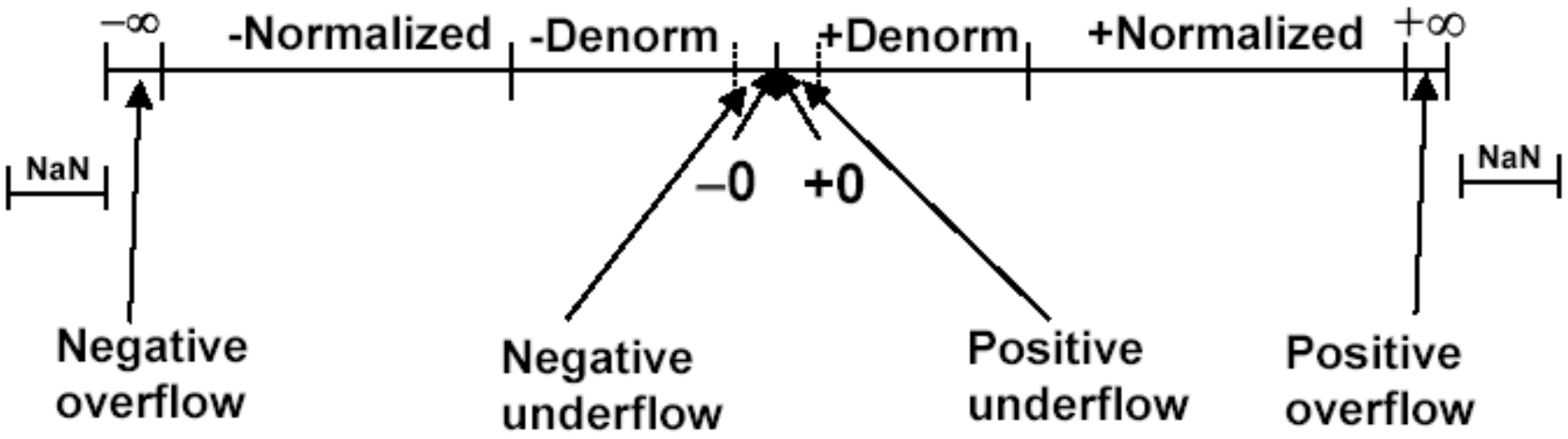
- IEEE 754 Standard
  - Special Values Summary

Normalized:	±	0<E<max	Any bit pattern
Denormalized:	±	0	Any nonzero bit pattern
zero:	±	0	0
Infinity:	±	11...1	0
NaN:	±	11...1	Any nonzero bit pattern

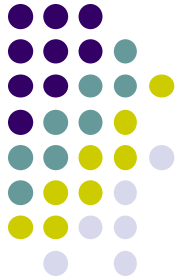
9.7 Floating-Point Numbers and Operations



- IEEE 754 Standard
  - Special Values Summary

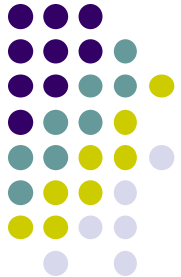


9.7 Floating-Point Numbers and Operations



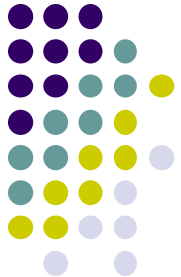
- Arithmetic Operations On FP Numbers
  - Addition and Subtraction
    - Alignment
      - If their exponents differ, it is necessary to manipulate the two summands so that the two exponents are equal.
      - Example: Decimal addition $(123\times10^{0}) + (456\times10^{-2})$
        $(123\times10^{0}) + (456\times10^{-2}) = (123\times10^{0})+(4.56\times10^{0})$
        $=127.56\times10^{0}$
      - The alignment is achieved by shifting the magnitude portion of the mantissa right 1 digit and incrementing the exponent until the two exponents are equal. (小阶向大阶看齐)

9.7 Floating-Point Numbers and Operations



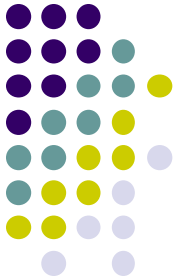
- Arithmetic Operations On FP Numbers
  - Addition and Subtraction
    - Add/Subtract Rule (for IEEE single precision)
      - ① Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
      - ② Set the exponent of the result equal to the larger exponent.
      - ③ Perform addition/subtraction on the mantissas and determine the sign of the result.
      - ④ Normalize the resulting value, if necessary.

9.7 Floating-Point Numbers and Operations



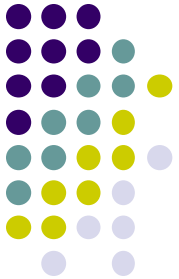
- Arithmetic Operations On FP Numbers
  - Multiplication and Division
    - Multiply Rule (for IEEE single precision)
      - ① Add the exponents and subtract 127.
      - ② Multiply the mantissas and determine the sign of the result.
      - ③ Normalize the resulting value, if necessary.
    - Divide Rule (for IEEE single precision)
      - ① Subtract the exponents and add 127.
      - ② Divide the mantissas and determine the sign of the result.
      - ③ Normalize the resulting value, if necessary.

9.7 Floating-Point Numbers and Operations



- Problems Considerable in FP Arithmetic
  - Guard bits
    - The additional bits retained in the mantissa are referred to as guard bits.
    - The guard bits are used to pad out the right end of the mantissa with 0s.
    - It is important to retain guard bits during the intermediate steps. This yields maximum accuracy in the final results.
    - Example: X= 1.00…00 $\times$ 2^1, Y= 1.11…11 $\times$ 2^0 (X and Y are all in IEEE single precision format). Calculate Z= X – Y

9.7 Floating-Point Numbers and Operations



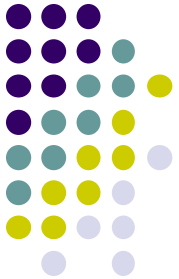
- Problems Considerable in FP Arithmetic
  - Guard bits
    - Example
      - Without guard bits

$$X = 1.000\ldots00 \times 2^{1}$$
$$-Y = 0.111\ldots11 \times 2^{1}$$
$$Z = 0.000\ldots01 \times 2^{1}$$
$$= 1.000\ldots00 \times 2^{-22}$$

      - With guard bits

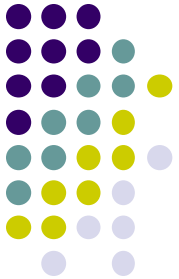
$$X = 1.000\ldots00\ \ 0000 \times 2^{1}$$
$$-Y = 0.111\ldots11\ \ 1000 \times 2^{1}$$
$$Z = 0.000\ldots00\ \ 1000 \times 2^{1}$$
$$= 1.000\ldots00\ \ 0000 \times 2^{-23}$$

9.7 Floating-Point Numbers and Operations



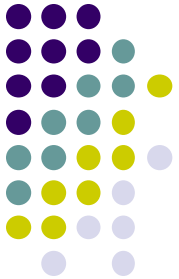
- Problems Considerable in FP Arithmetic
  - Truncation
    - Chopping
      - Remove the guard bits and make no changes in the retained bits.
      - Example: Truncate $0.b_{-1}b_{-2}b_{-3}\,010$ to three bits.
        - $0.b_{-1}b_{-2}b_{-3}\,010 \rightarrow 0.b_{-1}b_{-2}b_{-3}$
        - Actually, all fractions in the range $0.b_{-1}b_{-2}b_{-3}\,000$ to $0.b_{-1}b_{-2}b_{-3}111$ are truncated to $0.b_{-1}b_{-2}b_{-3}$.
        - The error range in the 3-bit result is from 0 to 0.000111.
      - Error Range
        - The error range in chopping is from 0 to almost 1 in the least significant position of the retained bits.

9.7 Floating-Point Numbers and Operations



- Problems Considerable in FP Arithmetic
  - Truncation
    - Von Neumann Rounding
      - ① If the bits to be removed are all 0s, they are simply dropped, with no changes to the retained bits.
      - ② If any of the bits to be removed are 1, the least significant bit of the retained bits is set to 1.
      - Example: Truncate $0.b_{-1}b_{-2}b_{-3}\ 000 - 0.b_{-1}b_{-2}b_{-3}\ 111$ to three bits.
        - ① $0.b_{-1}b_{-2}b_{-3}\ 000 \rightarrow 0.b_{-1}b_{-2}b_{-3}$
        - ② $0.b_{-1}b_{-2}b_{-3}\ 001 - 0.b_{-1}b_{-2}b_{-3}\ 111 \rightarrow 0.b_{-1}b_{-2}1$
      - Error Range
        - The error range in Von Neumann rounding is between – 1 and +1 in the LSB position of the retained bits.

9.7 Floating-Point Numbers and Operations

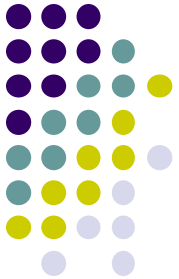


- Problems Considerable in FP Arithmetic
  - Truncation
    - Rounding (Round to the nearest number)
      - A 1 is added to the LSB position of the bits to be retained if there is a 1 in the MSB position of the bits being removed.
      - Example: Truncate $0.b_{-1}b_{-2}b_{-3}\ 000 - 0.b_{-1}b_{-2}b_{-3}\ 111$ to three bits.
        - ① $0.b_{-1}b_{-2}b_{-3}\ 000 - 0.b_{-1}b_{-2}b_{-3}\ 011 \rightarrow 0.b_{-1}b_{-2}b_{-3}$
        - ② $0.b_{-1}b_{-2}b_{-3}\ 101 - 0.b_{-1}b_{-2}b_{-3}\ 111 \rightarrow 0.b_{-1}b_{-2}b_{-3} + 0.001$
        - ③ $0.b_{-1}b_{-2}b_{-3}\ 100$ (tie situation): Choose the retained bits to be nearest even number.

          $0.b_{-1}b_{-2}0100 \rightarrow 0.b_{-1}b_{-2}0$

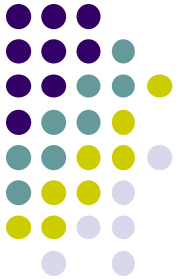
          $0.b_{-1}b_{-2}1100 \rightarrow 0.b_{-1}b_{-2}1 + 0.001$

9.7 Floating-Point Numbers and Operations



- Problems Considerable in FP Arithmetic
  - Truncation
    - Rounding (Round to the nearest number)
      - Error Range
        - The error range is – ½ to + ½ in the LSB position of the retained bits.
      - Rounding achieves the closest approximation to the number being truncated and is an unbiased technique.
      - Rounding is the default mode for truncation specified in the IEEE floating-point standard.

9.7 Floating-Point Numbers and Operations



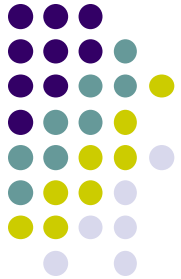
- Problems Considerable in FP Arithmetic
  - Normalization
    - If a number is not normalized, it can always be put in normalized form by shifting the mantissa and adjusting the exponent.
    - Example:
0	10001000	0010110…

        (There is no implicit 1 to the left of the binary point)

        Value = $+0.0010110\ldots \times 2^{9}$
0	10000101	0110 …

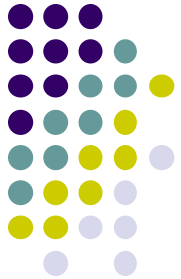
        Value = $+1.0110\ldots \times 2^{6}$

9.7 Floating-Point Numbers and Operations



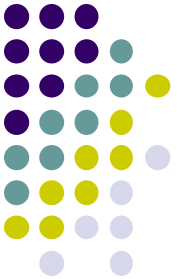
- Problems Considerable in FP Arithmetic
  - Overflow
    - As computation proceed, a number that does not fall in the representable range of normal numbers might be generated.
    - Exponent Overflow
      - A positive exponent exceeds the maximum possible exponent value.
      - Example: In IEEE single precision, E > 127
      - In some systems, it may be designated as plus infinity or minus infinity.
    - Exponent Underflow
      - A negative exponent is less than the minimum possible exponent value.
      - Example: In IEEE single precision, E < -126
      - This means that the number is too small to be represented, it may be reported as 0.

9.7 Floating-Point Numbers and Operations

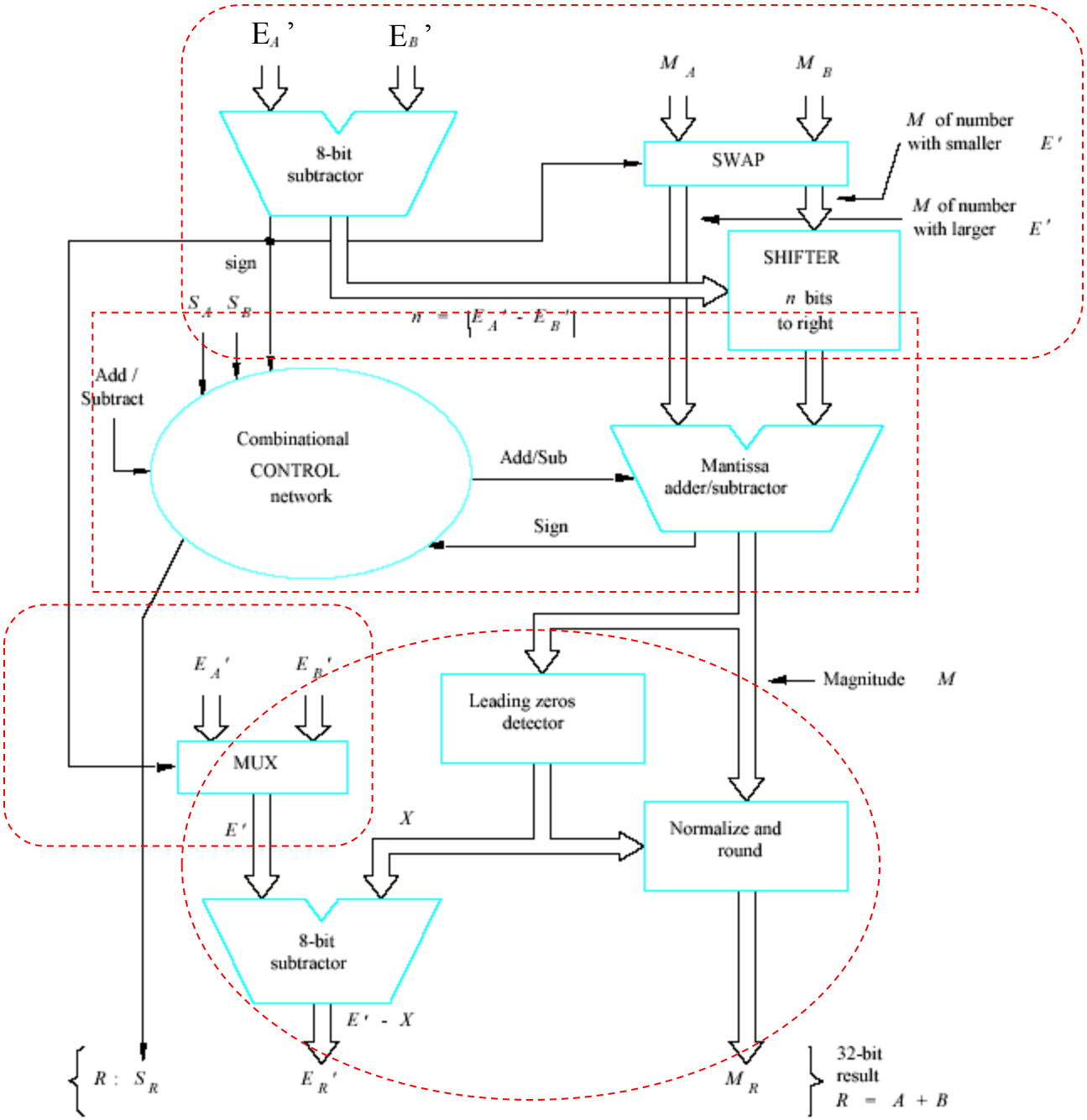


- Problems Considerable in FP Arithmetic
  - Overflow
    - Mantissa Overflow
      - The addition of two mantissas of the same sign may result in a carry out of the most significant bit.
      - If so, the mantissa of the result is shifted right and the exponent is incremented.
    - Mantissa Underflow
      - In the process of aligning exponent , digits may flow off the right end of the mantissa.
      - This can be resolved by using guard bits and some method of truncation.

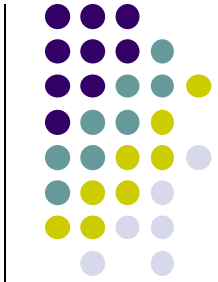
9.7 Floating-Point Numbers and Operations



- Implementing Floating-Point Operations
  - Implementation Methods
    - Software Routines
    - Hardware Routines
      - Computers will provide machine instructions for floating-point operations.
      - Note
        - In either case, the computer must be able to convert input and output from and to the use's decimal representation of numbers.
  - Example of Hardware Implementation
    - 32-bit operands , {A: S_A, E_A', M_A}, {B: S_B, E_B', M_B}

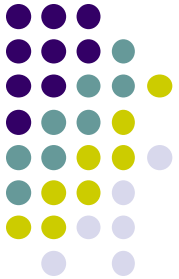

E_A'
E_B'
8-bit subtractor
M_A
M_B
SWAP
M of number with smaller E'
M of number with larger E'
SHIFTER
n bits to right
sign
S_A S_B
$n = |E_A' - E_B'|$
Add / Subtract
Combinational CONTROL network
Add/Sub
Mantissa adder/subtractor
Sign
Magnitude M
E_A'
E_B'
MUX
Leading zeros detector
E'
X
Normalize and round
8-bit subtractor
$E' - X$
E_R'
M_R
$R : S_R$
32-bit result
$R = A + B$

Summary



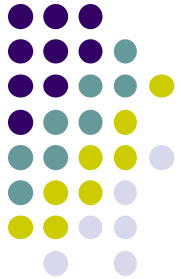
- Integer Representation
  - unsigned integer
    - non-negative
    - 2’s-complement
  - signed integer
    - signed-magnitude
    - 1’s-complement
    - 2’s-complement
  - Converting between different bit lengths
    - signed-magnitude numbers
    - signed 2’s-complement number
  - Rules of Addition and subtraction of signed numbers (2 ’s-complement form)
    - Addition
    - Subtraction

Summary



- Arithmetic overflow (2 ’s-complement form)
- 1-bit Full-adder
- n-bit ripple-carry Adder
  - Propagation delay of n-bit ripple-carry Adder
- Carry-lookahead addition
- Sequential multiplication (Unsigned Multiplication)
- Booth algorithm (Signed Multiplication)
- Restoring division or non-restoring division
- IEEE 754 Floating-Point number
- Addition, subtraction, multiplication, and division of FP
- Guard bits, truncation, normalization, overflow of FP

Homework



- 9.1
- 9.6
- 9.9 (a)
- 9.20
- 9.21
- 9.22
- 9.32