

Chapter 5

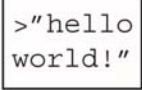


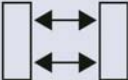
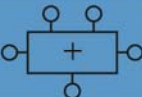
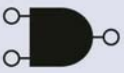
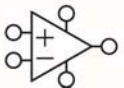

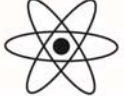
Digital Design and Computer Architecture, 2nd Edition

David Money Harris and Sarah L. Harris



Chapter 5 :: Topics

- Introduction
- Arithmetic Circuits
- Number Systems
- Sequential Building Blocks
- Memory Arrays
- Logic Arrays

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

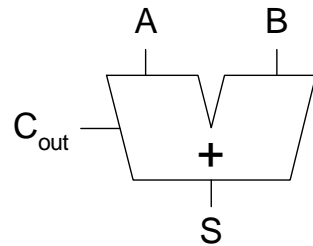
Introduction

- **Digital building blocks:**
 - Gates, multiplexers, decoders, registers, arithmetic circuits, counters, memory arrays, logic arrays
- **Building blocks demonstrate hierarchy, modularity, and regularity:**
 - Hierarchy of simpler components
 - Well-defined interfaces and functions
 - Regular structure easily extends to different sizes
- **Will use these building blocks in Chapter 7 to build microprocessor**



1-Bit Adders

Half Adder

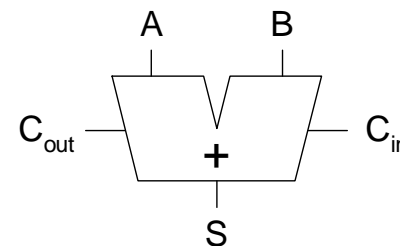


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B$$

$$C_{out} = AB$$

Full Adder



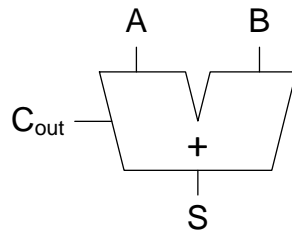
C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

Half Adder

Half Adder



$$\begin{array}{r} \text{C} \\ 0A \\ + \quad 0B \\ \hline CS \end{array}$$

A	B	C _{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

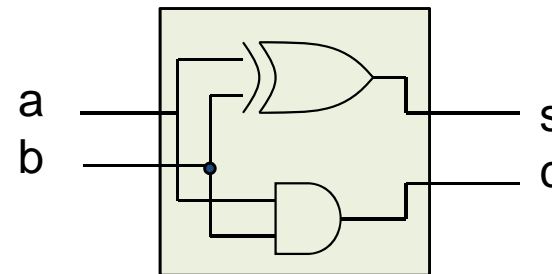
$$\begin{aligned} S &= A \oplus B \\ C_{out} &= AB \end{aligned}$$

A \ B	0	1
0	0	0
1	0	1

$$c_{out} = a b$$

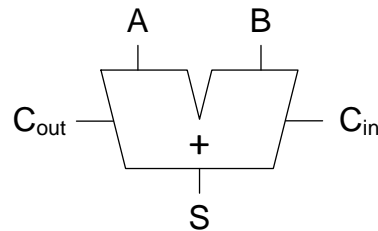
A \ B	0	1
0	0	1
1	1	0

$$s = a \oplus b$$



Full Adder

Full Adder



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

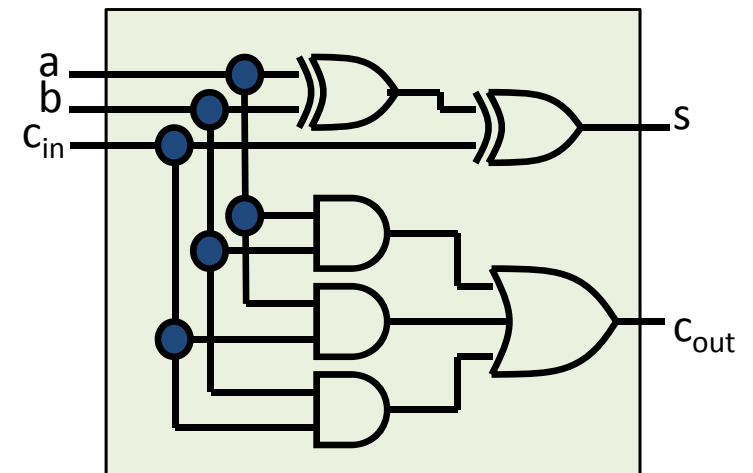
$C_{in} \backslash ab$	00	01	11	10
0	0	0	1	0
1	0	1	1	1

$$C_{out} = ab + c_{in}a + c_{in}b$$

$$\begin{array}{r}
 C_{out}C_{in} \\
 0 \ a \\
 0 \ b \\
 \hline
 C_{out}S
 \end{array}$$

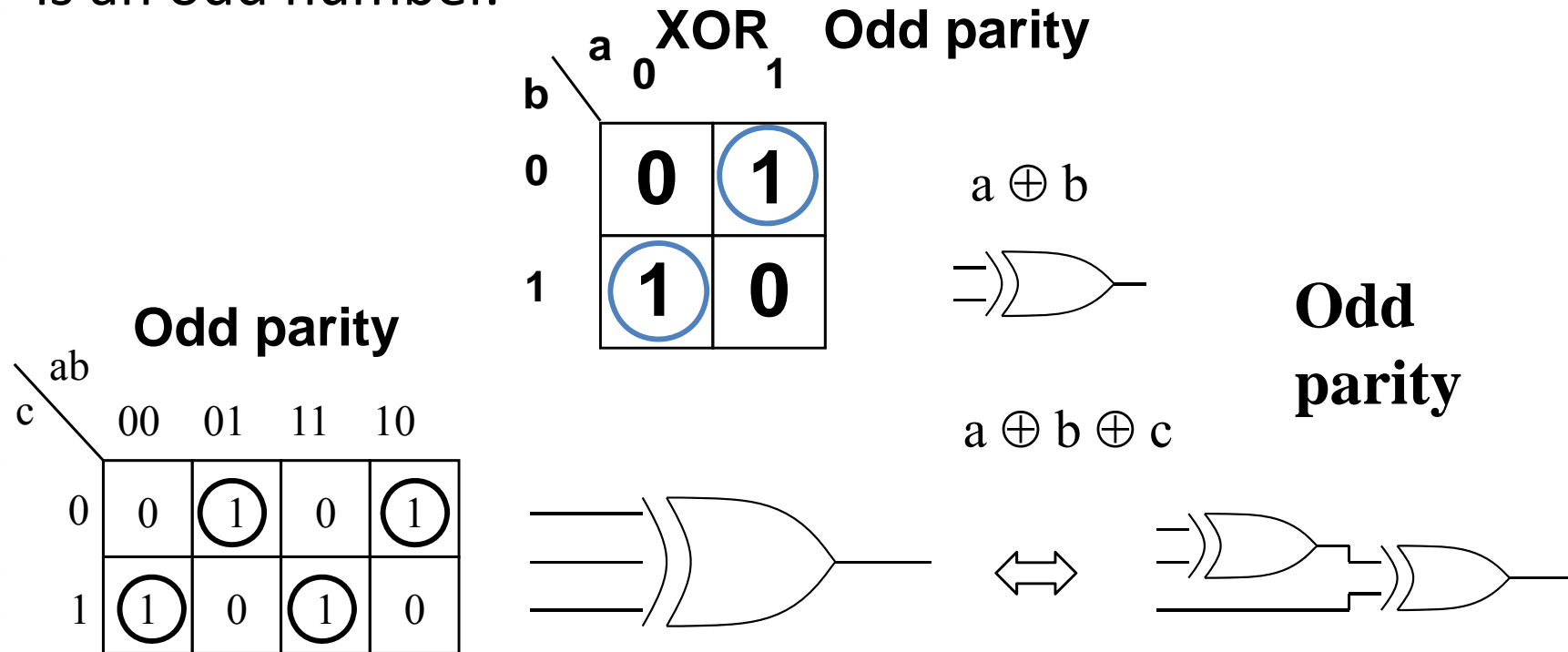
$C_{in} \backslash ab$	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$$s = a \oplus b \oplus c_{in}$$

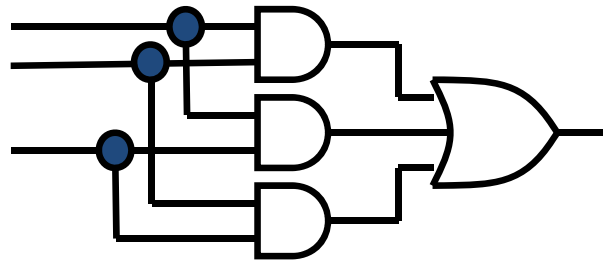


Sum function = Odd parity

The Full Adder sum function is the "odd" parity function. This is the XOR function's natural extension to more variables than two. Odd parity is when the number of 1's on the inputs is an odd number.

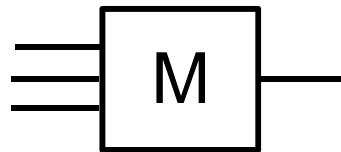


Carry function = Majority function

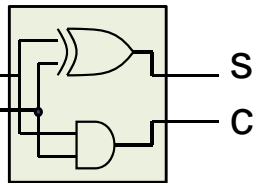


	00	01	11	10
0	0	0	1	0
1	0	1	1	1

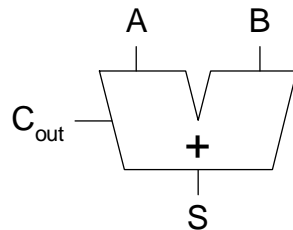
- **Majority function.** Output assumes same value 1/0 as a majority of the inputs.



1-Bit Adders



Half Adder

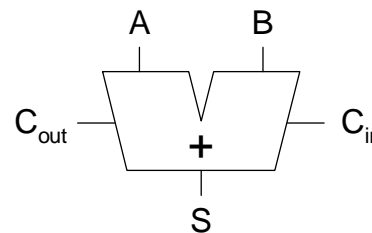


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B$$

$$C_{out} = AB$$

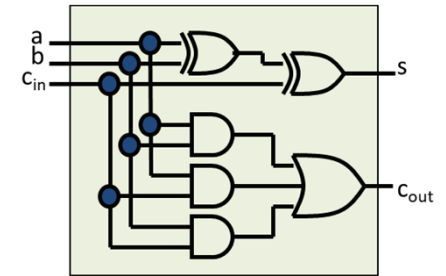
Full Adder



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

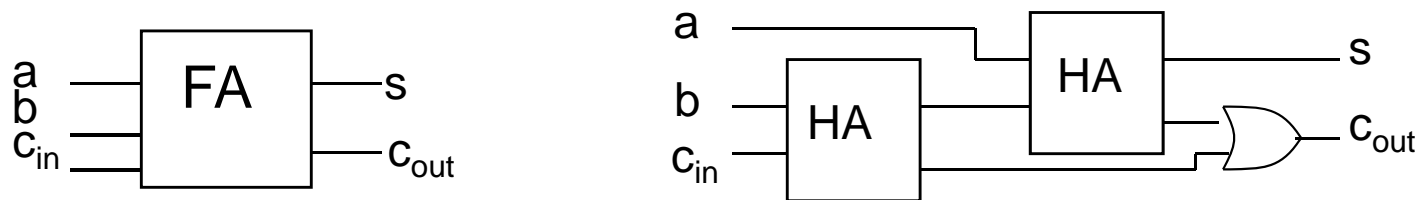
$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$



Full adder from two half-adders

- We may also construct a full adder with the help of two half-adder and an OR gate



- Composition allows us to construct new systems using known building blocks

Full adder from two half-adders

a	b	c _{in}	c _o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

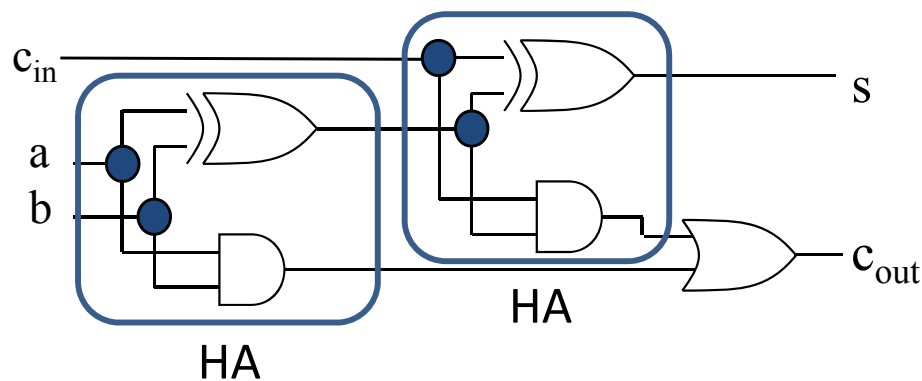
		ab			
c _{in}		00	01	11	10
	0	0	0	1	0
	1	0	1	1	1

$a \cdot b$
 $c_{in}(a \oplus b)$
 $c_{out} = ab + c_{in}(a \oplus b)$

		ab			
c _{in}		00	01	11	10
	0	0	0	1	0
	1	0	1	1	1

$$c_{out} = ab + c_{in}a + c_{in}b$$

$$= ab + c_{in}(a+b)$$



		ab			
c _{in}		00	01	11	10
	0	0	1	0	1
	1	1	0	1	0

$$s = a \oplus b \oplus c_{in}$$

Full Adder Tattoo?!

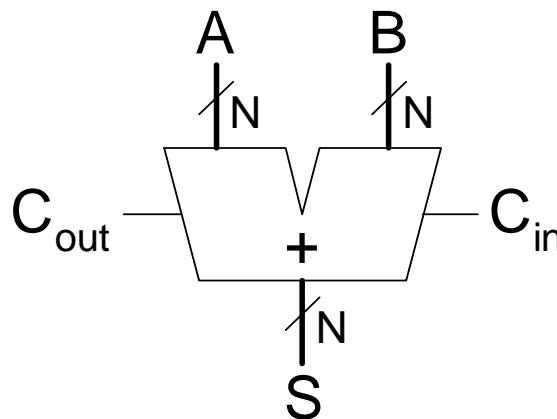


Tattoos are forever! Unfortunately this is not the "best" adder, not if you want fast computers...

Multibit Adders (CPAs)

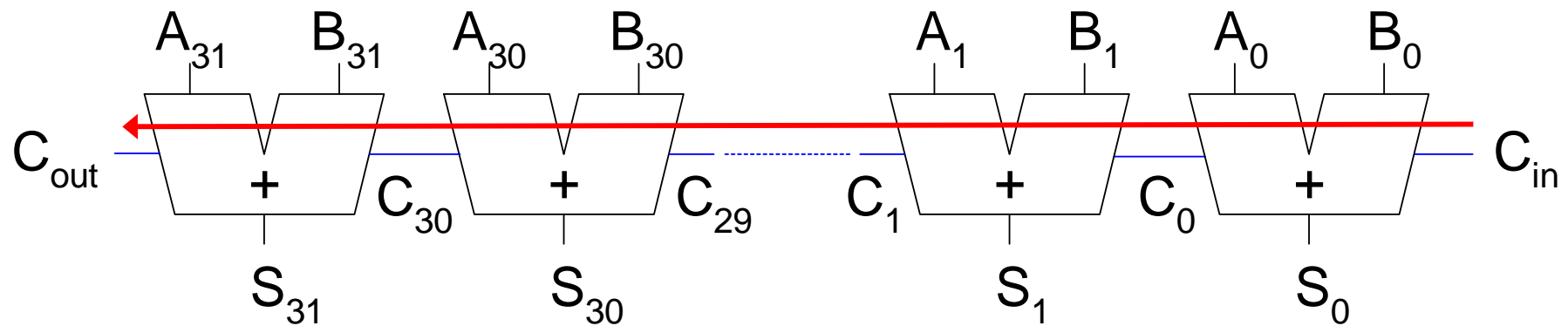
- Types of carry propagate adders (CPAs):
 - Ripple-carry (slow)
 - Carry-lookahead (fast)
 - Prefix (faster)
- Carry-lookahead and prefix adders faster for large adders but require more hardware

Symbol

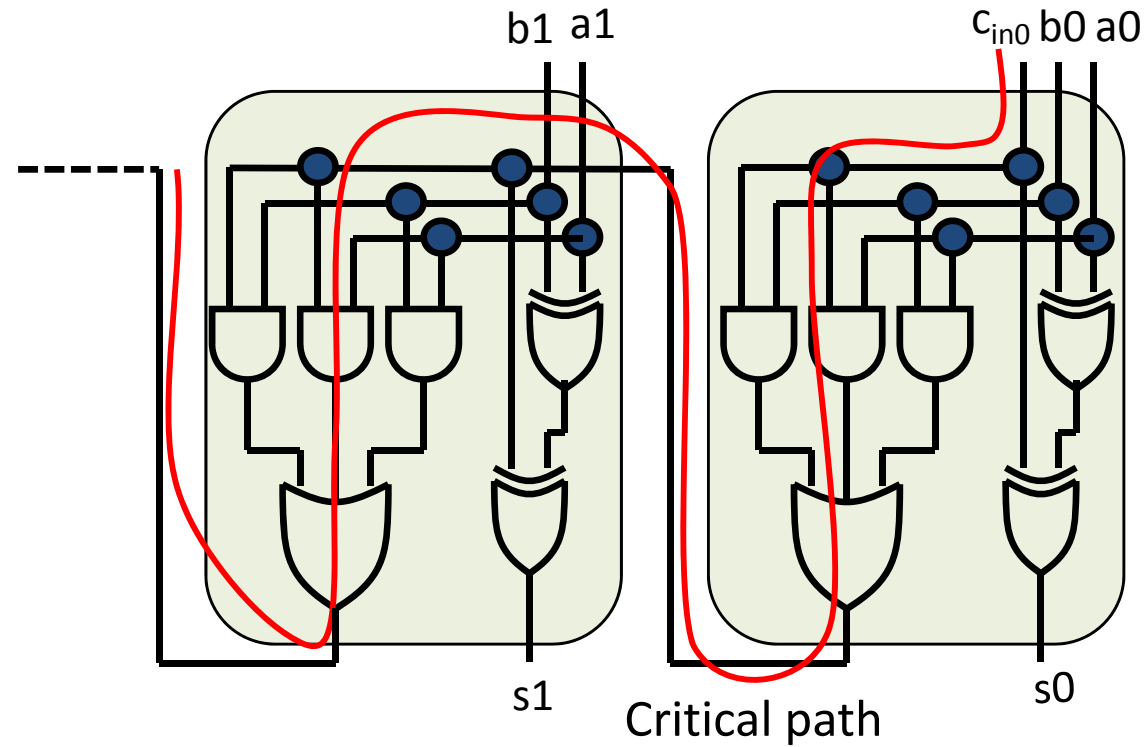


Ripple-Carry Adder

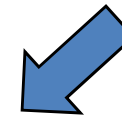
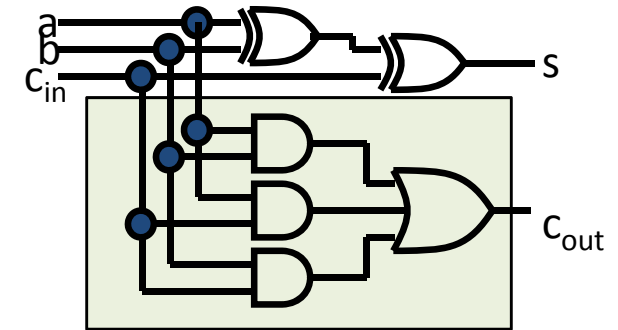
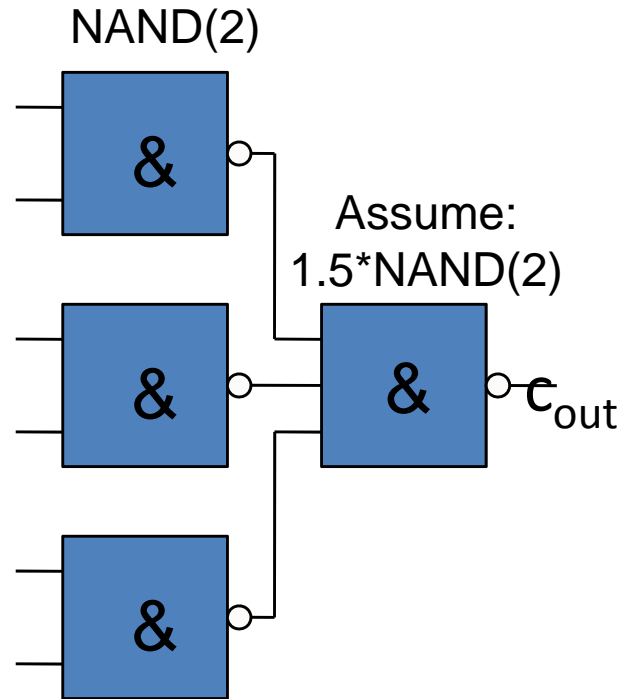
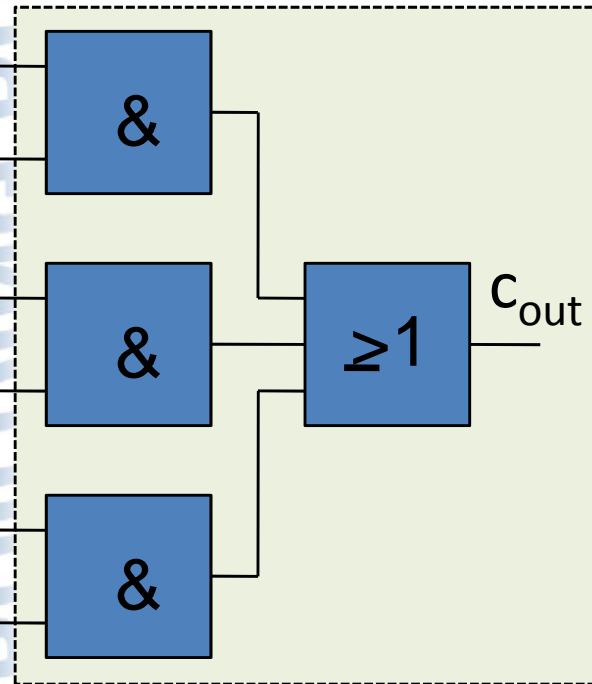
- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: **slow**



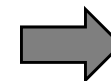
Ripple-Carry Adder Critical Path



Carry out Function with NAND Gates



Carry-out function is in the critical path and thus the delay is important!

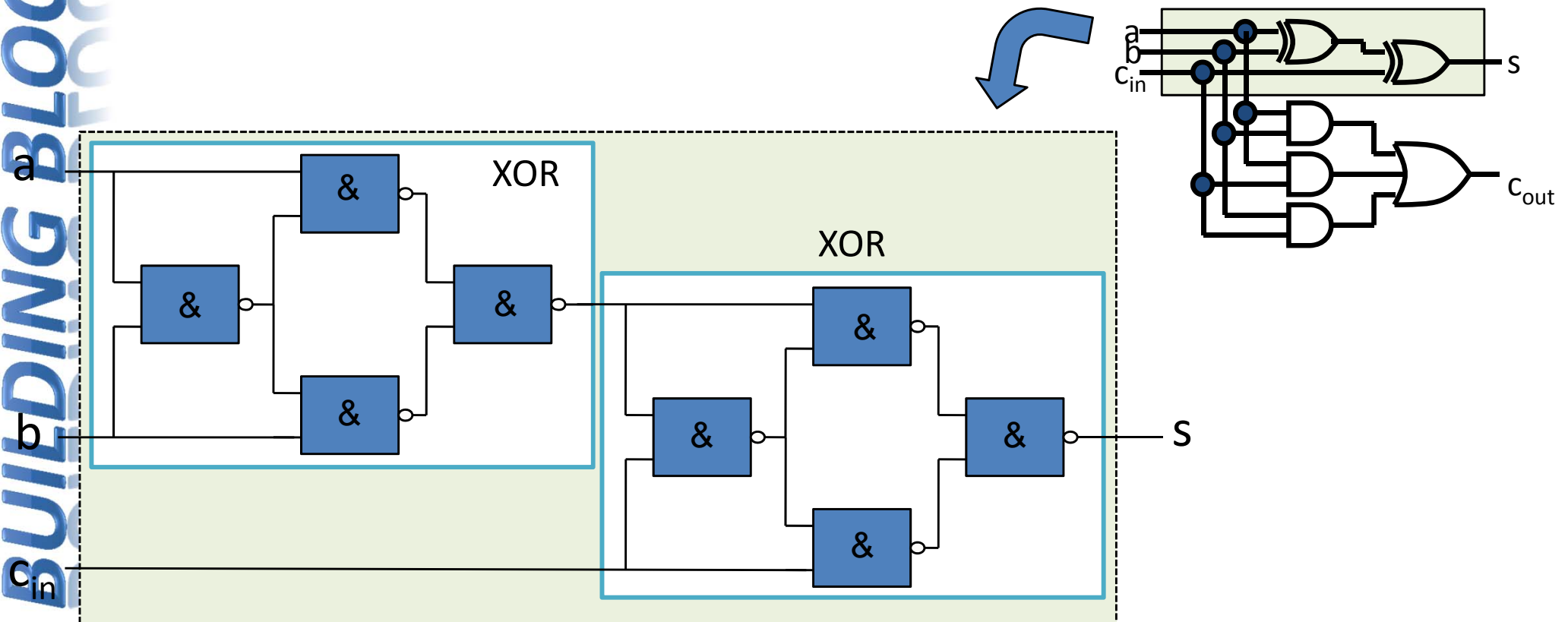


$$A_{\text{NAND-NAND}} = 4.5 \times A_{\text{NAND}}$$

$$T_{\text{NAND-NAND}} = 2.5 \times T_{\text{NAND}}$$



Sum Function with NAND Gates



Sum function is not in the critical path and thus the delay is not important!

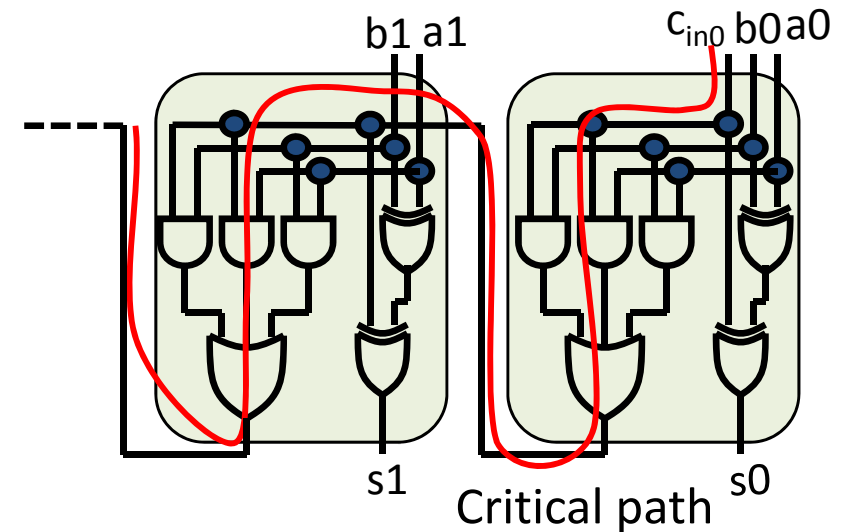
$$A_{\text{NAND-NAND}} = 8A_{\text{NAND}}$$

$$T_{\text{NAND}} = 6 * T_{\text{NAND}}$$

Area and Delay Calculations

Ripple-Carry Adder:

- $T = n * T_{FA} = n * (2.5 * T_{NAND})$
- $A = n * A_{FA} = n * (4.5 * A_{NAND} + 8A_{NAND}) =$
 $n * (12.5A_{NAND})$



Ripple-Carry Adder Delay

$$t_{\text{ripple}} = Nt_{FA}$$

where t_{FA} is the delay of a 1-bit full adder

Can we construct a faster adder?

- The delay of a Ripple Carry Adder (RCA) grows proportionally to the number of bits
- For 32 bits, the delay will be very large

How to reduce the critical path?
Carry-Look-ahead Adder (CLA)

Carry-Lookahead Adder

- Compute carry out (C_{out}) for k -bit blocks using *generate* and *propagate* signals
- **Some definitions:**
 - Column i produces a carry out by either *generating* a carry out or *propagating* a carry in to the carry out
 - Generate (G_i) and propagate (P_i) signals for each column:
 - Column i will generate a carry out if A_i AND B_i are both 1.

$$G_i = A_i B_i$$

- Column i will propagate a carry in to the carry out if A_i OR B_i is 1.

$$P_i = A_i + B_i$$

- The carry out of column i (C_i) is:

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1}$$



Carry-Lookahead Addition

- **Step 1:** Compute G_i and P_i for all columns
- **Step 2:** Compute G and P for k -bit blocks
- **Step 3:** C_{in} propagates through each k -bit propagate/generate block

Carry-Lookahead Adder

- **Example:** 4-bit blocks ($G_{3:0}$ and $P_{3:0}$) :

$$G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0))$$

$$P_{3:0} = P_3 P_2 P_1 P_0$$

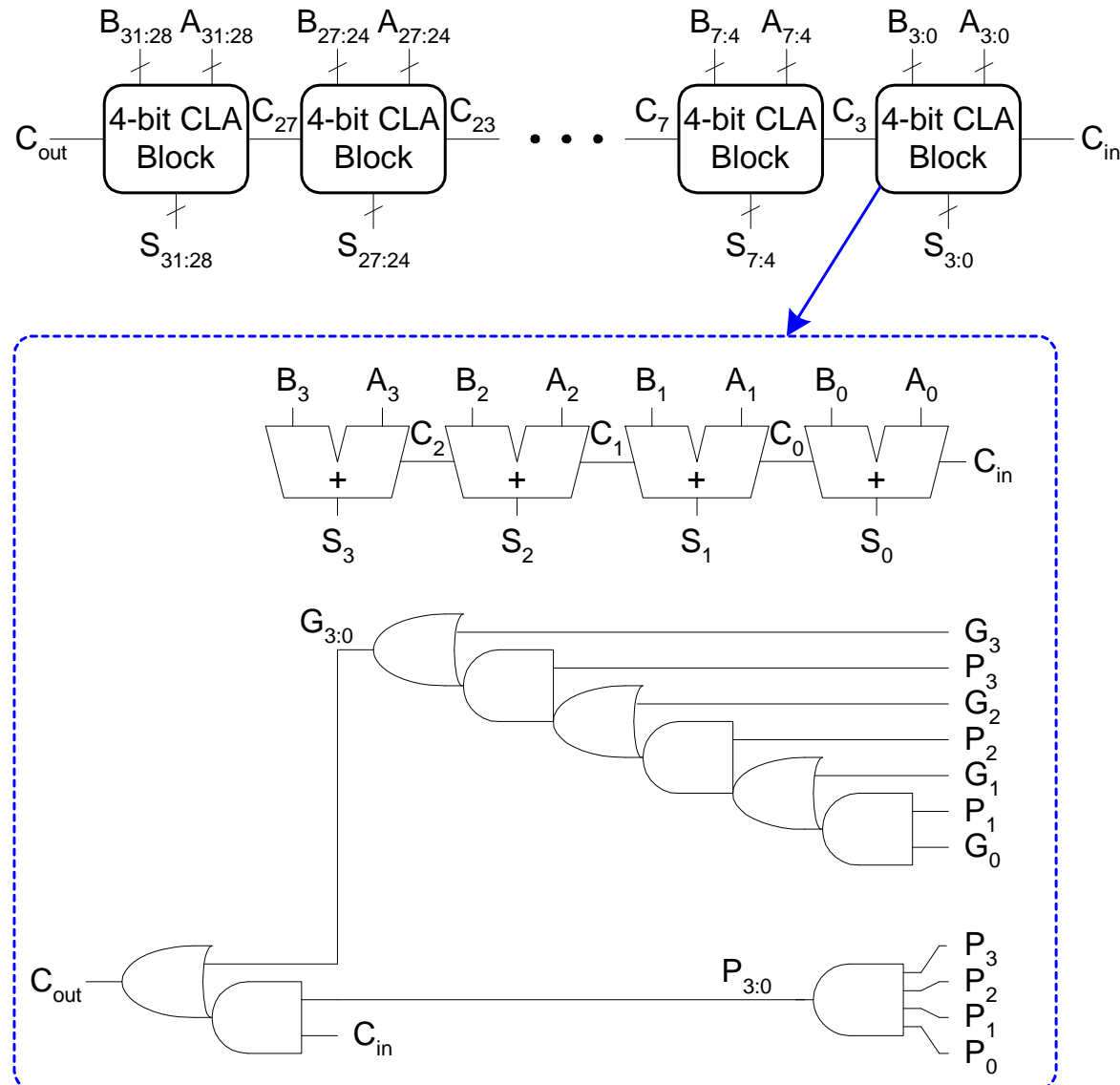
- **Generally,**

$$G_{i:j} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} G_j))$$

$$P_{i:j} = P_i P_{i-1} P_{i-2} P_j$$

$$C_i = G_{i:j} + P_{i:j} C_{j-1}$$

32-bit CLA with 4-bit Blocks



Carry-Lookahead Adder Delay

For N -bit CLA with k -bit blocks:

$$t_{CLA} = t_{pg} + t_{pg_block} + (N/k - 1)t_{AND_OR} + kt_{FA}$$

- t_{pg} : delay to generate all P_i, G_i
- t_{pg_block} : delay to generate all $P_{i:j}, G_{i:j}$
- t_{AND_OR} : delay from C_{in} to C_{out} of final AND/OR gate in k -bit CLA block

An N -bit carry-lookahead adder is generally much faster than a ripple-carry adder for $N > 16$



Carry-Lookahead Adder

- Carry-bit c_0

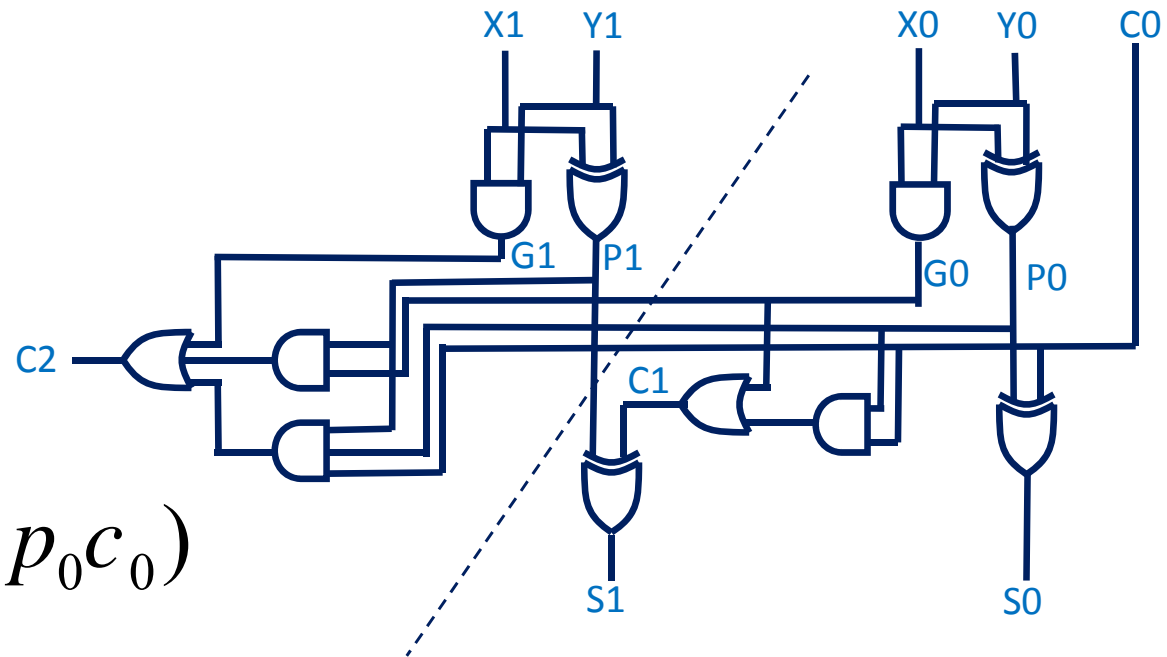
$$c_1 = g_0 + p_0 c_0$$

- Carry-bit c_1

$$c_2 = g_1 + p_1 c_1$$

$$= g_1 + p_1(g_0 + p_0 c_0)$$

$$= g_1 + p_1 g_0 + p_1 p_0 c_0$$



c_2 will be produced after 3 gate delays
 s_2 will be produced after 4 gate delays

Carry-Lookahead Adder

$$C_1 = g_0 + p_0 c_0$$

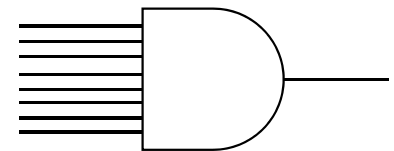
$$C_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$$

$$C_3 = g_2 + p_2 c_2 = g_2 + p_2 (g_1 + p_1 g_0 + p_1 p_0 c_0) = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$

...

$$C_8 = g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4 + p_7 p_6 p_5 p_4 g_3 + p_7 p_6 p_5 p_4 p_3 g_2 + p_7 p_6 p_5 p_4 p_3 p_2 g_1 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 g_0 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0 c_0$$

Ooops!



C_8 will be produced after 3 gate delays (larger gates)

S_8 will be produced after 4 gate delays (larger gates)

Area($n = 8$) = VERY BIG!

Block Propagate and Generate

Now use column Propagate and Generate signals to compute ***Block Propagate*** and ***Generate*** signals for k-bit blocks, i.e.:

- Compute if a k-bit group will propagate a carry in (to the block) to the carry out (of the block)
- Compute if a k-bit group will generate a carry out (of the block)

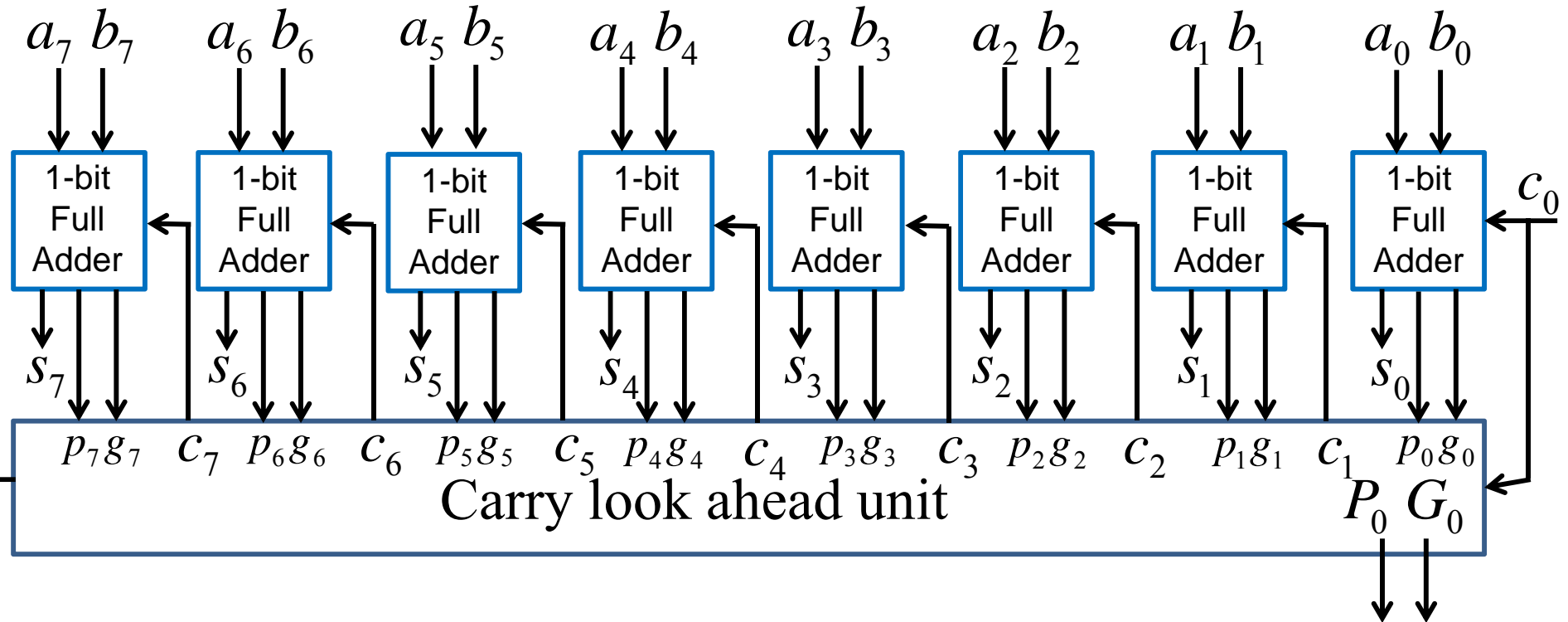
N-bit Carry-Lookahead Adder

To reduce the area, we can use *hierarchical* approach

- To design a 32-bit adder, we divide it into 4 eight-bit blocks and implement each block as an 8-bit carry look-ahead adder
- A second-level carry-look-ahead can be used to produce quickly the carry between the blocks

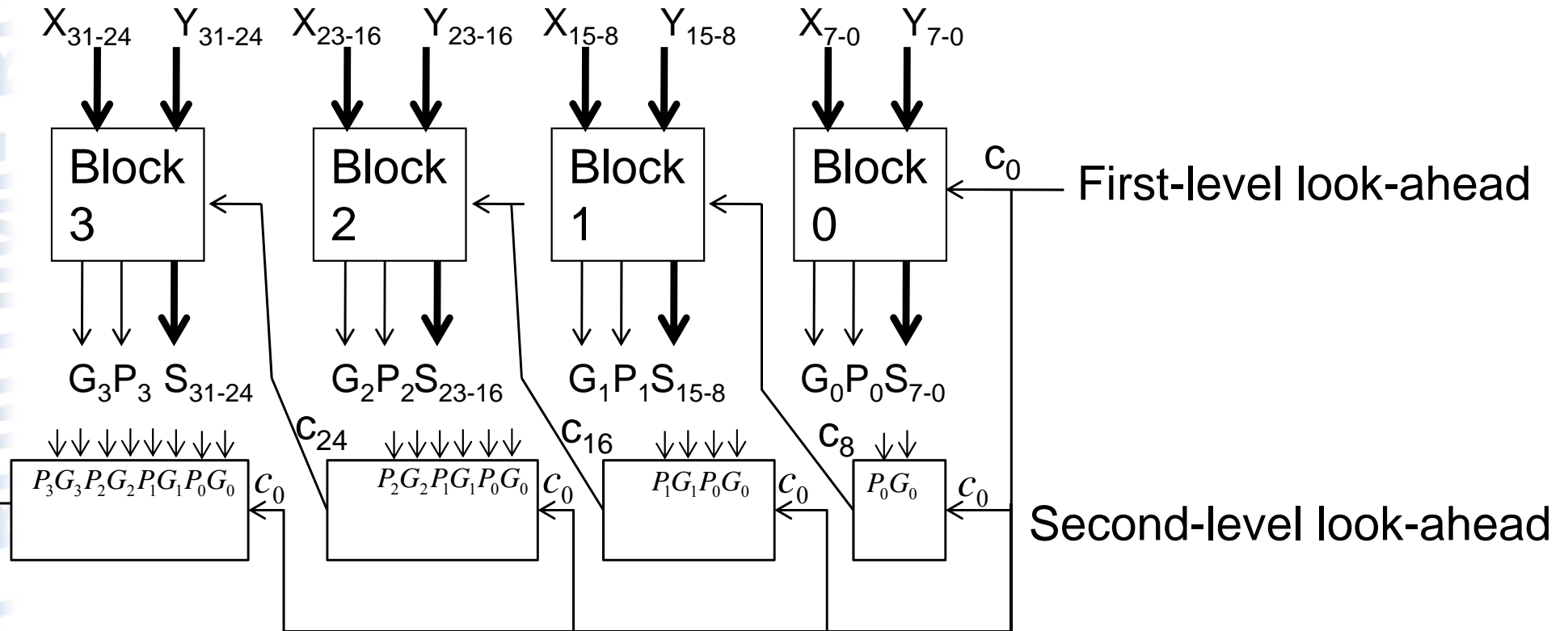
Hierarchical expansion

$$c_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0 = a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0 + b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$$



$$c_8 = g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4 + p_7 p_6 p_5 p_4 g_3 + p_7 p_6 p_5 p_4 p_3 g_2 + p_7 p_6 p_5 p_4 p_3 p_2 g_1 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 g_0 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0 c_0$$

Hierarchical expansion



$$P_0 = p_7p_6p_5p_4p_3p_2p_1p_0 \quad G_0 = g_7 + p_7g_6 + p_7p_6g_5 + \dots + p_7p_6p_5p_4p_3p_2p_1p_0$$



Carry-Select-Adder (CSA)

- Ripple Carry Adder (RCA) is simple but slow!
- Carry-Look-ahead Adder (CLA) is fast but complex with large area!

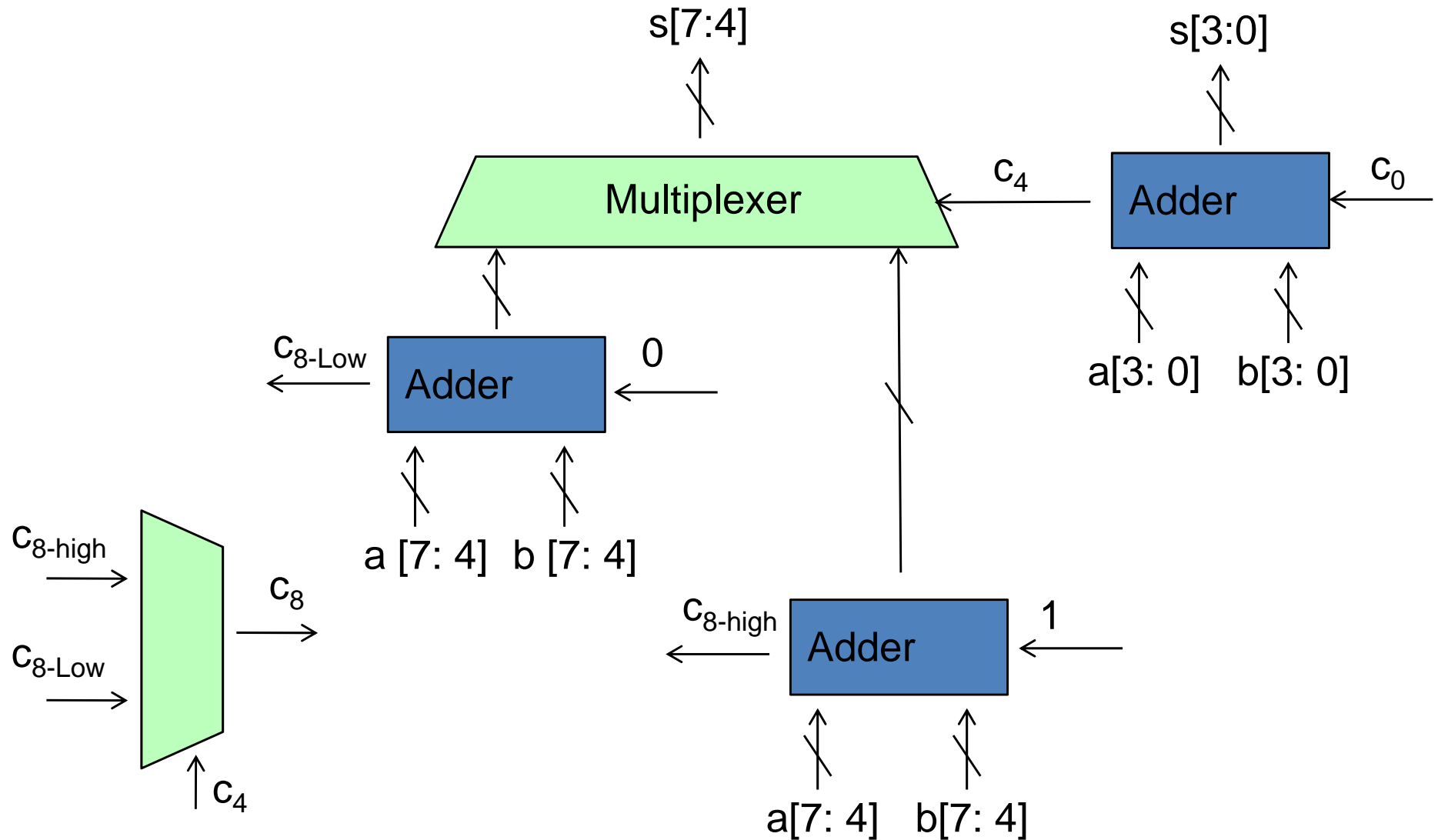
Any other alternative?
Carry-Select Adder (CSA)

Carry-Select-Adder (CSA)

- Divide an adder in two stages with the same number of bits
- To speed up the process, the result of the second step is estimated in advance for two cases
 - Carry-in = 0
 - Carry-in = 1
- When the calculation of the carry bit is completed for the first step, one chooses a result of the second step depending on carry-bit value



8-bit Carry-Select Adder



Adder Delay Comparisons

Compare delay of: 32-bit ripple-carry, carry-lookahead adders

- CLA has 4-bit blocks
- 2-input gate delay = 100 ps; full adder delay = 300 ps

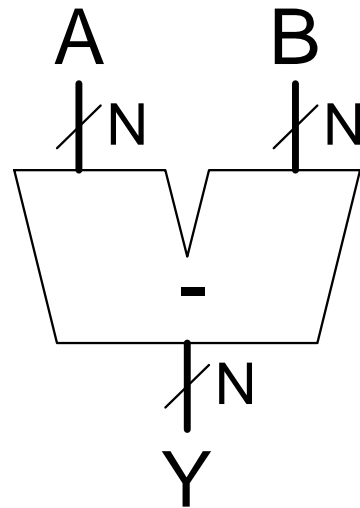
$$\begin{aligned}t_{\text{ripple}} &= Nt_{FA} = 32(300 \text{ ps}) \\ &= \mathbf{9.6 \text{ ns}}\end{aligned}$$

$$\begin{aligned}t_{CLA} &= t_{pg} + t_{pg_block} + (N/k - 1)t_{AND_OR} + kt_{FA} \\ &= [100 + 600 + (7)200 + 4(300)] \text{ ps} \\ &= \mathbf{3.3 \text{ ns}}\end{aligned}$$

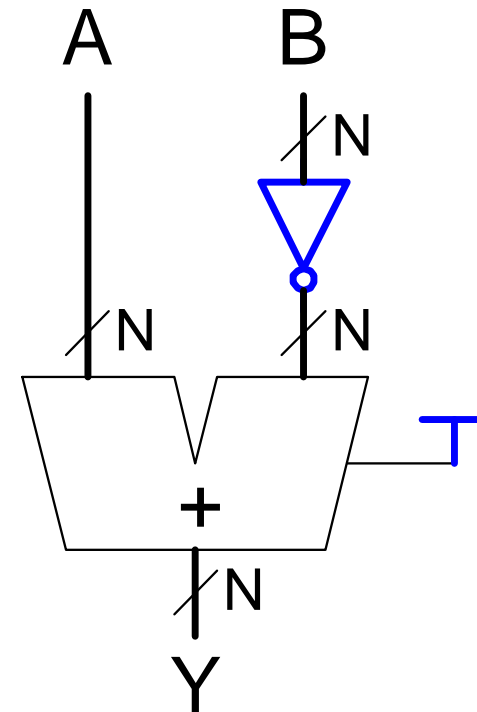


Subtractor

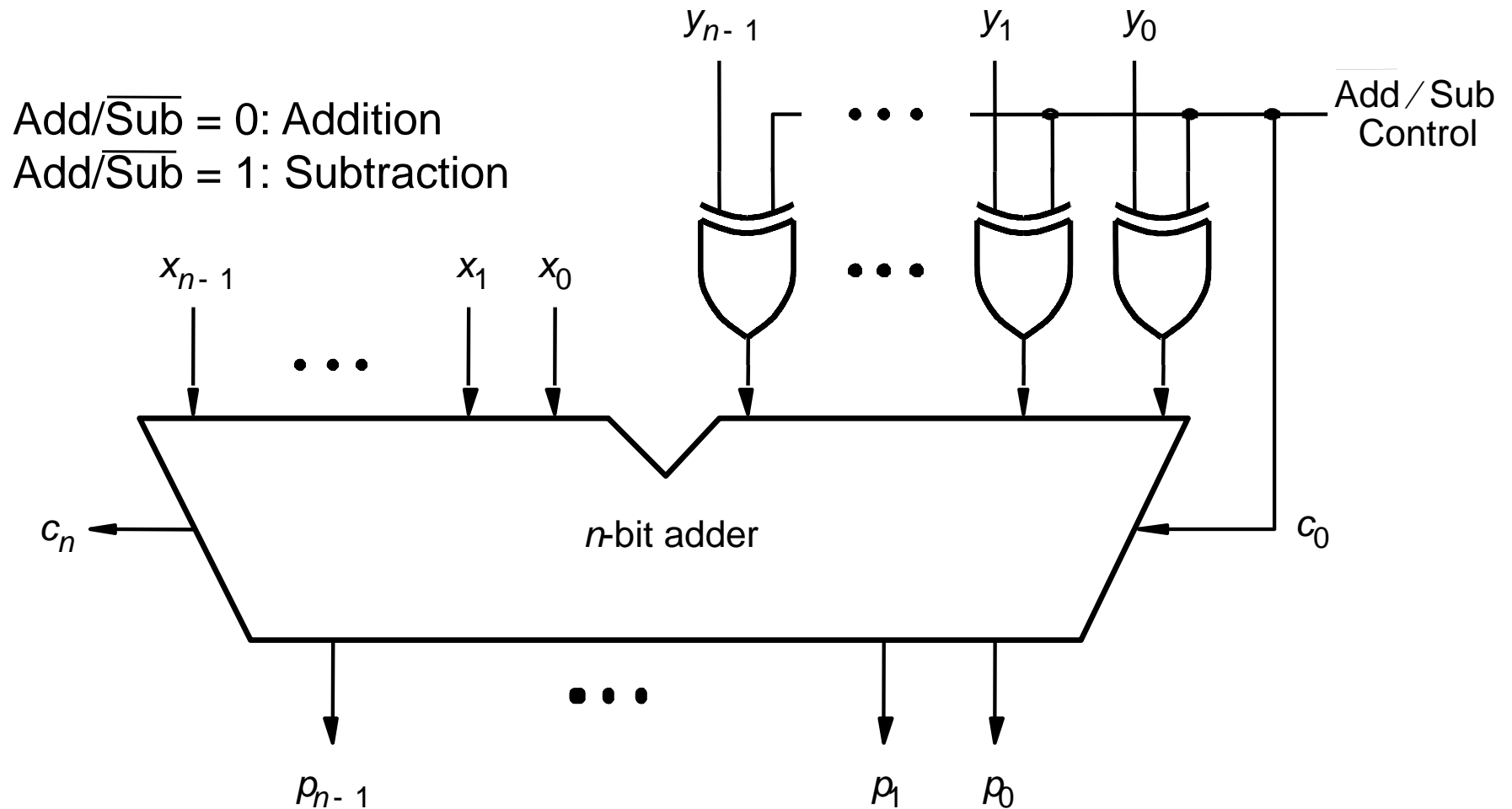
Symbol



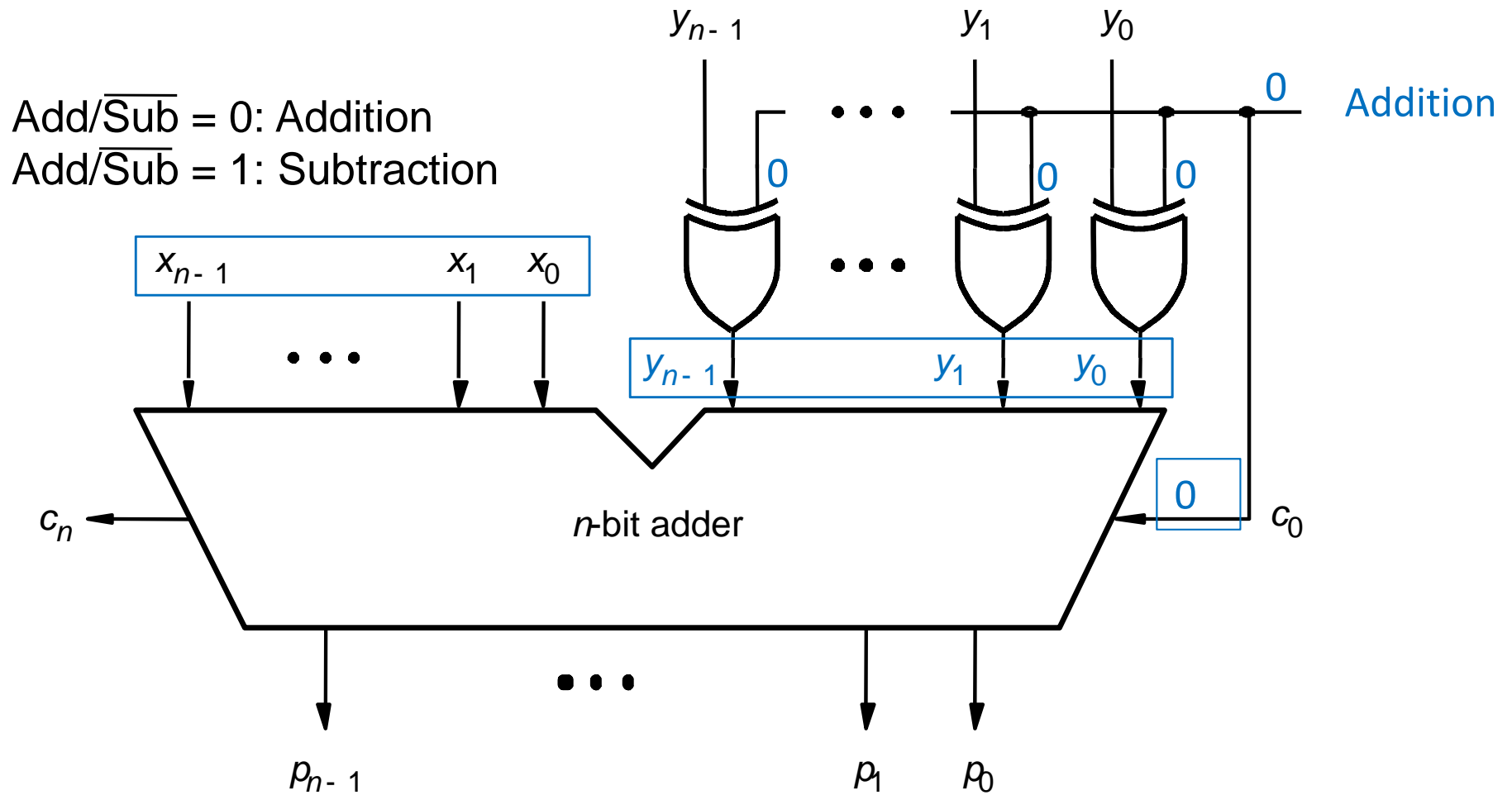
Implementation



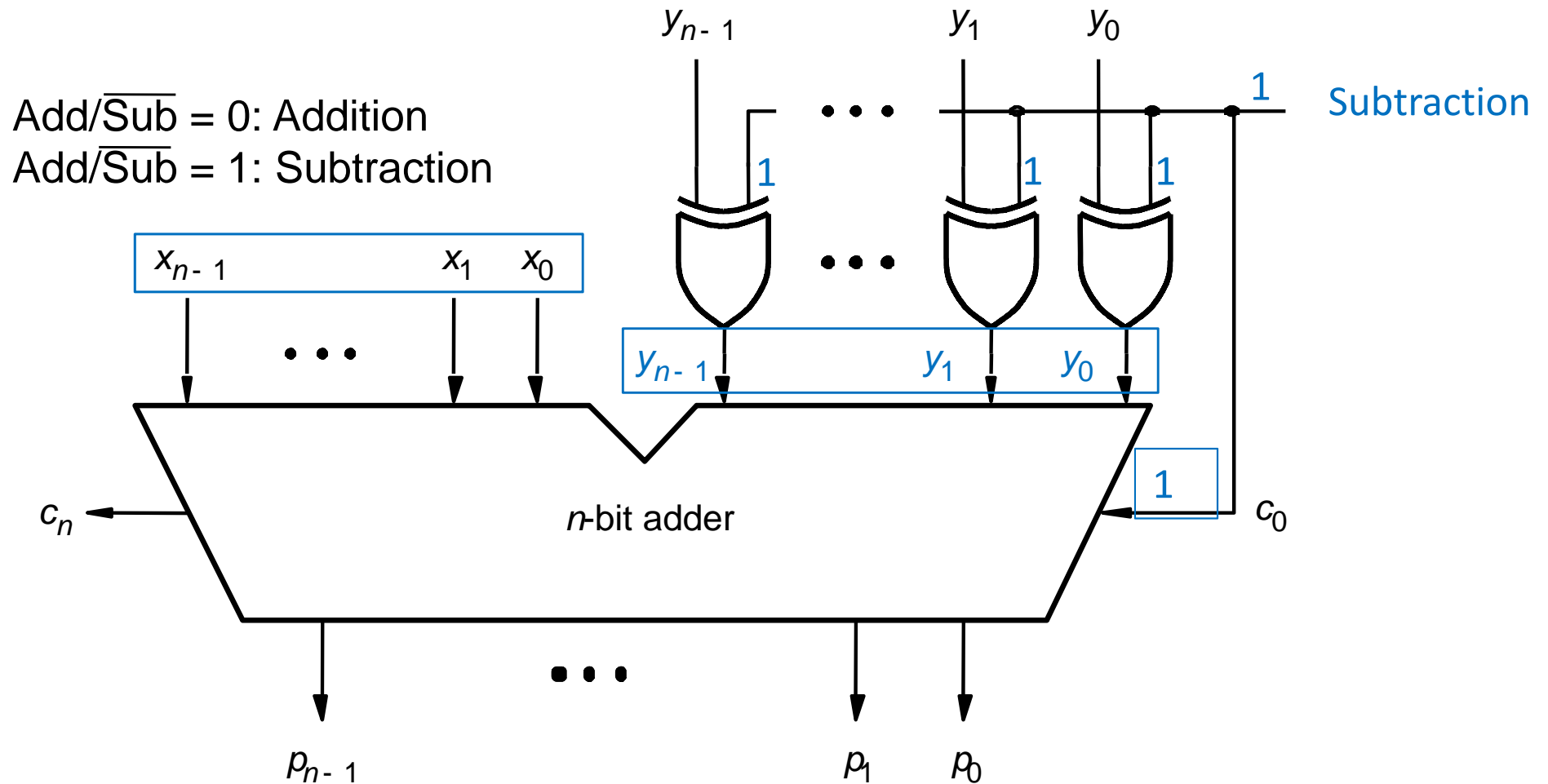
Addition / Subtraction



Addition

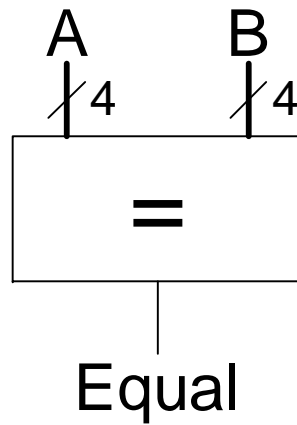


Subtraction

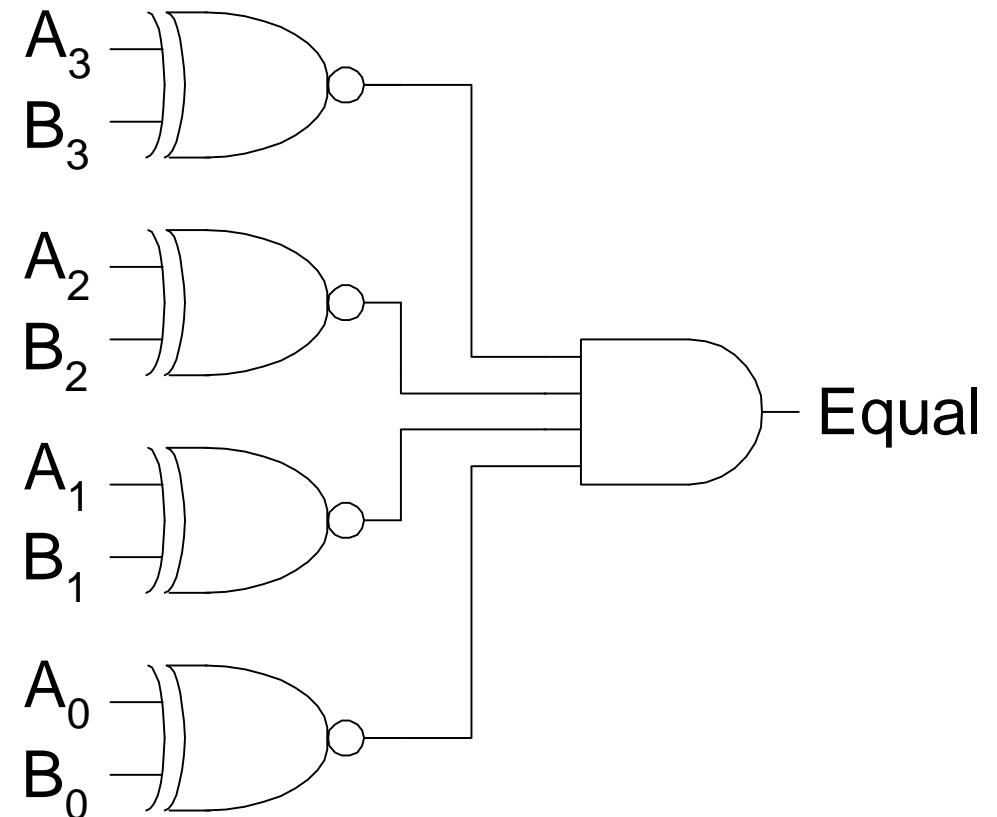


Comparator: Equality

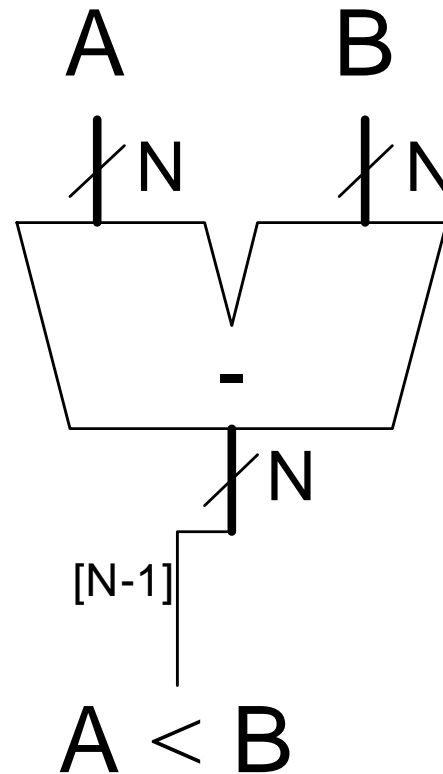
Symbol



Implementation

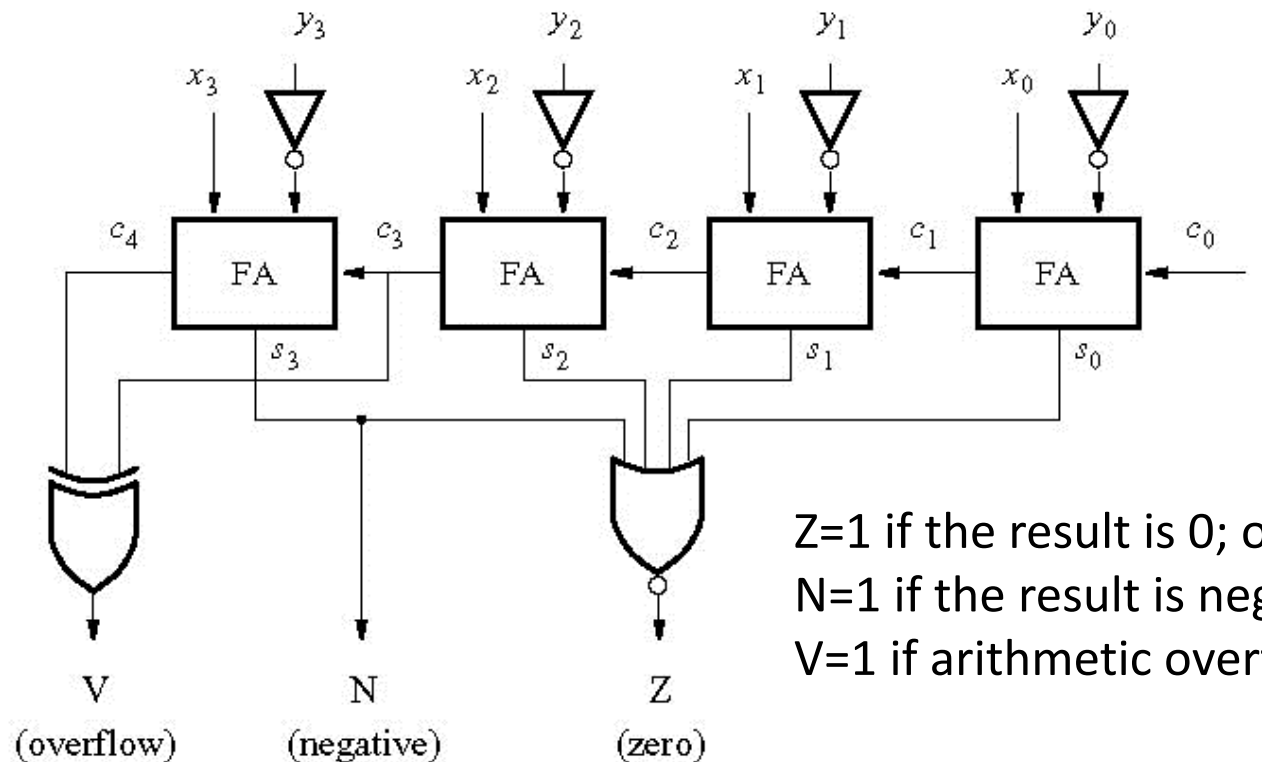


Comparator: Less Than



Comparator

- The comparator can be implemented as a subtraction $X-Y$



$Z=1$ if the result is 0; otherwise $Z=0$
 $N=1$ if the result is negative
 $V=1$ if arithmetic overflow occurs

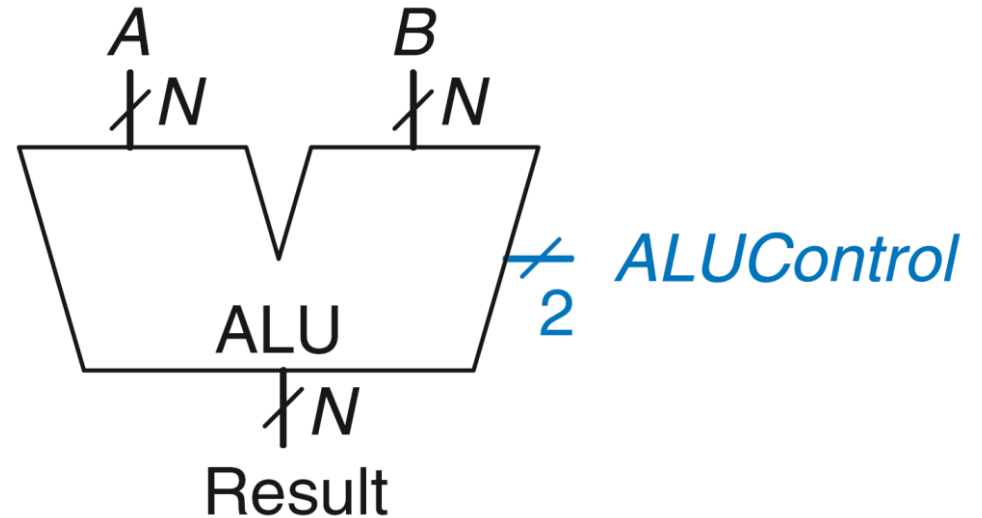
ALU: Arithmetic Logic Unit

ALU should perform:

- Addition
- Subtraction
- AND
- OR

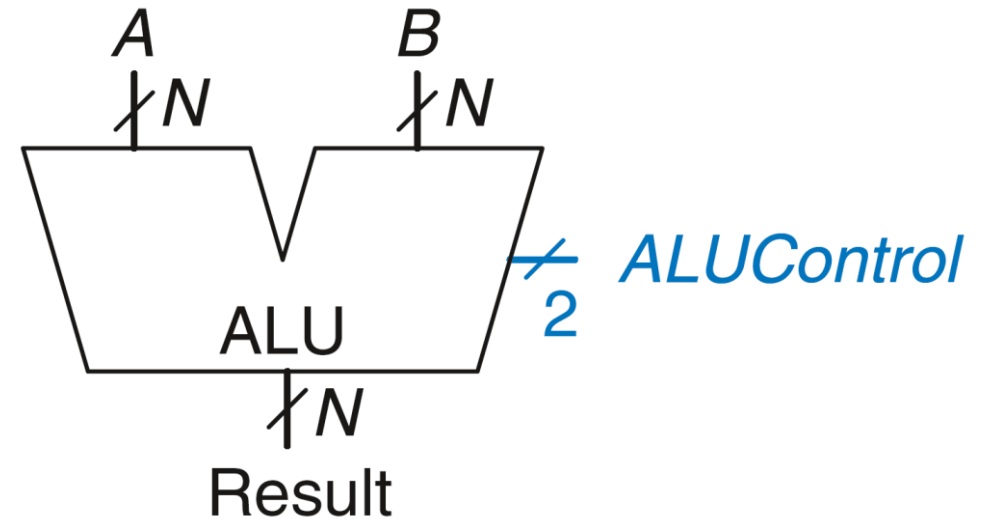
ALU: Arithmetic Logic Unit

ALUControl _{1:0}	Function
00	Add
01	Subtract
10	AND
11	OR



ALU: Arithmetic Logic Unit

ALUControl _{1:0}	Function
00	Add
01	Subtract
10	AND
11	OR



Example: Perform $A + B$

$ALUControl = 00$

$Result = A + B$

ALU: Arithmetic Logic Unit

ALUControl _{1:0}	Function
00	Add
01	Subtract
10	AND
11	OR

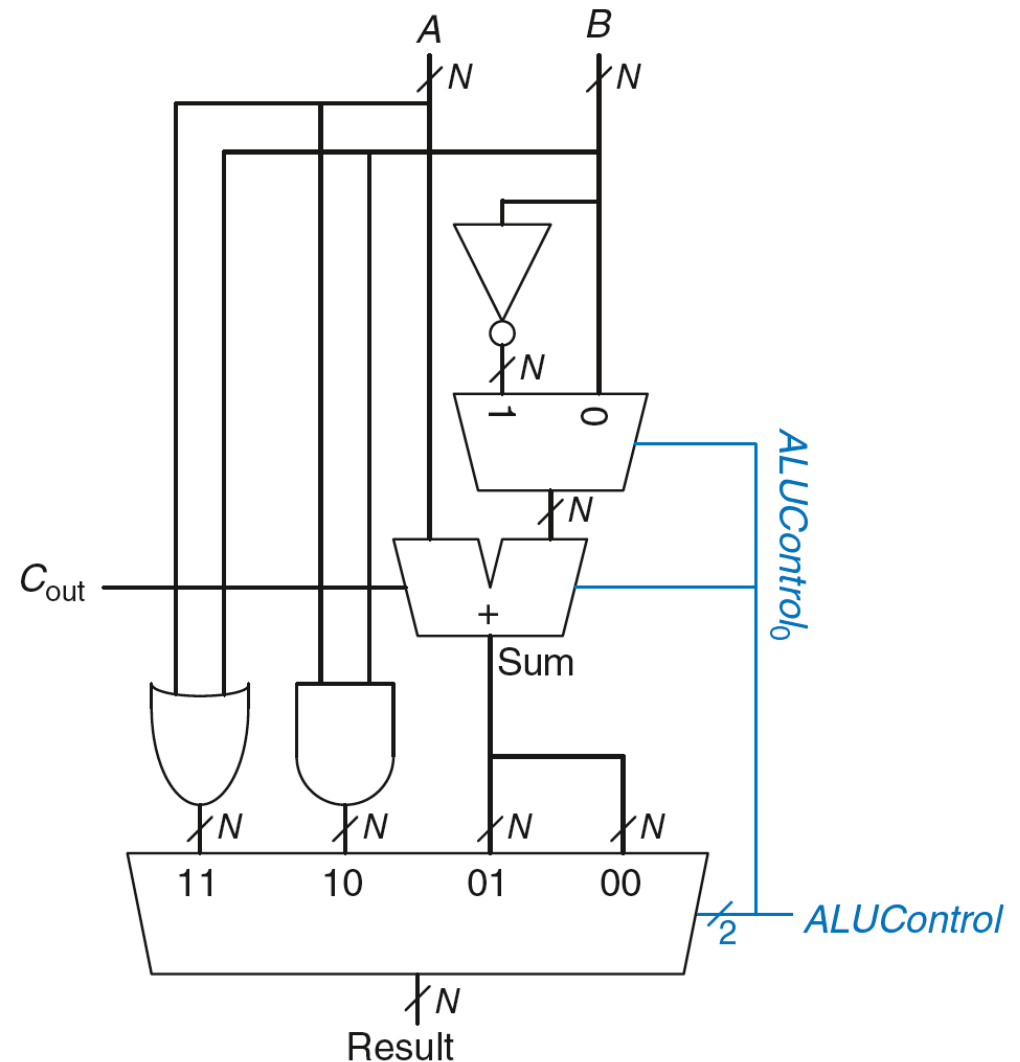
Example: Perform A OR B

$$ALUControl_{1:0} = 11$$

Mux selects output of OR gate as

Result, so

***Result* = A OR B**



ALU: Arithmetic Logic Unit

ALUControl _{1:0}	Function
00	Add
01	Subtract
10	AND
11	OR

Example: Perform $A + B$

$$ALUControl_{1:0} = 00$$

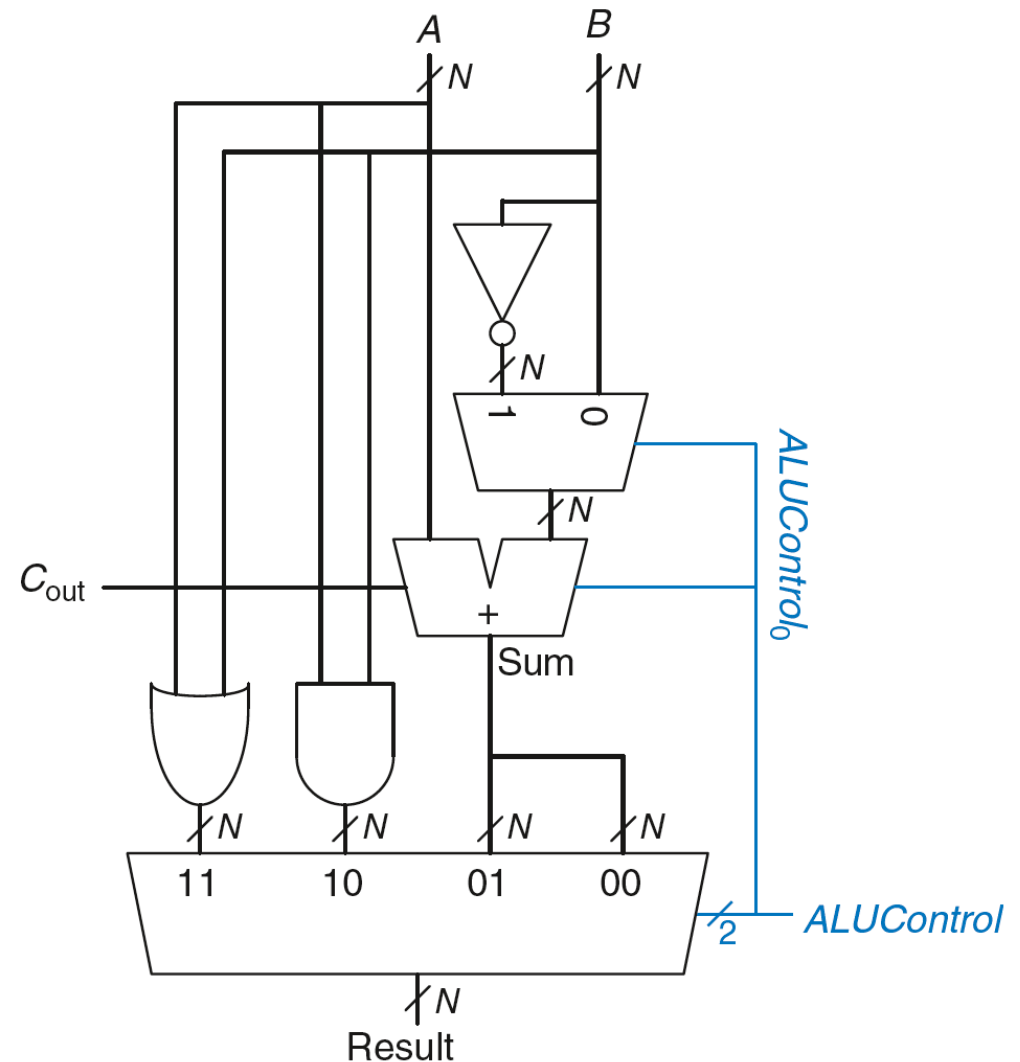
$ALUControl_0 = 0$, so:

Cin to adder = 0

2nd input to adder is B

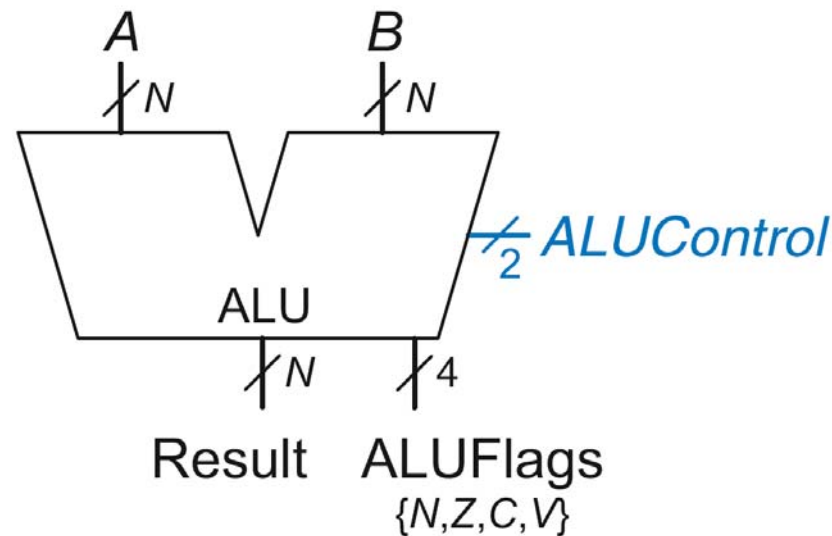
Mux selects *Sum* as *Result*, so

***Result* = A + B**

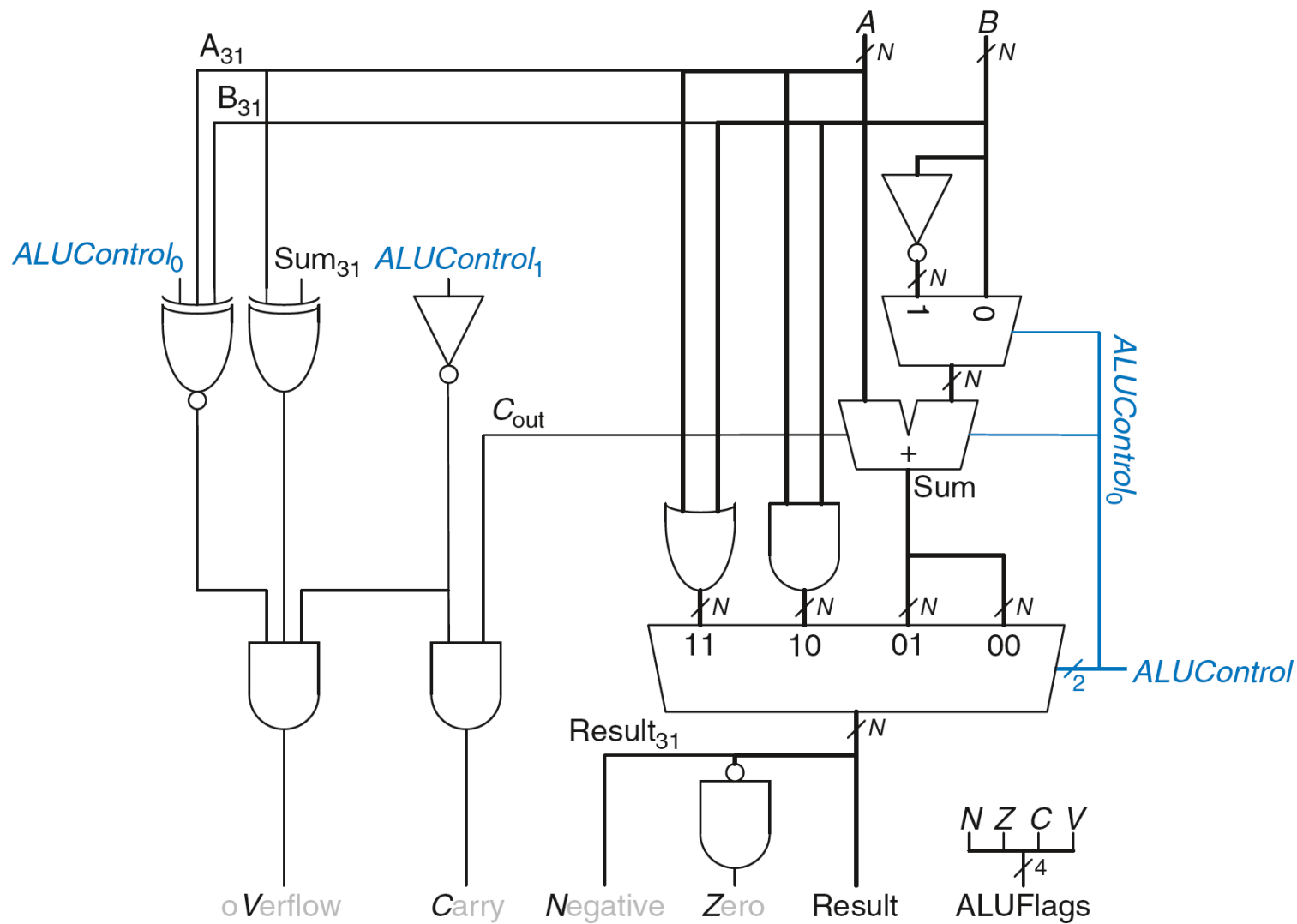


ALU with Status Flags

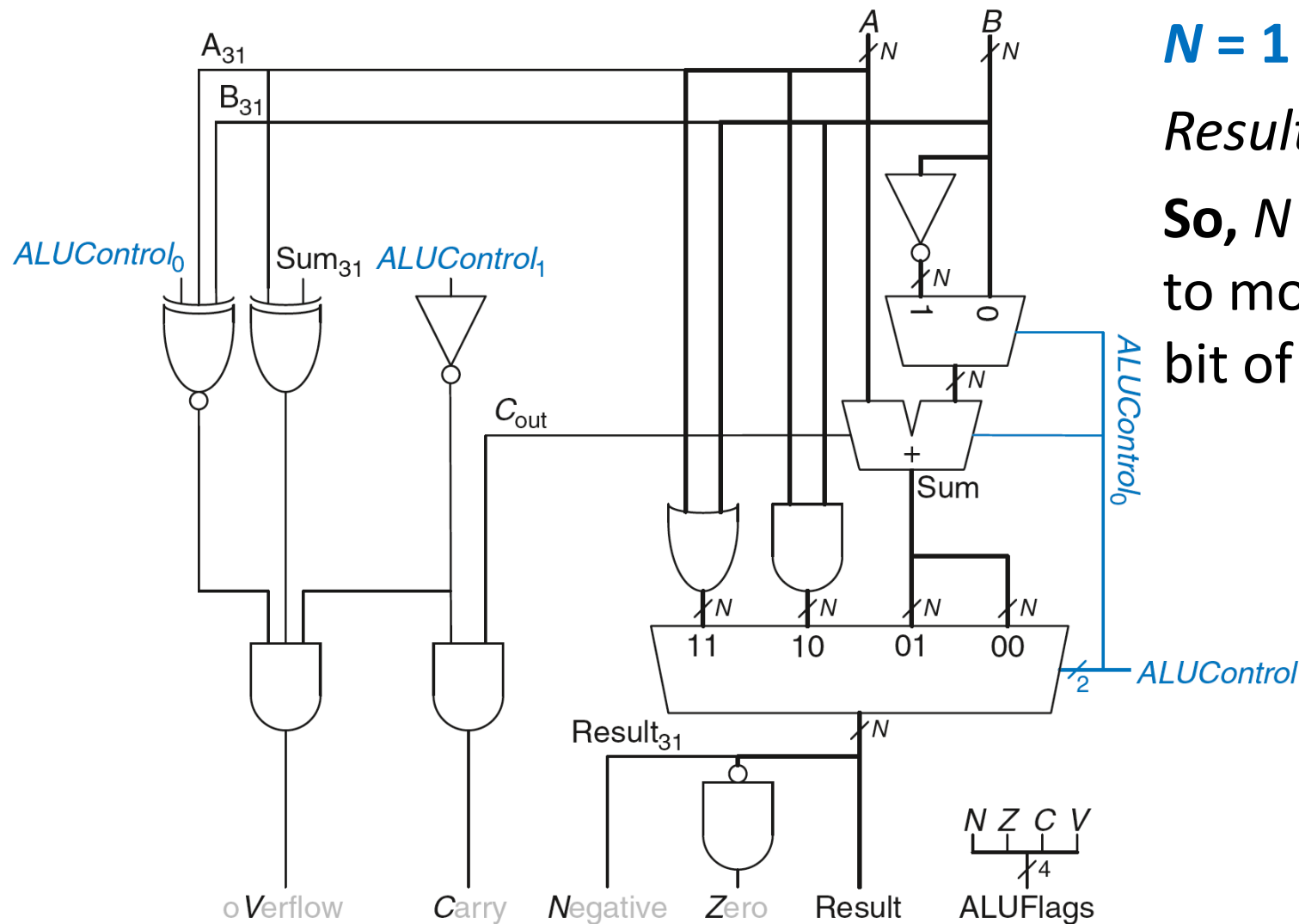
Flag	Description
<i>N</i>	Result is N egative
<i>Z</i>	Result is Z ero
<i>C</i>	Adder produces C arry out
<i>V</i>	Adder o V erflowed



ALU with Status Flags



ALU with Status Flags: Negative

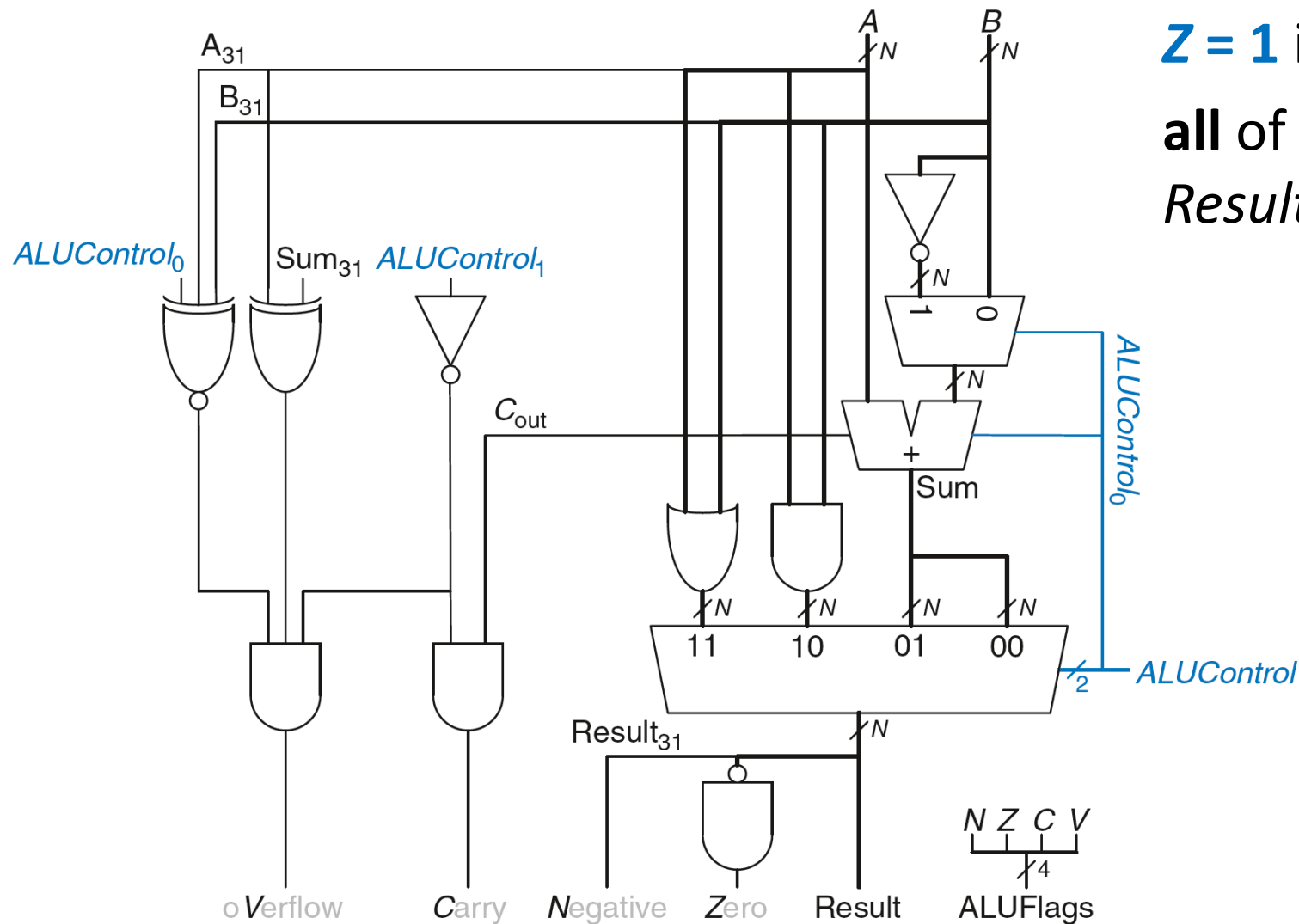


$N = 1$ if:

Result is **negative**

So, N is connected to most significant bit of *Result*

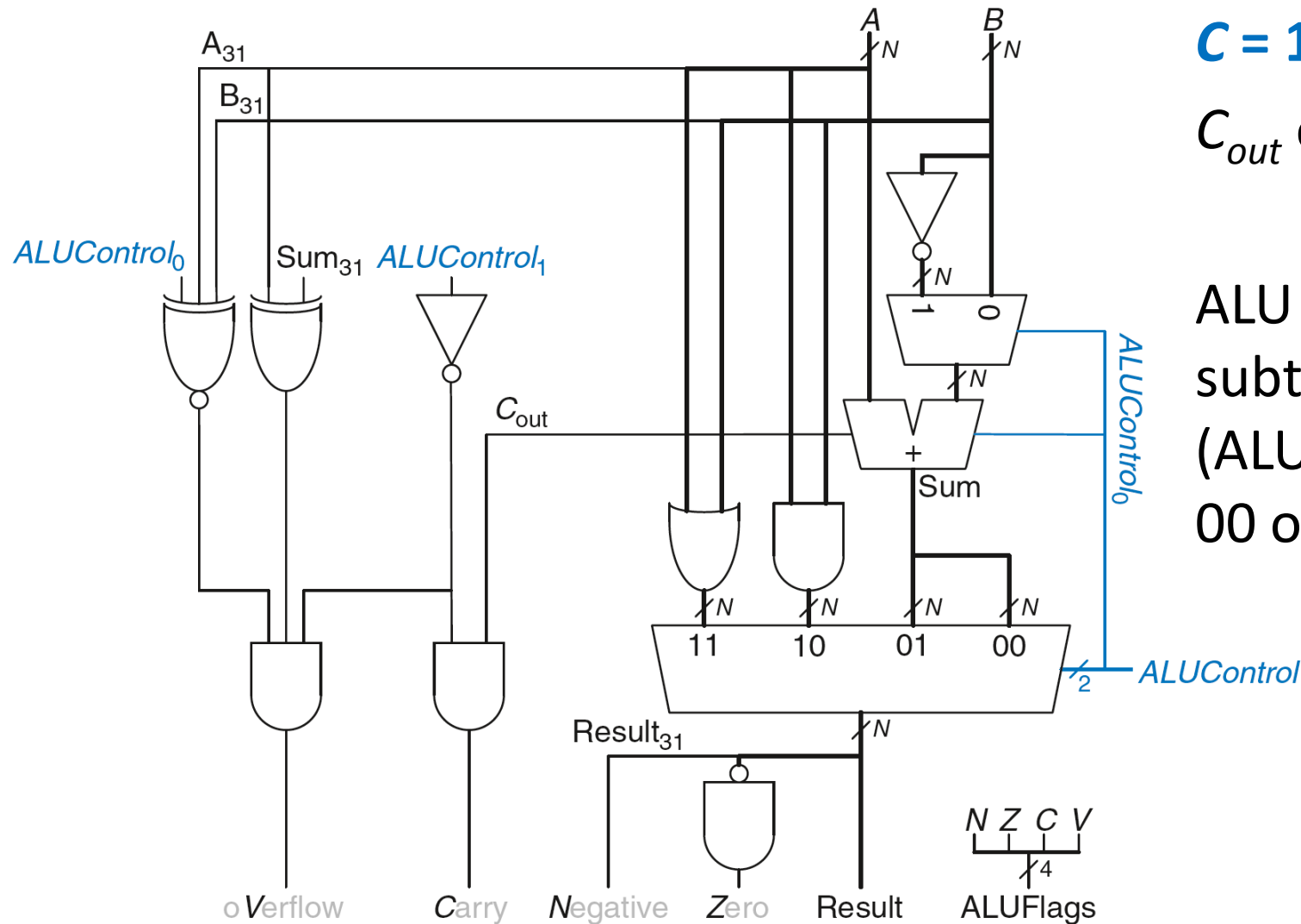
ALU with Status Flags: Zero



Z = 1 if:

all of the bits of *Result* are 0

ALU with Status Flags: Carry



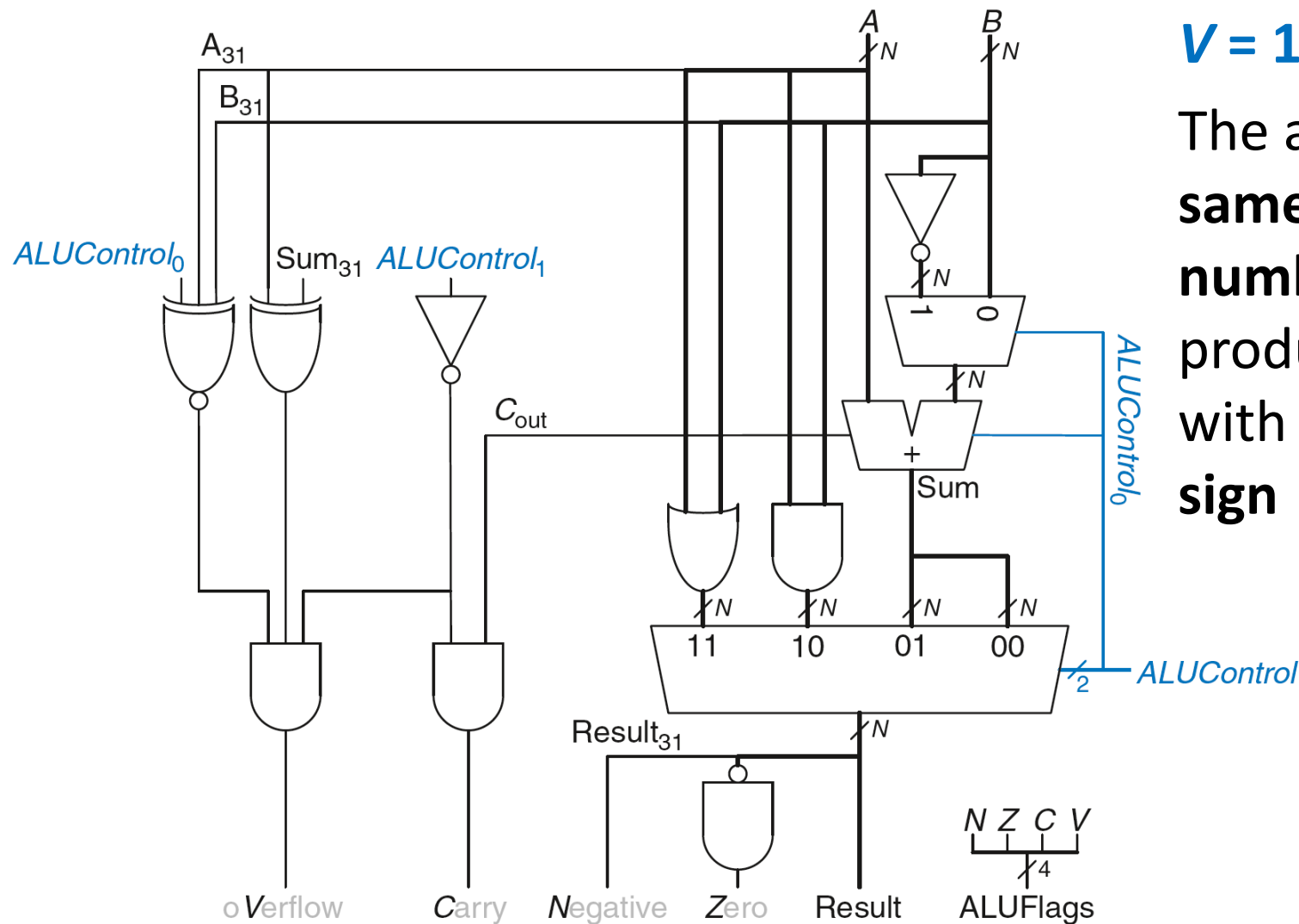
C = 1 if:

C_{out} of Adder is 1

AND

ALU is adding or subtracting
(ALUControl is 00 or 01)

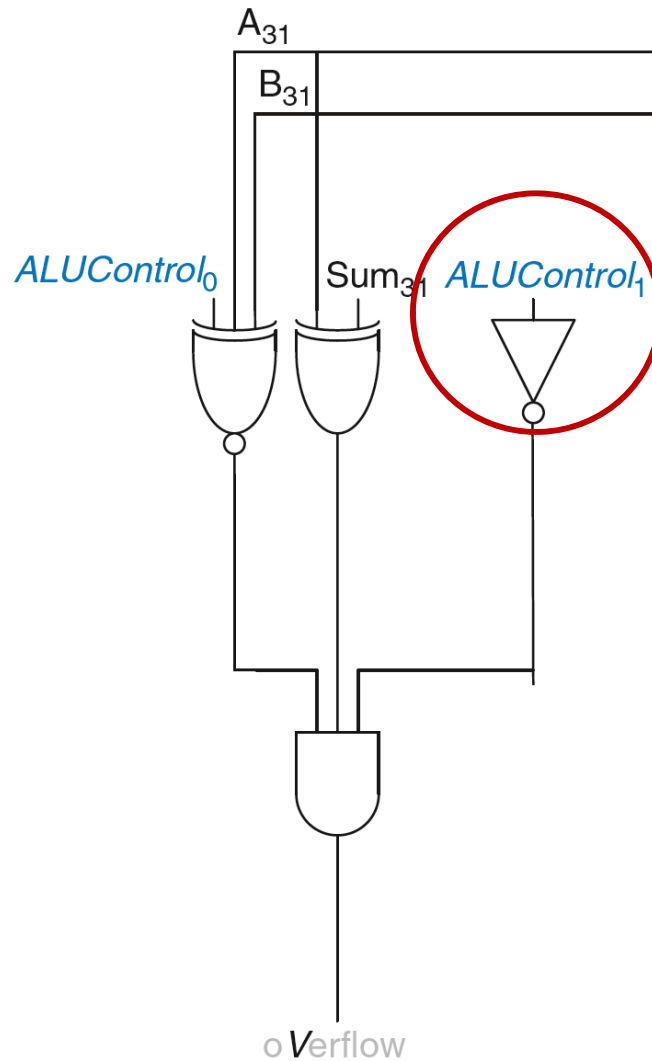
ALU with Status Flags: oVerflow



$V = 1$ if:

The addition of 2 **same-signed numbers** produces a result with the **opposite sign**

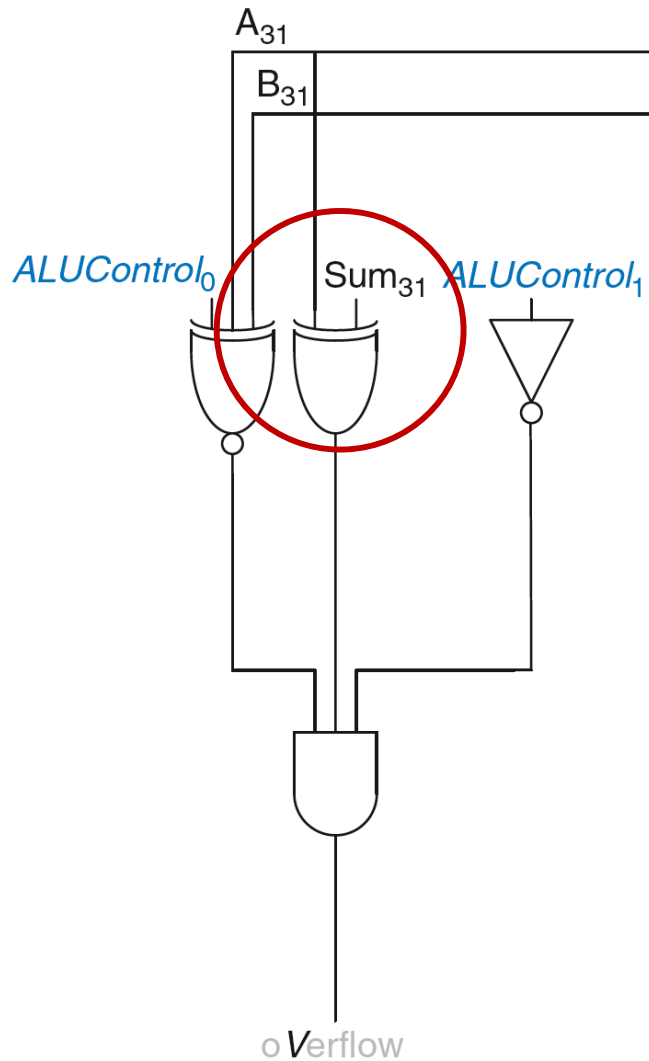
ALU with Status Flags: oVerflow



$V = 1$ if:

ALU is performing
addition or subtraction
($ALUControl_1 = 0$)

ALU with Status Flags: oVerflow



$V = 1$ if:

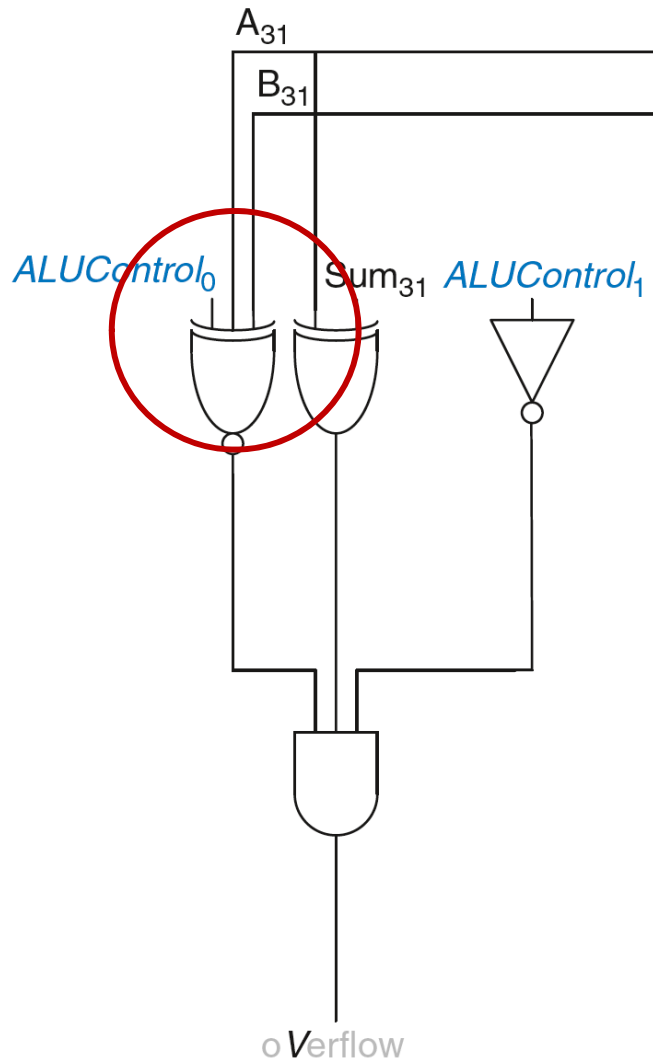
ALU is performing addition or subtraction

($ALUControl_1 = 0$)

AND

A and Sum have opposite signs

ALU with Status Flags: oVerflow



V = 1 if:

ALU is performing addition or subtraction

($ALUControl_1 = 0$)

AND

A and Sum have opposite signs

AND

A and B have same signs upon addition ($ALUControl_0 = 0$)

OR

A and B have different signs upon subtraction

($ALUControl_0 = 1$)

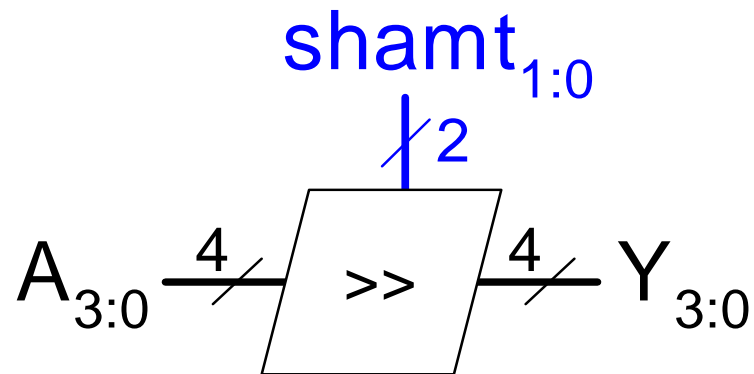
Shifters

- **Logical shifter:** shifts value to left or right and fills empty spaces with 0's
 - Ex: $11001 \gg 2 =$
 - Ex: $11001 \ll 2 =$
- **Arithmetic shifter:** same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb).
 - Ex: $11001 \ggg 2 =$
 - Ex: $11001 \lll 2 =$
- **Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end
 - Ex: $11001 \text{ ROR } 2 =$
 - Ex: $11001 \text{ ROL } 2 =$

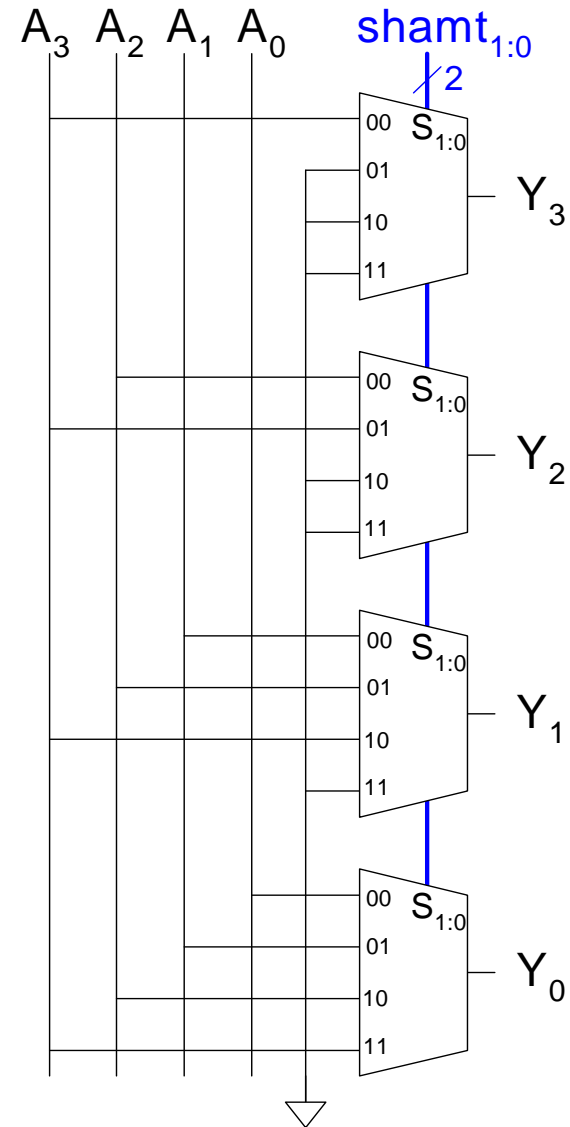
Shifters

- **Logical shifter:**
 - Ex: 11001 >> 2 = 00110
 - Ex: 11001 << 2 = 00100
- **Arithmetic shifter:**
 - Ex: 11001 >>> 2 = 11110
 - Ex: 11001 <<< 2 = 00100
- **Rotator:**
 - Ex: 11001 ROR 2 = 01110
 - Ex: 11001 ROL 2 = 00111

Shifter Design

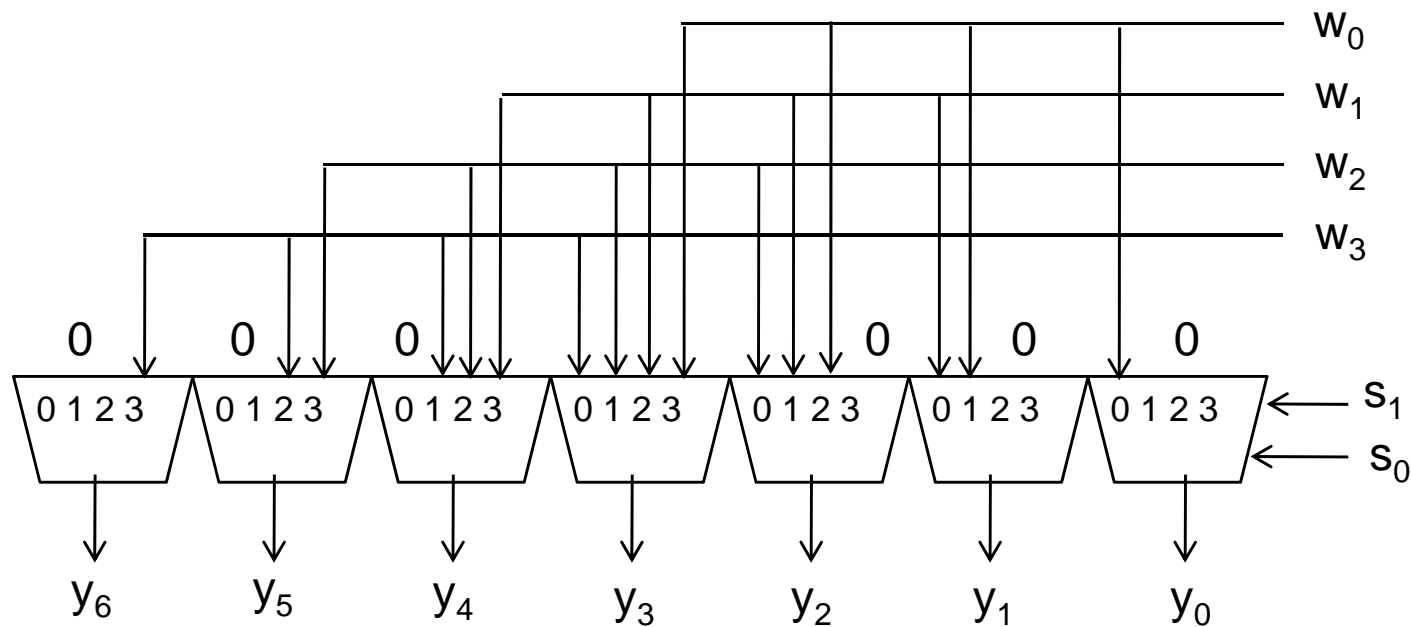


$S_1 S_0$	$Y_3 Y_2 Y_1 Y_0$
0 0	$A_3 A_2 A_1 A_0$
0 1	0 $A_3 A_2 A_1$
1 0	0 0 $A_3 A_2$
1 1	0 0 0 A_3

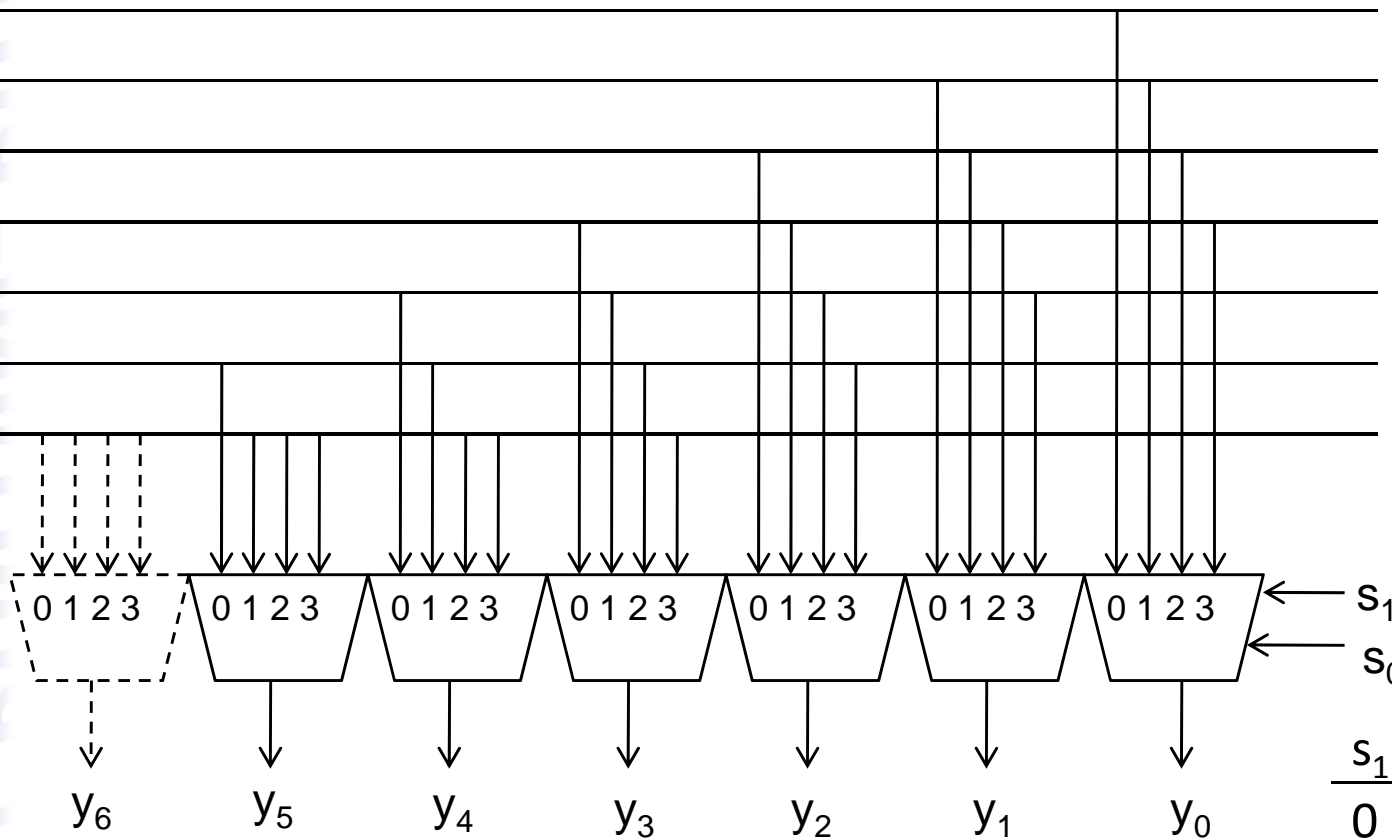


Logical/Arithmetic Shift to the Left

s_1	s_0	y_6	y_5	y_4	y_3	y_2	y_1	y_0
0	0	0	0	0	w_3	w_2	w_1	w_0
0	1	0	0	w_3	w_2	w_1	w_0	0
1	0	0	w_3	w_2	w_1	w_0	0	0
1	1	w_3	w_2	w_1	w_0	0	0	0



Arithmetic Shift to the Right



s_1	s_0	y_5	y_4	y_3	y_2	y_1	y_0
0	0	w_5	w_4	w_3	w_2	w_1	w_0
0	1	w_6	w_5	w_4	w_3	w_2	w_1
1	0	w_6	w_6	w_5	w_4	w_3	w_2
1	1	w_6	w_6	w_6	w_5	w_4	w_3

Shifters as Multipliers, Dividers

- $A \ll N = A \times 2^N$
 - **Example:** $00001 \ll 2 = 00100$ ($1 \times 2^2 = 4$)
 - **Example:** $11101 \ll 2 = 10100$ ($-3 \times 2^2 = -12$)

Compare with multiplication by 10 in base 10:

$63 * 10 = 630$, $-63 * 10 = -630$ etc.

- $A \ggg N = A \div 2^N$
 - **Example:** $01000 \ggg 2 = 00010$ ($8 \div 2^2 = 2$)
 - **Example:** $10000 \ggg 2 = 11100$ ($-16 \div 2^2 = -4$)

Multiplication by 2^n

- A multiplication by 2^n can be done by shifting all bits n positions to the left and filling in with zeros
- Calculate $13 * 8$:
 - $13 * 8$ can be calculated by shifting (01011) three bits to the left
 - Result: 01011000 corresponds to $(104)_{10}$
 - Note that you need more bits to represent the result!



Multipliers

- **Partial products** formed by multiplying a single digit of the multiplier with multiplicand
- **Shifted** partial products **summed** to form result

Decimal

$$\begin{array}{r}
 230 \\
 \times 42 \\
 \hline
 460 \\
 + 920 \\
 \hline
 9660
 \end{array}$$

multiplicand
 multiplier
 partial
 products

result

$$230 \times 42 = 9660$$

Binary

$$\begin{array}{r}
 0101 \\
 \times 0111 \\
 \hline
 0101 \\
 0101 \\
 0101 \\
 + 0000 \\
 \hline
 0100011
 \end{array}$$

$$5 \times 7 = 35$$

Multiplication of unsigned integers

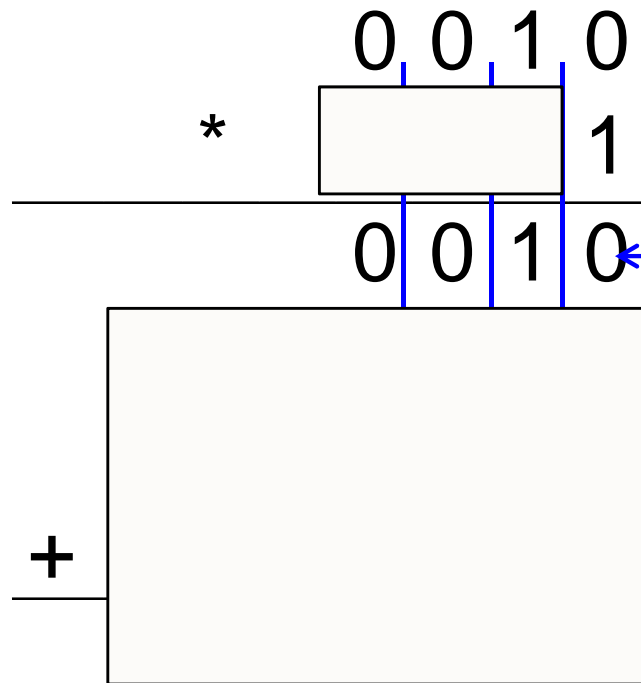
$$\begin{array}{r}
 0010 \\
 * 1011 \\
 \hline
 0010 \\
 0010 \\
 0000 \\
 + 0010 \\
 \hline
 0010110
 \end{array}$$

2 Multiplicand

11 Multiplier

22 Product

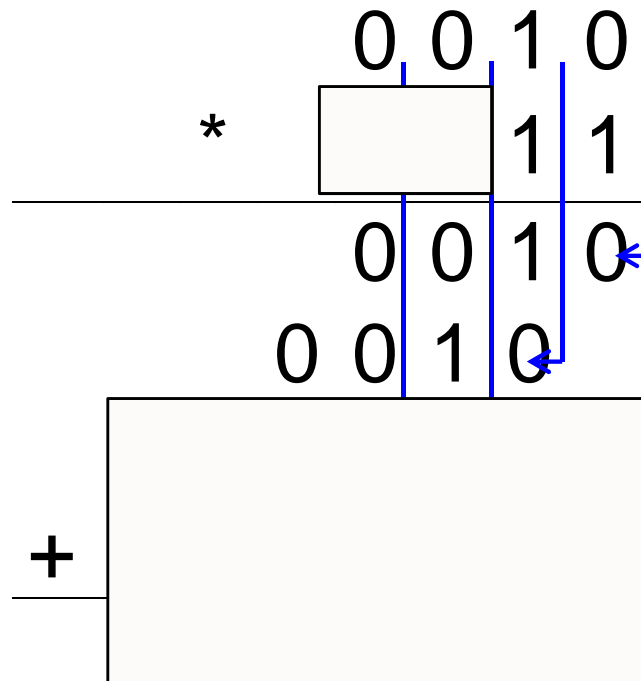
Multiplication of unsigned integers



2 Multiplicand
11 Multiplier

22 Product

Multiplication of unsigned integers

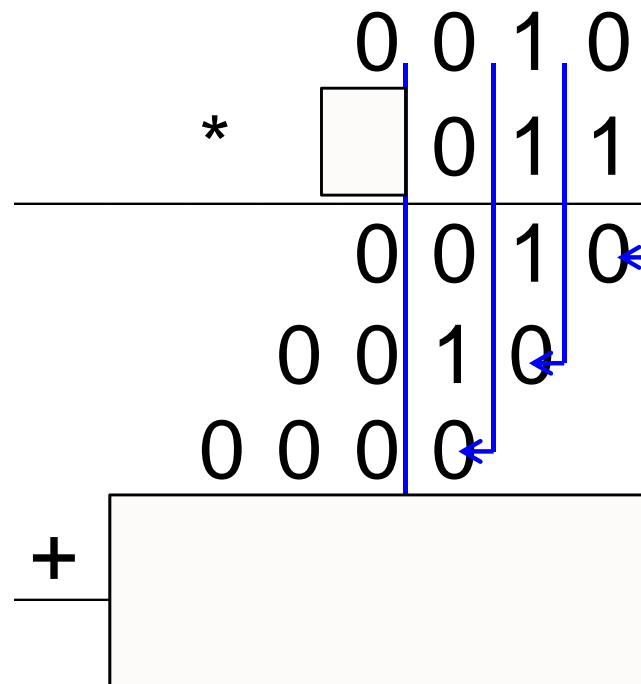


2 Multiplicand

11 Multiplier

22 Product

Multiplication of unsigned integers



2 Multiplicand

11 Multiplier

22 Product

Multiplication of unsigned integers

$$\begin{array}{r}
 \\
 \\
 \\
 \\
 \\
 + \\
 \hline
 0010110
 \end{array}$$

The diagram illustrates the multiplication of the multiplicand 0010 (2) by the multiplier 1011 (11). Blue arrows indicate the shifting of partial products: the first partial product is 0010, shifted 0 positions; the second is 0010, shifted 1 position; the third is 0010, shifted 2 positions; and the fourth is 0010, shifted 3 positions. The final product is 0010110 (22).

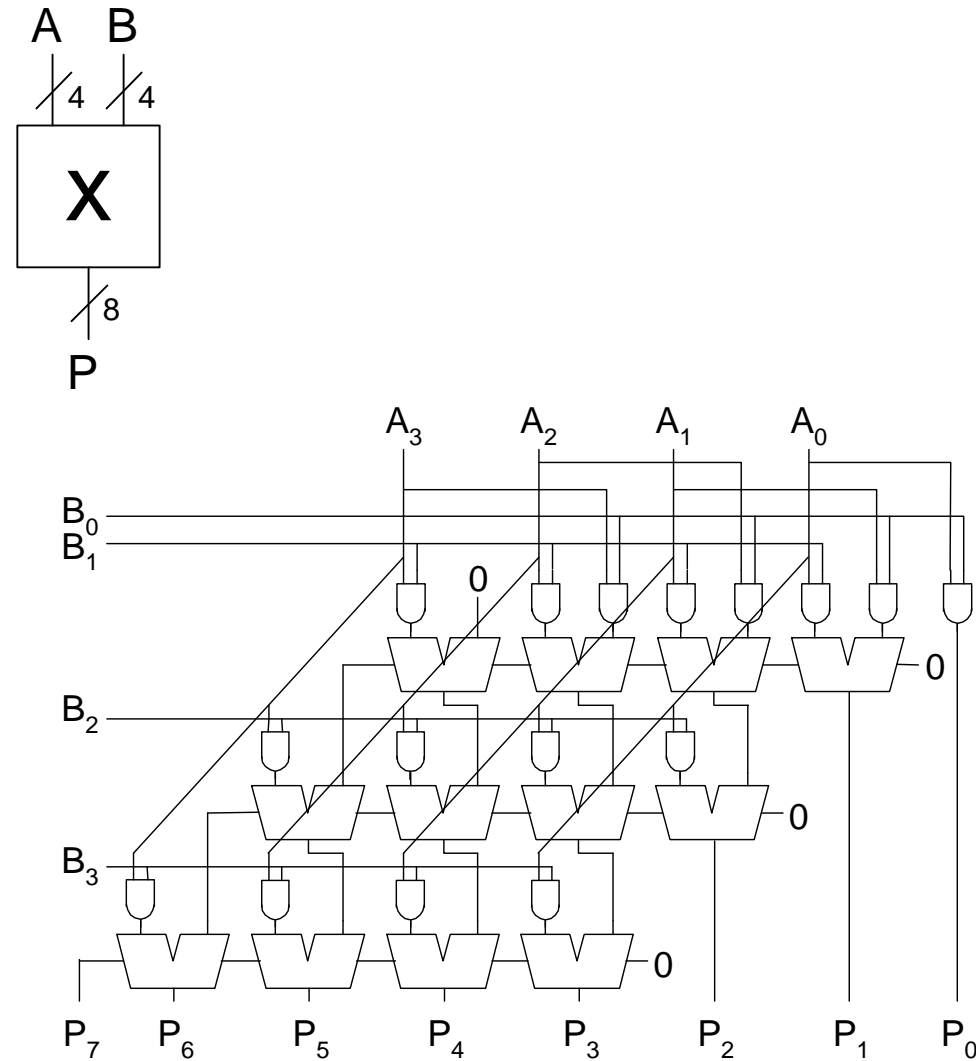
2 Multiplicand

11 Multiplier

22 Product

4 x 4 Multiplier

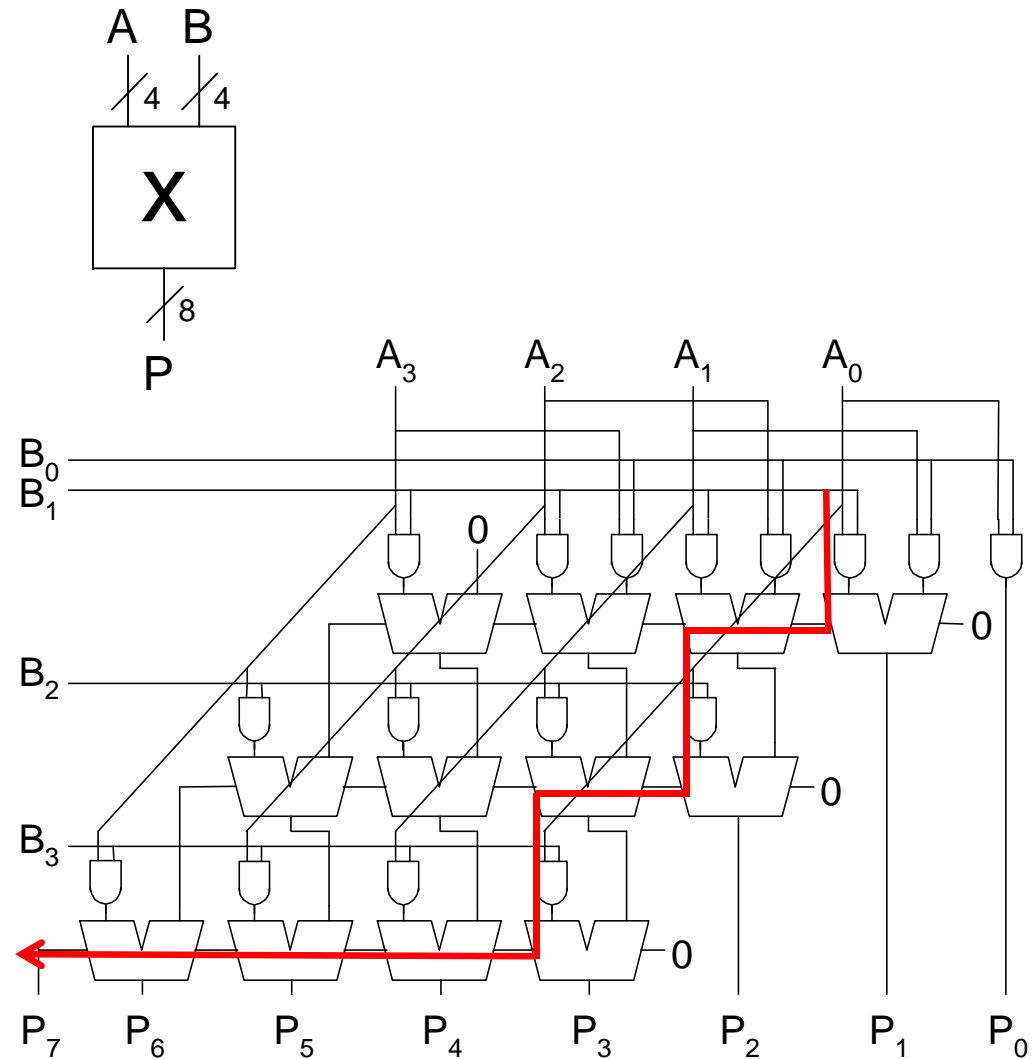
$$\begin{array}{r}
 \begin{array}{cccc}
 & A_3 & A_2 & A_1 & A_0 \\
 \times & B_3 & B_2 & B_1 & B_0 \\
 \hline
 & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\
 & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 \\
 & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 \\
 + & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 \\
 \hline
 P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
 \end{array}
 \end{array}$$



Quick Question-Critical Path

$$T_{MUL} = 8 * T_{FA} + AND(2)$$

				A ₃	A ₂	A ₁	A ₀	
	x			B ₃	B ₂	B ₁	B ₀	
				A ₃ B ₀	A ₂ B ₀	A ₁ B ₀	A ₀ B ₀	
				A ₃ B ₁	A ₂ B ₁	A ₁ B ₁	A ₀ B ₁	
				A ₃ B ₂	A ₂ B ₂	A ₁ B ₂	A ₀ B ₂	
+				A ₃ B ₃	A ₂ B ₃	A ₁ B ₃	A ₀ B ₃	
	P ₇	P ₆	P ₅	P ₄	P ₃	P ₂	P ₁	P ₀



Multiplication of signed integers

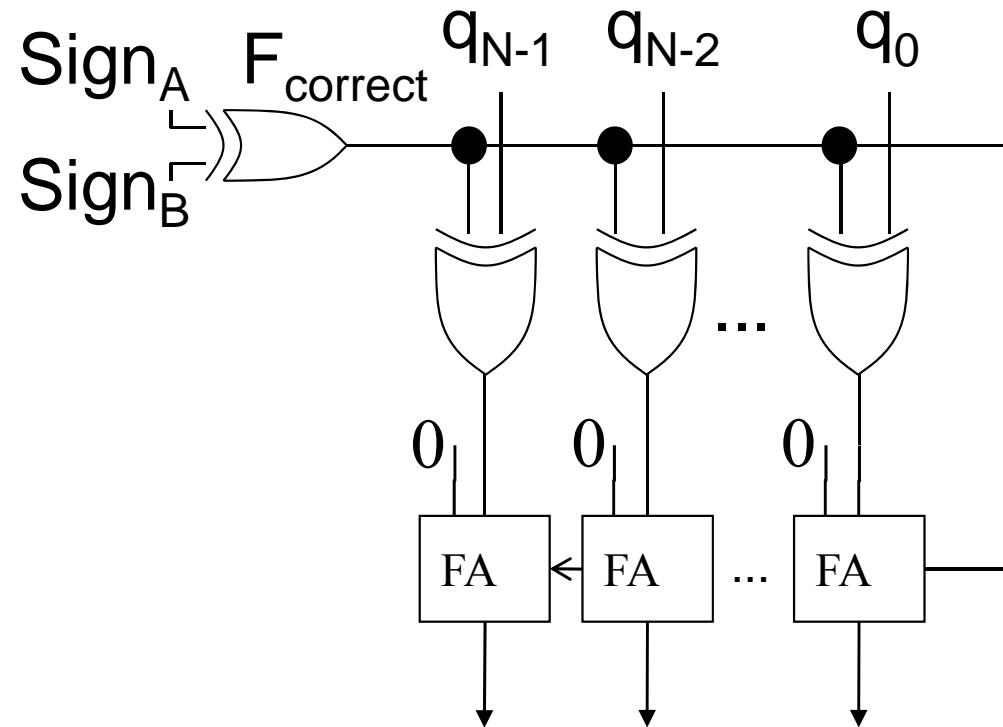
- Use only positive numbers in the multiplication
 - Convert negative numbers to positive numbers by taking 2's complement
 - Keep track of the result's sign
 - $(+ +) \Rightarrow +$; $(+, -) \Rightarrow -$; $(-, +) \Rightarrow -$; $(-, -) \Rightarrow +$
 - Take 2's complement from the result to adjust the sign if necessary

Sign Adjustment

		Sign _A	
		0	1
Sign _B	0	0	1
	1	1	0

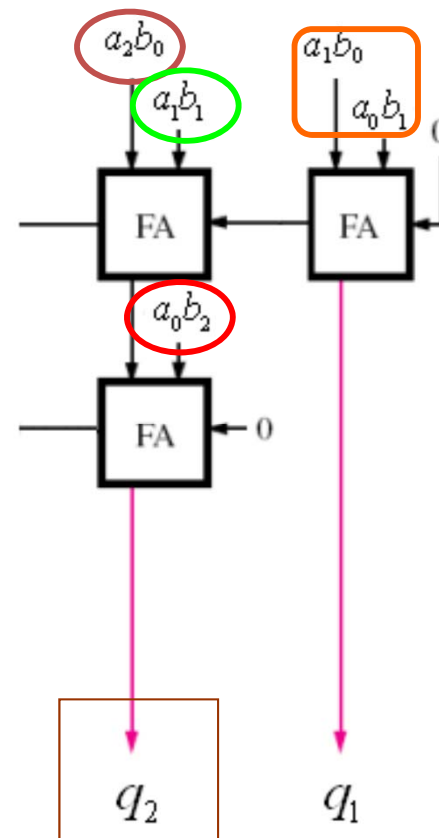
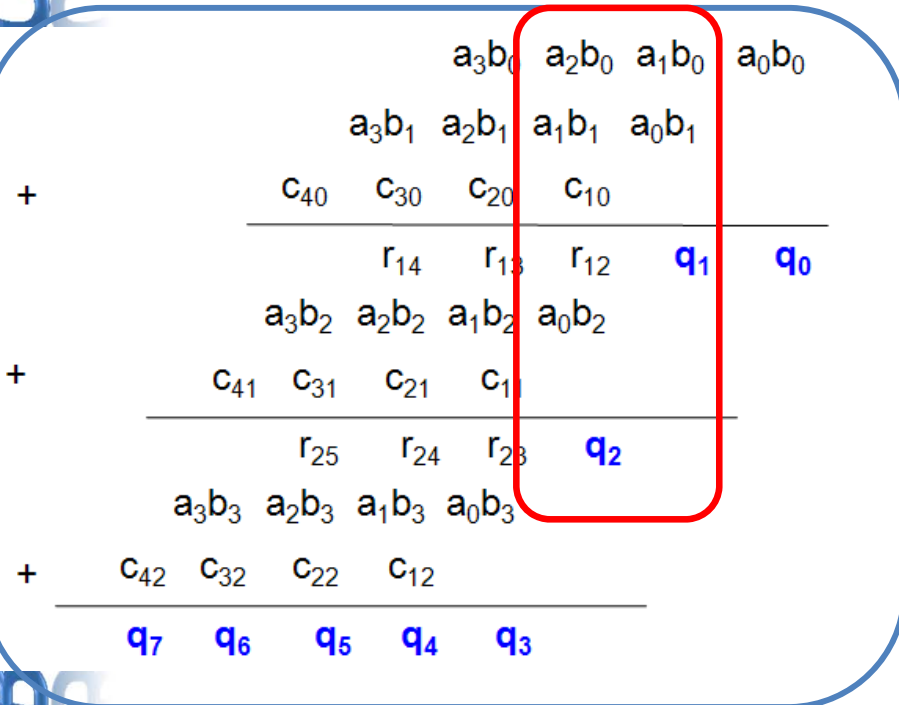
$$F_{\text{correct}} = \text{Sign}_A \text{ xor } \text{Sign}_B$$

The correction is done by inverting the bits, and then add 1



**2's complement of product
(when correction is needed)**

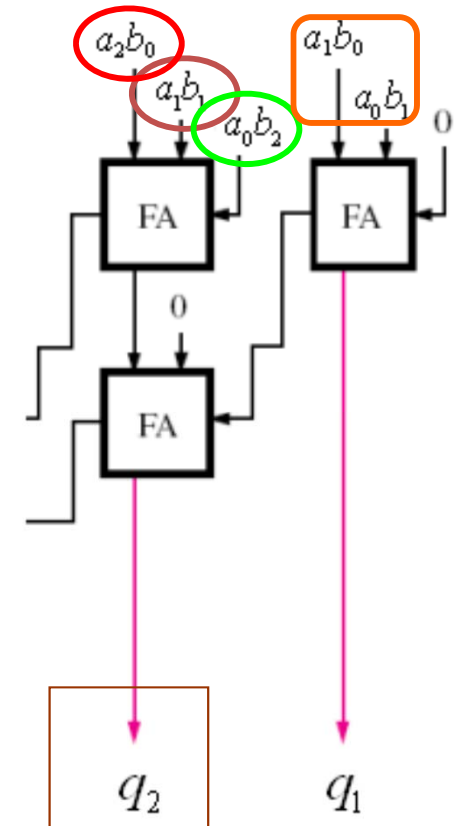
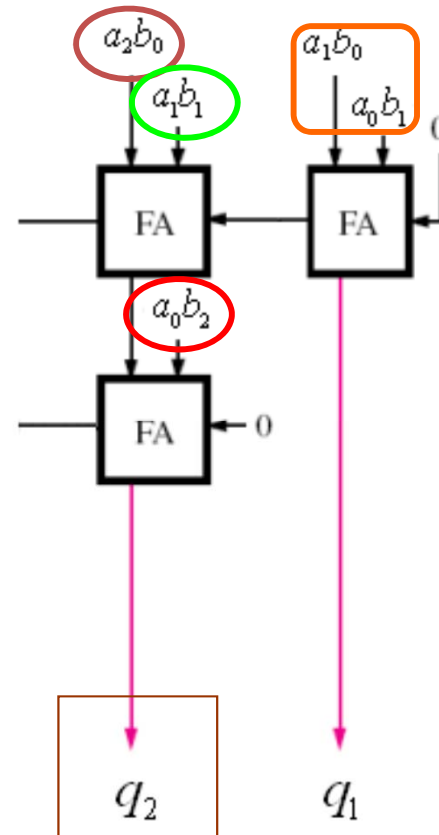
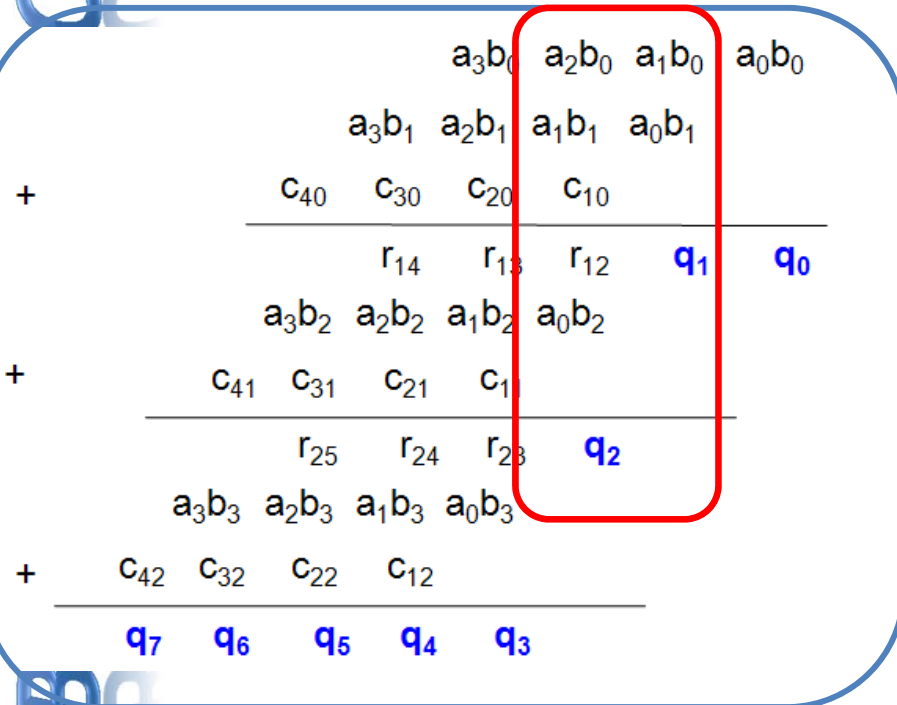
Carry-Save Multiplier



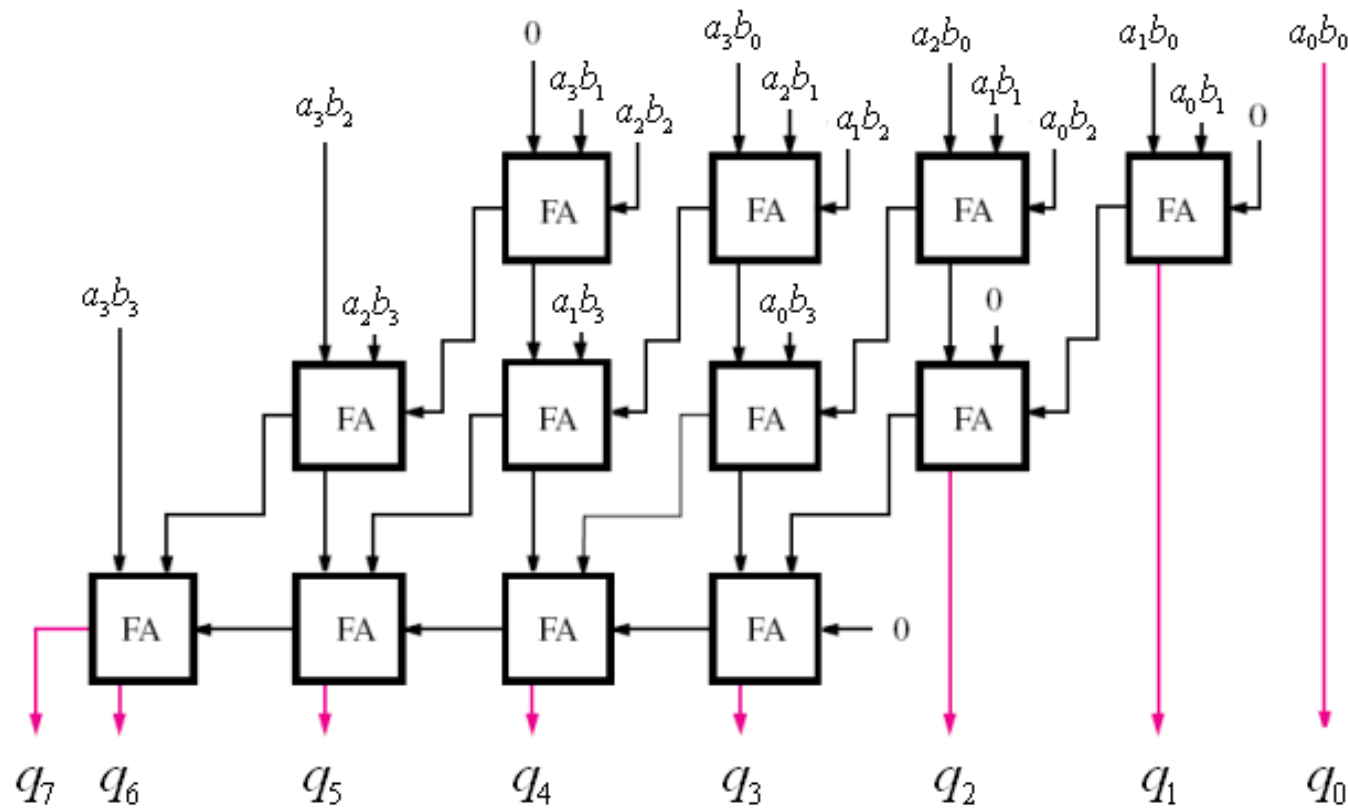
Carry-Save Multiplier

BOOKS

DIGITAL



Carry-Save Multiplier

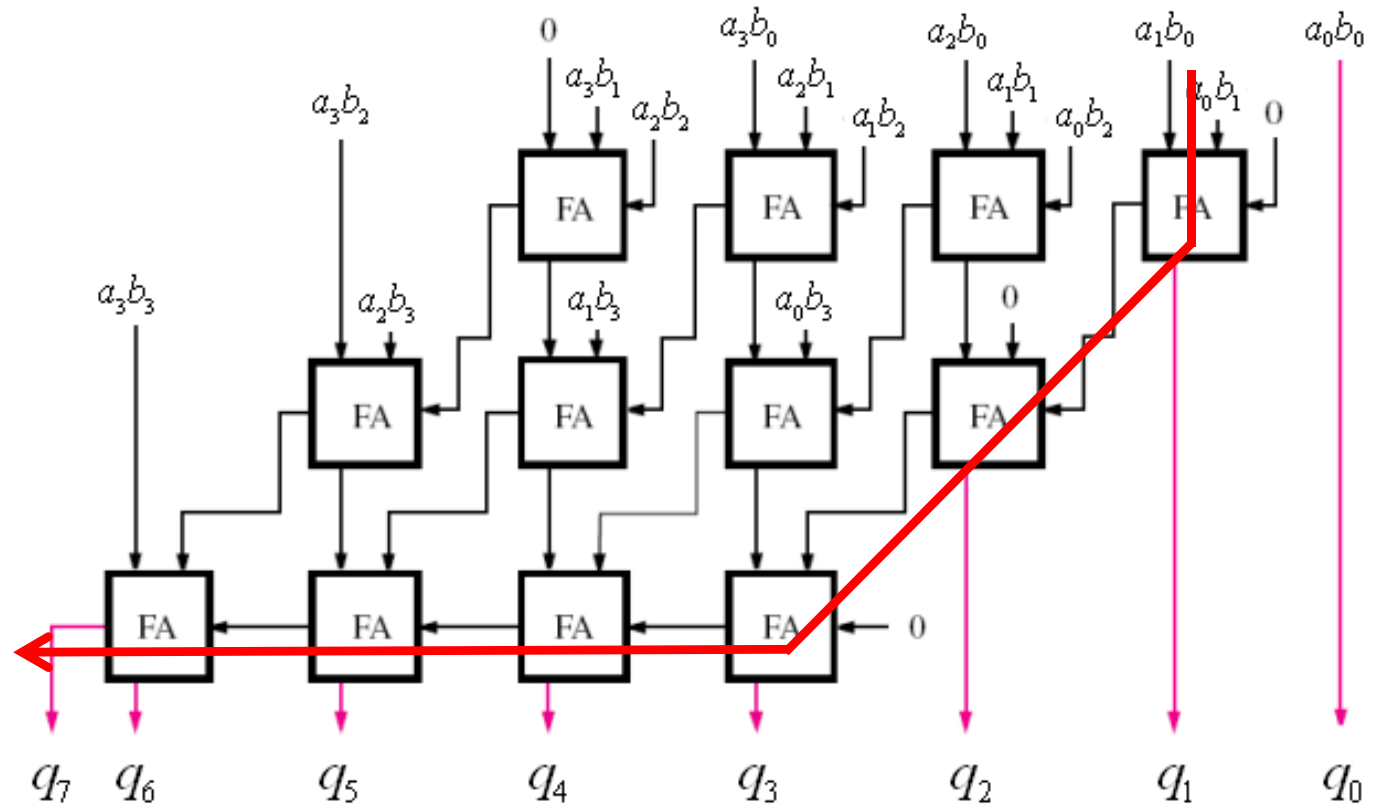


Quick Question-Critical Path

$$T_{\text{MUL}} \sim (2 \cdot N - 2) \cdot T_{\text{FA}} + \text{AND}(2)$$

Reduces the delay because the carry is fed directly to the next row!

Carry is delayed through 6 stages!



Dividers

$$A/B = Q + R$$

Decimal Example: 2584/15 = 172 R4

Dividers

$$A/B = Q + R$$

Decimal Example: $2584/15 = 172 \text{ R}4$

Long-Hand:

		172	← Q Quotient
Divisor	B →	15 $\overline{) 2584}$	← A Dividend
		-15	
		<hr/> 108	
		-105	
		<hr/> 34	
		-30	
		<hr/> 4	← R Remainder

Dividers

$$A/B = Q + R$$

Decimal Example: 2584/15 = 172 R4

Long-Hand:

$$\begin{array}{r}
 172 \text{ R}4 \\
 15 \overline{) 2584} \\
 \underline{-15} \\
 108 \\
 \underline{-105} \\
 34 \\
 \underline{-30} \\
 4
 \end{array}$$

Long-Hand Revisited:

$ \begin{array}{r} 0002 \\ -15 \\ \hline -13 \end{array} $	$ \begin{array}{r} 0 \\ 3 \overline{) 210} \end{array} $
$ \begin{array}{r} 0025 \\ -15 \\ \hline 10 \end{array} $	$ \begin{array}{r} 01 \\ 3 \overline{) 210} \end{array} $
$ \begin{array}{r} 0108 \\ -105 \\ \hline 3 \end{array} $	$ \begin{array}{r} 017 \\ 3 \overline{) 210} \end{array} $
$ \begin{array}{r} 0034 \\ -30 \\ \hline 4 \end{array} $	$ \begin{array}{r} 0172 \\ 3 \overline{) 210} \end{array} $

Dividers

$$A/B = Q + R$$

Decimal: 2584/15 = 172 R4 **Binary:** 1101/10 = 0110 R1

$$\begin{array}{r} 0002 \\ - 15 \\ \hline -13 \end{array}$$

$$\begin{array}{r} 0 \\ \hline 3 \end{array} \begin{array}{r} \\ 2 \end{array} \begin{array}{r} \\ 1 \end{array} \begin{array}{r} \\ 0 \end{array}$$

$$\begin{array}{r} 0025 \\ - 15 \\ \hline 10 \end{array}$$

$$\begin{array}{r} 0 1 \\ \hline 3 \end{array} \begin{array}{r} \\ 2 \end{array} \begin{array}{r} \\ 1 \end{array} \begin{array}{r} \\ 0 \end{array}$$

$$\begin{array}{r} 0108 \\ - 105 \\ \hline 3 \end{array}$$

$$\begin{array}{r} 0 1 7 \\ \hline 3 \end{array} \begin{array}{r} \\ 2 \end{array} \begin{array}{r} \\ 1 \end{array} \begin{array}{r} \\ 0 \end{array}$$

$$\begin{array}{r} 0034 \\ - 30 \\ \hline 4 \end{array}$$

$$\begin{array}{r} 0 1 7 2 \\ \hline 3 \end{array} \begin{array}{r} \\ 2 \end{array} \begin{array}{r} \\ 1 \end{array} \begin{array}{r} \\ 0 \end{array}$$

$$\begin{array}{r} 0001 \\ - 0010 \\ \hline 1111 \end{array}$$

$$\begin{array}{r} 0 \\ \hline 3 \end{array} \begin{array}{r} \\ 2 \end{array} \begin{array}{r} \\ 1 \end{array} \begin{array}{r} \\ 0 \end{array}$$

$$\begin{array}{r} 0011 \\ - 0010 \\ \hline 0001 \end{array}$$

$$\begin{array}{r} 0 1 \\ \hline 3 \end{array} \begin{array}{r} \\ 2 \end{array} \begin{array}{r} \\ 1 \end{array} \begin{array}{r} \\ 0 \end{array}$$

$$\begin{array}{r} 0010 \\ - 0010 \\ \hline 0000 \end{array}$$

$$\begin{array}{r} 0 1 1 \\ \hline 3 \end{array} \begin{array}{r} \\ 2 \end{array} \begin{array}{r} \\ 1 \end{array} \begin{array}{r} \\ 0 \end{array}$$

$$\begin{array}{r} 0001 \\ - 0010 \\ \hline 1111 \end{array}$$

$$\begin{array}{r} 0 1 1 0 \\ \hline 3 \end{array} \begin{array}{r} \\ 2 \end{array} \begin{array}{r} \\ 1 \end{array} \begin{array}{r} \\ 0 \end{array} \text{ R1}$$



Divider Algorithm

$$A/B = Q + R/B$$

$$R' = 0$$

for $i = N-1$ to 0

$$R = \{R' \ll 1, A_i\}$$

$$D = R - B$$

if $D < 0$, $Q_i = 0$; $R' = R$

else $Q_i = 1$; $R' = D$

$$R' = R$$

Binary: $1101/10 = 0110 \text{ R}1$

$$\begin{array}{r} 0001 \\ -0010 \\ \hline 1111 \end{array} \quad \begin{array}{r} 0 \\ \hline 3 \end{array} \begin{array}{r} 2 \\ \hline 1 \end{array} \begin{array}{r} 1 \\ \hline 0 \end{array}$$

$$\begin{array}{r} 0011 \\ -0010 \\ \hline 0001 \end{array} \quad \begin{array}{r} 0 \\ \hline 3 \end{array} \begin{array}{r} 1 \\ \hline 2 \end{array} \begin{array}{r} 1 \\ \hline 1 \end{array} \begin{array}{r} 0 \\ \hline 0 \end{array}$$

$$\begin{array}{r} 0010 \\ -0010 \\ \hline 0000 \end{array} \quad \begin{array}{r} 0 \\ \hline 3 \end{array} \begin{array}{r} 1 \\ \hline 2 \end{array} \begin{array}{r} 1 \\ \hline 1 \end{array} \begin{array}{r} 0 \\ \hline 0 \end{array}$$

$$\begin{array}{r} 0001 \\ -0010 \\ \hline 1111 \end{array} \quad \begin{array}{r} 0 \\ \hline 3 \end{array} \begin{array}{r} 1 \\ \hline 2 \end{array} \begin{array}{r} 1 \\ \hline 1 \end{array} \begin{array}{r} 0 \\ \hline 0 \end{array} \text{ R}1$$

Division Example

$$\begin{array}{r}
 \text{Quotient} \\
 0101 \\
 \text{Divisor } 10 \overline{) 1011} \text{ Dividend}
 \end{array}$$

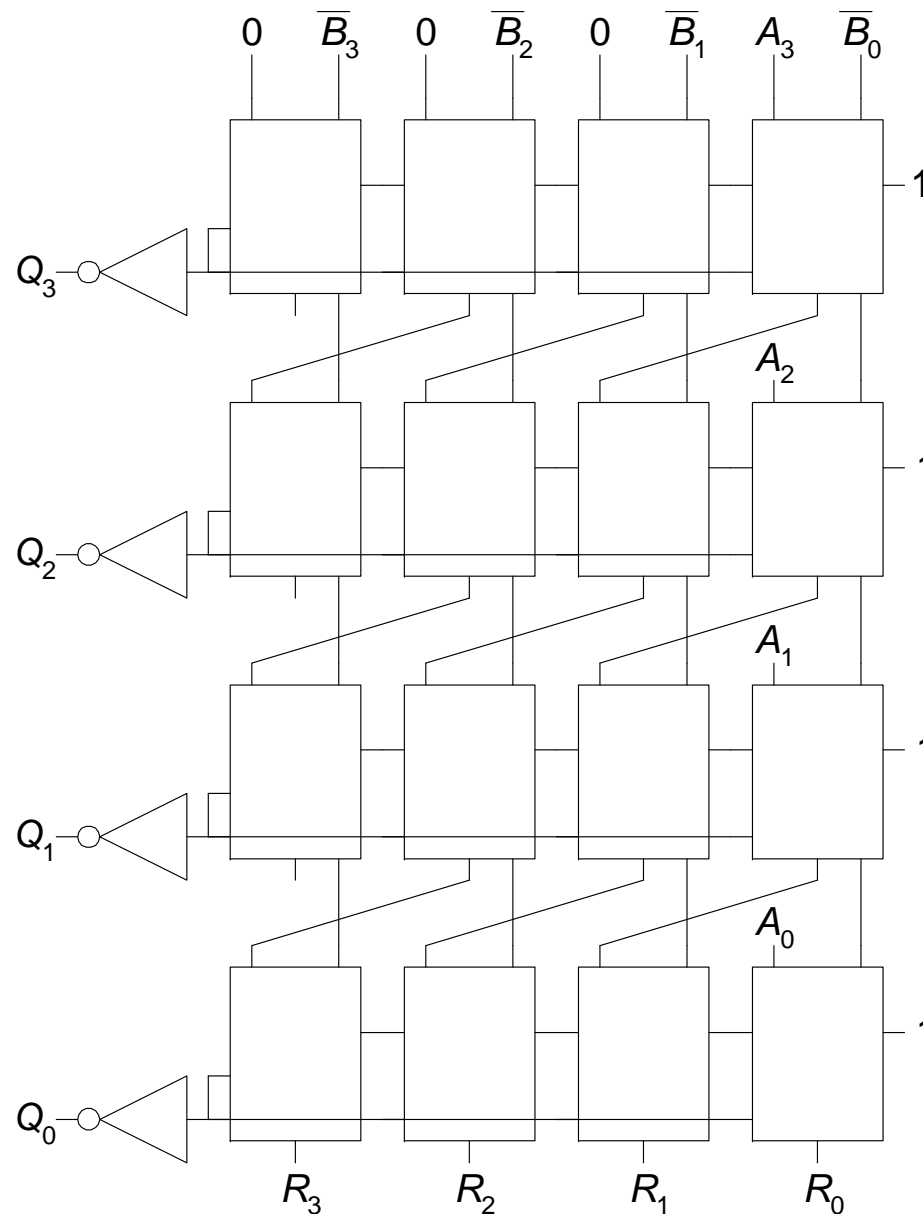
Division Example

$$\begin{array}{r}
 0101 \\
 10 \overline{) 01011} \\
 \underline{- 00} \\
 10 \\
 \underline{- 10} \\
 001 \\
 \underline{- 00} \\
 011 \\
 \underline{- 10} \\
 1
 \end{array}$$

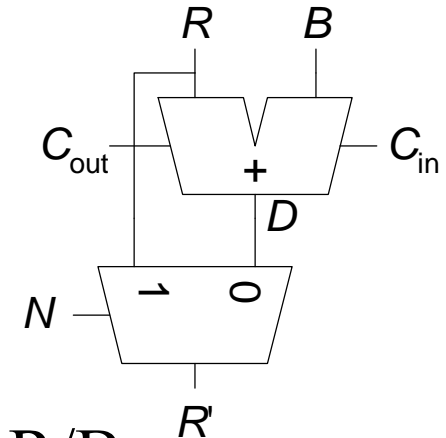
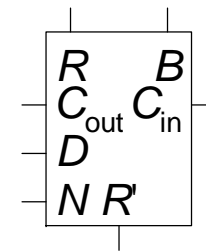
$$11/2 = 5$$

$$\text{Remainder} = 1$$

4 x 4 Divider



Legend



$$A/B = Q + R/B$$

Algorithm:

$$R' = 0$$

for $i = N-1$ to 0

$$R = \{R' \ll 1, A_i\}$$

$$D = R - B$$

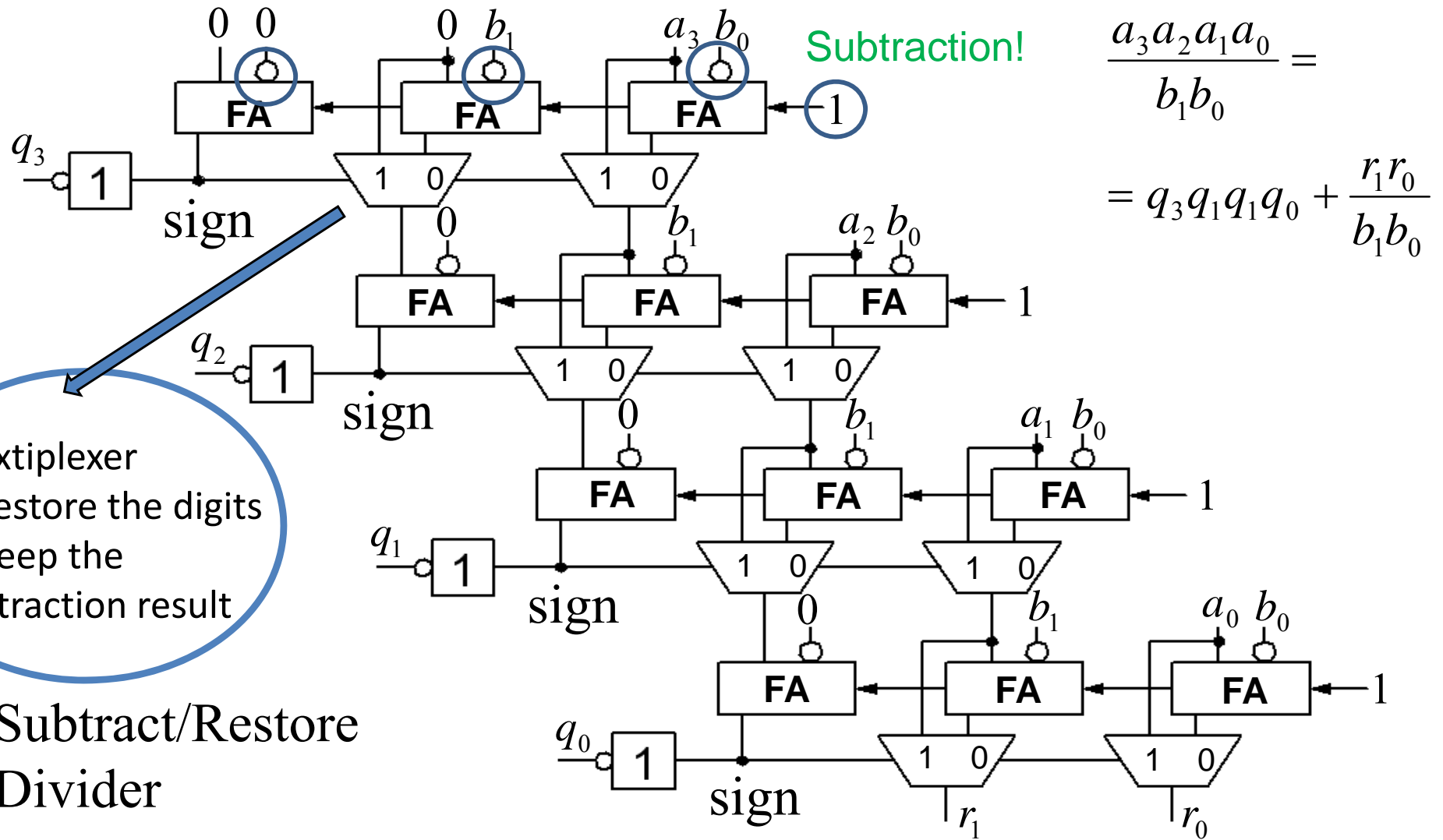
if $D < 0$, $Q_i = 0$, $R' = R$

else $Q_i = 1$, $R' = D$

$$R' = R$$



4 x 4 Divider



Division of signed integers

- Use only positive numbers in the division
 - Convert negative numbers to positive numbers by taking 2's complement
 - Keep track of the result's sign
 - $(+ +) \Rightarrow +$; $(+, -) \Rightarrow -$; $(-, +) \Rightarrow -$; $(-, -) \Rightarrow +$
 - Take 2's complement from the result to adjust the sign if necessary

Number Systems

- Numbers we can represent using binary representations
 - **Positive numbers**
 - Unsigned binary
 - **Negative numbers**
 - Two's complement
 - Sign/magnitude numbers
- What about **fractions**?



Numbers with Fractions

- Two common notations:
 - **Fixed-point:** binary point fixed
 - **Floating-point:** binary point floats to the right of the most significant 1



Fixed-Point Numbers

- 6.75 using 4 integer bits and 4 fraction bits:

01101100

0110.1100

$$2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$$

- Binary point is implied
- The number of integer and fraction bits must be agreed upon beforehand



Fixed-Point Number Example

- Represent 7.5_{10} using 4 integer bits and 4 fraction bits.

Fixed-Point Number Example

- Represent 7.5_{10} using 4 integer bits and 4 fraction bits.

01111000

Signed Fixed-Point Numbers

- **Representations:**
 - Sign/magnitude
 - Two's complement
- **Example:** Represent -7.5_{10} using 4 integer and 4 fraction bits
 - **Sign/magnitude:**
 - **Two's complement:**

Signed Fixed-Point Numbers

- **Representations:**
 - Sign/magnitude
 - Two's complement
- **Example:** Represent -7.5_{10} using 4 integer and 4 fraction bits

- **Sign/magnitude:**

11111000

- **Two's complement:**

1. +7.5: 01111000

2. Invert bits: 10000111

3. Add 1 to lsb: + 1

10001000



Operations on Fixed-Point Numbers

- Logic circuits which deal with fixed-point numbers are essentially the same as those used for integers
- Fixed-point numbers have a range that is limited by the significant digits used to represent the number
- In scientific computations it is often necessary to deal with very large numbers
 - (5.213×10^{43})

Floating-Point Numbers

- Binary point floats to the right of the most significant 1
- Similar to decimal scientific notation

- For example, write 273_{10} in scientific notation:

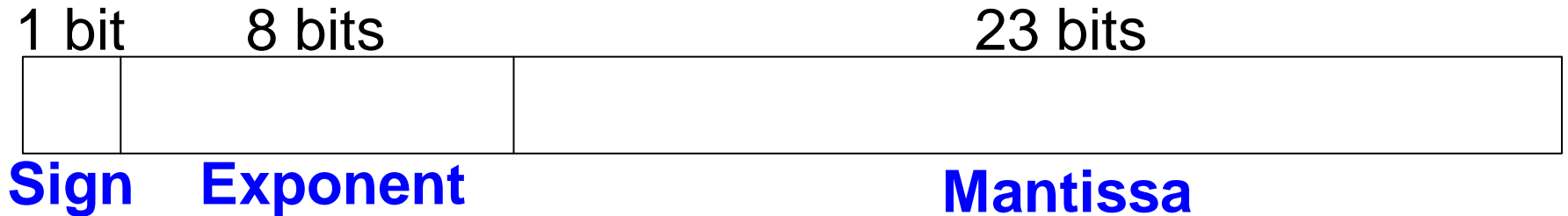
$$273 = 2.73 \times 10^2$$

- In general, a number is written in scientific notation as:

$$\pm M \times B^E$$

- M = mantissa
- B = base
- E = exponent
- In the example, $M = 2.73$, $B = 10$, and $E = 2$

Floating-Point Numbers



- **Example:** represent the value 228_{10} using a 32-bit floating point representation

We show three versions –final version is called the **IEEE 754 floating-point standard**



Floating-Point Representation 1

1. Convert decimal to binary (**don't reverse steps 1 & 2!**):

$$228_{10} = 11100100_2$$

2. Write the number in “binary scientific notation”:

$$11100100_2 = 1.11001_2 \times 2^7$$

3. Fill in each field of the 32-bit floating point number:

- The sign bit is positive (0)
- The 8 exponent bits represent the value 7
- The remaining 23 bits are the mantissa

1 bit	8 bits	23 bits
0	00000111	11 1001 0000 0000 0000 0000
Sign	Exponent	Mantissa



Floating-Point Representation 2

- First bit of the mantissa is always 1:
 - $228_{10} = 11100100_2 = \mathbf{1.11001} \times 2^7$
- So, no need to store it: *implicit leading 1*
- Store just fraction bits in 23-bit field

1 bit	8 bits	23 bits
0	00000111	110 0100 0000 0000 0000 0000
Sign	Exponent	Fraction

Floating-Point Representation 3

- *Biased exponent*: bias = 127 (01111111_2)
 - Biased exponent = bias + exponent
 - Exponent of 7 is stored as:

$$127 + 7 = 134 = 0x10000110_2$$
- The **IEEE 754 32-bit floating-point representation** of 228_{10}

1 bit	8 bits	23 bits
0	10000110	110 0100 0000 0000 0000 0000
Sign	Biased Exponent	Fraction

in hexadecimal: **0x43640000**



Floating-Point Example

Write -58.25_{10} in floating point (IEEE 754)

Floating-Point Example

Write -58.25_{10} in floating point (IEEE 754)

1. Convert decimal to binary:

$$58.25_{10} = 111010.01_2$$

2. Write in binary scientific notation:

$$1.1101001 \times 2^5$$

3. Fill in fields:

Sign bit: 1 (negative)

8 exponent bits: $(127 + 5) = 132 = 10000100_2$

23 fraction bits: 110 1001 0000 0000 0000 0000

1 bit	8 bits	23 bits
1	100 0010 0	110 1001 0000 0000 0000 0000
Sign	Exponent	Fraction

in hexadecimal: 0xC2690000



Floating-Point: Special Cases

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000000000000
∞	0	11111111	00000000000000000000000000000000
$-\infty$	1	11111111	00000000000000000000000000000000
NaN	X	11111111	non-zero

Floating-Point Precision

- **Single-Precision:**
 - 32-bit
 - 1 sign bit, 8 exponent bits, 23 fraction bits
 - bias = 127
- **Double-Precision:**
 - 64-bit
 - 1 sign bit, 11 exponent bits, 52 fraction bits
 - bias = 1023

Floating-Point: Rounding

- **Overflow:** number too large to be represented
- **Underflow:** number too small to be represented
- **Rounding modes:**
 - Down
 - Up
 - Toward zero
 - To nearest
- **Example:** round 1.100101 (1.578125) to only 3 fraction bits
 - Down: 1.100
 - Up: 1.101
 - Toward zero: 1.100
 - To nearest: 1.101 (1.625 is closer to 1.578125 than 1.5 is)



Floating-Point Addition

1. Extract exponent and fraction bits
2. Prepend leading 1 to form mantissa
3. Compare exponents
4. Shift smaller mantissa if necessary
5. Add mantissas
6. Normalize mantissa and adjust exponent if necessary
7. Round result
8. Assemble exponent and fraction back into floating-point format

Floating-Point Addition-

$$\begin{array}{ll}
 & \text{normalized} \qquad \qquad \text{aligned} \\
 a = 123456.7 & = 1.234567 \cdot 10^5 \\
 b = 101.7654 & = 1.017654 \cdot 10^2 = 0.001017654 \cdot 10^5
 \end{array}$$

The **number which is smaller** (here **b**) is shifted (aligned) so that both numbers will have the **same** exponent

$$\begin{array}{r}
 c = a + b \\
 \begin{array}{r}
 1.234567 \cdot 10^5 \\
 + 0.001017654 \cdot 10^5 \\
 \hline
 1.235584654 \cdot 10^5
 \end{array}
 \end{array}$$

The result have to be normalized (shifted)

Floating-Point Addition

- Given two floating-point numbers:

$$a = a_{frac} \cdot 2^{a_{exp}}$$

$$b = b_{frac} \cdot 2^{b_{exp}}$$

- The sum of these numbers is:

$$c = a + b$$

The smaller number is shifted

$$= \begin{cases} (a_{frac} + (b_{frac} \cdot 2^{-(a_{exp} - b_{exp})})) * 2^{a_{exp}} & , \text{if } a_{exp} \geq b_{exp} \\ (b_{frac} + (a_{frac} \cdot 2^{-(b_{exp} - a_{exp})})) * 2^{b_{exp}} & , \text{if } b_{exp} \geq a_{exp} \end{cases}$$



Floating-Point Subtraction

- Given two floating-point numbers:

$$a = a_{frac} \cdot 2^{a_{exp}}$$

$$b = b_{frac} \cdot 2^{b_{exp}}$$

- The difference between these numbers is:

$$c = a - b$$

The smaller number is shifted

$$= \begin{cases} (a_{frac} - (b_{frac} \cdot 2^{-(a_{exp} - b_{exp})})) * 2^{a_{exp}} & , \text{if } a_{exp} \geq b_{exp} \\ (b_{frac} - (a_{frac} \cdot 2^{-(b_{exp} - a_{exp})})) * 2^{b_{exp}} & , \text{if } b_{exp} \geq a_{exp} \end{cases}$$



Floating-Point Multiplication-Decimal

The result has too many digits – round off

$$c = a \cdot b$$

$$a = 4,734612 \cdot 10^3 \quad b = 5,417242 \cdot 10^5$$

$$c = 4,734612 \cdot 5,417242 \cdot 10^{3+5} = 25,648538980104 \cdot 10^8$$

$$c = 2,564854 \cdot 10^9 \quad \text{normalize}$$

Multiplication involves:

- Addition of exponents
- Multiplication of fractional parts
- Normalization of the answer (shifting)



Floating-Point Multiplication

- Given two floating-point numbers:

$$a = a_{frac} \cdot 2^{a_{exp}}$$

$$b = b_{frac} \cdot 2^{b_{exp}}$$

- The product of these numbers is:

$$\begin{aligned} c &= a * b \\ &= \left(a_{frac} * b_{frac} \cdot 2^{a_{exp} + b_{exp}} \right) \end{aligned}$$

Floating-Point Division

- Given two floating-point numbers:

$$a = a_{frac} \cdot 2^{a_{exp}}$$

$$b = b_{frac} \cdot 2^{b_{exp}}$$

- The ratio of these numbers is:

$$\begin{aligned} c &= a / b \\ &= \left(a_{frac} / b_{frac} \cdot 2^{a_{exp} - b_{exp}} \right) \end{aligned}$$

Normalization

- When a floating-point operation is complete, it must be normalized
 - Mantissa is shifted until its first bit is 1
 - For each shift step, exponent is increased or decreased.
 - Mantissa's bits to the right of the first 1 are saved

$$V(B) = (-1)^s * (1.M) * 2^{E-(127)}$$

- If the exponent is zero, the first mantissa's bit is 0

$$V(B) = (-1)^s * (0.M) * 2^{-(126)}$$

Floating-Point Addition Example

Add the following floating-point numbers:

0x3FC00000

0x40500000

Floating-Point Addition Example

1. Extract exponent and fraction bits

1 bit	8 bits	23 bits
0	01111111	100 0000 0000 0000 0000 0000
Sign	Exponent	Fraction
1 bit	8 bits	23 bits
0	10000000	101 0000 0000 0000 0000 0000
Sign	Exponent	Fraction

For first number (N1):

$S = 0, E = 127, F = .1$

For second number (N2):

$S = 0, E = 128, F = .101$

2. Prepend leading 1 to form mantissa

N1: 1.1

N2: 1.101



Floating-Point Addition Example

3. Compare exponents

$127 - 128 = -1$, so shift N1 right by 1 bit

4. Shift smaller mantissa if necessary

shift N1's mantissa: $1.1 \gg 1 = 0.11$ ($\times 2^1$)

5. Add mantissas

$$\begin{array}{r}
 0.11 \times 2^1 \\
 + 1.101 \times 2^1 \\
 \hline
 10.011 \times 2^1
 \end{array}$$

Floating Point Addition Example

6. **Normalize mantissa and adjust exponent if necessary**
 $10.011 \times 2^1 = 1.0011 \times 2^2$
7. **Round result**
 No need (fits in 23 bits)
8. **Assemble exponent and fraction back into floating-point format**

$$S = 0, E = 2 + 127 = 129 = 10000001_2, F = 001100..$$

1 bit	8 bits	23 bits
0	10000001	001 1000 0000 0000 0000 0000
Sign	Exponent	Fraction

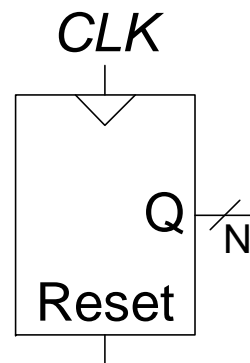
in hexadecimal: **0x40980000**



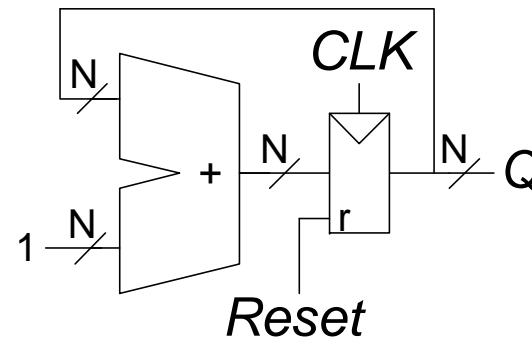
Counters

- Increments on each clock edge
- Used to cycle through numbers. For example,
 - 000, 001, 010, 011, 100, 101, 110, 111, 000, 001...
- Example uses:
 - Digital clock displays
 - Program counter: keeps track of current instruction executing

Symbol



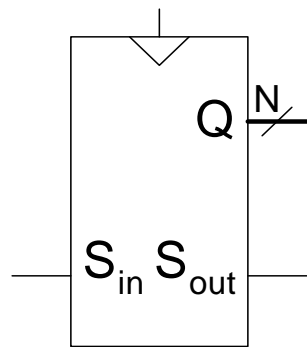
Implementation



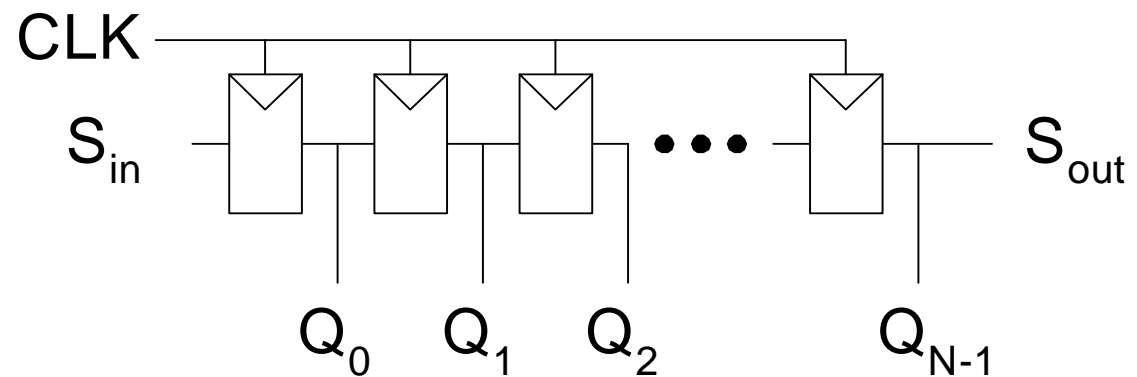
Shift Registers

- Shift a new bit in on each clock edge
- Shift a bit out on each clock edge
- *Serial-to-parallel converter*: converts serial input (S_{in}) to parallel output ($Q_{0:N-1}$)

Symbol:

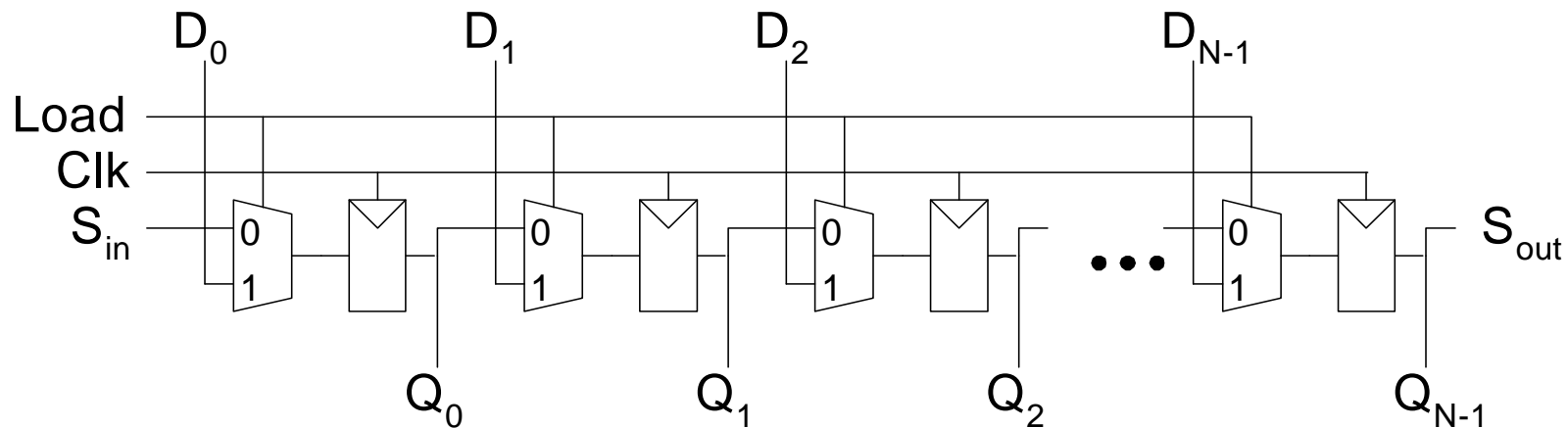


Implementation:



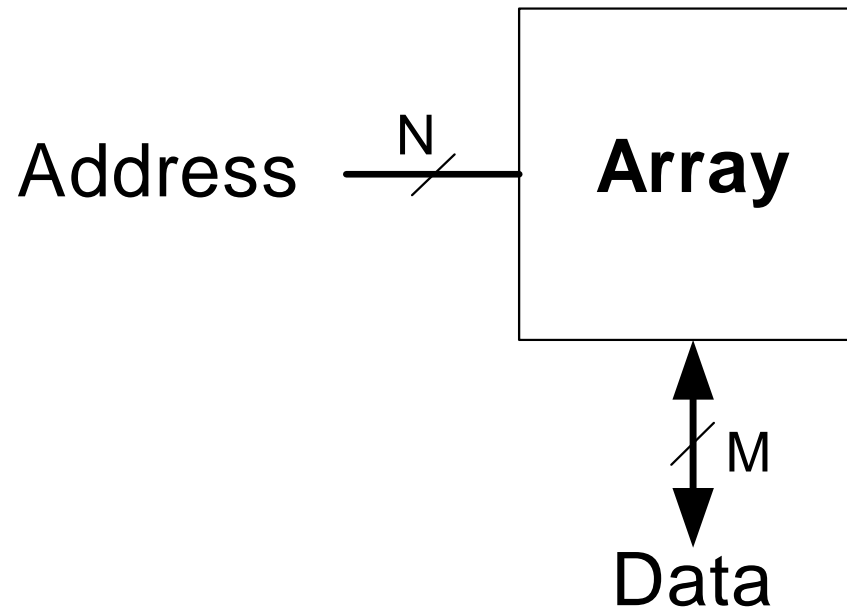
Shift Register with Parallel Load

- When $Load = 1$, acts as a normal N -bit register
- When $Load = 0$, acts as a shift register
- Now can act as a *serial-to-parallel converter* (S_{in} to $Q_{0:N-1}$) or a *parallel-to-serial converter* ($D_{0:N-1}$ to S_{out})



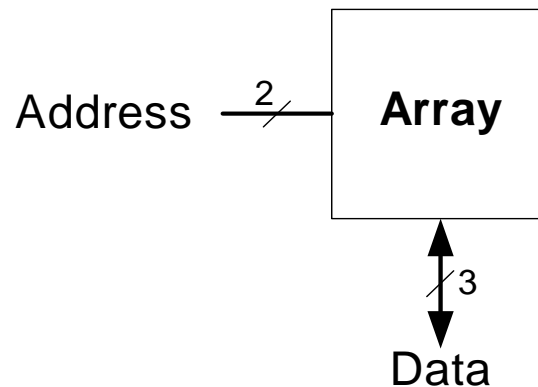
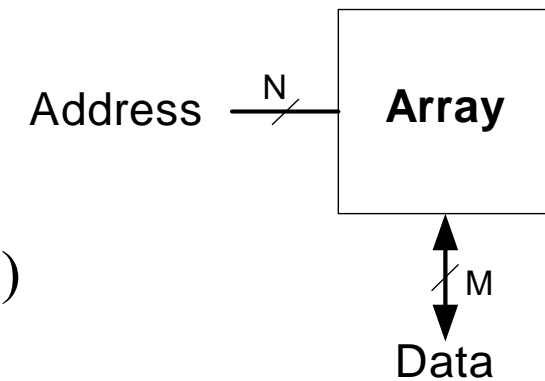
Memory Arrays

- Efficiently store large amounts of data
- 3 common types:
 - Dynamic random access memory (DRAM)
 - Static random access memory (SRAM)
 - Read only memory (ROM)
- M -bit data value read/ written at each unique N -bit address



Memory Arrays

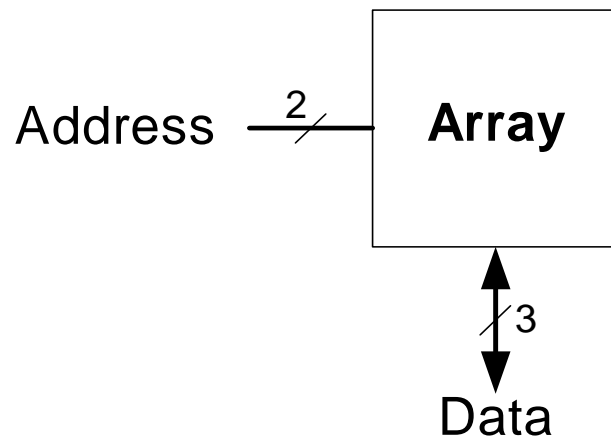
- 2-dimensional array of bit cells
- Each bit cell stores one bit
- N address bits and M data bits:
 - 2^N rows and M columns
 - **Depth:** number of rows (number of words)
 - **Width:** number of columns (size of word)
 - **Array size:** depth \times width = $2^N \times M$



Address	Data			
11	0	1	0	depth ↑ ↓
10	1	0	0	
01	1	1	0	
00	0	1	1	
	width ←→			

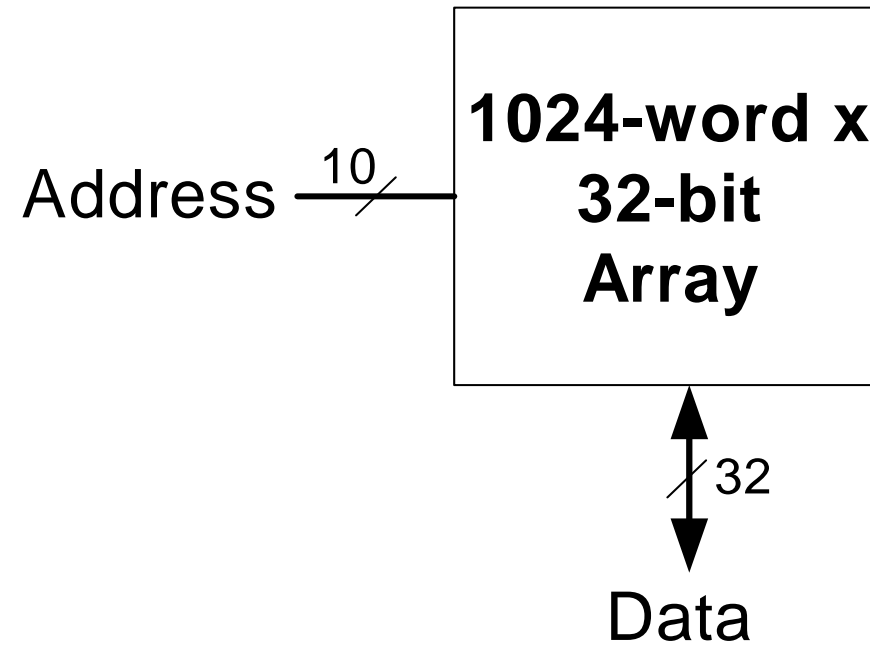
Memory Array Example

- $2^2 \times 3$ -bit array
- Number of words: 4
- Word size: 3-bits
- For example, the 3-bit word stored at address 10 is 100

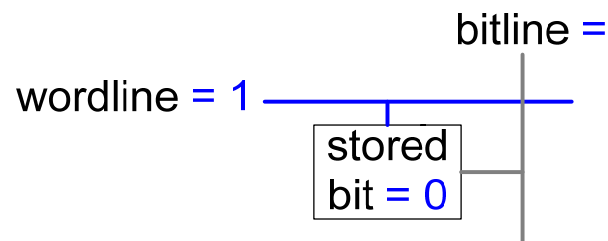
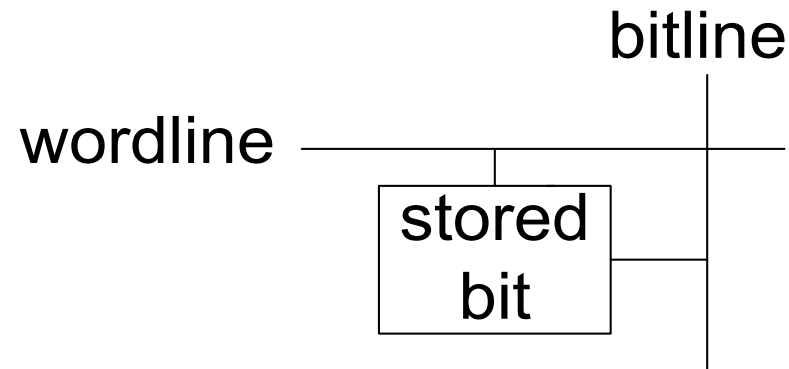


Address	Data			
11	0	1	0	depth ↑ ↓
10	1	0	0	
01	1	1	0	
00	0	1	1	
	width ←→			

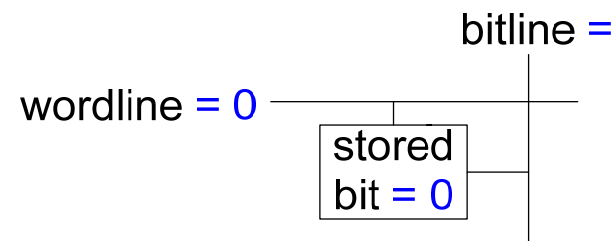
Memory Arrays



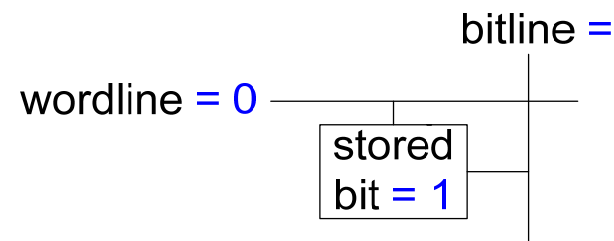
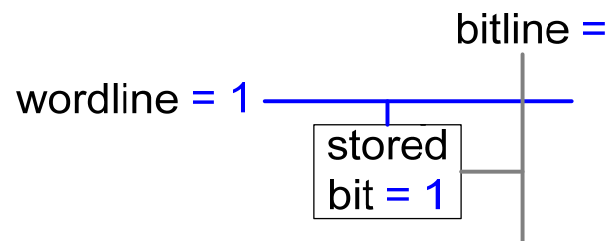
Memory Array Bit Cells



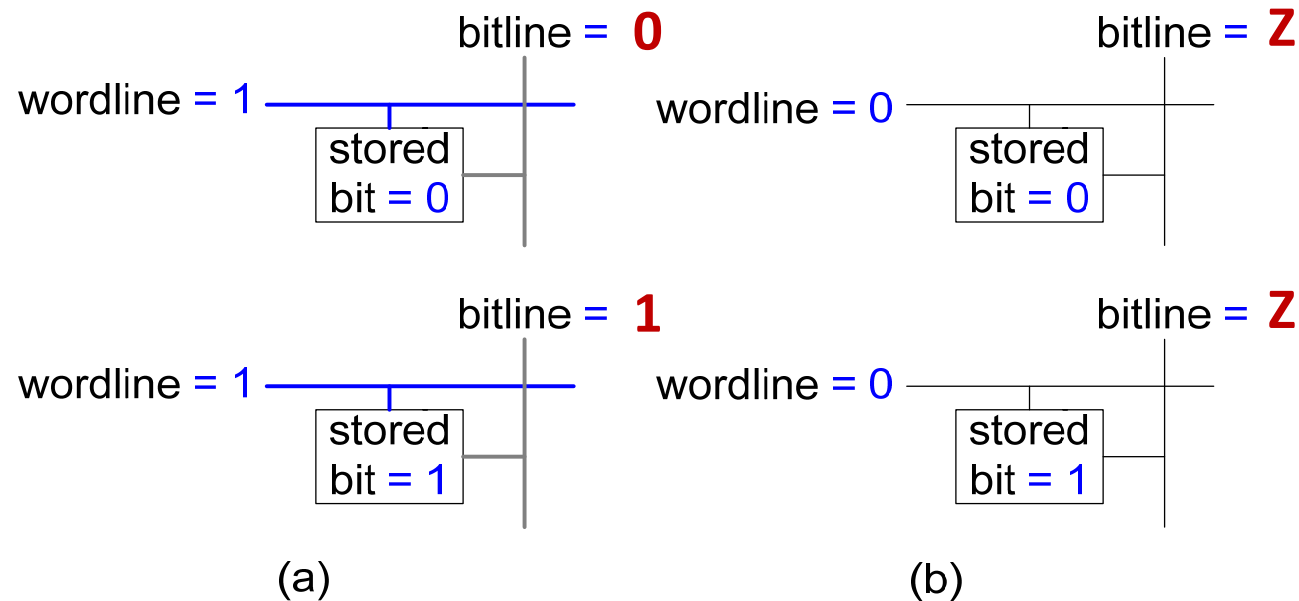
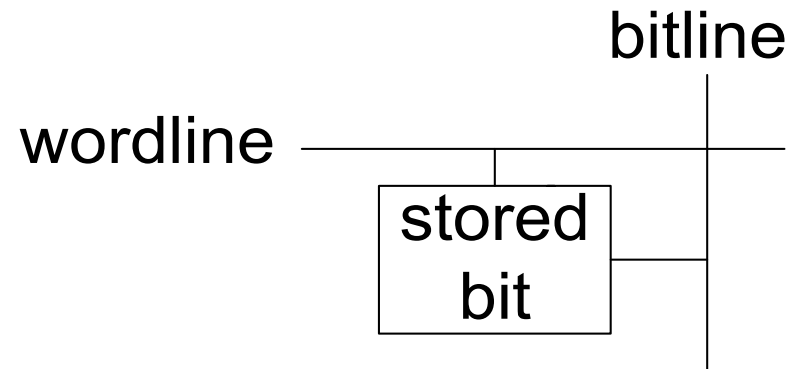
(a)



(b)

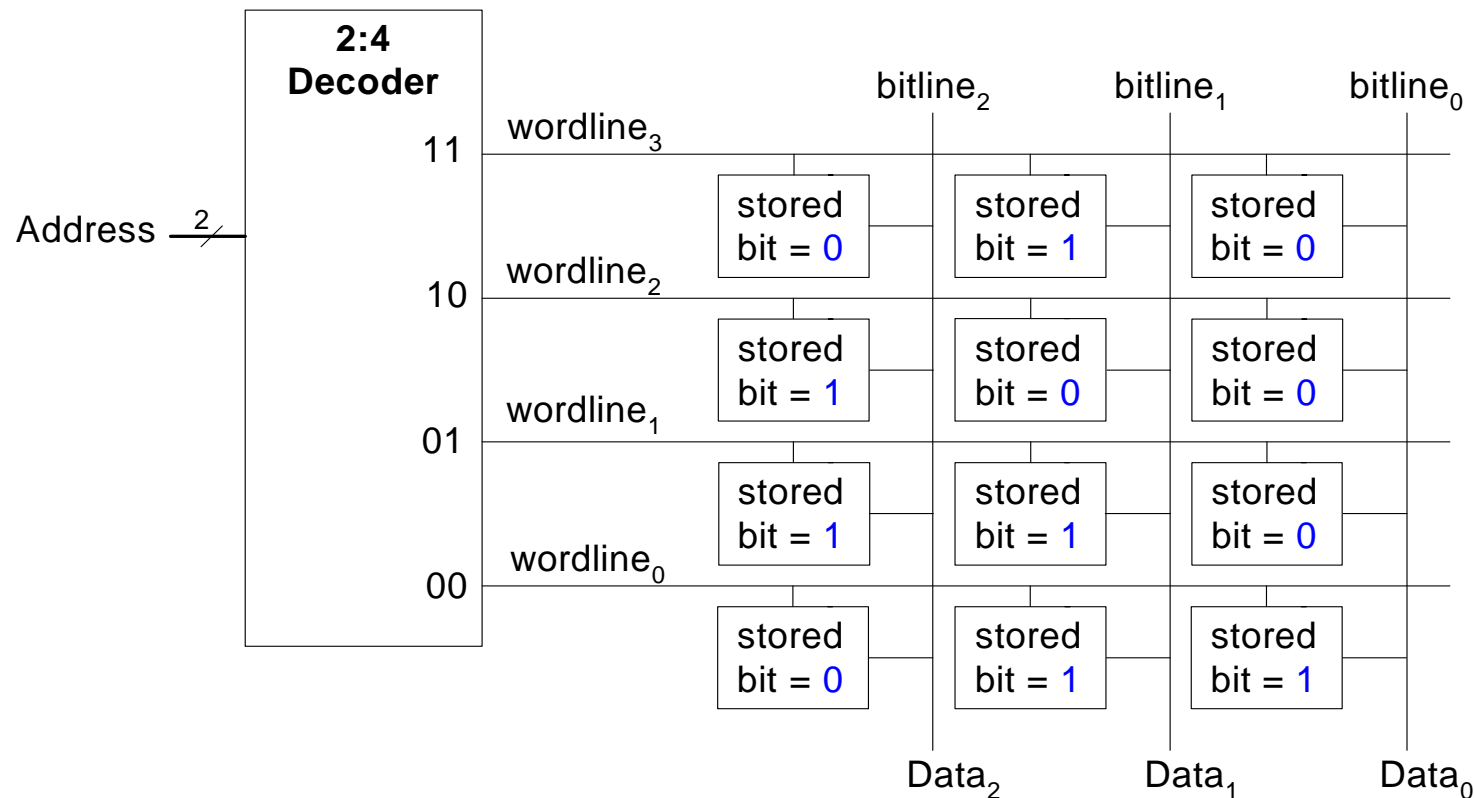


Memory Array Bit Cells



Memory Array

- **Wordline:**
 - like an enable
 - single row in memory array read/written
 - corresponds to unique address
 - only one wordline HIGH at once



Types of Memory

- Random access memory (RAM): **volatile**
- Read only memory (ROM): **nonvolatile**

Types of Memory

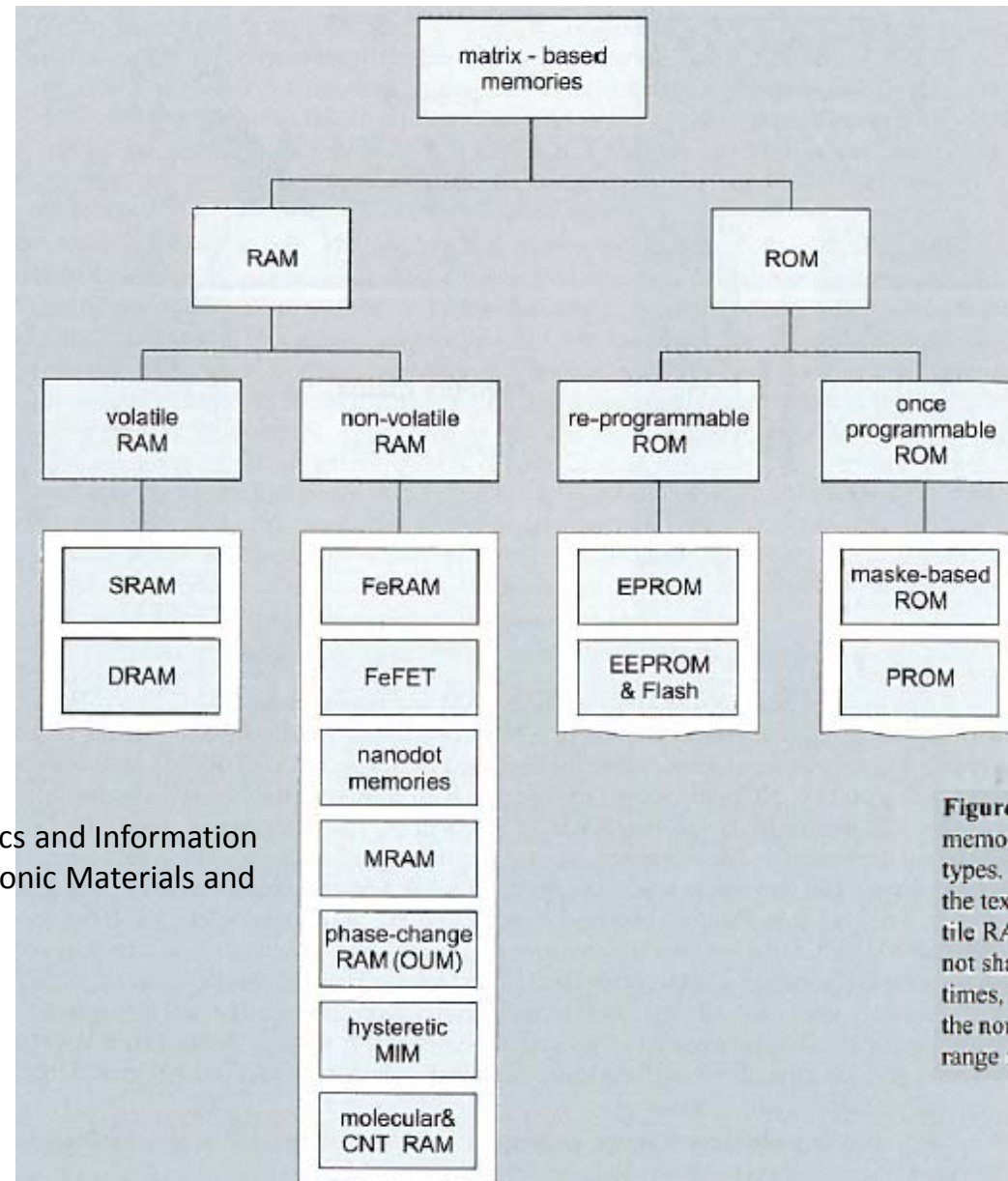


Figure 4: Categories of matrix-based memory chips showing the most relevant types. The abbreviations are explained in the text. The boundary between non-volatile RAM and re-programmable ROM is not sharp. Today, it is given by the write times, which are in the ns to μ s range for the non-volatile RAMs and in the seconds range for the re-programmable ROM.

Rainer Waser, Nanoelectronics and Information Technology: Advanced Electronic Materials and Novel Devices, Wiley 2005

RAM: Random Access Memory

- **Volatile:** loses its data when power off
- Read and written quickly
- Main memory in your computer is RAM (DRAM)

Historically called *random* access memory because any data word accessed as easily as any other (in contrast to sequential access memories such as a tape recorder)



Original Random Access Memory

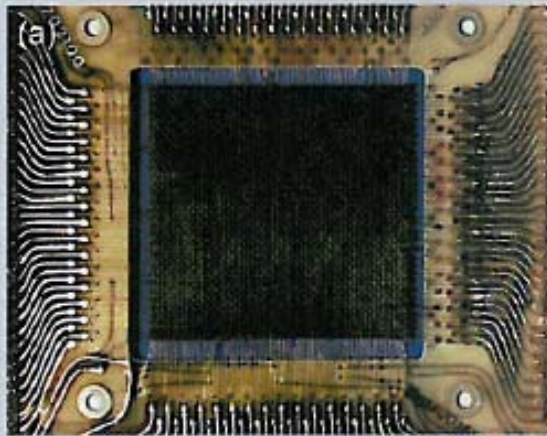
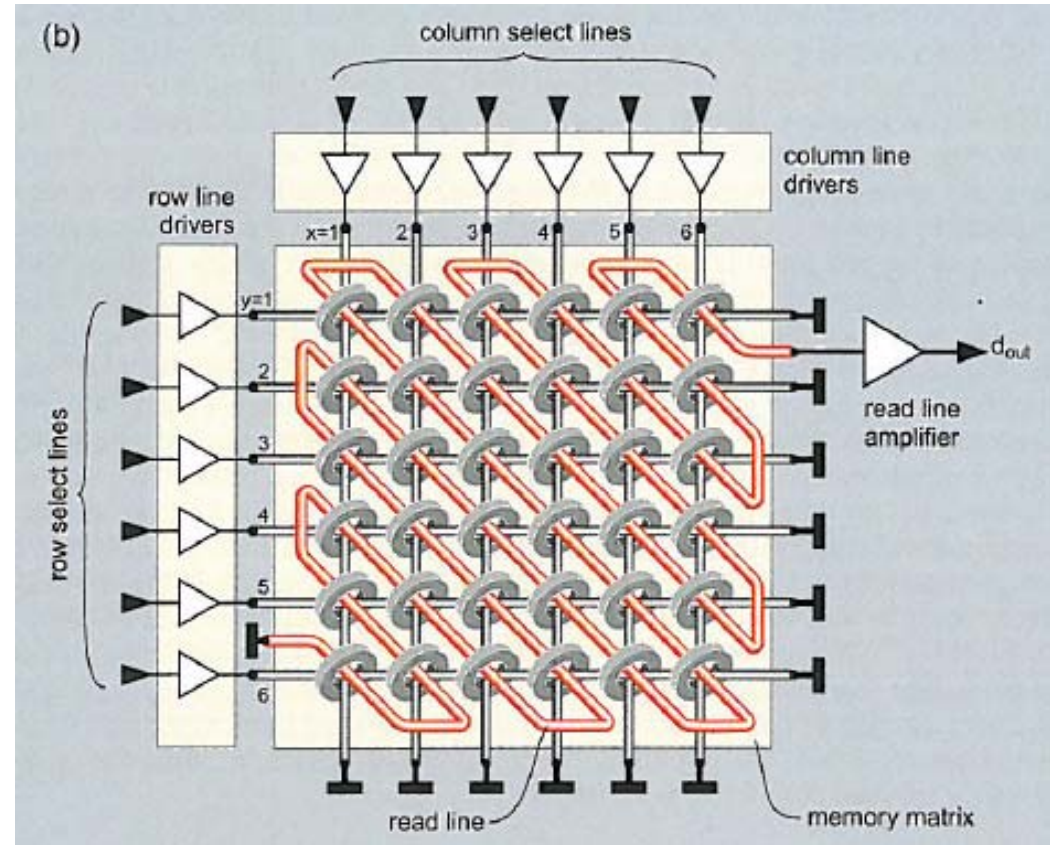


Figure 1: Magnetic core memory.
 (a) Photograph of an historical 1kb ferrite-based magnetic-core memory. One side of the square has a length of 10 cm.
 (b) Schematics of the magnetic-core memory.
 (c) A ferrite core at the intersection (node) of the row and column lines used for writing the magnetization direction which represents the binary logic states.



Rainer Waser, Nanoelectronics and Information Technology: Advanced Electronic Materials and Novel Devices, Wiley 2005

ROM: Read Only Memory

- **Nonvolatile:** retains data when power off
- Read quickly, but writing is impossible or slow
- Flash memory in cameras, thumb drives, and digital cameras are all ROMs

Historically called *read only* memory because ROMs were written at manufacturing time or by burning fuses. Once ROM was configured, it could not be written again. This is no longer the case for Flash memory and other types of ROMs.



Types of RAM

- **DRAM** (Dynamic random access memory)
- **SRAM** (Static random access memory)
- Differ in how they store data:
 - DRAM uses a capacitor
 - SRAM uses cross-coupled inverters

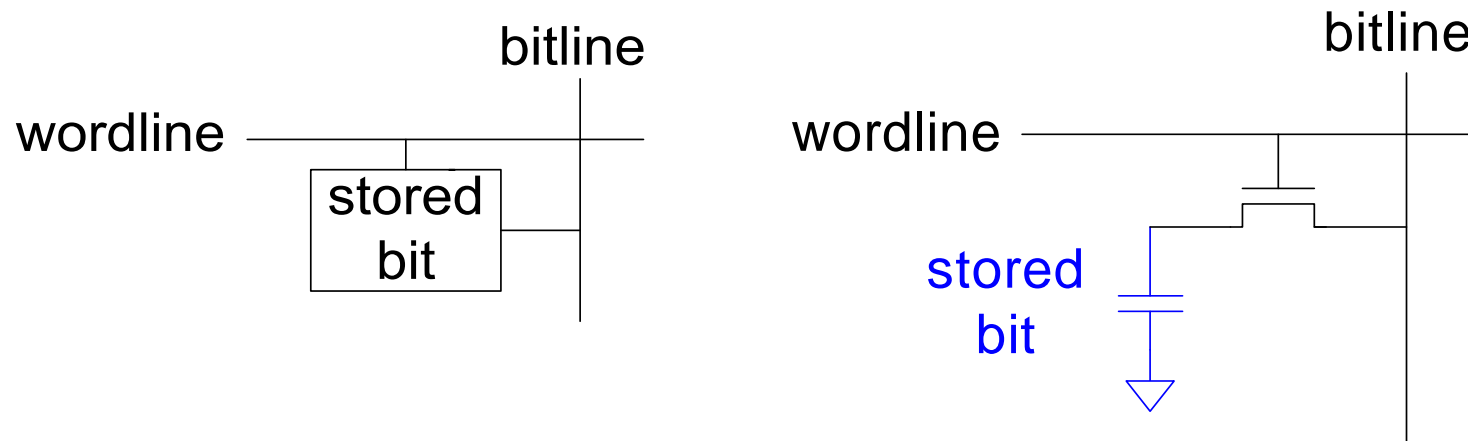
Robert Dennard, 1932 -

- Invented DRAM in 1966 at IBM
- Others were skeptical that the idea would work
- By the mid-1970's DRAM in virtually all computers

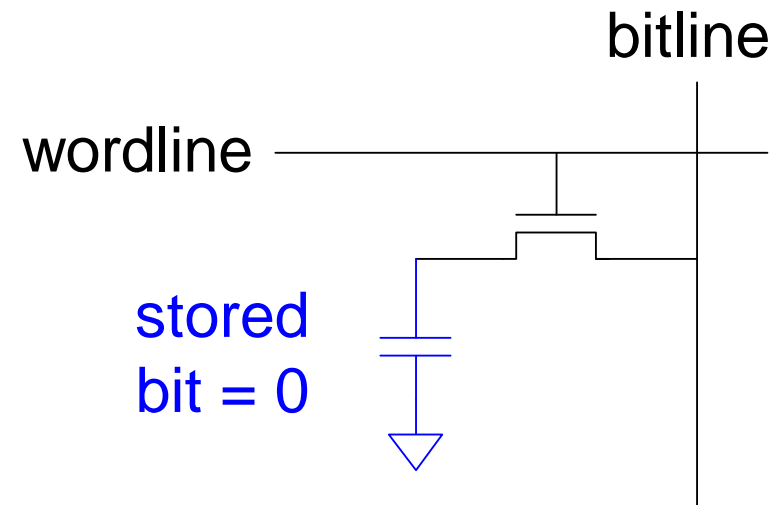
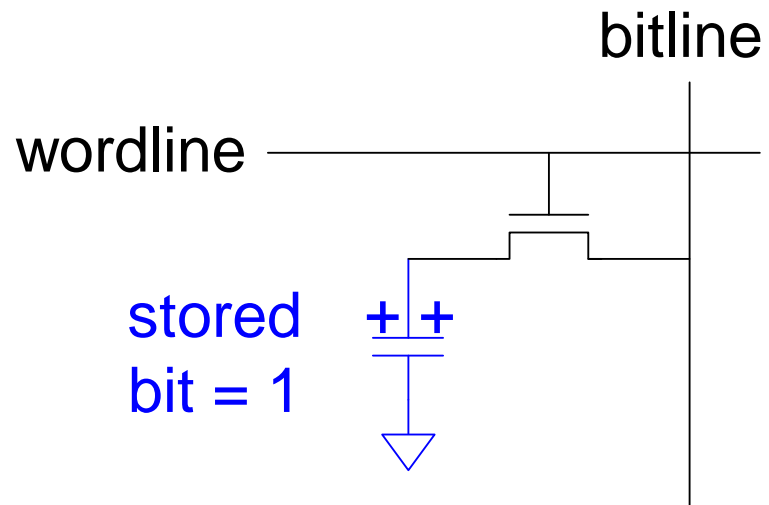


DRAM

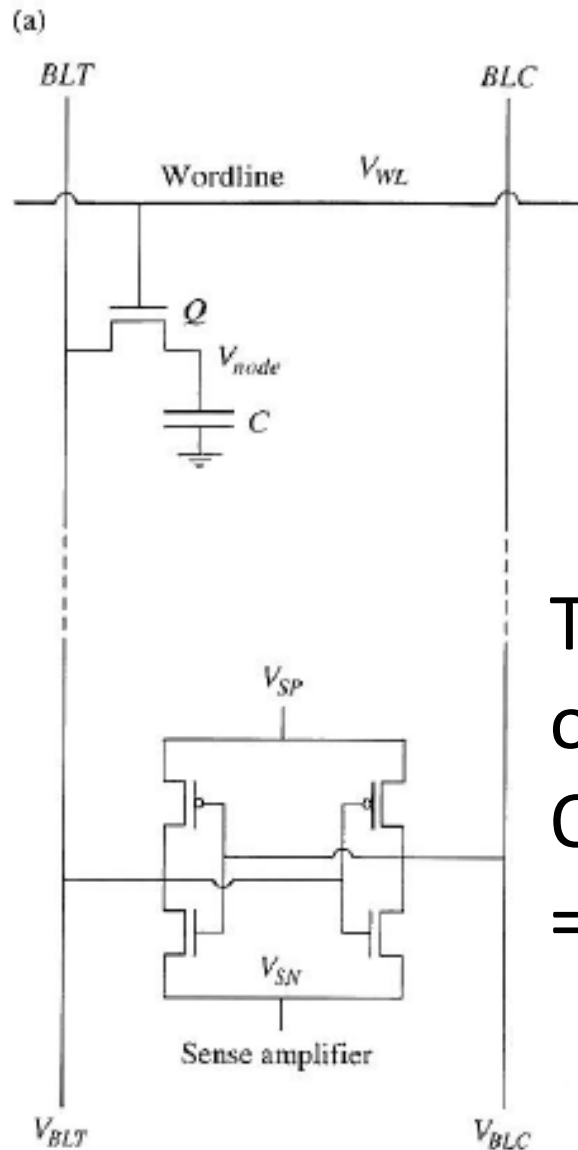
- Data bits stored on capacitor
- *Dynamic* because the value needs to be refreshed (rewritten) periodically and after read:
 - Charge leakage from the capacitor degrades the value
 - Reading destroys the stored value



DRAM



DRAM readout is destructive



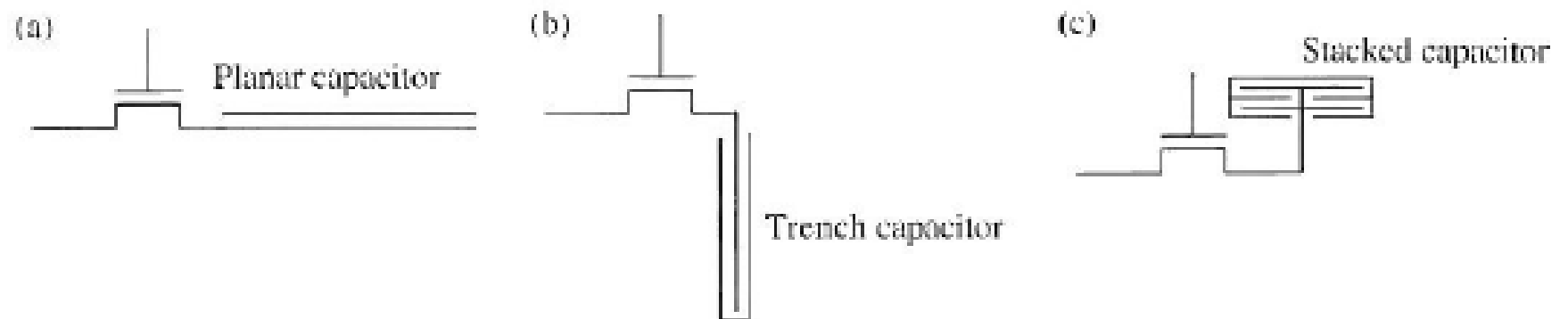
$$V_s = \left(V_{node} - \frac{V_{dd}}{2} \right) \frac{C_{cell}}{C_{cell} + C_{bitline}},$$

Typically $C_{cell} = 30$ fF at 1 V operation

$Q = CU$

$\Rightarrow 200\,000$ electrons/bit

DRAM: how to increase the capacitance



Schematics showing three DRAM cell structures: (a) planar-capacitor cell, (b) trench-capacitor cell, and (c) stacked-capacitor cell.

Trench DRAM cell

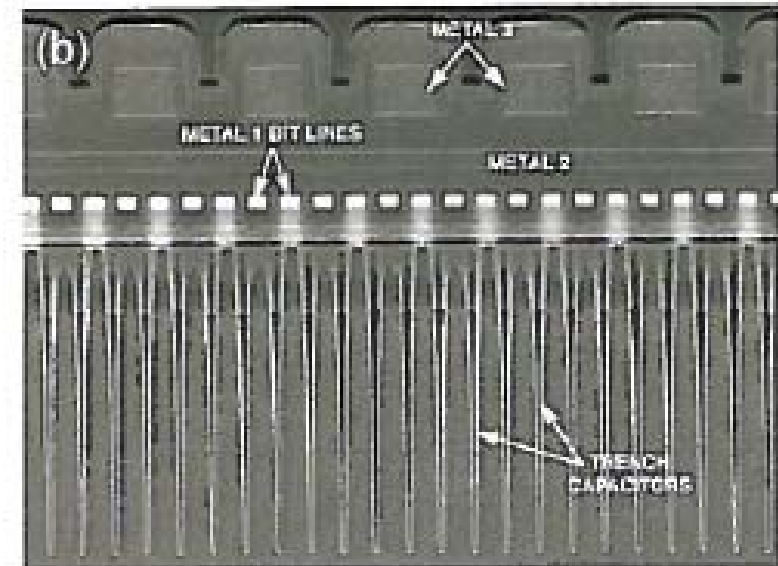
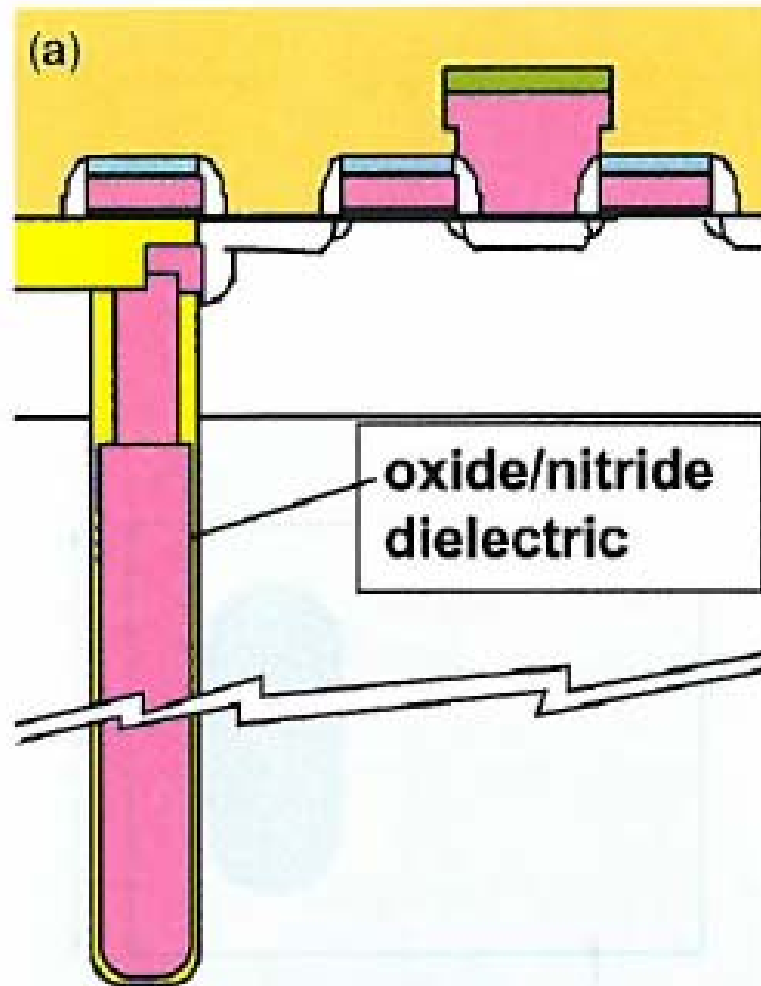


Figure 3: Example of a 3-D capacitor: (cross section) deep trench with oxide/nitride dielectric ($\epsilon_r \approx 7$).

(a) Schematic, from [8];

(b) Scanning electron micrograph for Toshiba/Infineon 64 Mb chip. The design is similar in 256 Mb and 1 Gb chips with much higher aspect ratios (height/ depth).

Rainer Waser, Nanoelectronics and Information Technology: Advanced Electronic Materials and Novel Devices, Wiley 2005

Stacked DRAM cell

Rainer Waser, Nanoelectronics and Information Technology: Advanced Electronic Materials and Novel Devices, Wiley 2005

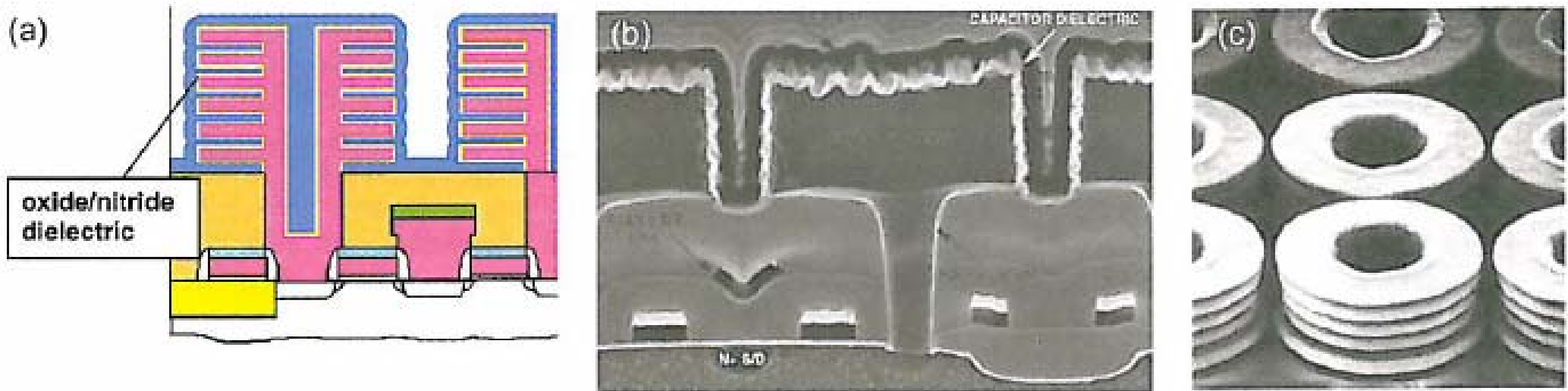
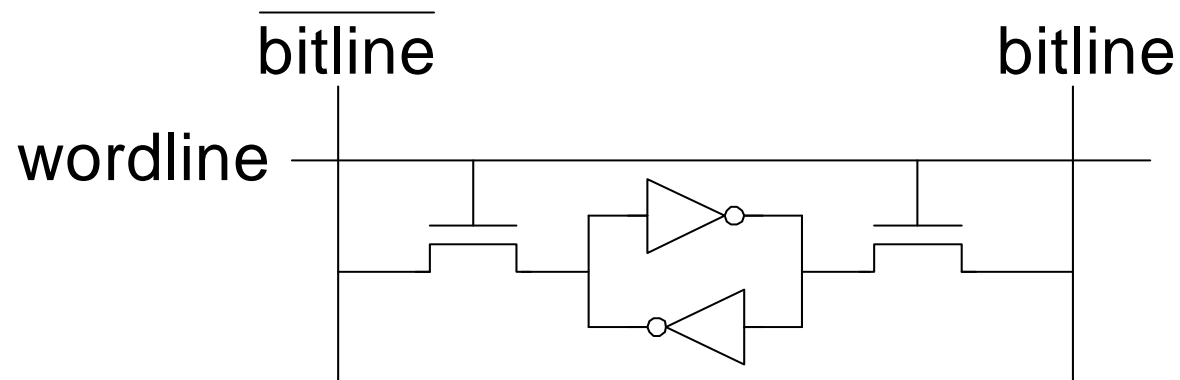
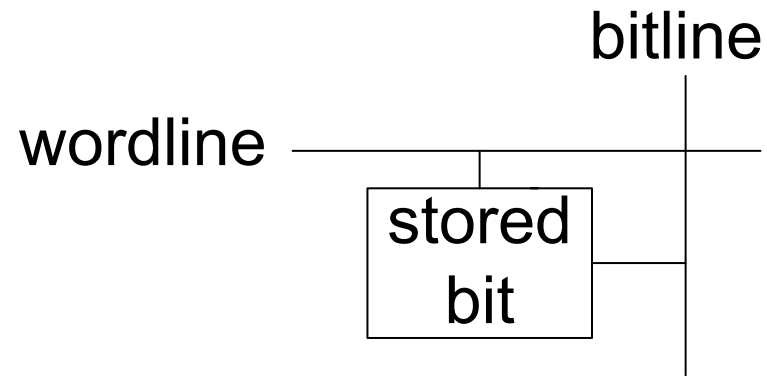


Figure 4: Example of a 3-D capacitor (cross section). Stack with oxide/nitride dielectric ($\epsilon_r \approx 7$) with COB design (capacitor over bit-line).
 (a) Schematic disk-type capacitor [8].
 (b) Scanning electron micrograph: Stacks with roughened surface of the bottom electrode (HSG-Si) for increased effective area, (Mitsubishi 64 Mb) [7].
 (c) Scanning electron micrograph: stacks as disks made of SiO₂ as base for capacitor (Mitsubishi 256 Mb) [9].

SRAM



SRAM basic 6 transistor cell

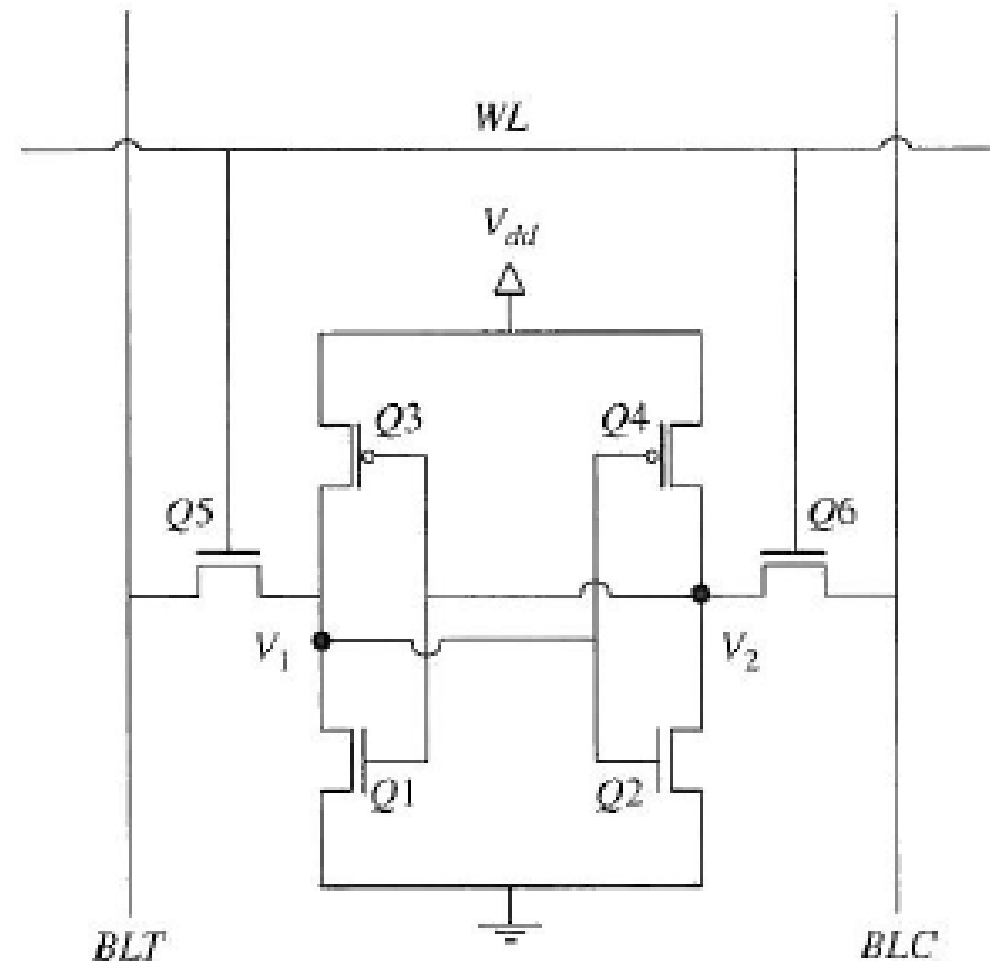
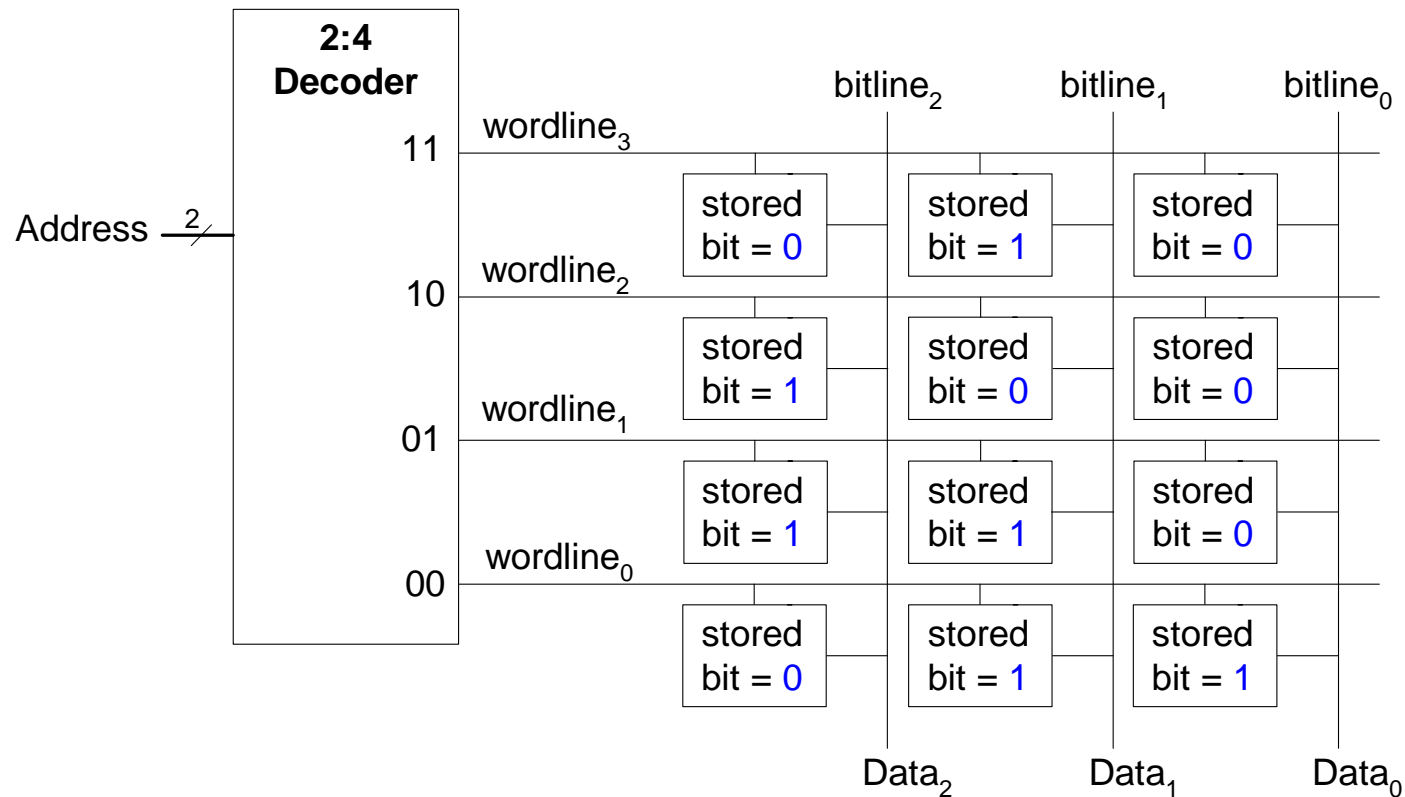
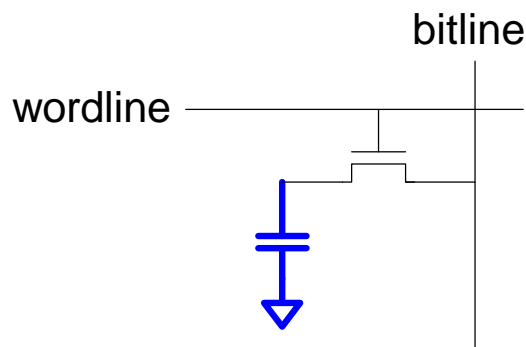


Figure 9.4. Circuit configuration of a CMOS SRAM cell. In the text, we assume the cell is storing a "1" when V_2 = high (V_1 = low) and a "0" when V_1 = high (V_2 = low).

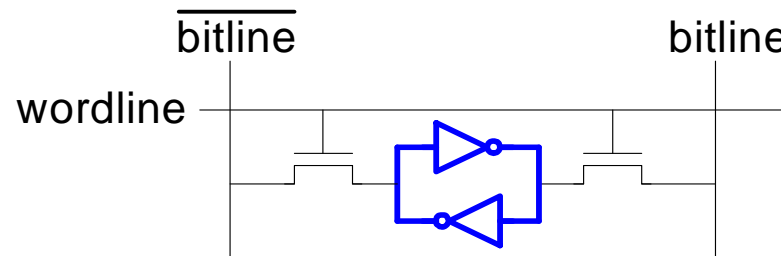
Memory Arrays Review



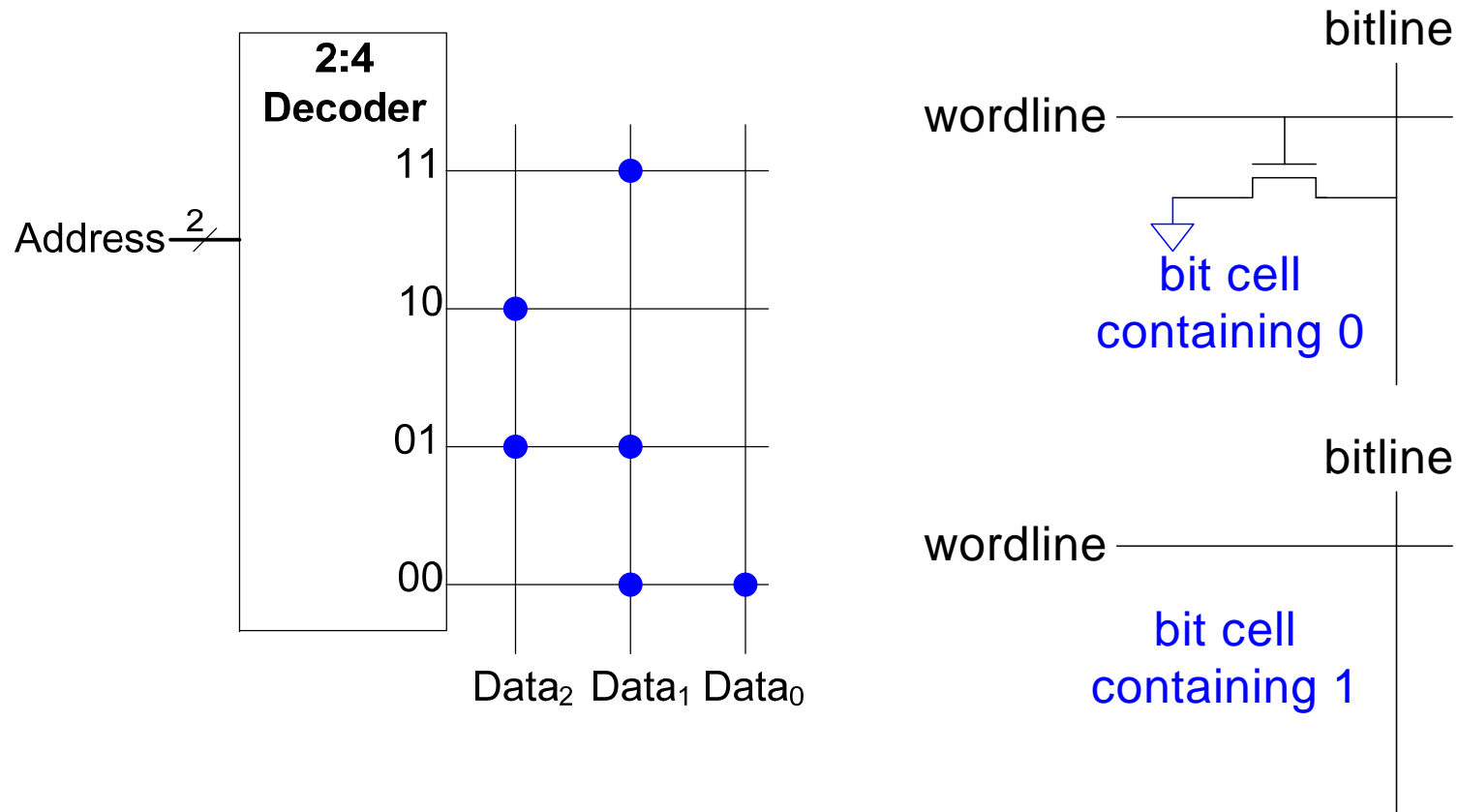
DRAM bit cell:



SRAM bit cell:



ROM: Dot Notation

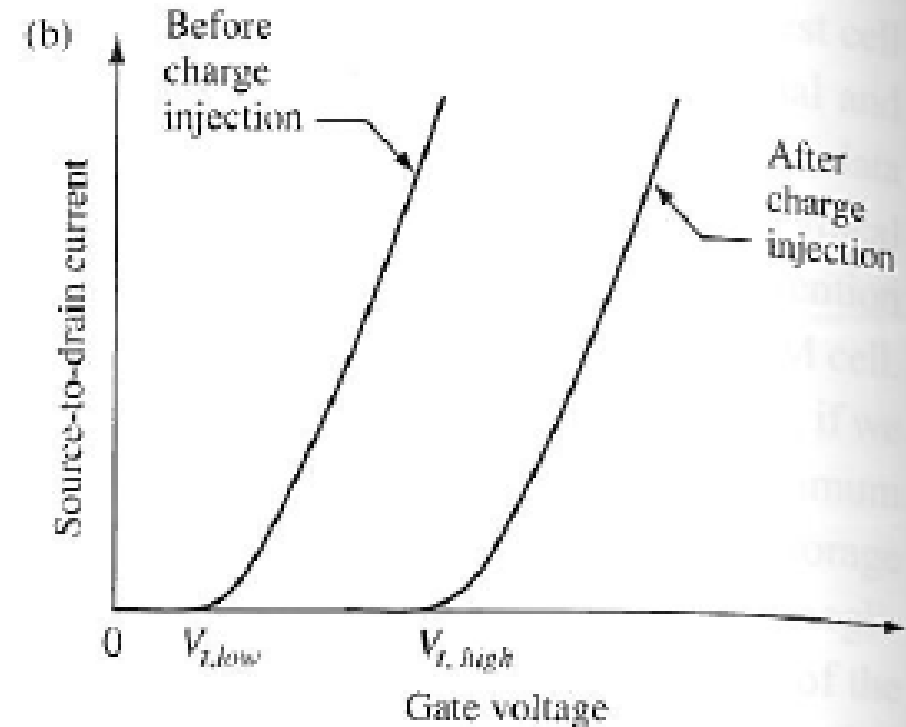
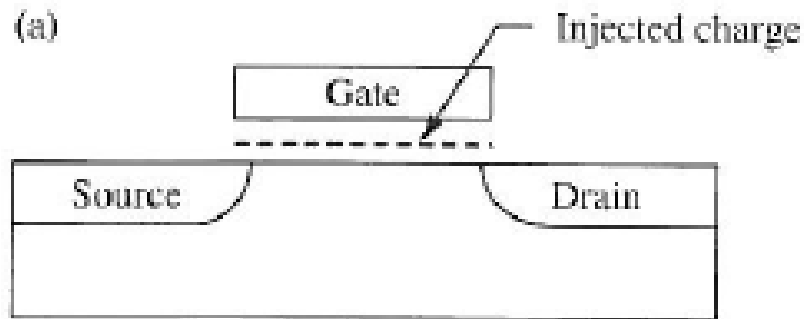


Fujio Masuoka, 1944 -

- Developed memories and high speed circuits at Toshiba, 1971-1994
- Invented Flash memory as an unauthorized project pursued during nights and weekends in the late 1970's
- The process of erasing the memory reminded him of the flash of a camera
- Toshiba slow to commercialize the idea; Intel was first to market in 1988
- Flash has grown into a \$25 billion per year market



Non-volatile Memory



19. (a) Schematic diagram of a MOSFET nonvolatile memory device. (b) The MOSFET threshold voltage shifts from $V_{t,low}$ to $V_{t,high}$ after electron injection.

Non-volatile Memory

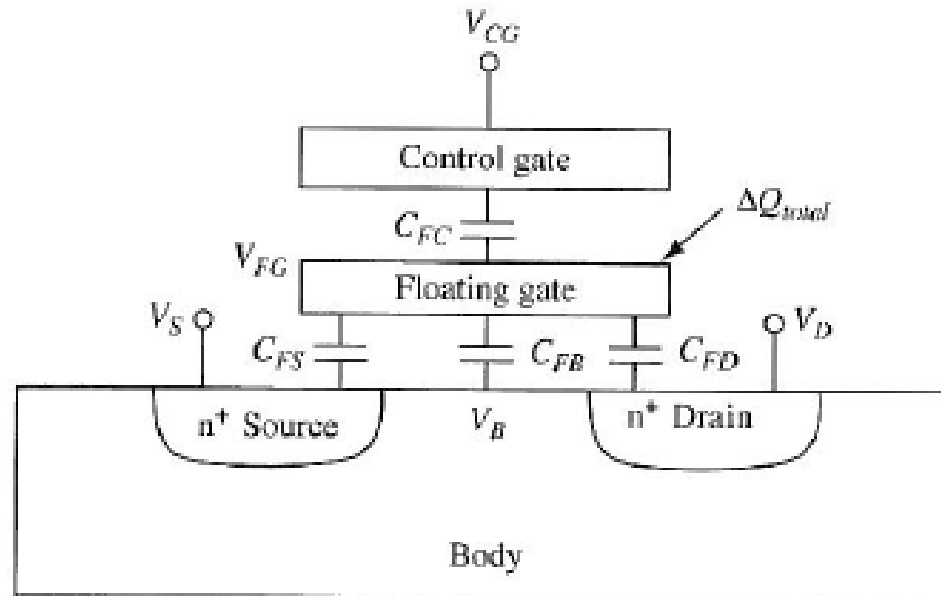
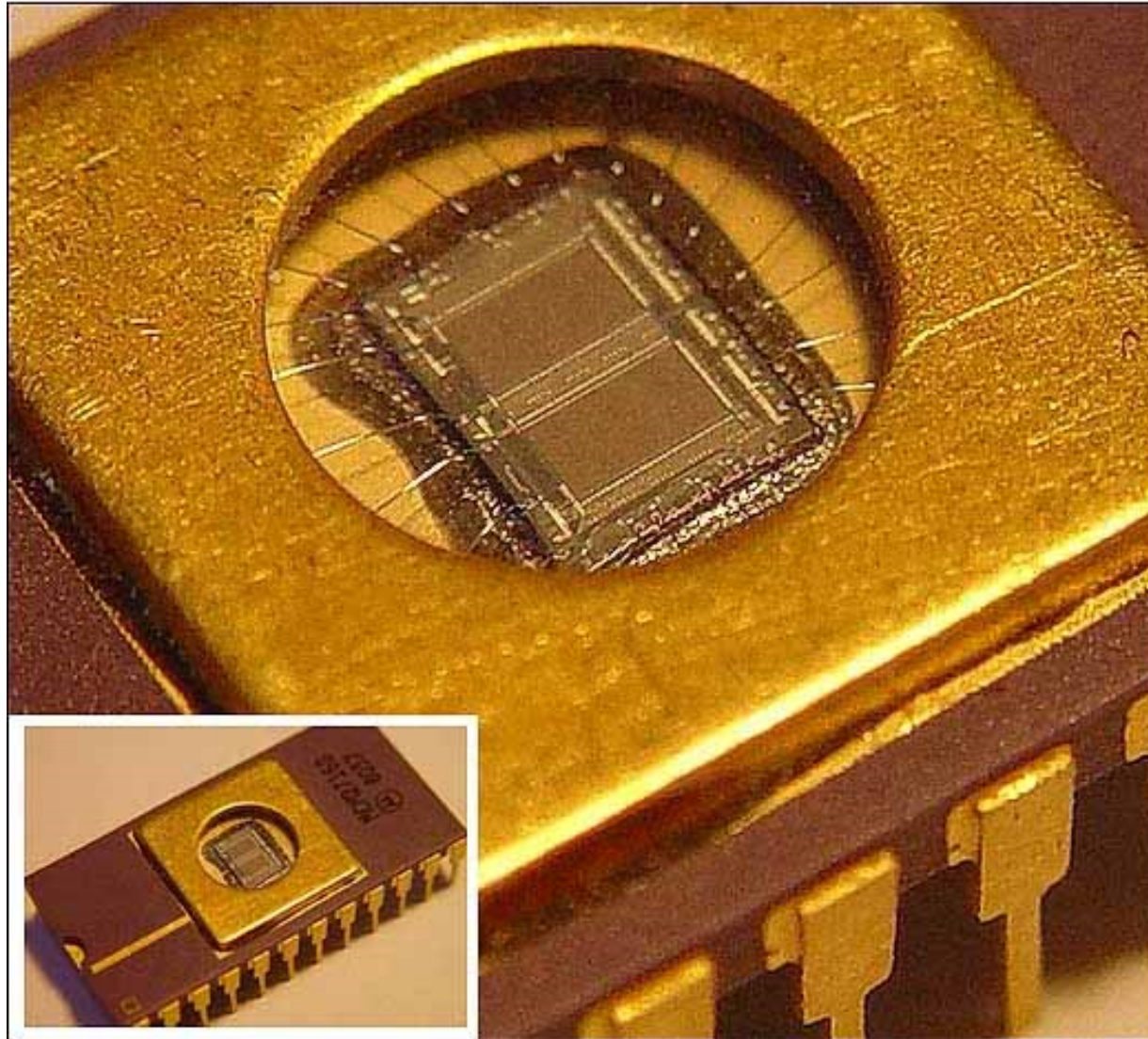


Figure 9.23. Capacitive coupling of the floating gate to other electrodes in an n-channel MOSFET nonvolatile memory device. Any depletion capacitance in the silicon body is absorbed in C_{FB} .

UV-erasable Memory, aka EPROM

From Computer Desktop Encyclopedia
© 2004 The Computer Language Co. Inc.



Flash Memory, aka EEPROM

NOR

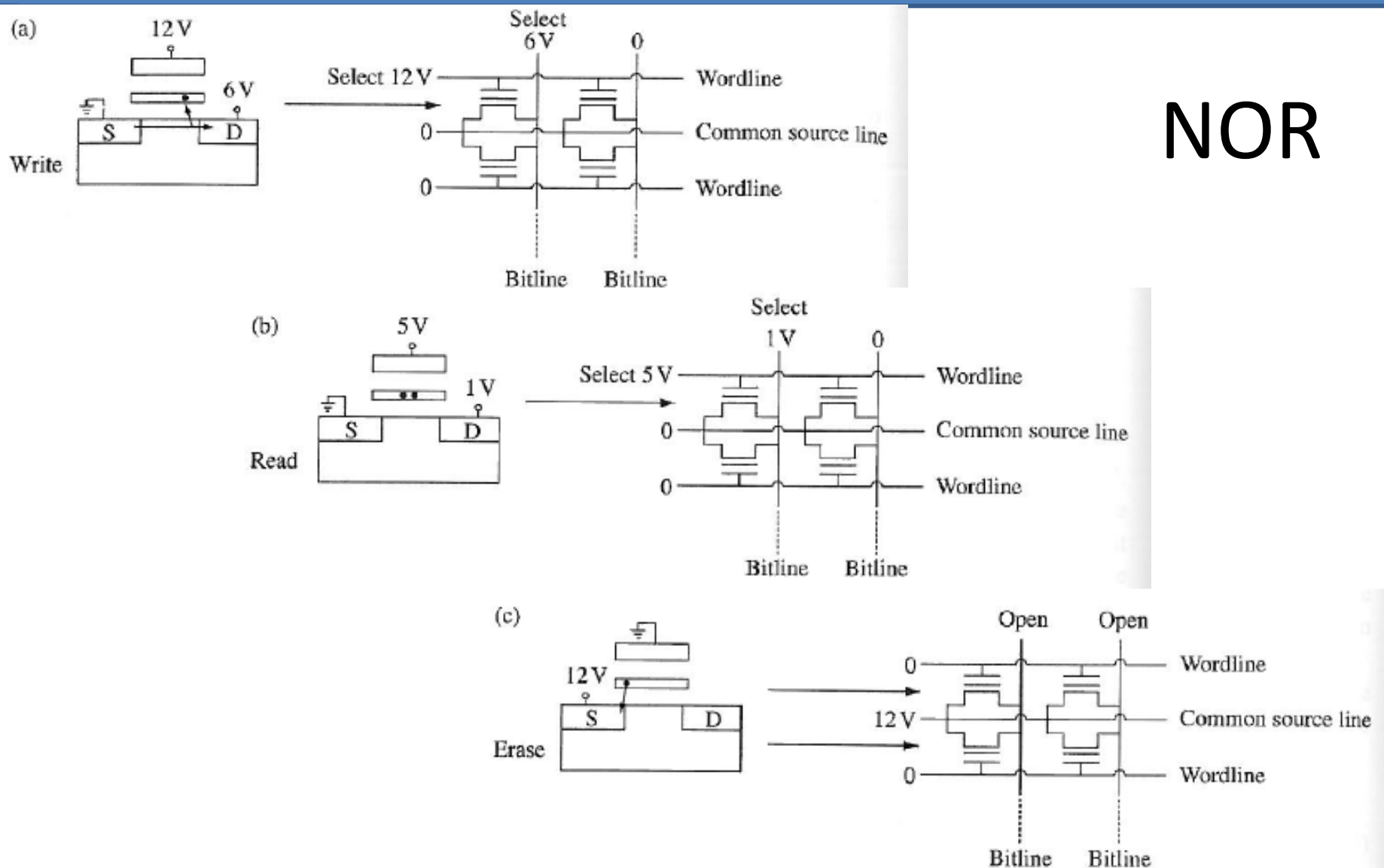
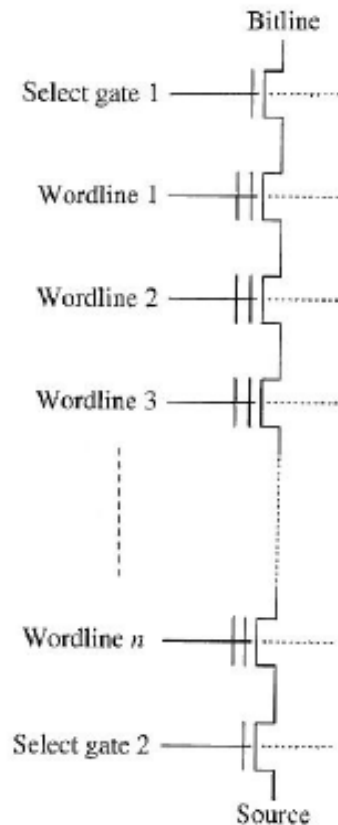


Figure 9.24. Schematics showing the connection of EEPROM devices to wordlines and bitlines in a memory array and their bias voltages for write, read, and erase operations. This is a *NOR* array. (After Itoh, 2001.)

Flash Memory, aka EEPROM



- Advantage of NAND: More bits per area
- Advantage of NOR: Faster
- Both types also called flash memory

Figure 9.25. Schematic showing the serial connection of EEPROM devices in one bitline in a *NAND* array.

Flash Memory: limited endurance

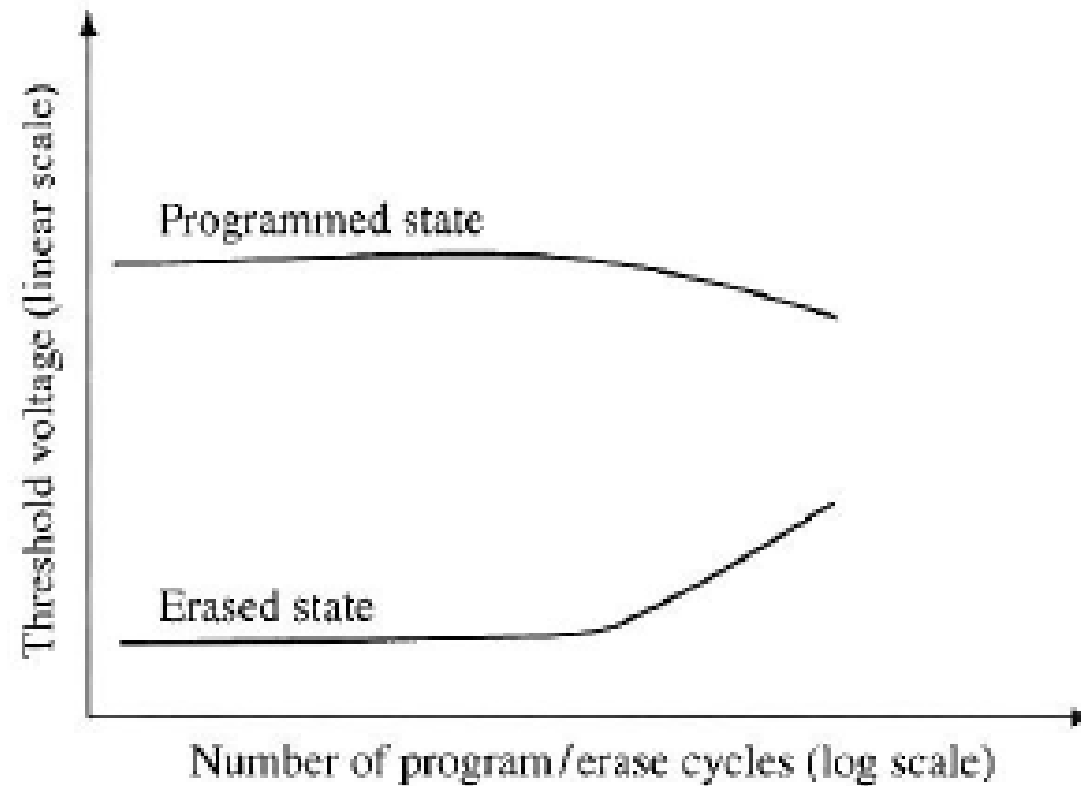


Figure 9.26. Schematic illustrating the collapse of the memory window as a function of the number of program and erase cycles.

Memory: comparison

Property	SRAM	eFlash	eDRAM	eFeRAM (projected)
Min. Voltage	> 0.5 V	> 12 V (± 6 V)	> 1 V	> 1 V
Write Time	< 10 ns	100 μ s / 1 s	< 20 ns	< 20 ns
Write Endurance	> 10^{15}	< 10^5	> 10^{15}	> 10^{15}
Read Time	< 10 ns	20 ns	< 20 ns	< 20 ns
Read Endurance	> 10^{15}	> 10^{15}	> 10^{15}	> 10^{15}
Nonvolatile	no	yes	no	yes
Cell Size (F =half metal pitch)	$\sim 80 F^2$	$\sim 8 F^2$	$\sim 8 F^2$	$\sim 15 F^2$
Mask Count Adder	0	5 – 8	5 – 9	2

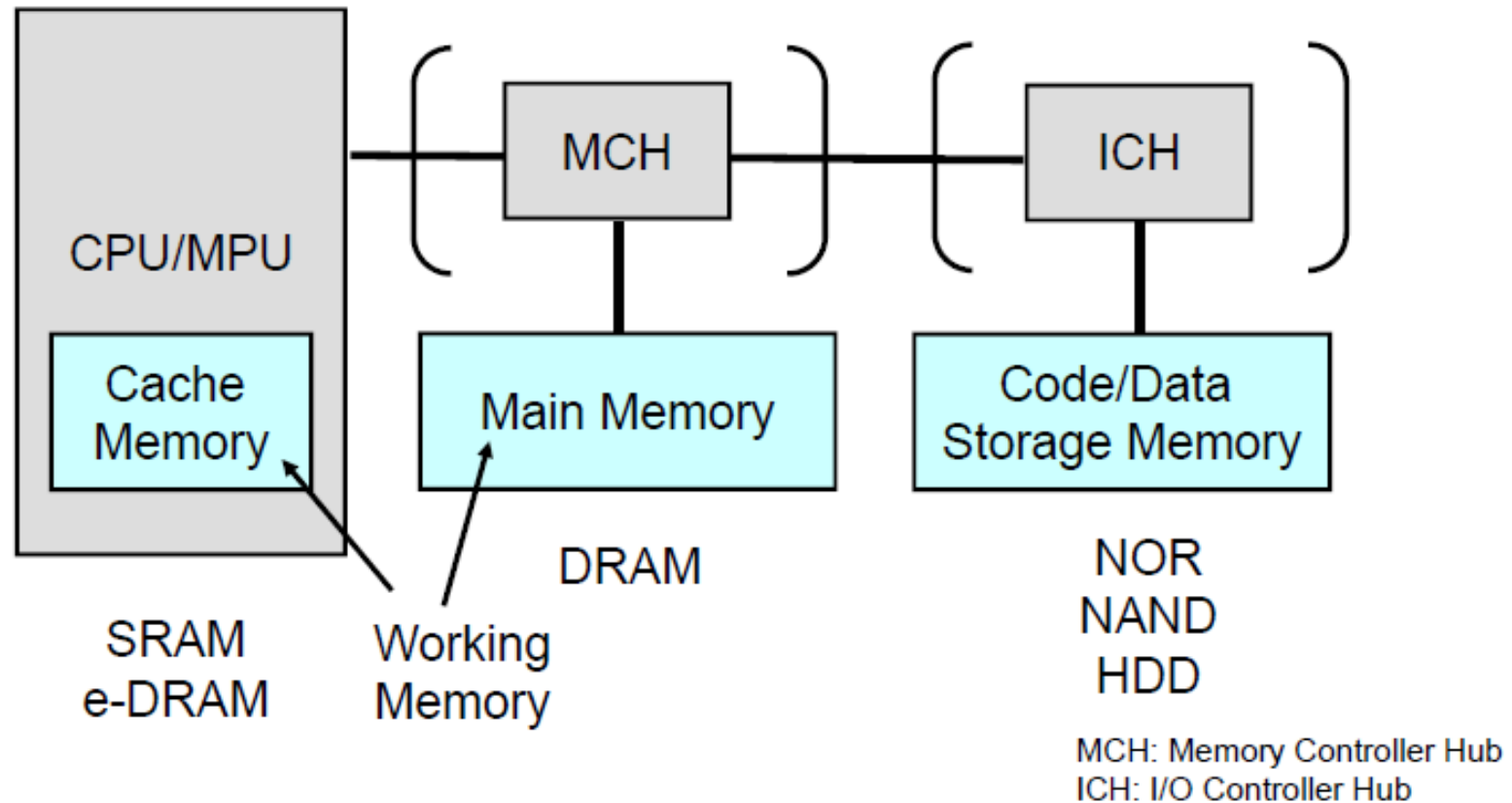
Table 1: Comparison of different memory technologies.

Rainer Waser, Nanoelectronics and Information Technology: Advanced Electronic Materials and Novel Devices, Wiley 2005



Memory: system in a computer

Memory System

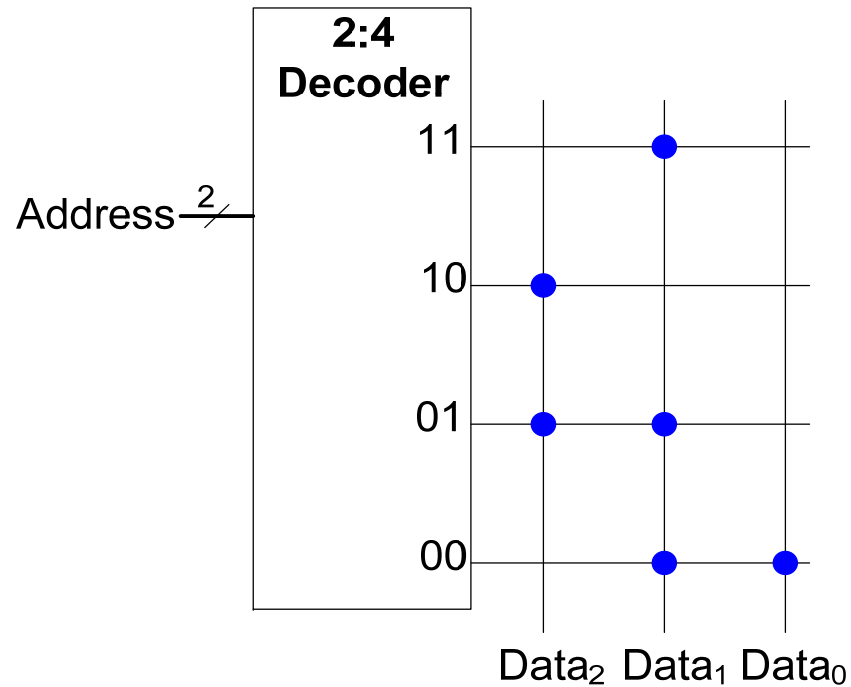


Akihiro Nitayama

IEDM 2009 Short Course • Low Power / Low Energy Circuits: From Device to System Aspects

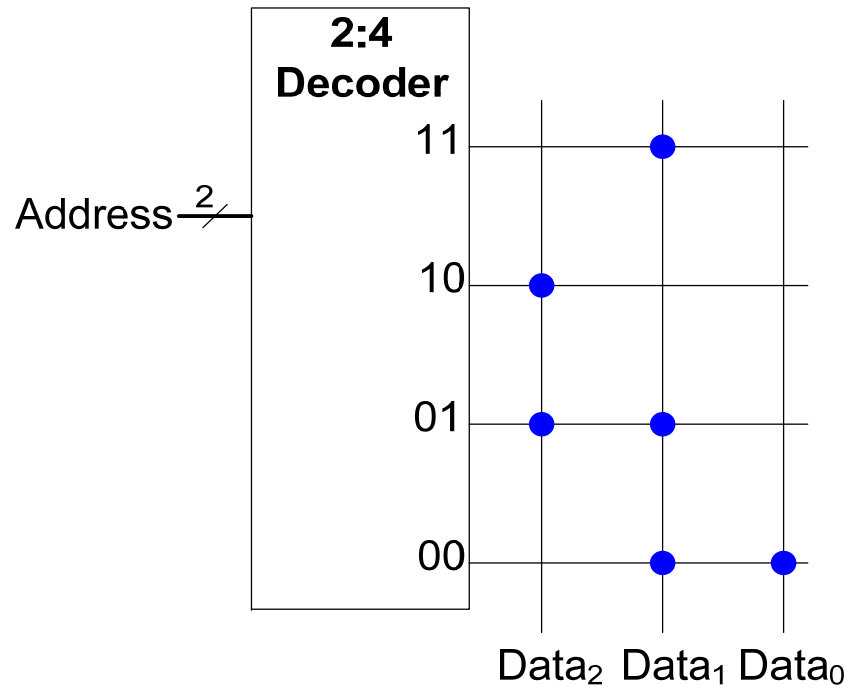


ROM Storage



Address	Data			depth ↑ ↓
11	0	1	0	
10	1	0	0	
01	1	1	0	
00	0	1	1	
			width ←→	

ROM Logic



$$Data_2 = A_1 \oplus A_0$$

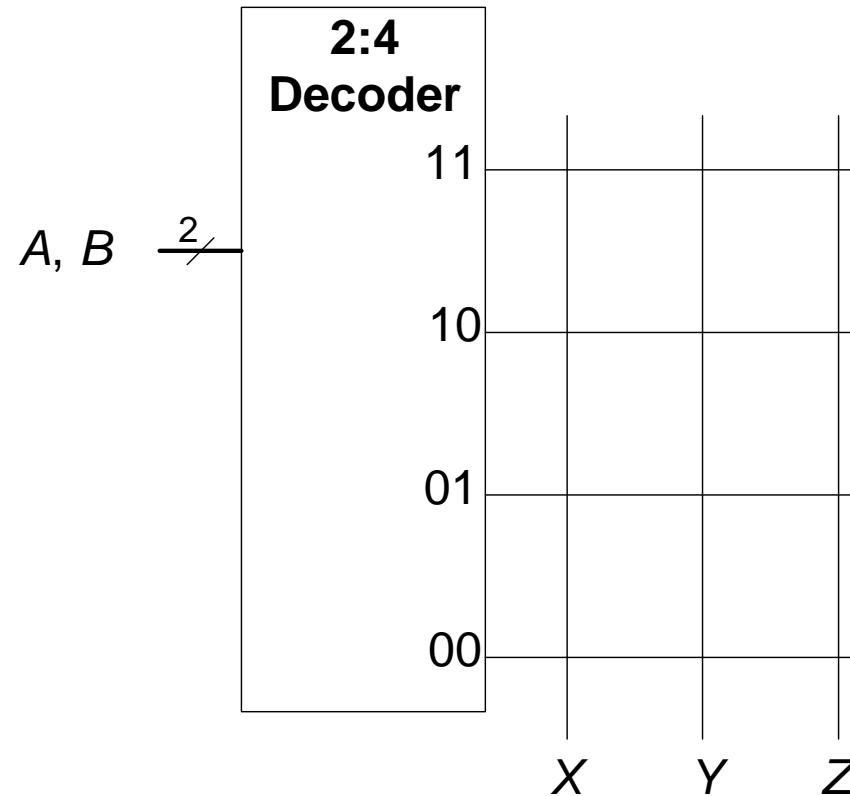
$$Data_1 = \overline{A_1} + A_0$$

$$Data_0 = \overline{A_1} \overline{A_0}$$

Example: Logic with ROMs

Implement the following logic functions using a $2^2 \times 3$ -bit ROM:

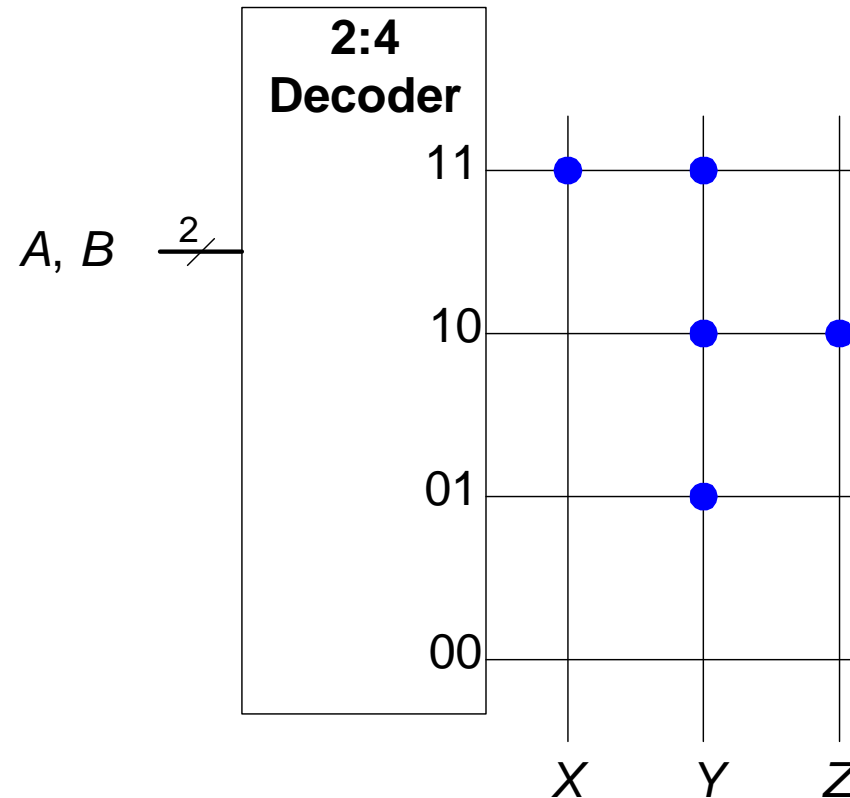
- $X = AB$
- $Y = A + B$
- $Z = A \overline{B}$



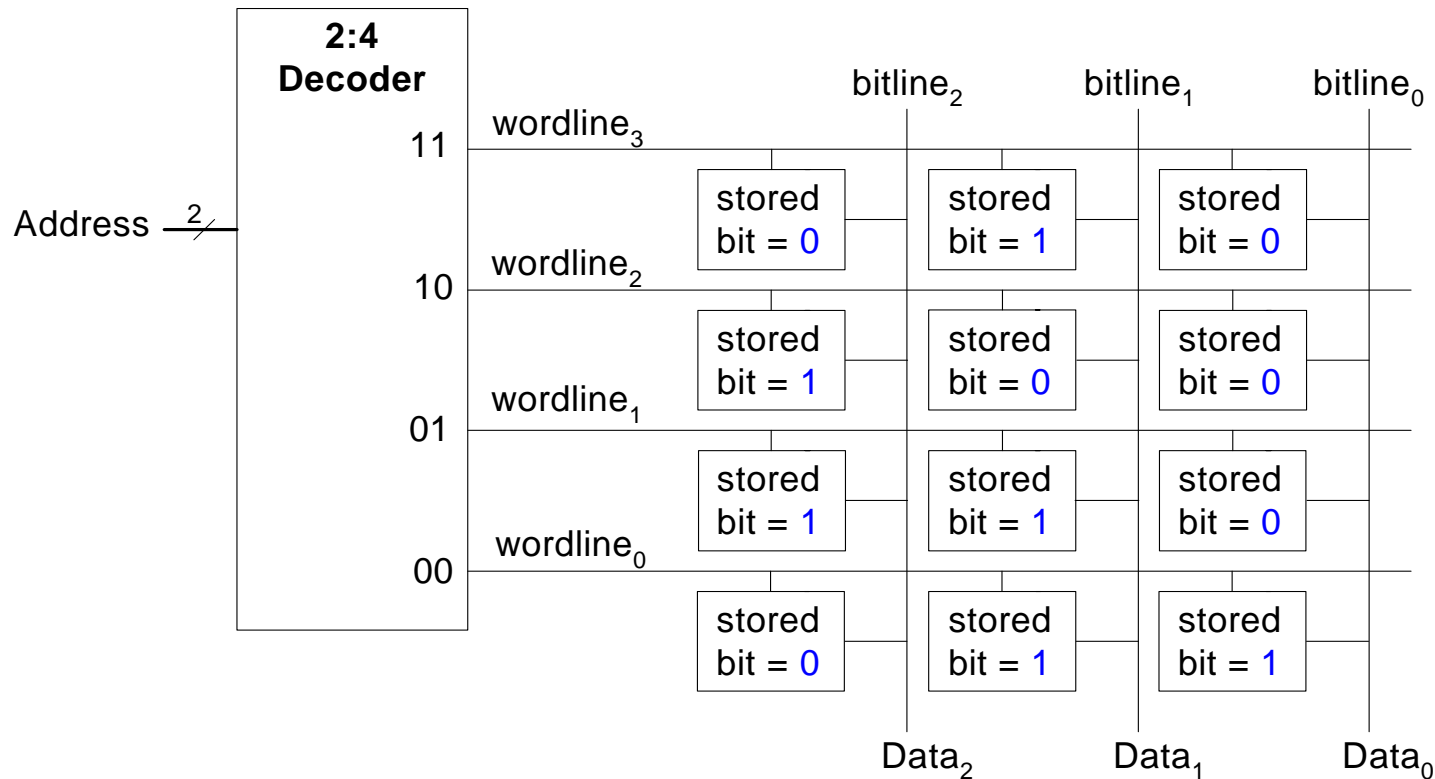
Example: Logic with ROMs

Implement the following logic functions using a $2^2 \times 3$ -bit ROM:

- $X = AB$
- $Y = A + B$
- $Z = A \overline{B}$



Logic with Any Memory Array



$$Data_2 = A_1 \oplus A_0$$

$$Data_1 = \bar{A}_1 + A_0$$

$$Data_0 = \bar{A}_1 \bar{A}_0$$

Logic with Memory Arrays

Implement the following logic functions using a $2^2 \times 3$ -bit memory array:

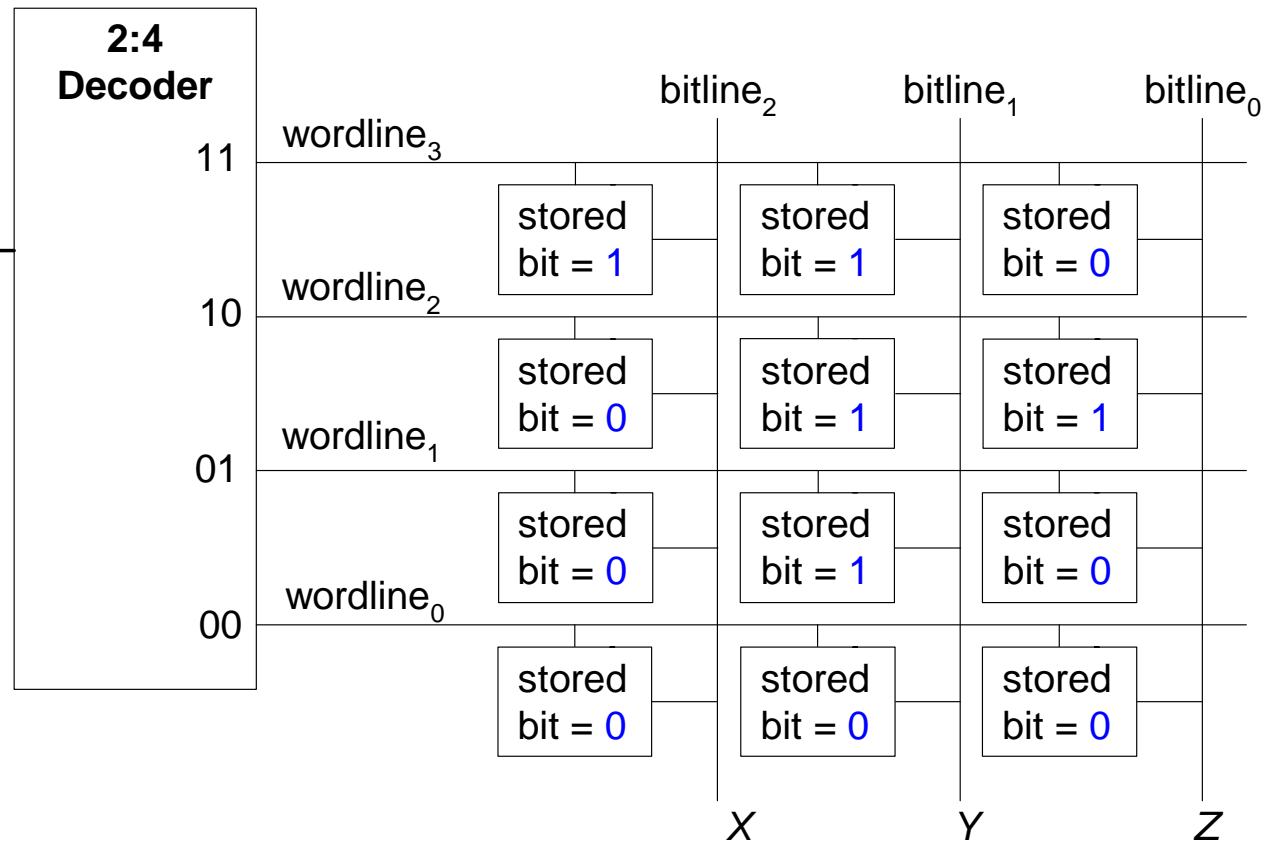
- $X = AB$
- $Y = A + B$
- $Z = A \overline{B}$

Logic with Memory Arrays

Implement the following logic functions using a $2^2 \times 3$ -bit memory array:

- $X = AB$
- $Y = A + B$
- $Z = A \overline{B}$

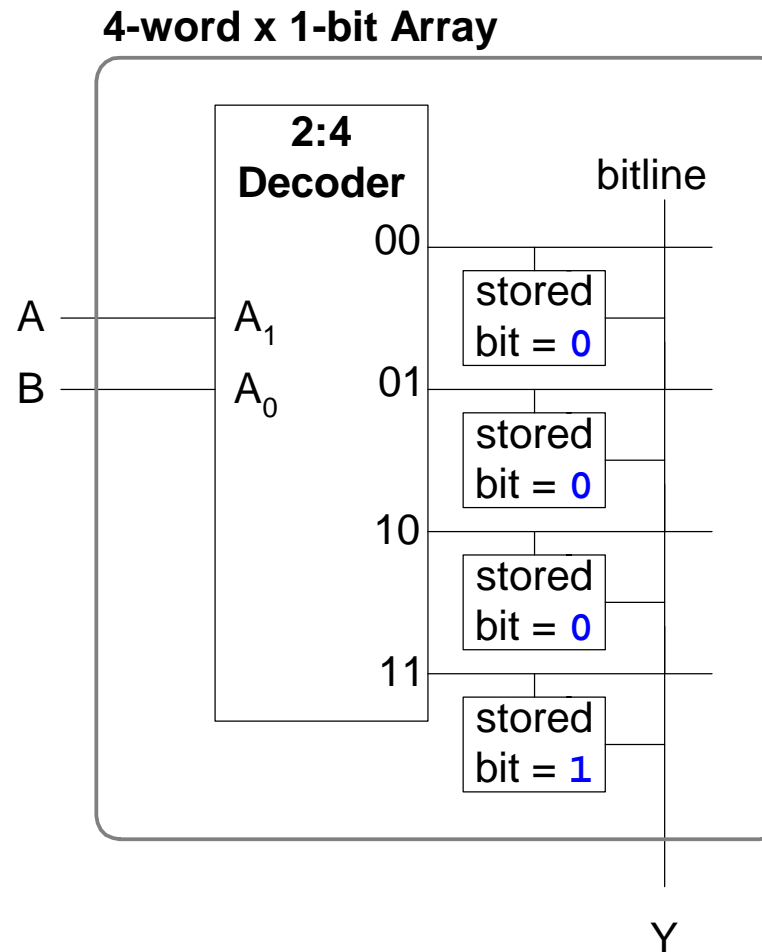
$A, B \xrightarrow{2/}$



Logic with Memory Arrays

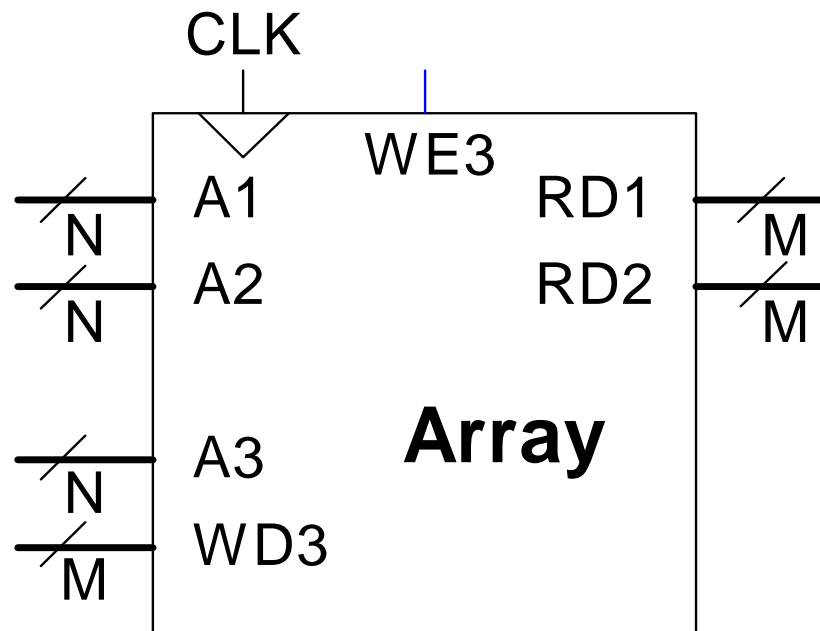
Called *lookup tables* (LUTs): look up output at each input combination (address)

Truth Table		
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1



Multi-ported Memories

- **Port:** address/data pair
- 3-ported memory
 - 2 read ports (A1/RD1, A2/RD2)
 - 1 write port (A3/WD3, WE3 enables writing)
- **Register file:** small multi-ported memory

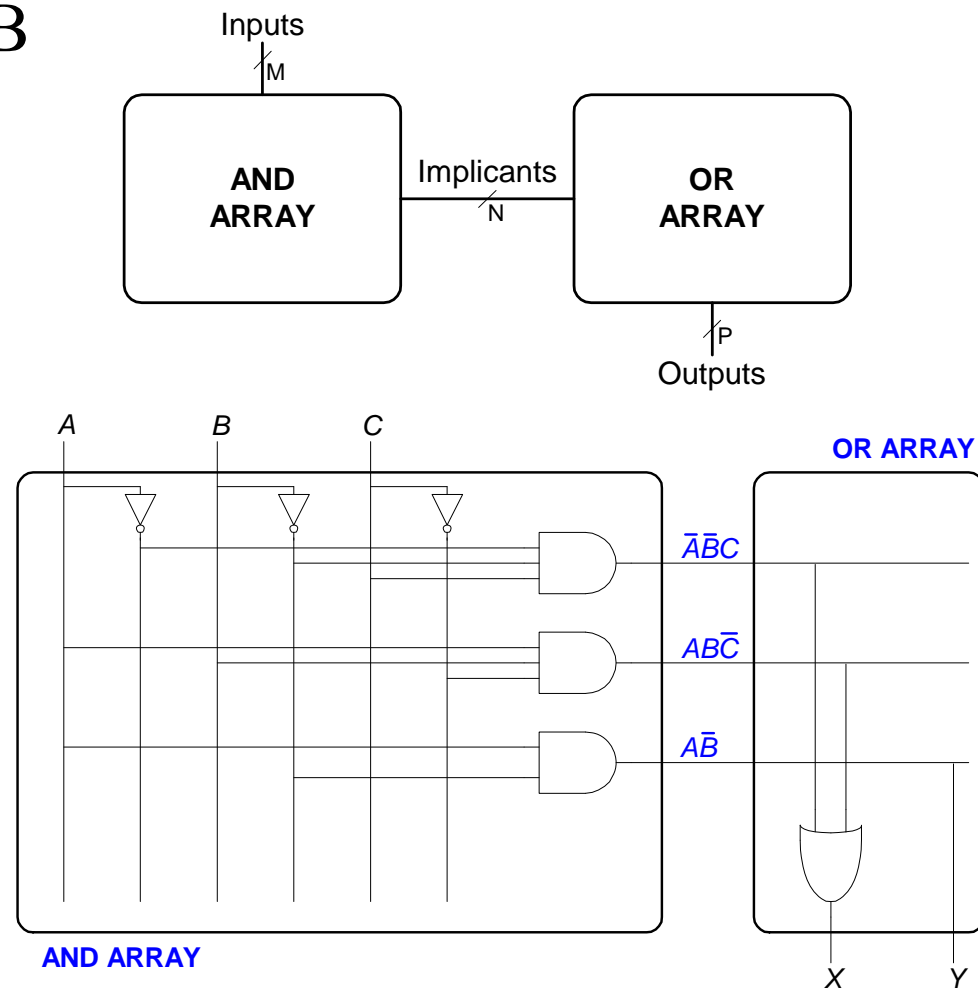


Logic Arrays

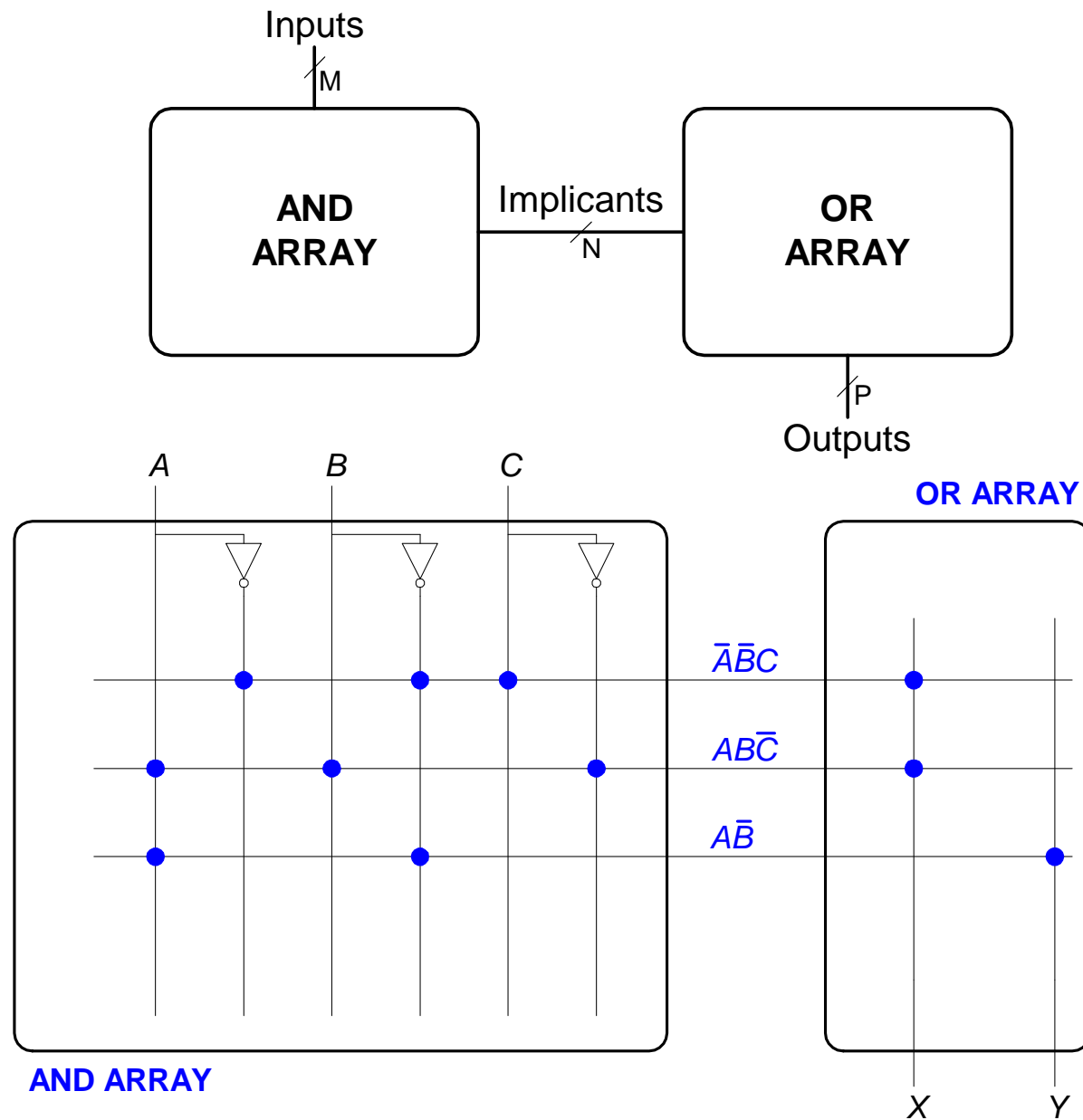
- **PLAs** (Programmable logic arrays)
 - AND array followed by OR array
 - Combinational logic only
 - Fixed internal connections
- **FPGAs** (Field programmable gate arrays)
 - Array of Logic Elements (LEs)
 - Combinational and sequential logic
 - Programmable internal connections

PLAs

- $X = \bar{A}\bar{B}C + A\bar{B}\bar{C}$
- $Y = A\bar{B}$



PLAs: Dot Notation

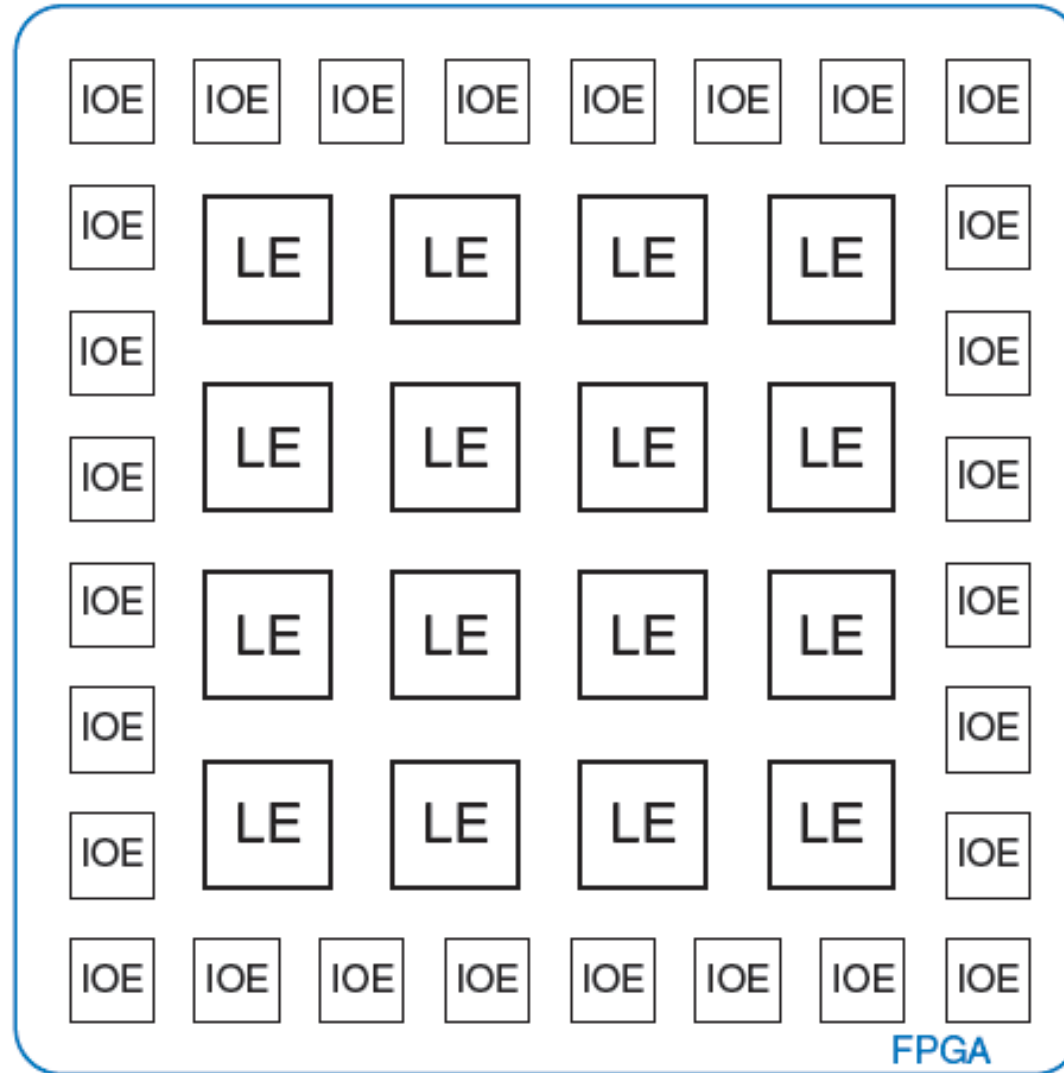


FPGA: Field Programmable Gate Array

- Composed of:
 - **LEs** (Logic elements): perform logic
 - **IOEs** (Input/output elements): interface with outside world
 - **Programmable interconnection:** connect LEs and IOEs
 - Some FPGAs include other building blocks such as multipliers and RAMs



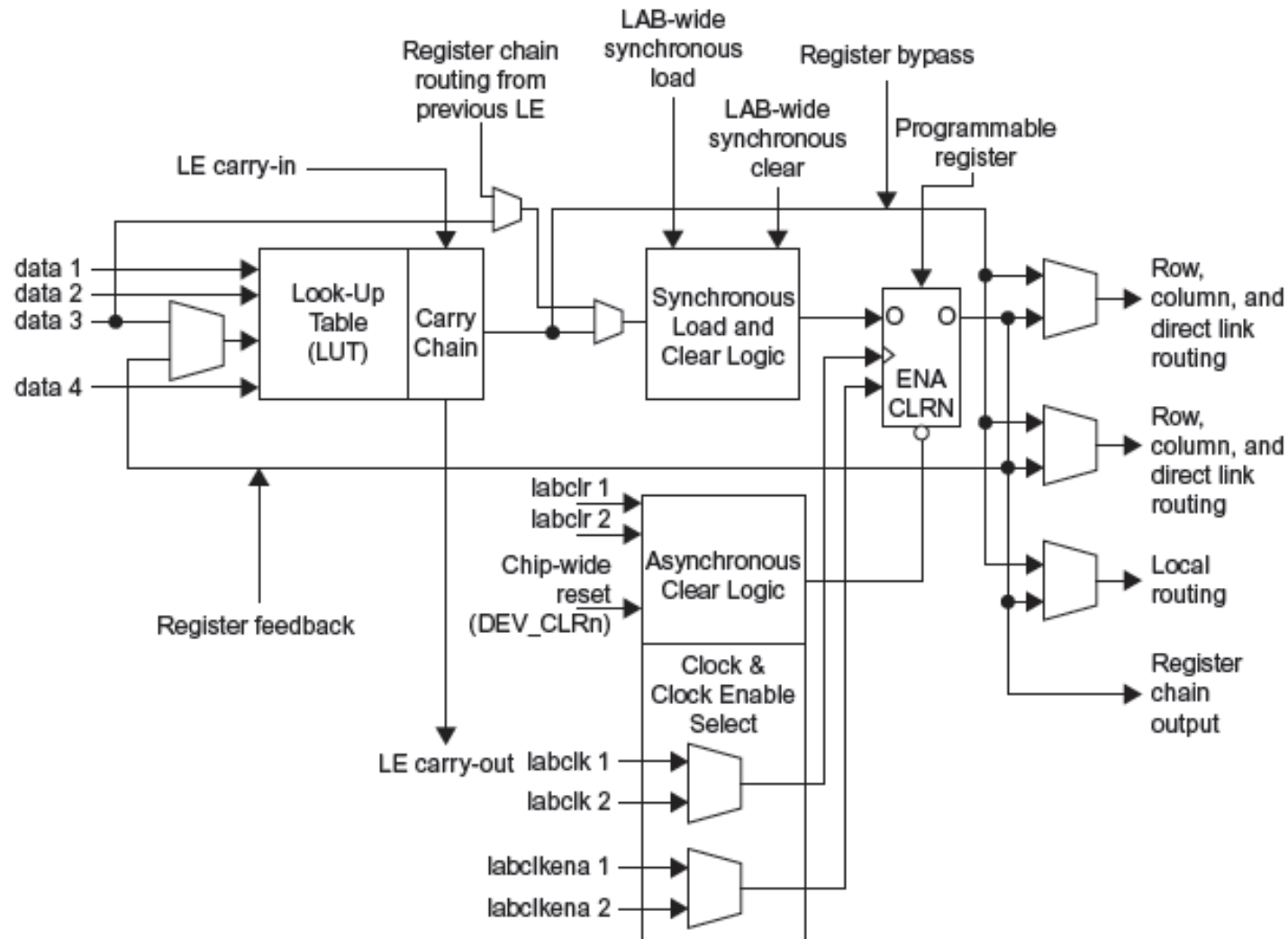
General FPGA Layout



LE: Logic Element

- Composed of:
 - **LUTs** (lookup tables): perform combinational logic
 - **Flip-flops**: perform sequential logic
 - **Multiplexers**: connect LUTs and flip-flops

Altera Cyclone IV LE



Altera Cyclone IV LE

- The Altera Cyclone IV LE has:
 - 1 four-input LUT
 - 1 registered output
 - 1 combinational output

LE Configuration Example

Show how to configure a Cyclone IV LE to perform the following functions:

- $X = \overline{A}\overline{B}C + A\overline{B}\overline{C}$
- $Y = A\overline{B}$

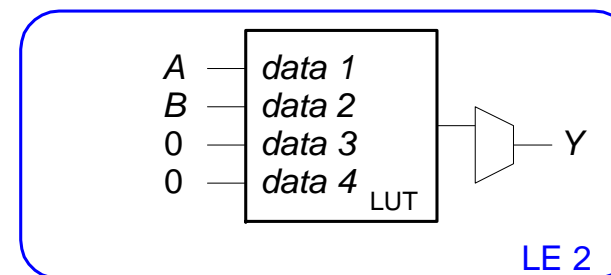
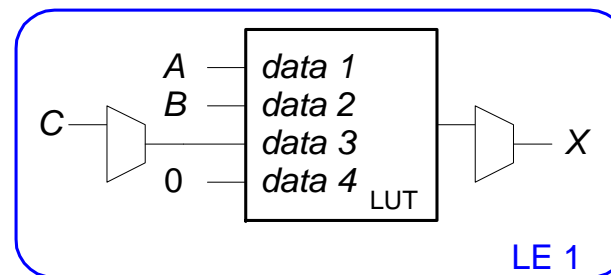
LE Configuration Example

Show how to configure a Cyclone IV LE to perform the following functions:

- $X = \overline{A}\overline{B}C + A\overline{B}\overline{C}$
- $Y = A\overline{B}$

(A) data 1	(B) data 2	(C) data 3	data 4	(X) LUT output
0	0	0	X	0
0	0	1	X	1
0	1	0	X	0
0	1	1	X	0
1	0	0	X	0
1	0	1	X	0
1	1	0	X	1
1	1	1	X	0

(A) data 1	(B) data 2	data 3	data 4	(Y) LUT output
0	0	X	X	0
0	1	X	X	0
1	0	X	X	1
1	1	X	X	0



FPGA Design Flow

Using a CAD tool (such as Altera's Quartus II)

- **Enter the design** using schematic entry or an HDL
- **Simulate** the design
- **Synthesize** design and map it onto FPGA
- **Download the configuration** onto the FPGA
- **Test** the design