

# Digital Design & Computer Architecture




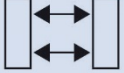
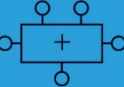
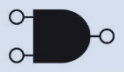
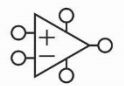


Sarah Harris & David Harris

## Chapter 5: Digital Building Blocks

*Modified by Younghwan Yoo, 2023*

# 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

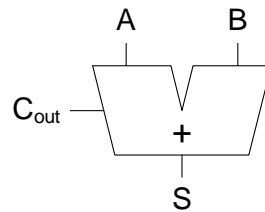
- **Digital building blocks:**
  - Gates, multiplexers, decoders, registers, arithmetic circuits, counters, memory arrays, logic arrays
- **Building blocks demonstrate hierarchy, modularity, and regularity:**
  - Hierarchy of simpler components
  - Well-defined interfaces and functions
  - Regular structure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

# Chapter 5: Digital Building Blocks

## **Adders**

# 1-Bit Adders

**Half Adder**

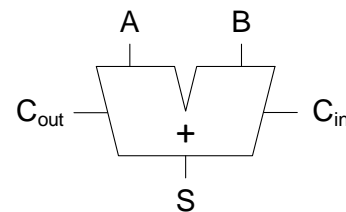


A	B	C <sub>out</sub>	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B$$

$$C_{out} = A \cdot B$$

**Full Adder**



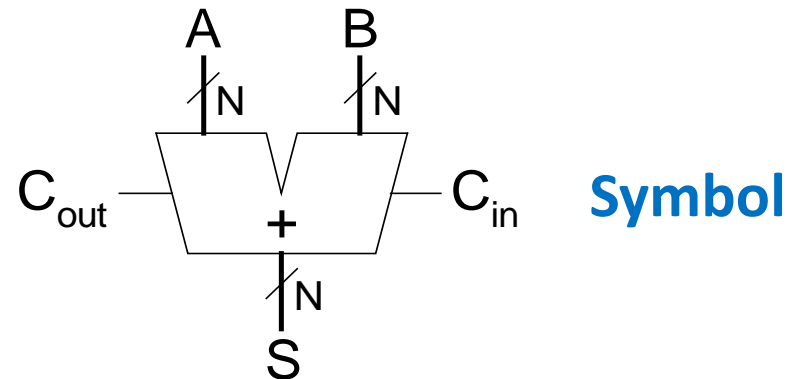
C <sub>in</sub>	A	B	C <sub>out</sub>	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

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

$$C_{out} = A \cdot B + (A \oplus B) \cdot C_{in}$$

# Multibit Adders: CPAs

- Multibit adders



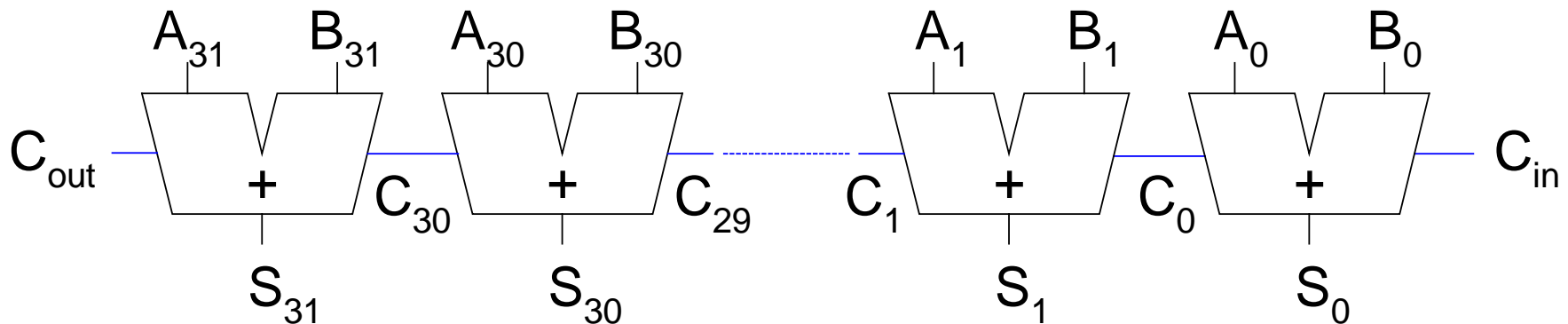
- Types of carry propagate adders (CPAs):
  - **Ripple-carry** (slow)
  - **Carry-lookahead** (fast)
  - **Prefix** (faster)
- Carry-lookahead and prefix adders are faster for large adders but require more hardware

## Chapter 5: Digital Building Blocks

# **Ripple Carry Addition**

# Ripple-Carry Adder

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

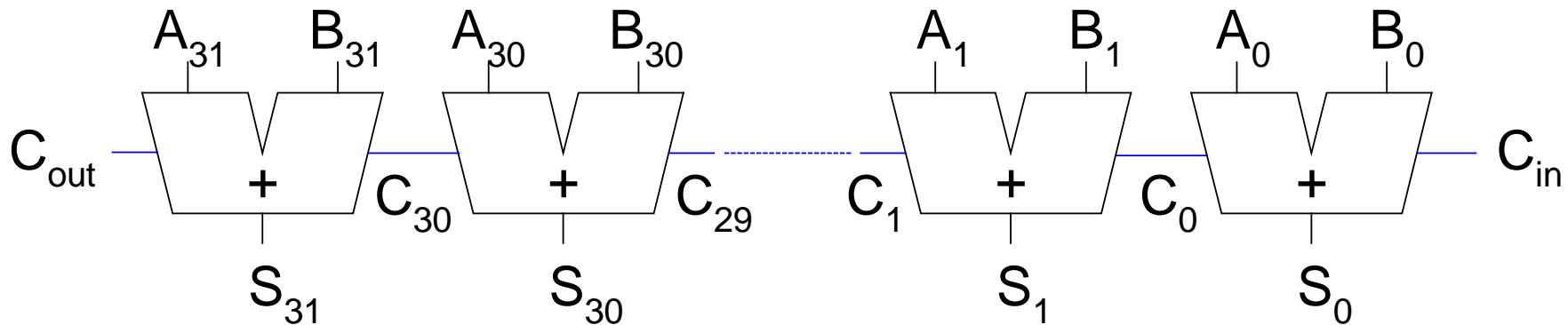




# Ripple-Carry Adder Delay

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

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



## Chapter 5: Digital Building Blocks

# **Carry Lookahead Addition**

# Carry-Lookahead Adder

Compute  $C_{\text{out}}$  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
- Calculate generate ( $G_i$ ) and propagate ( $P_i$ ) signals for each column:
  - **Generate:** Column  $i$  will generate a carry out if  $A_i$  **and**  $B_i$  are both 1.

$$G_i = A_i B_i$$

- **Propagate:** Column  $i$  will propagate a carry in to the carry out if  $A_i$  **or**  $B_i$  is 1.

$$P_i = A_i + B_i$$

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

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

# Propagate and Generate Signals

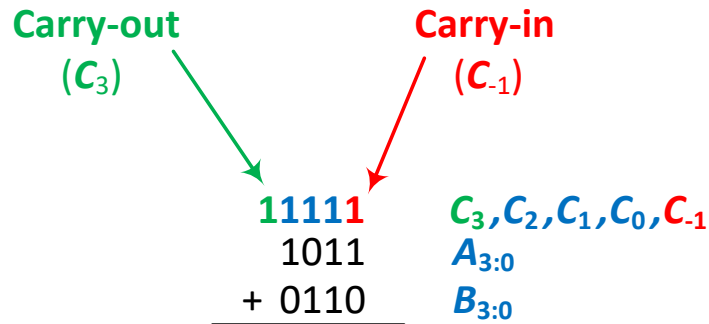
**Examples:** Column propagate and generate signals:

Column propagate:

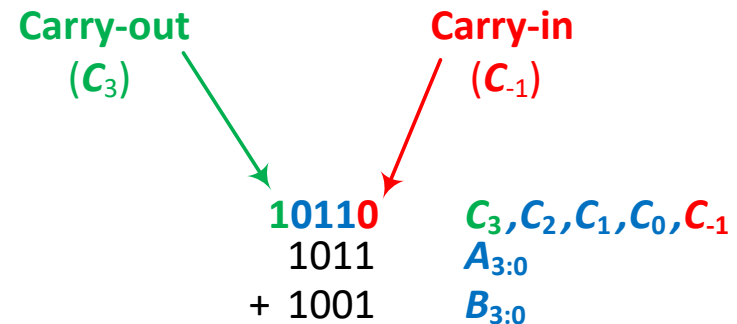
$$P_i = A_i + B_i$$

Column generate:

$$G_i = A_i B_i$$



1111	$P_3, P_2, P_1, P_0$
0010	$G_3, G_2, G_1, G_0$



1011	$P_3, P_2, P_1, P_0$
1001	$G_3, G_2, G_1, G_0$

$$C_i = G_i + P_i C_{i-1}$$

# Block Propagate and Generate

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

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

# Block Propagate and Generate

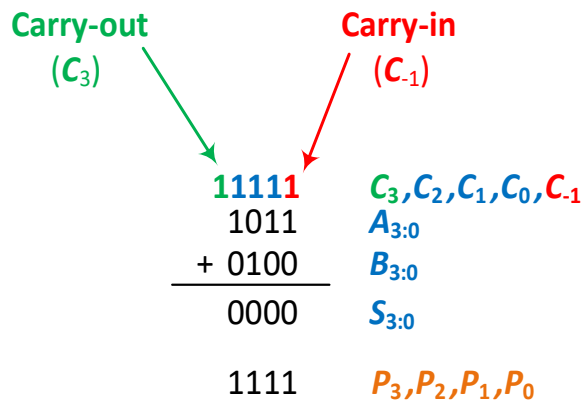
- **Example: 4-bit blocks**

- **Block propagate signal:  $P_{3:0}$**  (single-bit signal)

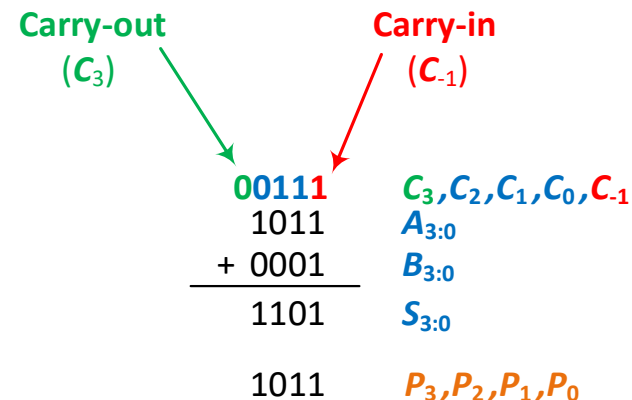
- A carry-in would propagate through all 4 bits of the block:

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

- **Examples:**



$$P_{3:0} = P_3 P_2 P_1 P_0 = 1$$



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

# Block Propagate and Generate

- **Example: 4-bit blocks**

- **Block propagate signal:  $P_{3:0}$**  (single-bit signal)

- A carry-in would propagate through all 4 bits of the block:

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

- **Block generate signal:  $G_{3:0}$**  (single-bit signal)

- A carry is generated:
  - in column 3, **or**
  - in column 2 and propagated through column 3, **or**
  - in column 1 and propagated through columns 2 and 3, **or**
  - in column 0 and propagated through columns 1-3

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

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

# Block Propagate and Generate

- **Example: 4-bit blocks**

- **Block generate signal:  $G_{3:0}$**  (single-bit signal)

- A carry is: generated in column 3, **or** generated in column 2 and propagated through column 3, **or** ...

$$G_{3:0} = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3$$

Carry-out ( $C_3$ )		Carry-out ( $C_3$ )		Carry-out ( $C_3$ )	
↓		↓		↓	
<b>10011</b>	$C_3, C_2, C_1, C_0, C_{-1}$	<b>11000</b>	$C_3, C_2, C_1, C_0, C_{-1}$	<b>01101</b>	$C_3, C_2, C_1, C_0, C_{-1}$
1001	$A_{3:0}$	1110	$A_{3:0}$	0110	$A_{3:0}$
+ 1100	$B_{3:0}$	+ 0100	$B_{3:0}$	+ 0010	$B_{3:0}$
0110	$S_{3:0}$	0010	$S_{3:0}$	1000	$S_{3:0}$
1101	$P_3, P_2, P_1, P_0$	1110	$P_3, P_2, P_1, P_0$	0110	$P_3, P_2, P_1, P_0$
1000	$G_3, G_2, G_1, G_0$	0100	$G_3, G_2, G_1, G_0$	0010	$G_3, G_2, G_1, G_0$
$G_{3:0} = 1$		$G_{3:0} = 1$		$G_{3:0} = 0$	



# Block Propagate and Generate

- **Example: 4-bit blocks**

- **Block propagate signal:  $P_{3:0}$**  (single-bit signal)

- A carry-in would propagate through all 4 bits of the block:

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

- **Block generate signal:  $G_{3:0}$**  (single-bit signal)

- A carry is generated:

- in column 3, **or**
- in column 2 and propagated through column 3, **or**
- in column 1 and propagated through columns 2 and 3, **or**
- in column 0 and propagated through columns 1-3

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

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

$$\boxed{C_3 = G_{3:0} + P_{3:0} C_{-1}}$$

# Block Propagate and Generate

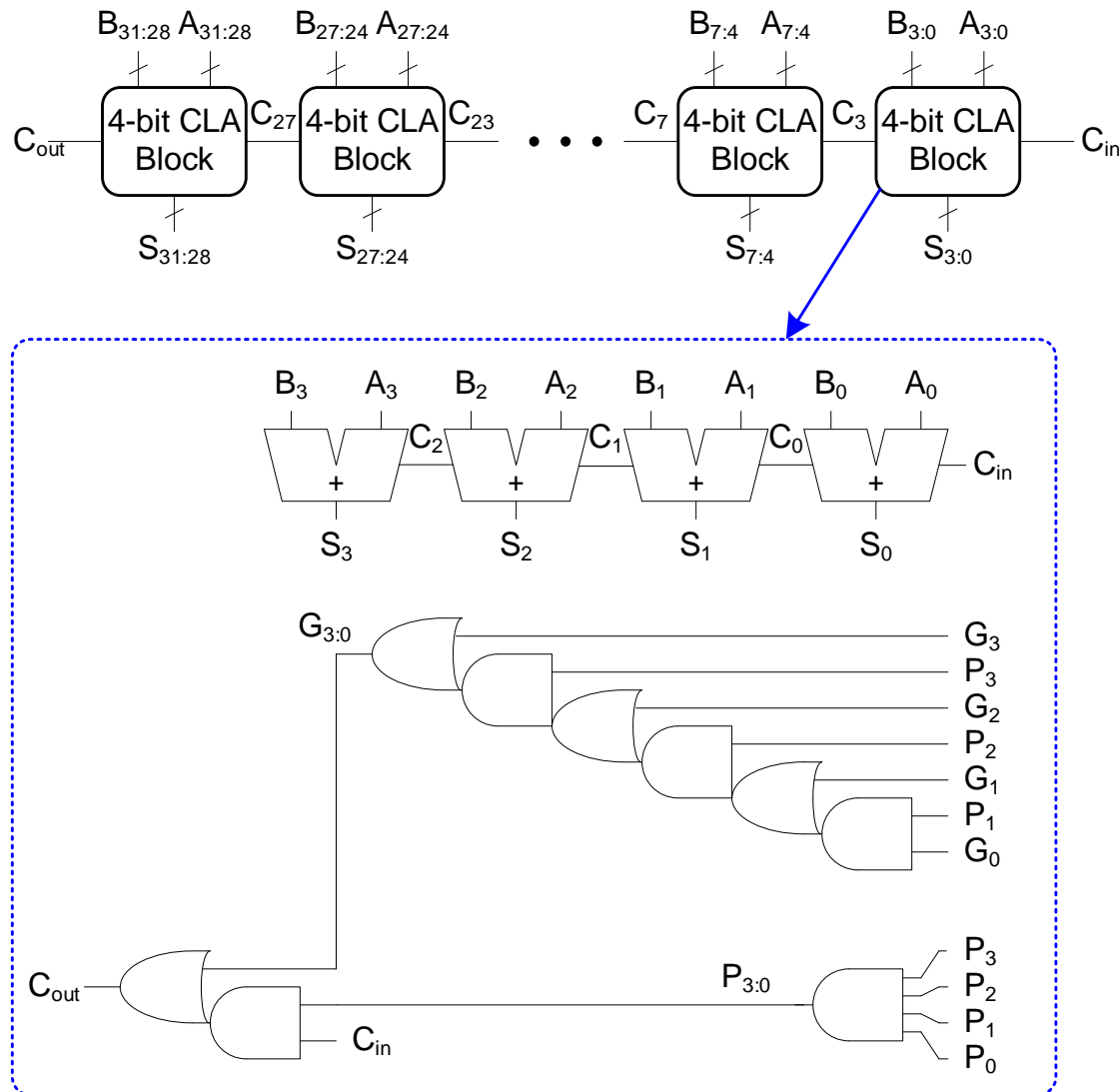
- **Example:** Block propagate and generate signals for 4-bit blocks ( $P_{3:0}$  and  $G_{3:0}$ ):

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

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

$$C_3 = G_{3:0} + P_{3:0} C_{-1}$$

# 32-bit CLA with 4-bit Blocks



# 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 logic (meanwhile computing sums)
- **Step 4:** Compute sum for most significant  $k$ -bit block

# Carry-Lookahead Addition

- **Step 1:** Compute  $G_i$  and  $P_i$  for all columns

$$G_i = A_i B_i$$

$$P_i = A_i + B_i$$

# Carry-Lookahead Addition

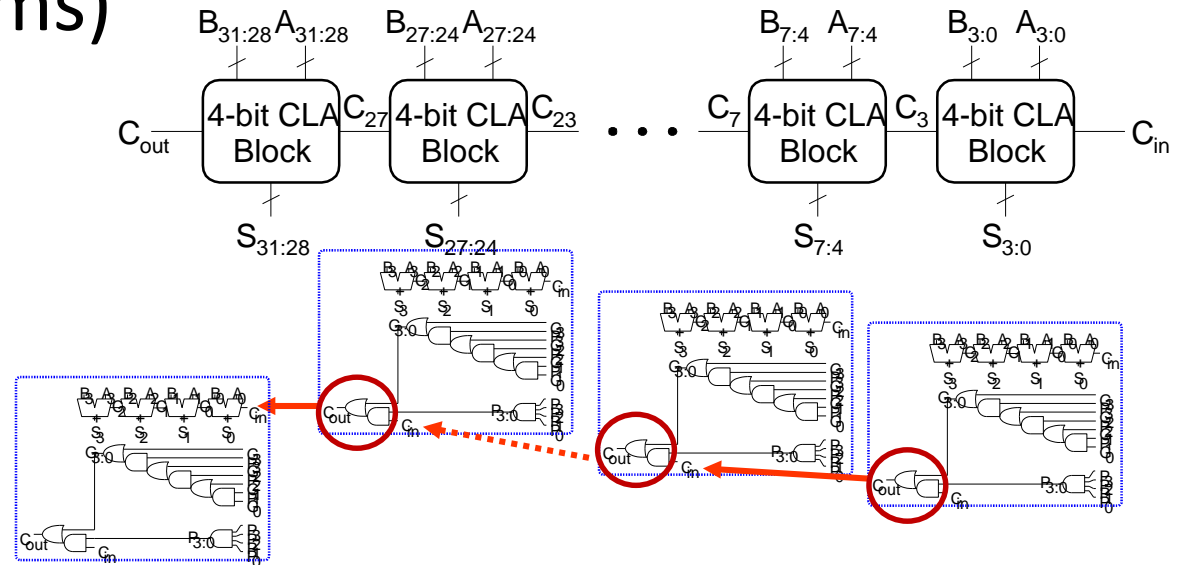
- **Step 1:** Compute  $G_i$  and  $P_i$  for all columns
- **Step 2:** Compute  $G$  and  $P$  for  $k$ -bit blocks

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

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

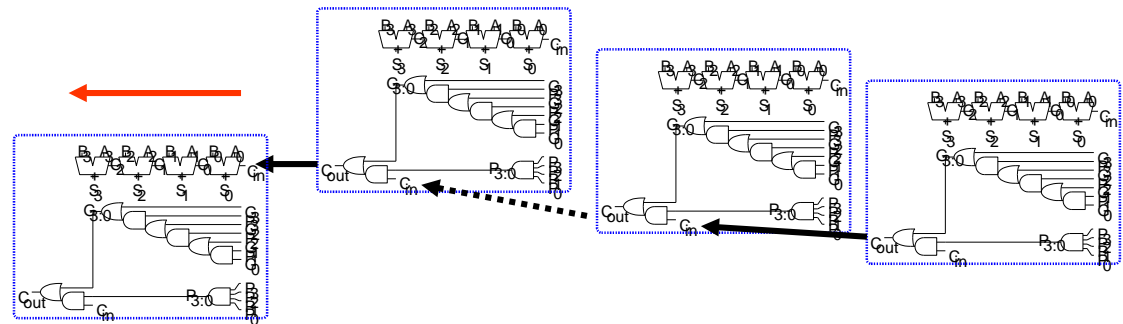
# Carry-Lookahead Addition

- **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 logic (meanwhile computing sums)



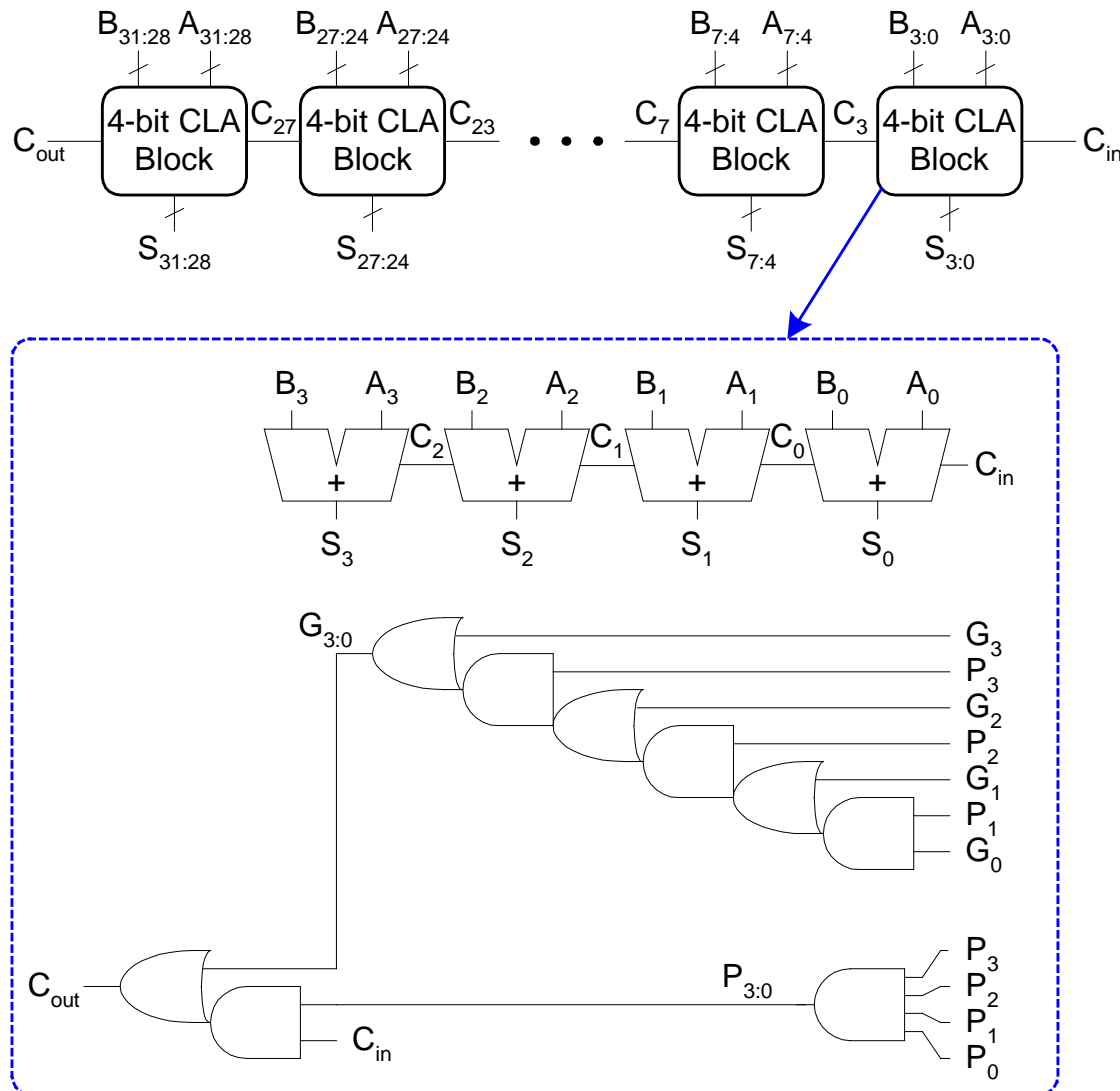
# Carry-Lookahead Addition

- **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 logic (meanwhile computing sums)
- **Step 4:** Compute sum for most significant  $k$ -bit block





# 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  $N > 16$

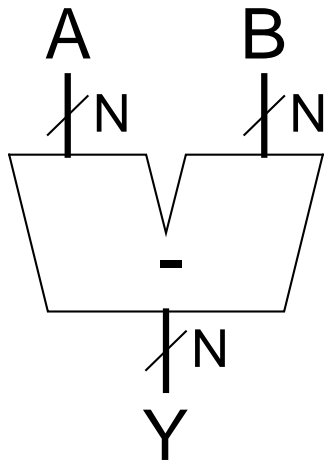
# Chapter 5: Digital Building Blocks

## **Subtractors & Comparators**

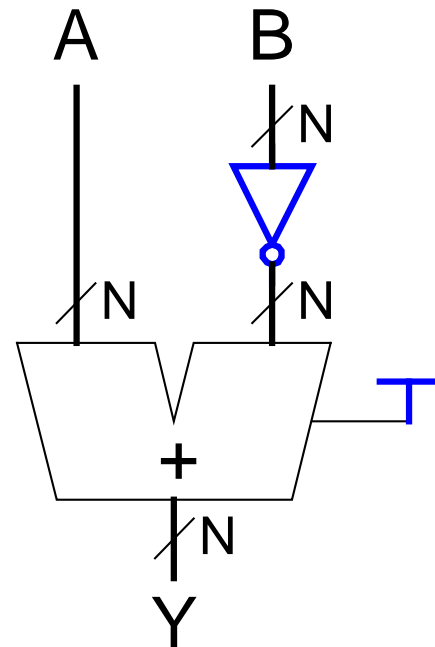
# Subtractor

$$A - B = A + \overline{B} + 1$$

## Symbol

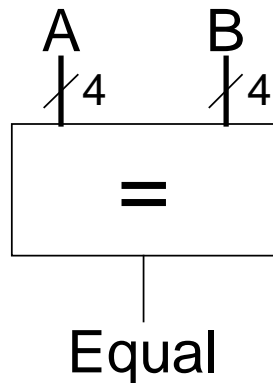


## Implementation

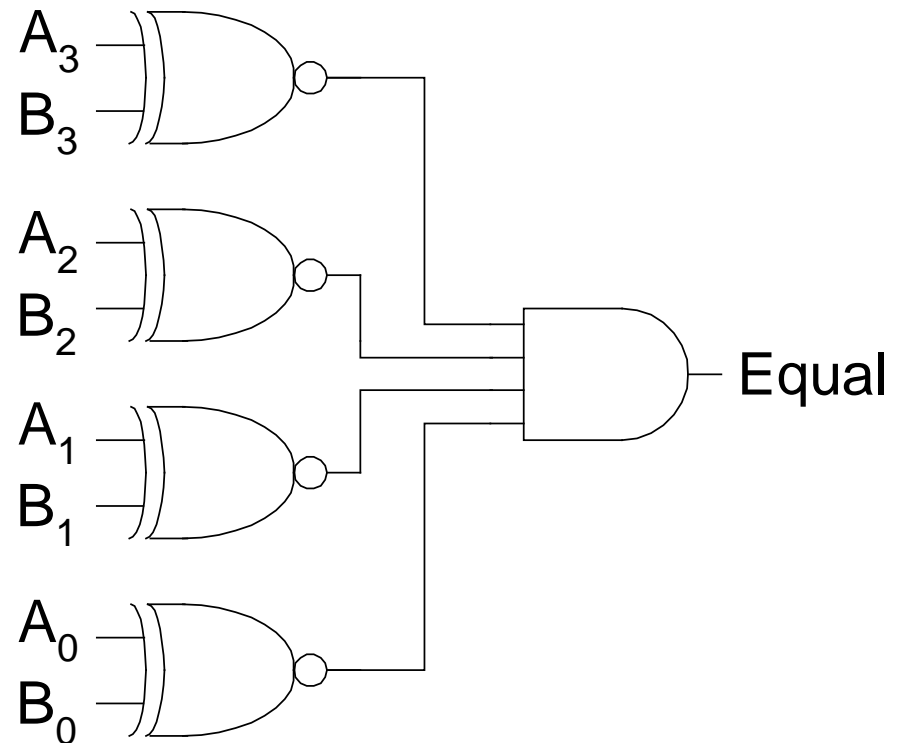


# Comparator: Equality

## Symbol



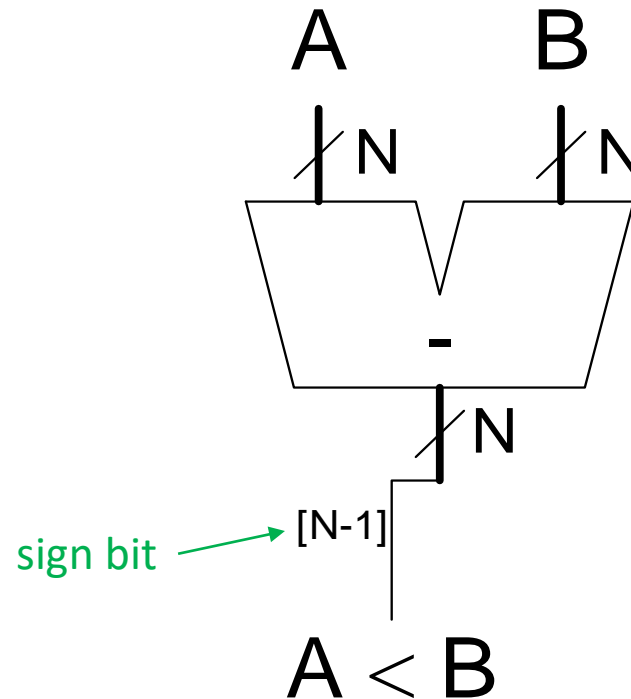
## Implementation



# Comparator: Signed Less Than

**$A < B$  if  $A - B$  is negative**

**Beware of overflow**



# Chapter 5: Digital Building Blocks

**ALU:**

**Arithmetic Logic Unit**

# ALU: Arithmetic Logic Unit

**ALU should perform:**

- Addition
- Subtraction
- AND
- OR



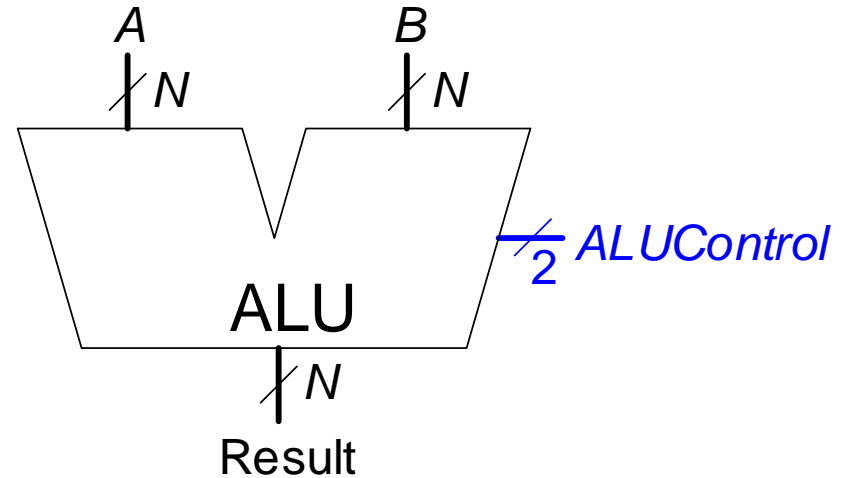
# ALU: Arithmetic Logic Unit

ALUControl <sub>1:0</sub>	Function
00	Add
01	Subtract
10	AND
11	OR

**Example: Perform  $A \text{ OR } B$**

$ALUControl_{1:0} = 11$

**Result =  $A \text{ OR } B$**



# ALU: Arithmetic Logic Unit

ALUControl <sub>1:0</sub>	Function
00	Add
01	Subtract
10	AND
11	OR

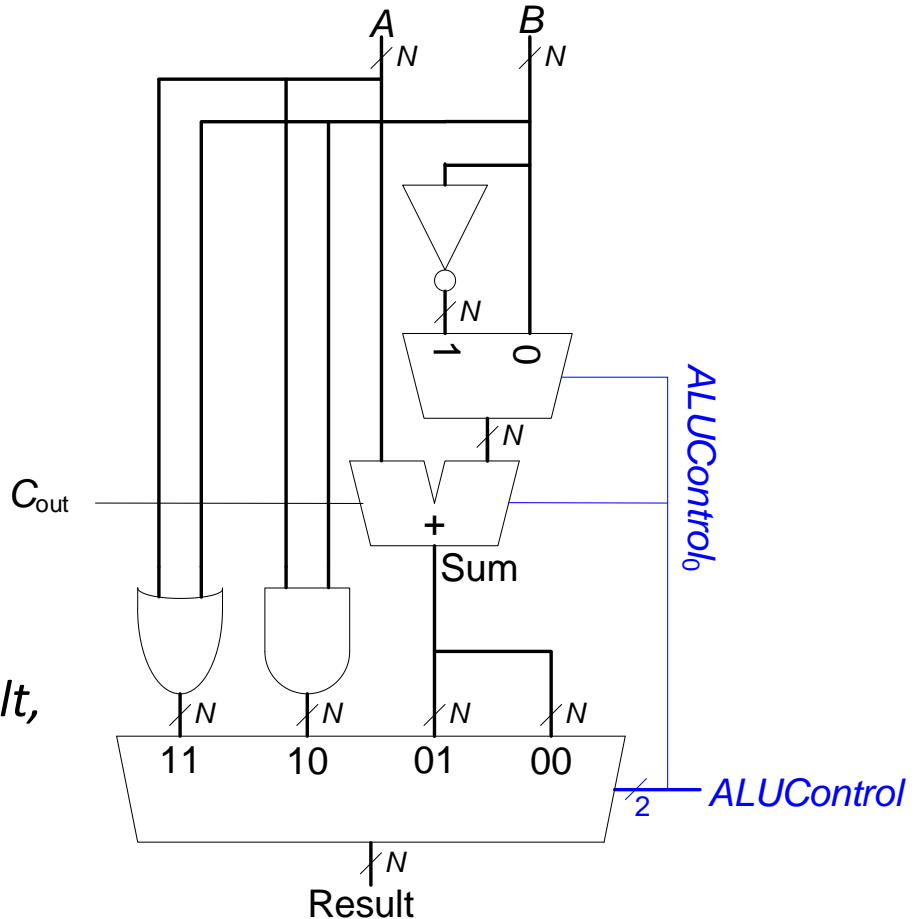
## Example: Perform $A \text{ OR } B$

$ALUControl_{1:0} = 11$

Mux selects output of OR gate as *Result*,

so:

**$Result = A \text{ OR } B$**



# ALU: Arithmetic Logic Unit

ALUControl <sub>1:0</sub>	Function
00	Add
01	Subtract
10	AND
11	OR

**Example: Perform  $A + B$**

$ALUControl_{1:0} = 00$

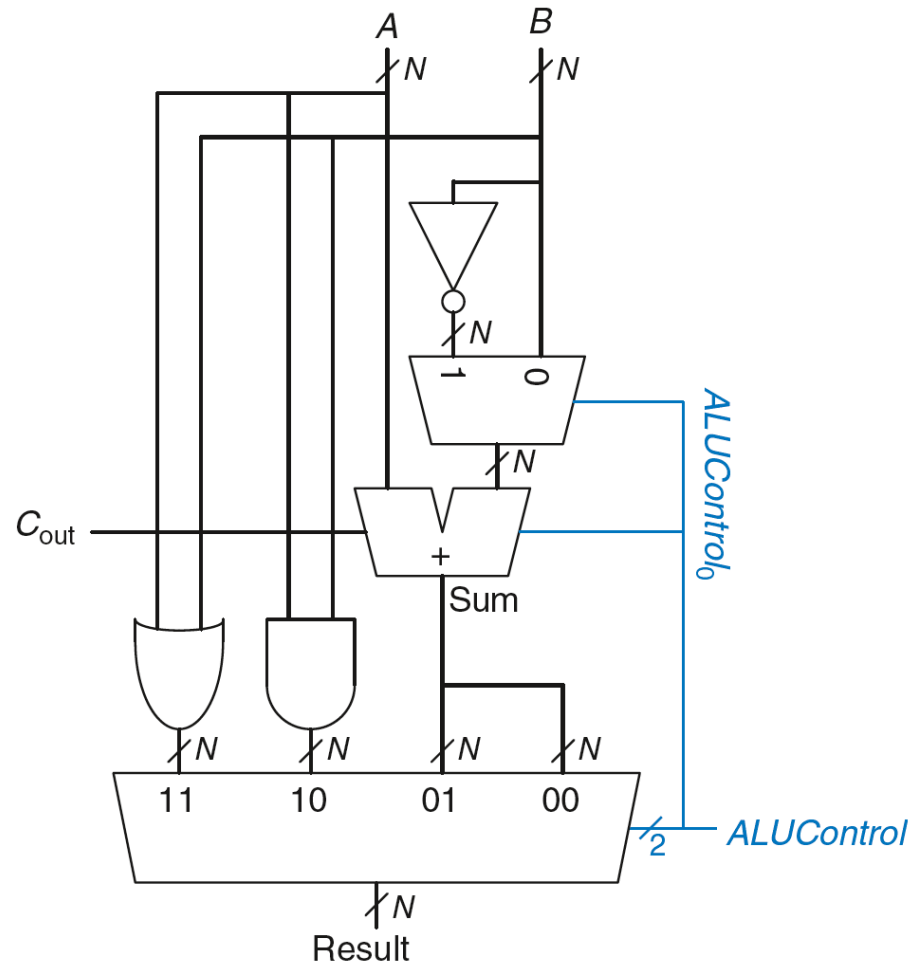
$ALUControl_0 = 0$ , so:

$C_{in}$  to adder = 0

2<sup>nd</sup> input to adder is  $B$

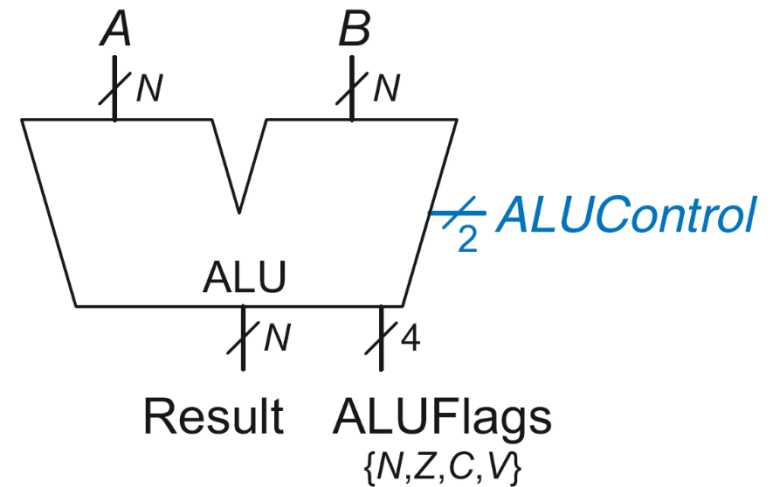
Mux selects *Sum* as *Result*, so

**$Result = A + B$**

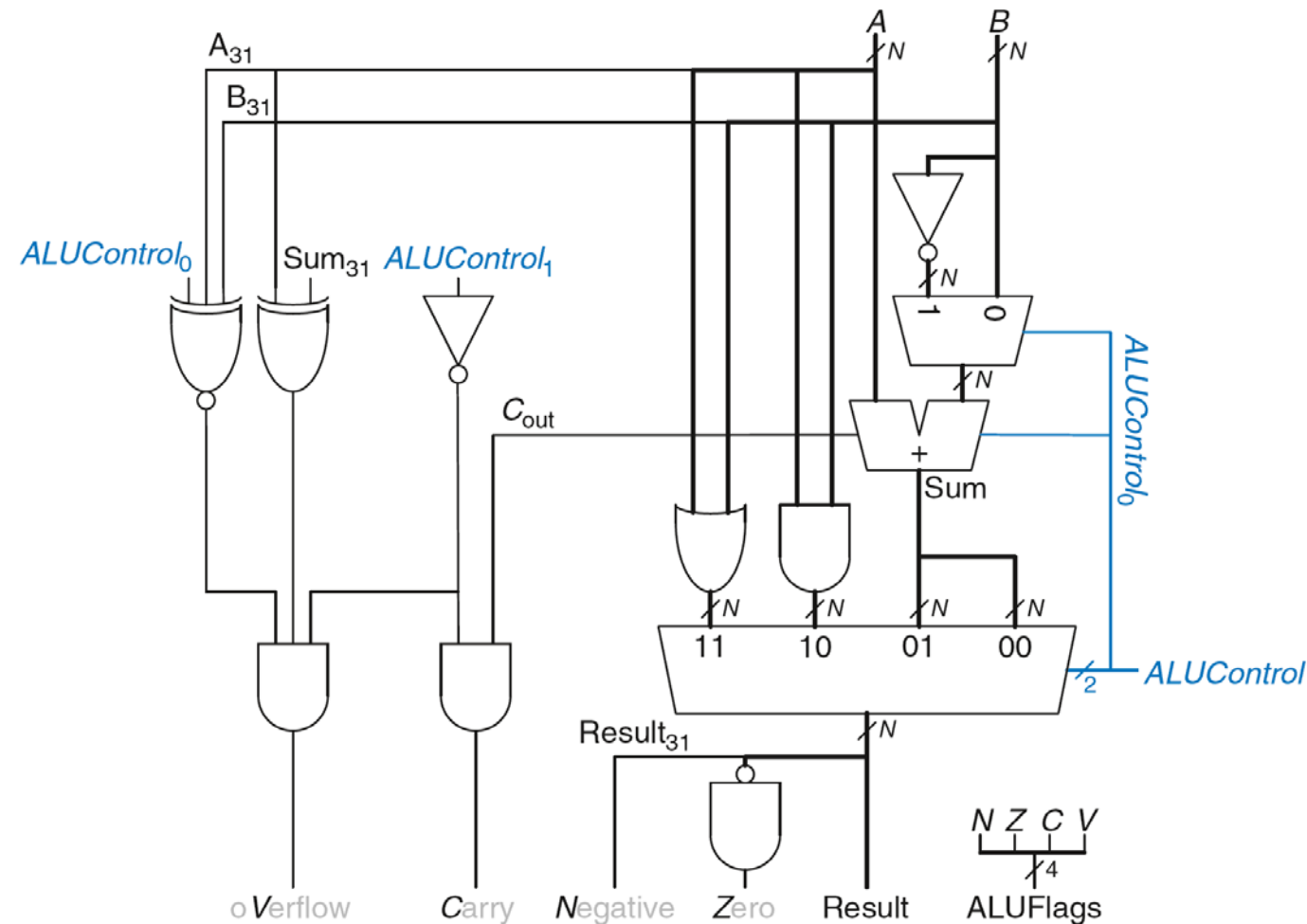


# ALU with Status Flags

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



# ALU with Status Flags

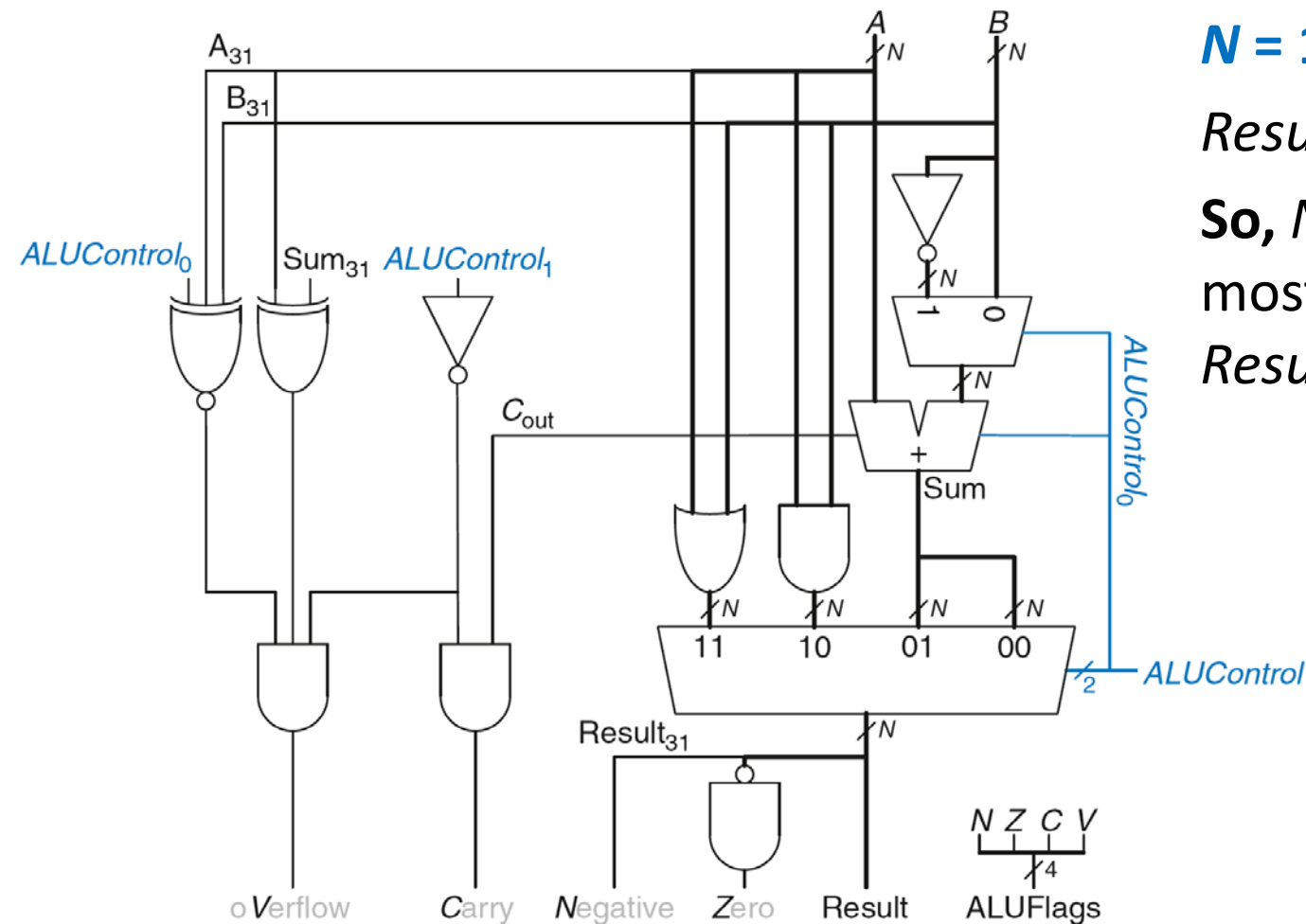


# ALU with Status Flags: **N**egative

**N** = 1 if:

*Result* is **negative**

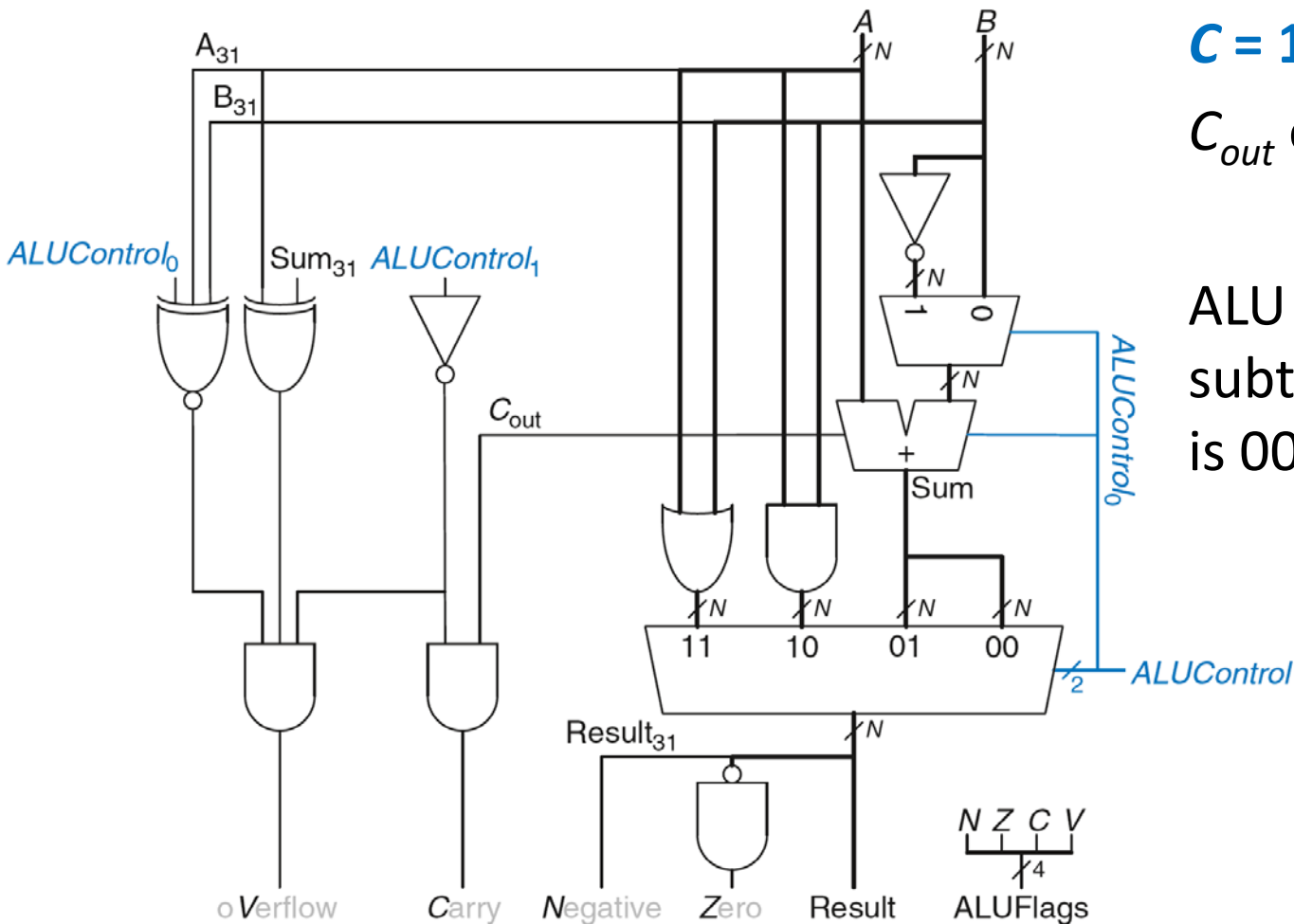
**So**, *N* is connected to most significant bit of *Result*.



# ALU with Status Flags: Zero



# ALU with Status Flags: Carry



**$C = 1$  if:**

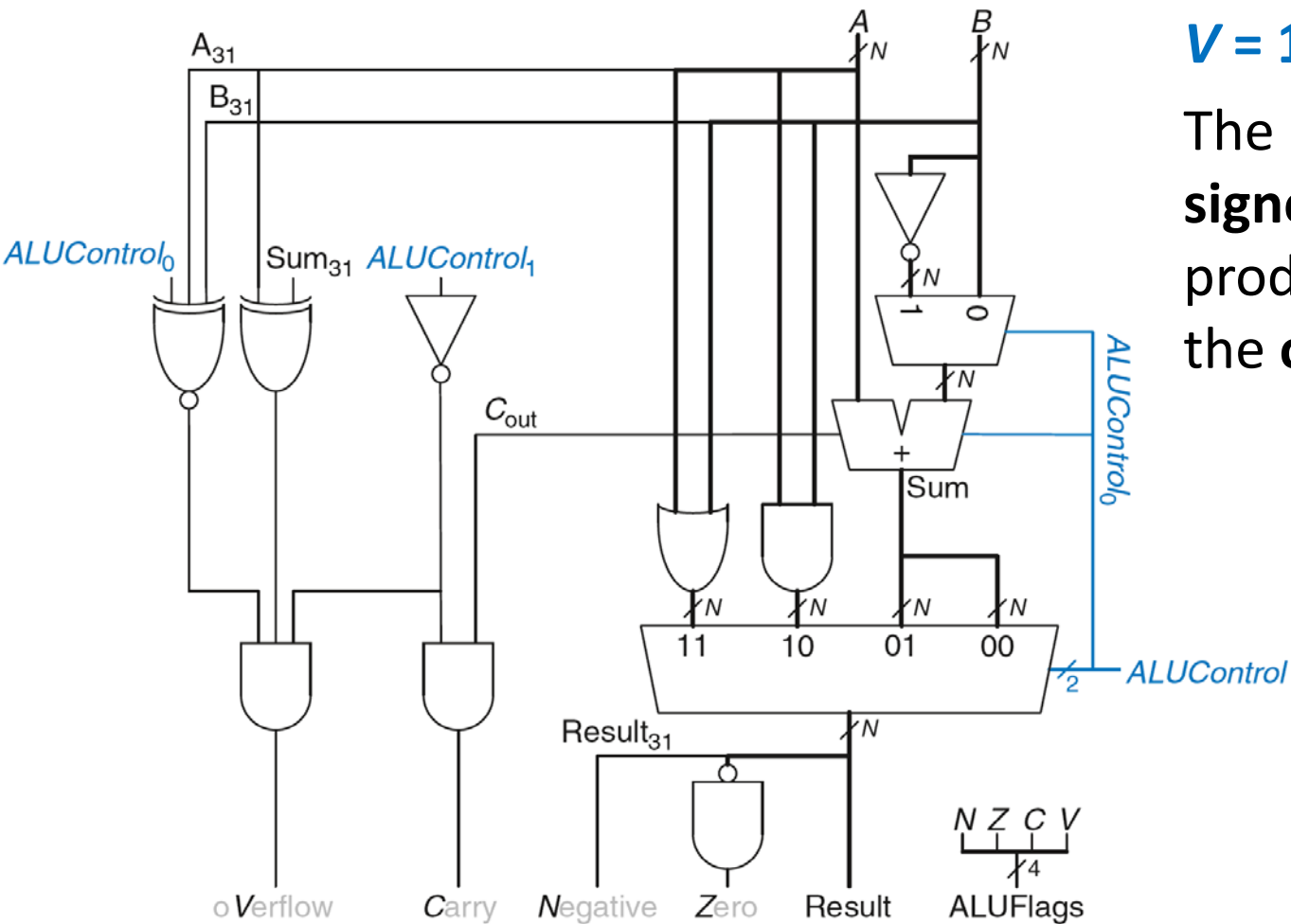
$C_{out}$  of Adder is 1

**AND**

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



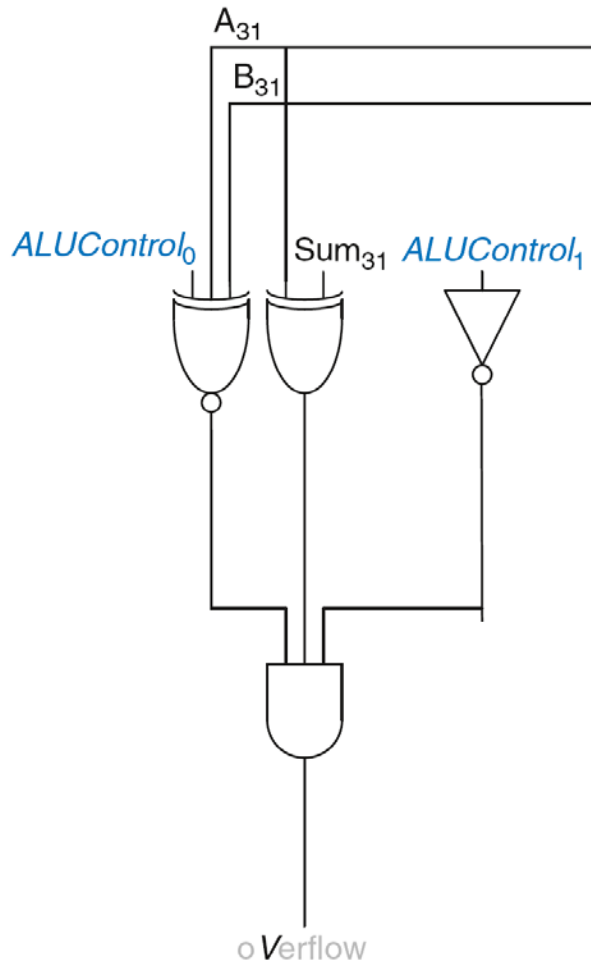
# ALU with Status Flags: o**v**erflow



**$V = 1$  if:**

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

# ALU with Status Flags: o**V**erflow



**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 for addition  
( $ALUControl_0 = 0$ )

**OR**

A and B have different signs for subtraction  
( $ALUControl_0 = 1$ )

# Comparison based on Flags

Compare by subtracting and checking flags

Different for signed and unsigned

Comparison	Signed	Unsigned
==	Z	Z
!=	$\sim Z$	$\sim Z$
<	$N \wedge V$	$\sim C$
<=	$Z \mid (N \wedge V)$	$Z \mid \sim C$
>	$\sim Z \ \& \ \sim(N \wedge V)$	$\sim Z \ \& \ C$
>=	$\sim(N \wedge V)$	C

# Chapter 5: Digital Building Blocks

## **Shifters, Multipliers, & Dividers**

# Shifters

**Logical shifter:** shifts value to left or right and fills empty spaces with 0's

- Ex: 11001 >> 2 = 00110
- Ex: 11001 << 2 = 00100

**Arithmetic shifter:** same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb)

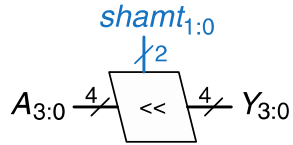
- Ex: 11001 >>> 2 = 11110
- Ex: 11001 <<< 2 = 00100

**Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end

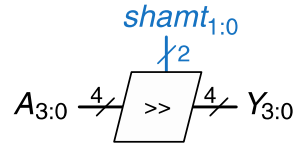
- Ex: 11001 ROR 2 = 01110
- Ex: 11001 ROL 2 = 00111

# Shifter Design

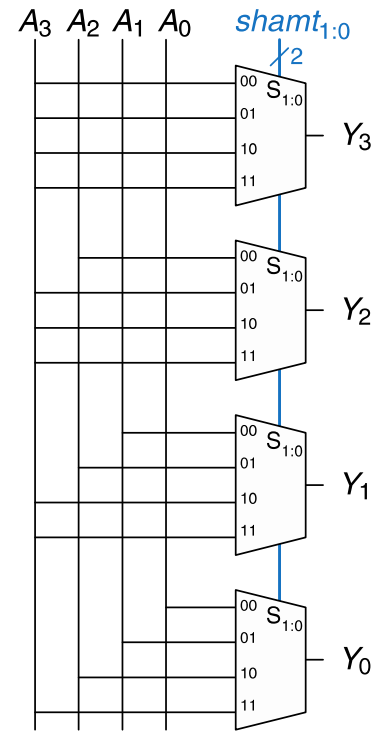
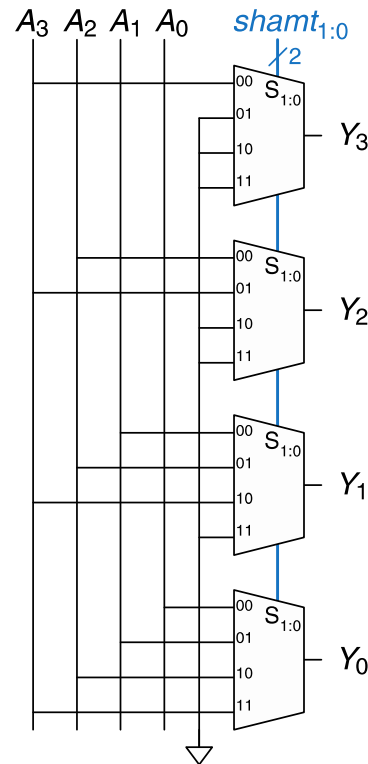
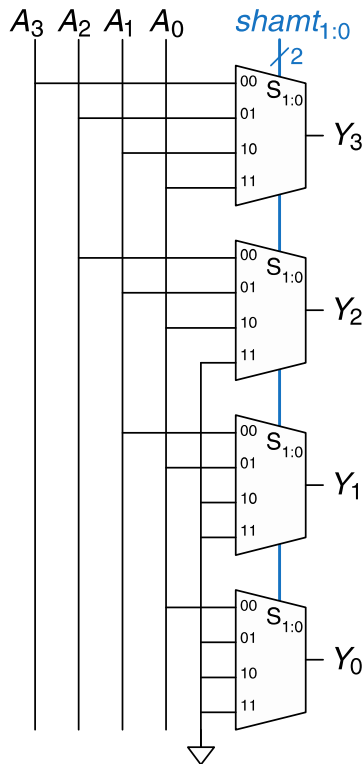
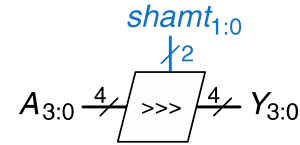
Shift Left



Logical Shift Right



Arithmetic Shift Right



# Shifters as Multipliers and Dividers

- $A \ll N = A \times 2^N$

- **Example:**  $00001 \ll 3 = 01000$  ( $1 \times 2^3 = 8$ )

- **Example:**  $11101 \ll 2 = 10100$  ( $-3 \times 2^2 = -12$ )

- $A \ggg N = A \div 2^N$

- **Example:**  $01000 \ggg 1 = 00100$  ( $8 \div 2^1 = 4$ )

- **Example:**  $10000 \ggg 2 = 11100$  ( $-16 \div 2^2 = -4$ )

# Chapter 5: Digital Building Blocks

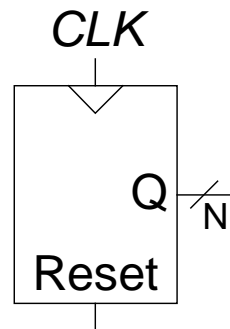
## **Counters & Shift Registers**



# Counters

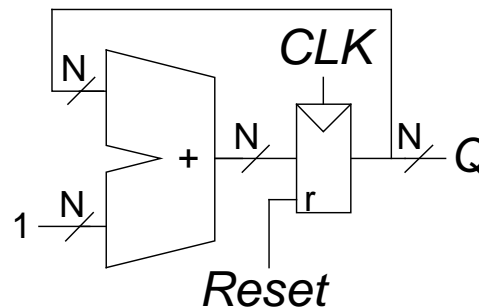
- Increments on each clock edge
- Used to cycle through numbers. For example,
  - 000, 001, 010, 011, 100, 101, 110, 111, 000, 001...
- **Example uses:**
  - Digital clock displays
  - Program counter: keeps track of current instruction executing

## Symbol



N-bit binary counter

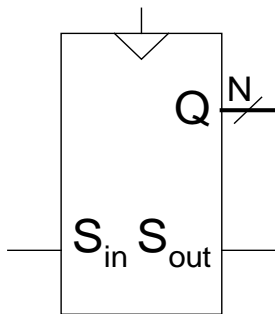
## Implementation



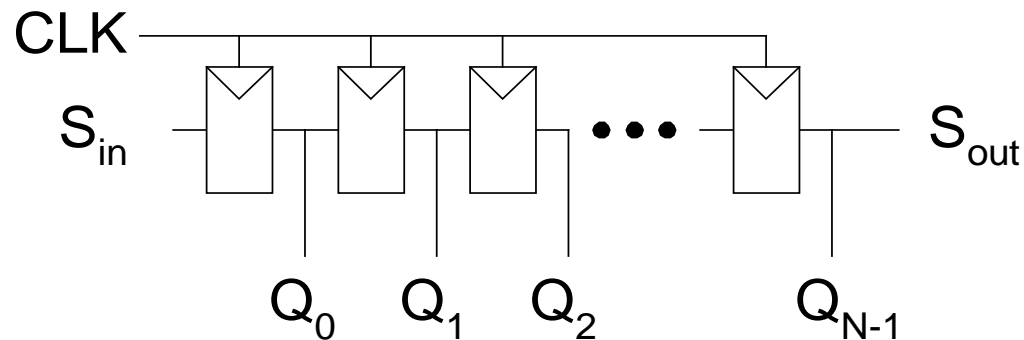
# Shift Registers

- 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:**

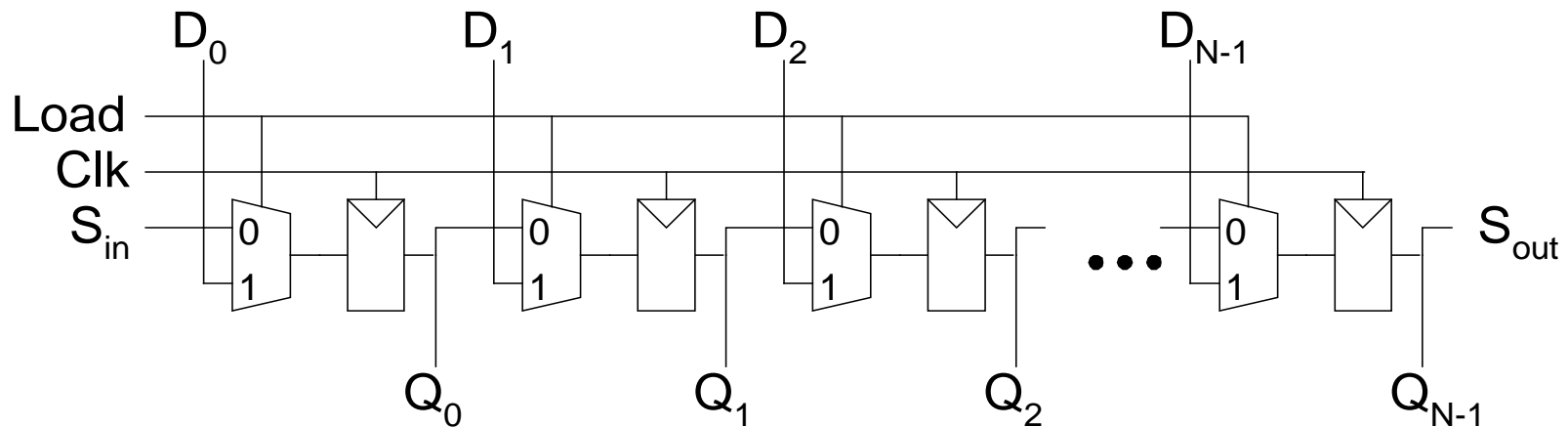


**Implementation:**



# 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  $S_{out}$ )

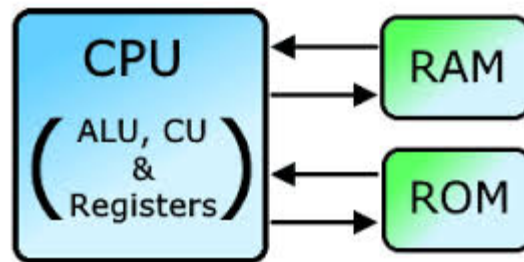
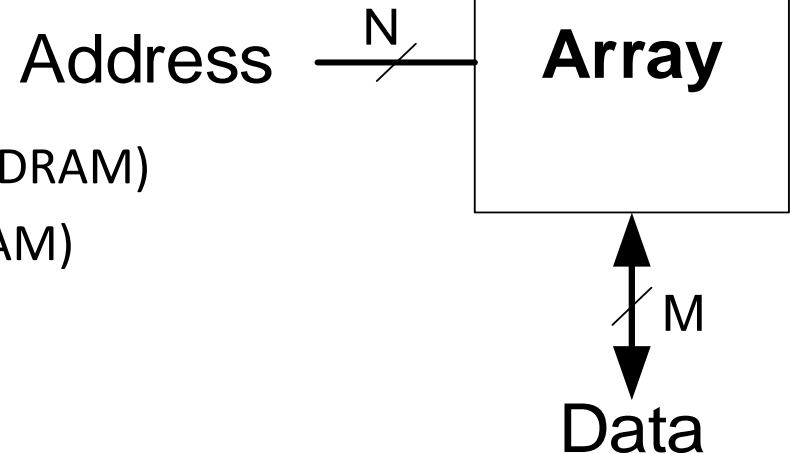


# Chapter 5: Digital Building Blocks

## **Memory**

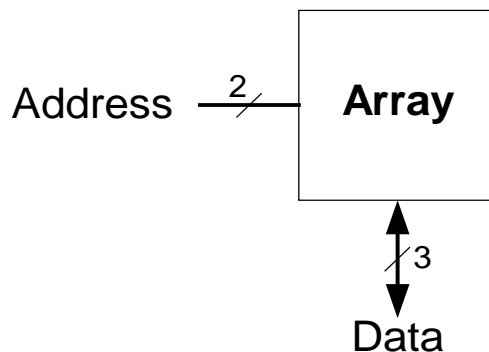
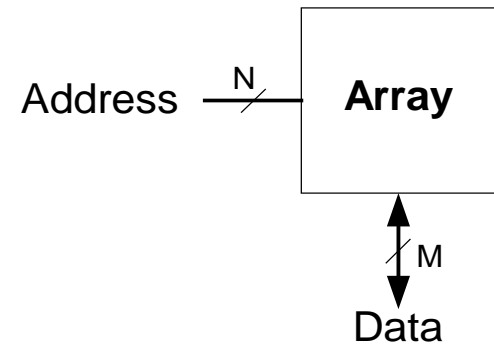
# Memory Arrays

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



# Memory Arrays

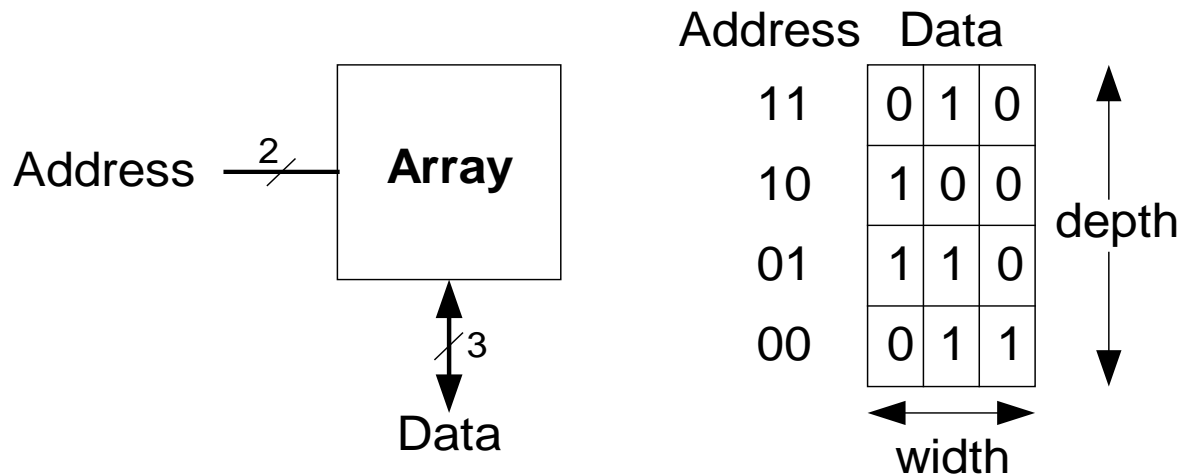
- 2-dimensional array of bit cells
- Each bit cell stores one bit
- $N$  address bits and  $M$  data bits:
  - $2^N$  rows and  $M$  columns
  - **Depth:** number of rows (number of words)
  - **Width:** number of columns (size of word)
  - **Array size:** depth  $\times$  width =  $2^N \times M$



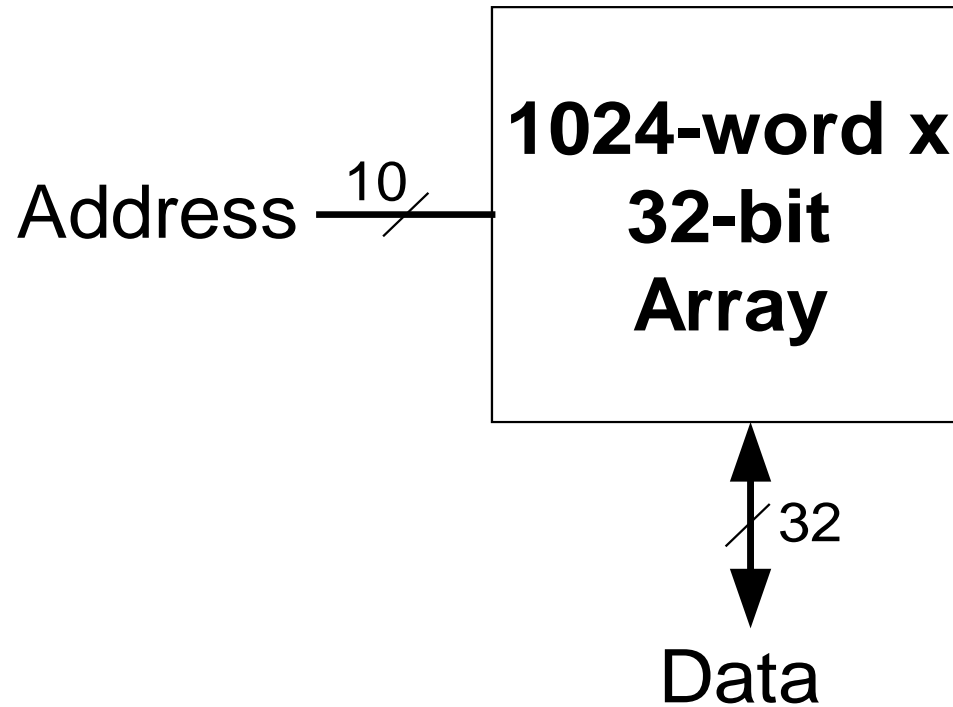
Address	Data			
11	0	1	0	depth ↑ ↓
10	1	0	0	
01	1	1	0	
00	0	1	1	
	width ←→			

# 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

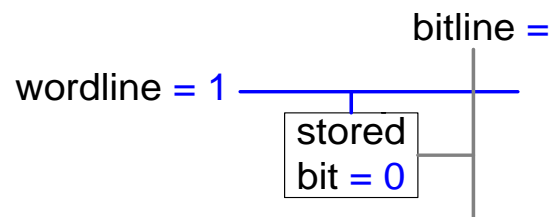
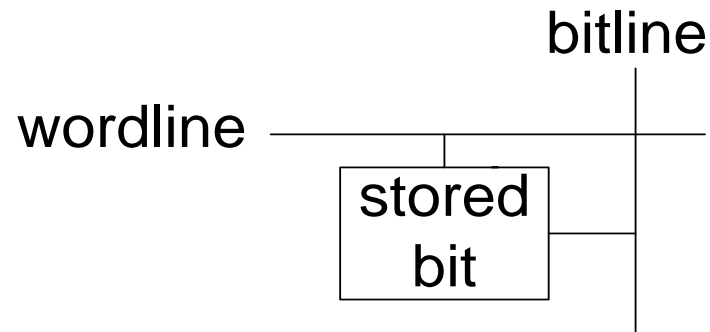


# Memory Arrays

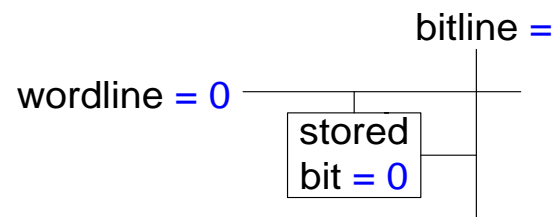




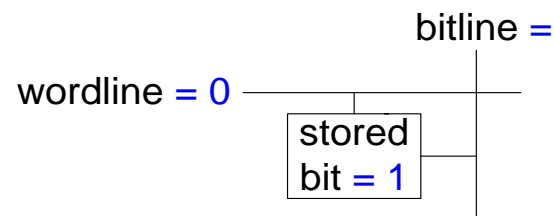
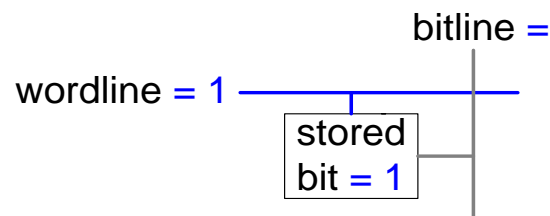
# Memory Array Bit Cells



(a)

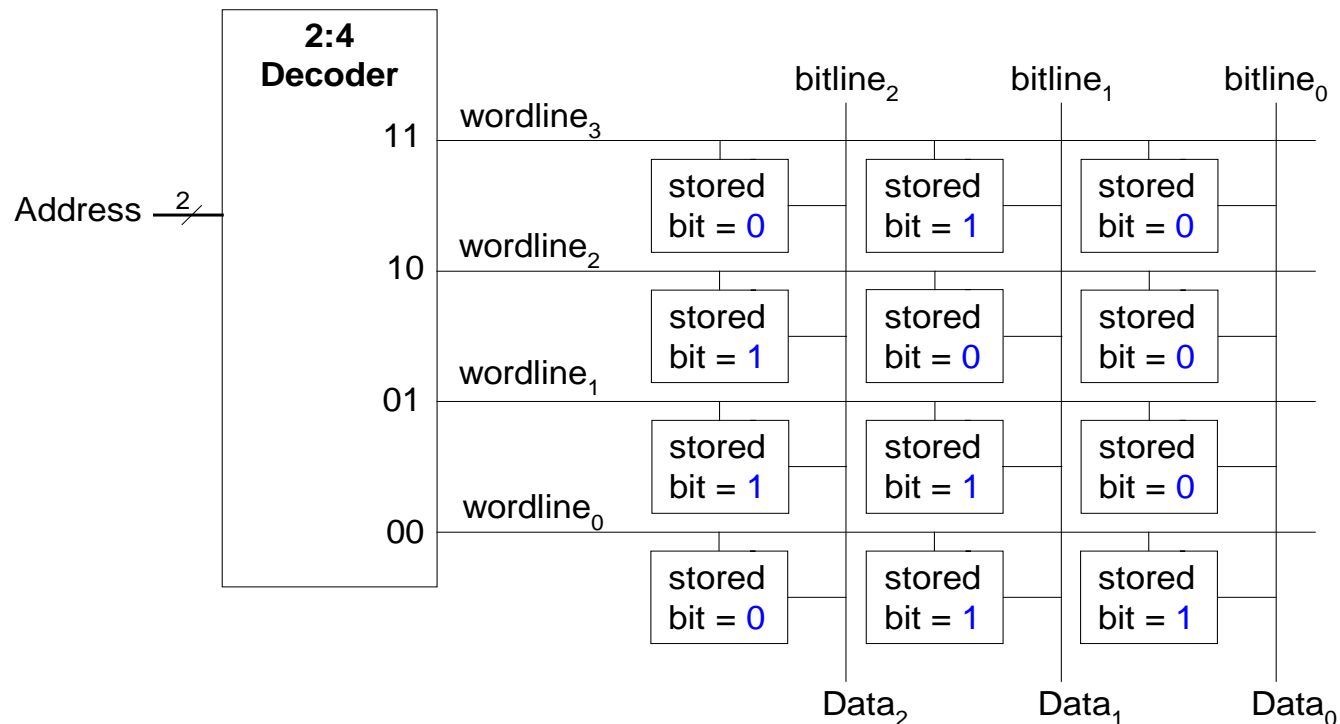


(b)



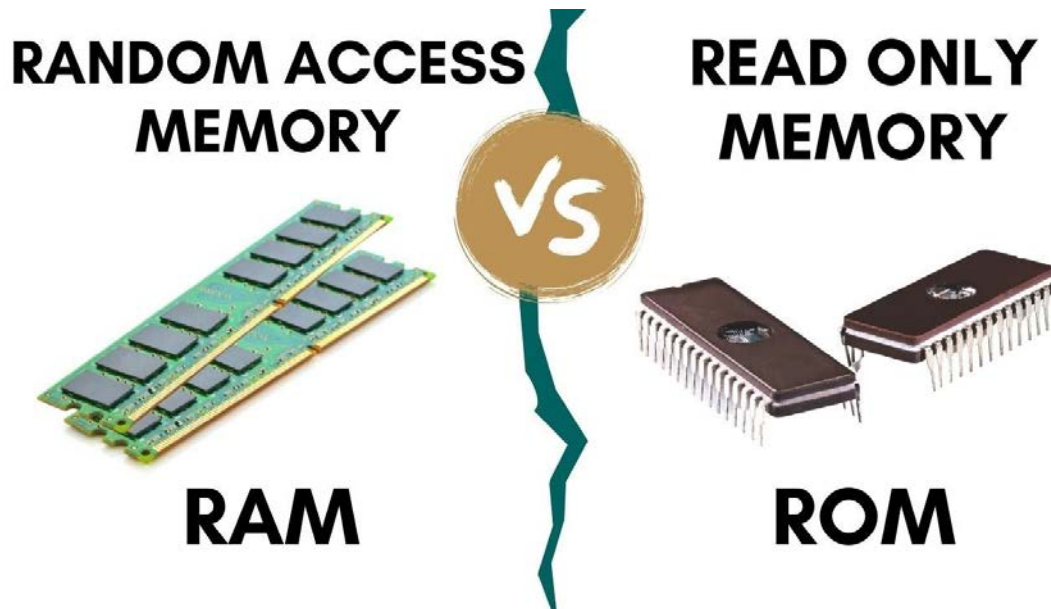
# Memory Array

- **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**



# 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 is accessed using its address with the same delay as any other (in contrast to sequential access memories such as a tape recorder)

# ROM: Read Only Memory

- **Nonvolatile:** retains data even if 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 time of fabrication or by burning fuses. Once a ROM was configured, it could not be written again. This is no longer the case for Flash memory and other types of ROMs.

# Chapter 5: Digital Building Blocks

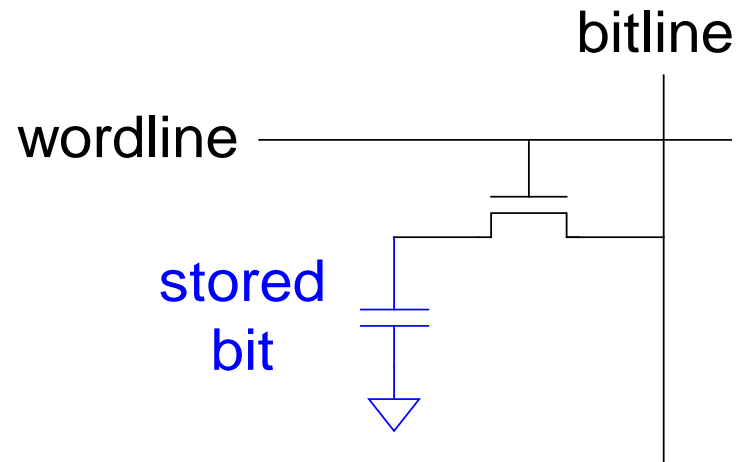
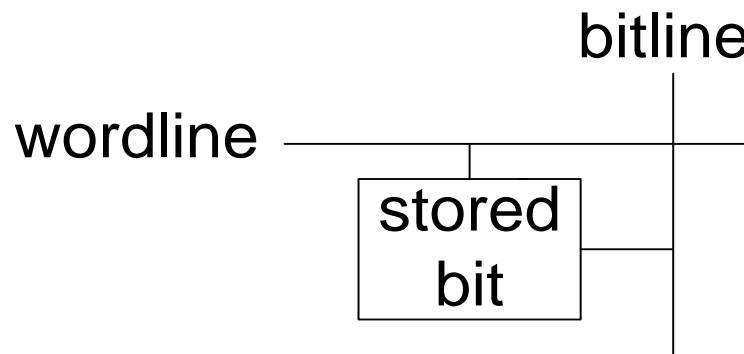
**RAM**

# 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

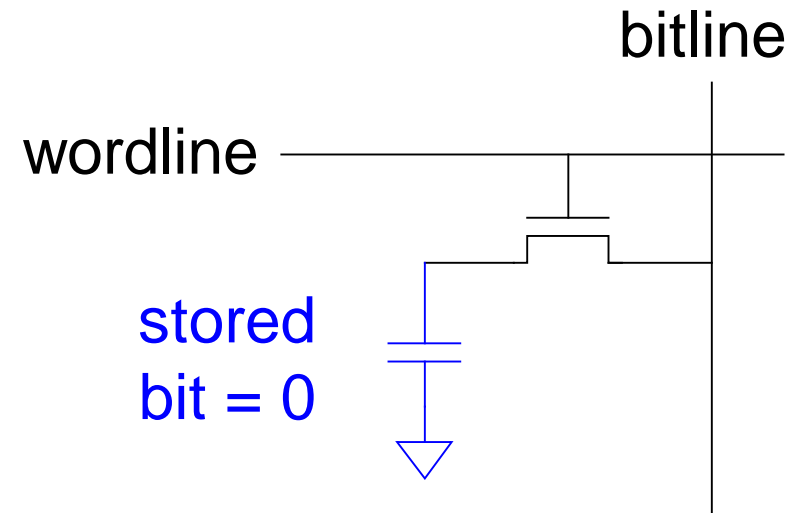
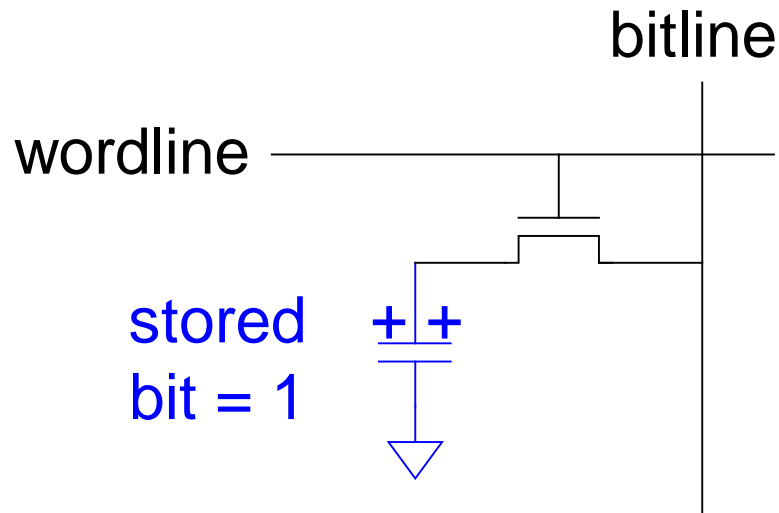
# DRAM

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



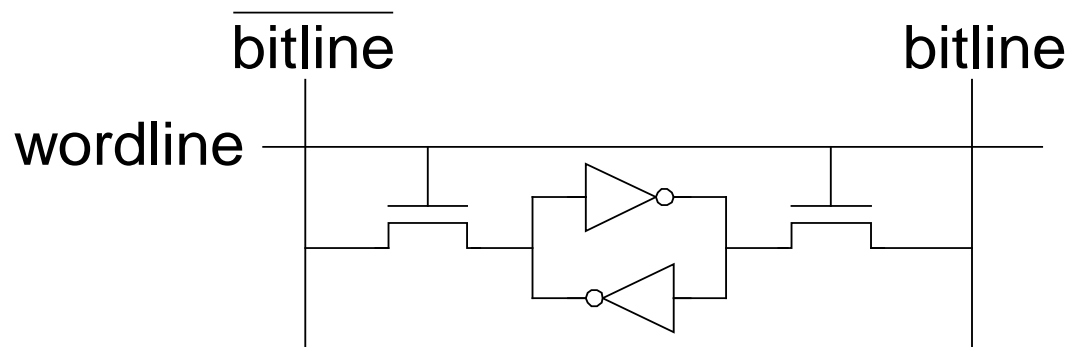
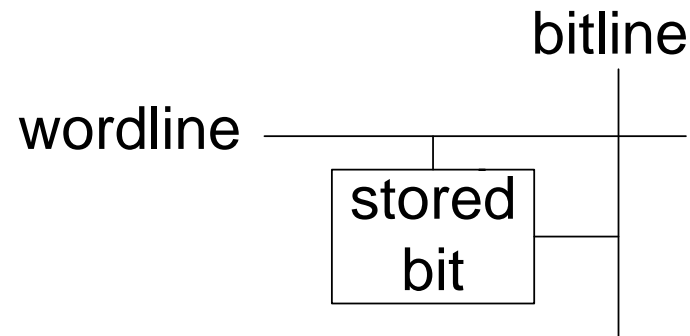


# DRAM

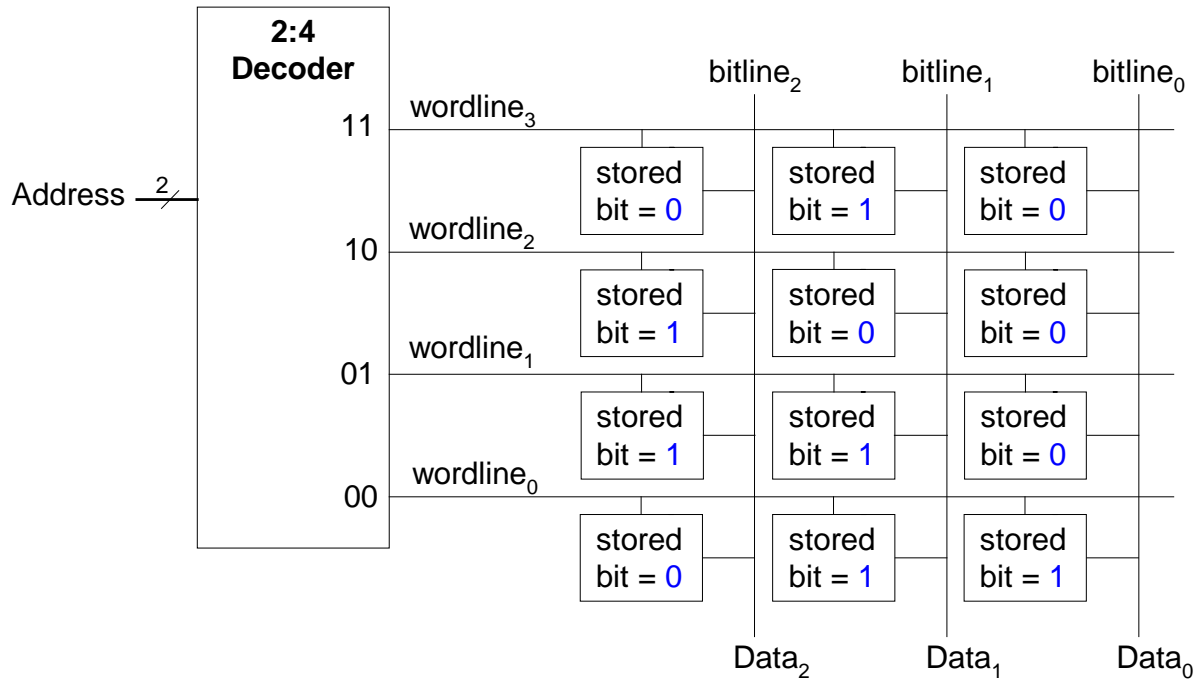


# SRAM

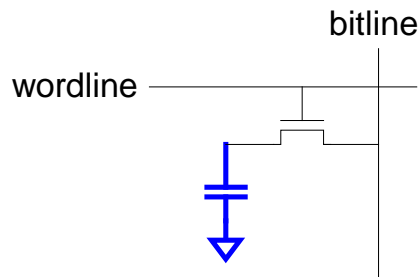
- *Static*: the value do not need to be **refreshed**



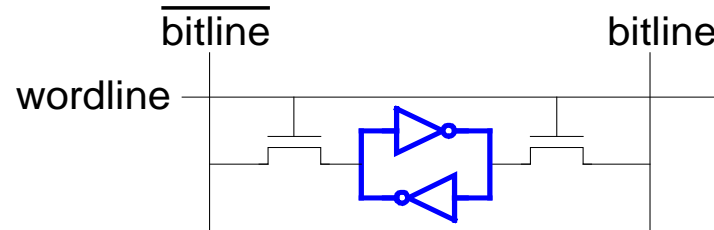
# Memory Arrays Review



**DRAM bit cell:**



**SRAM bit cell:**



# SRAM vs. DRAM

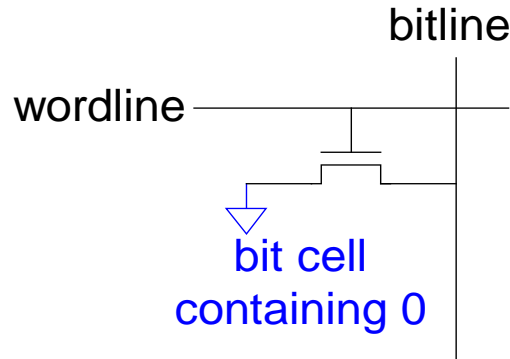


	SRAM	DRAM
Speed	Faster	Slower
Density	Low	High
Capacity	Small	Large
Power consumption	Low	High
Cost	Expensive	Cheap
Used as	Cache memory	Main memory

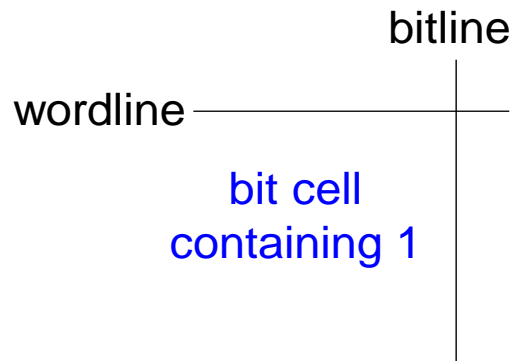
# Chapter 5: Digital Building Blocks

## **ROM**

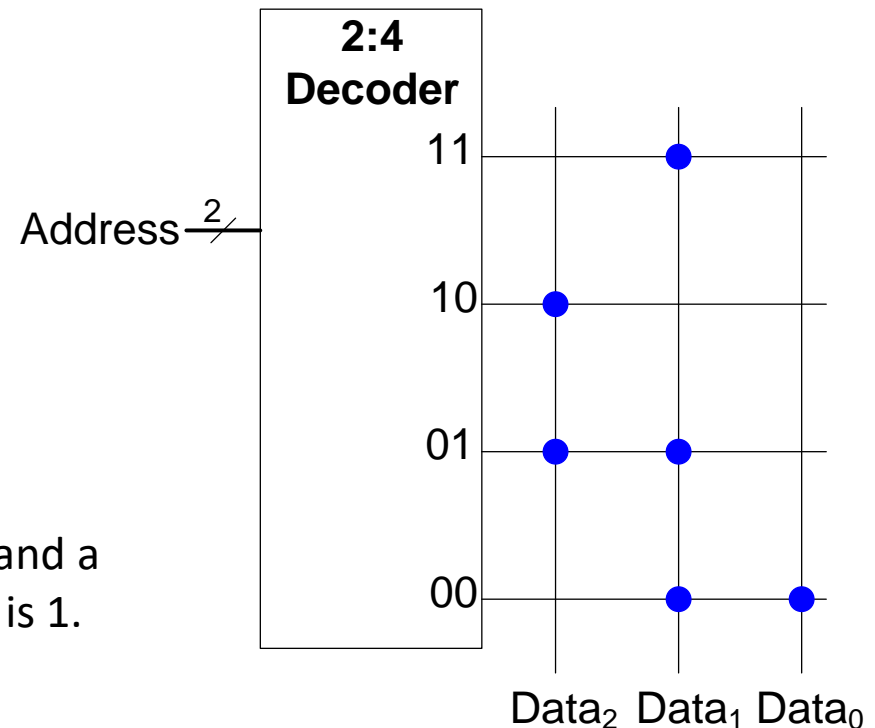
# ROM: Dot Notation



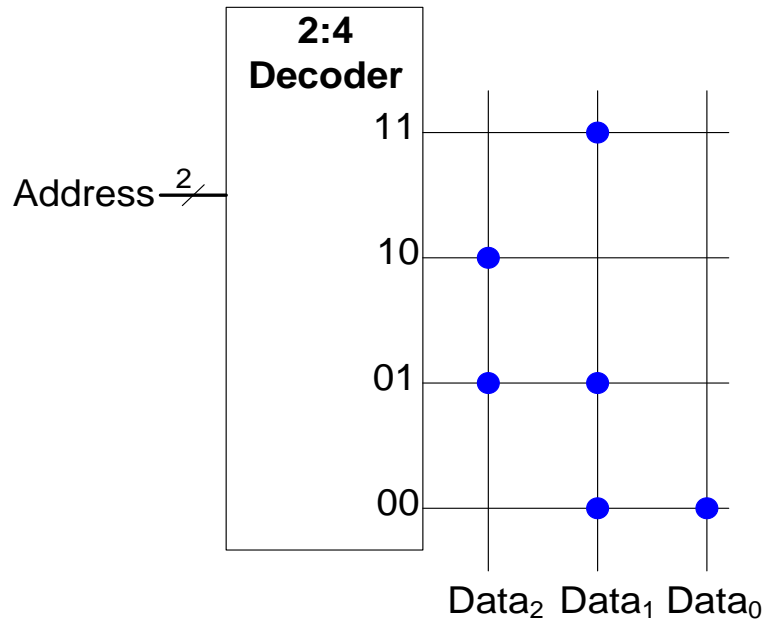
- Stores a bit as the presence or absence of a transistor
- When a bitline is pulled HIGH and the wordline is turned ON,
  - if a transistor is present, it pulls the bitline LOW
  - otherwise, the bitline remains HIGH



- A *dot* at the intersection of a row and a column indicates that the data bit is 1.

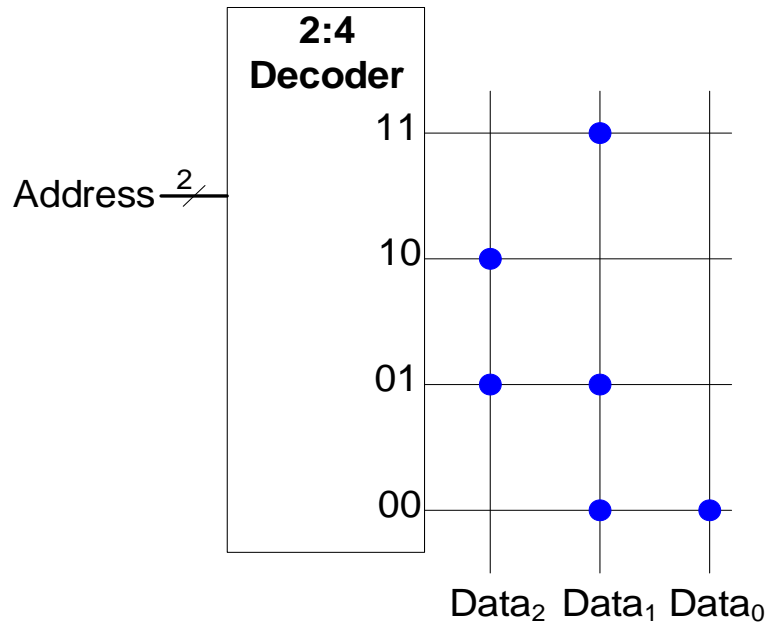


# ROM Storage



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

# ROM Logic



$$Data_2 = A_1 \oplus A_0$$

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

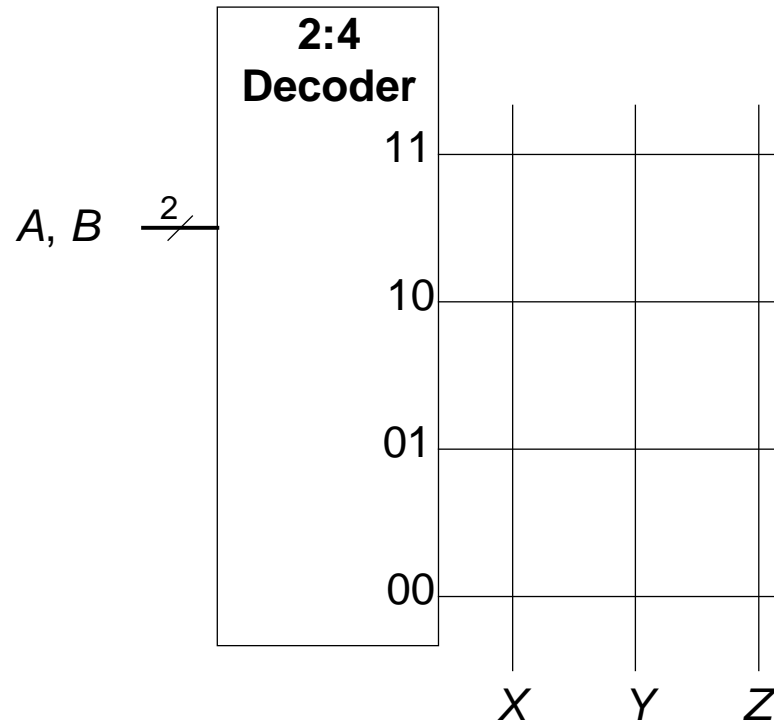
$$Data_0 = \bar{A}_1 \bar{A}_0$$



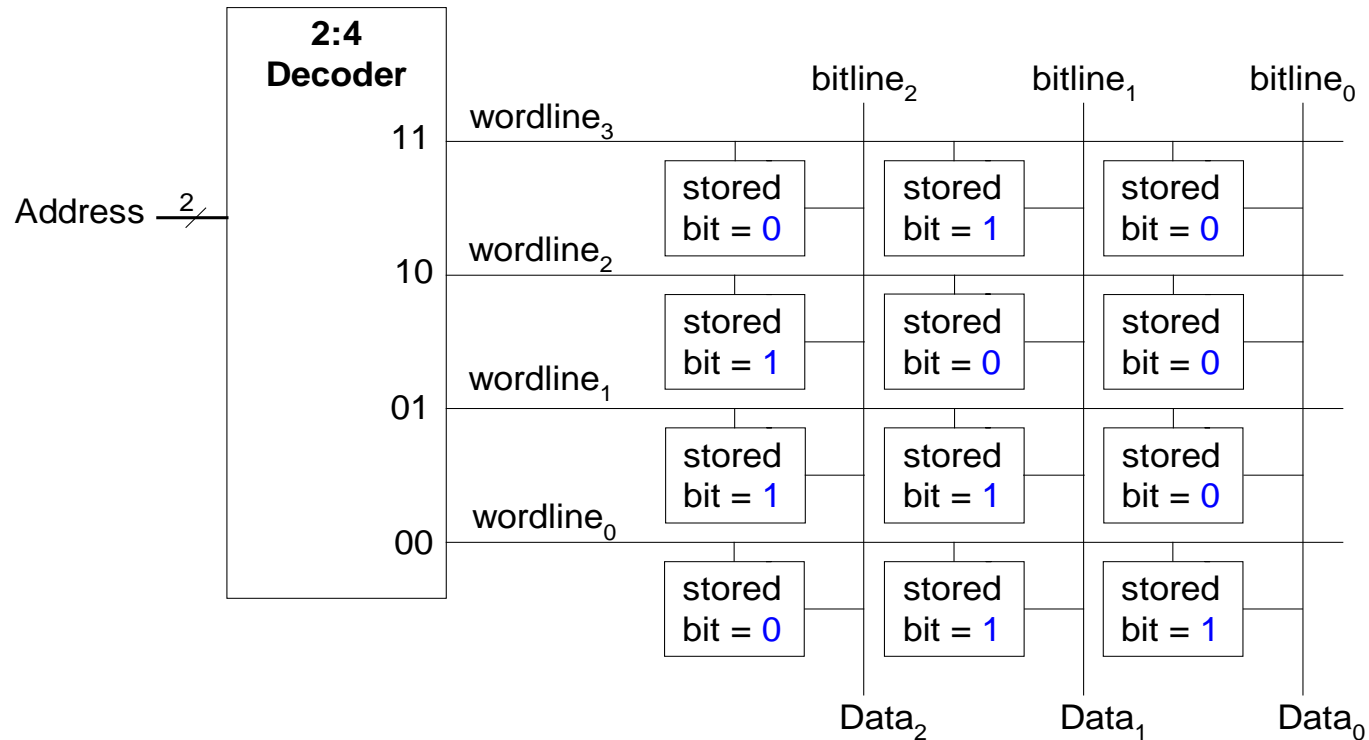
# Example: Logic with ROMs

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

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



# Logic with Any Memory Array



$$Data_2 = A_1 \oplus A_0$$

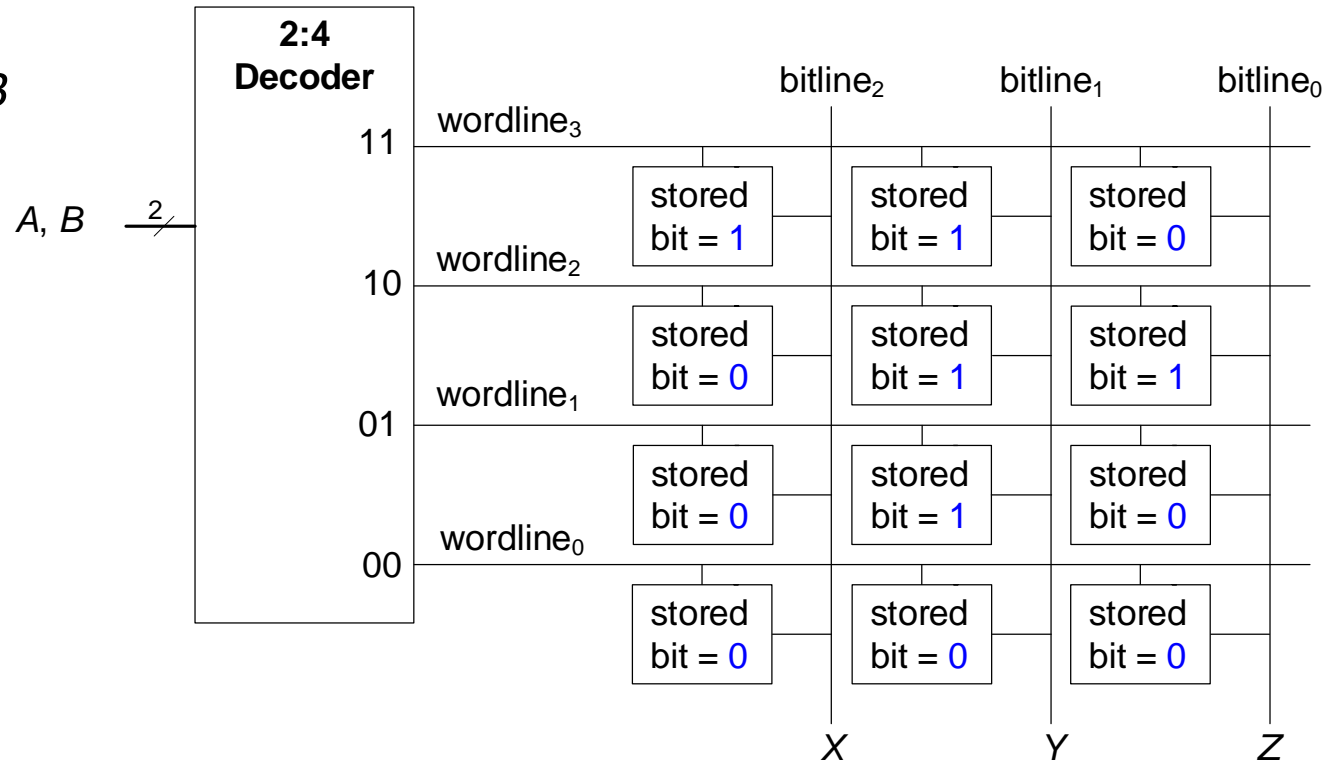
$$Data_1 = \overline{A_1} + A_0$$

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

# Logic with Memory Arrays

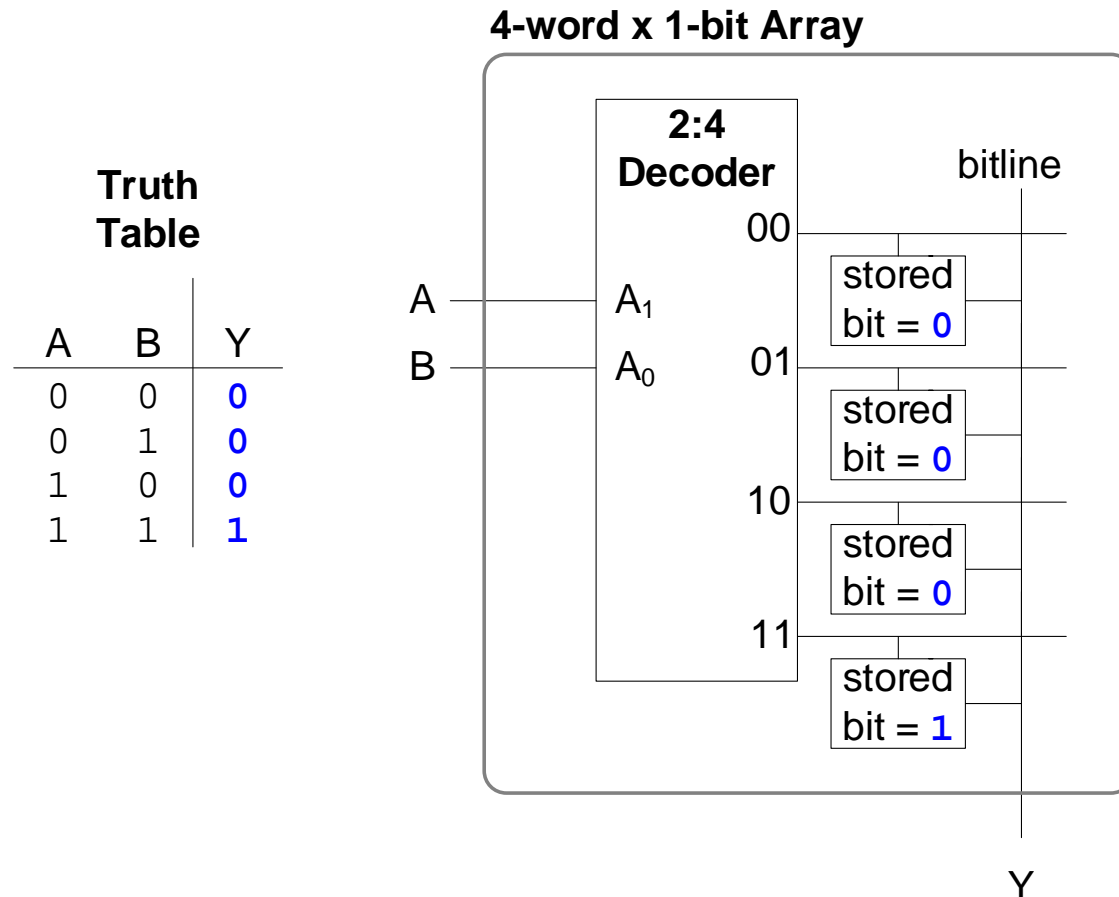
Implement the following logic functions using a  $2^2 \times 3$ -bit memory array:

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



# Logic with Memory Arrays

Can be used as *lookup tables* (**LUTs**) that look up output at each input combination (address)



# Chapter 5: Digital Building Blocks

## **Logic Arrays: PLAs & FPGAs**

# Logic Arrays

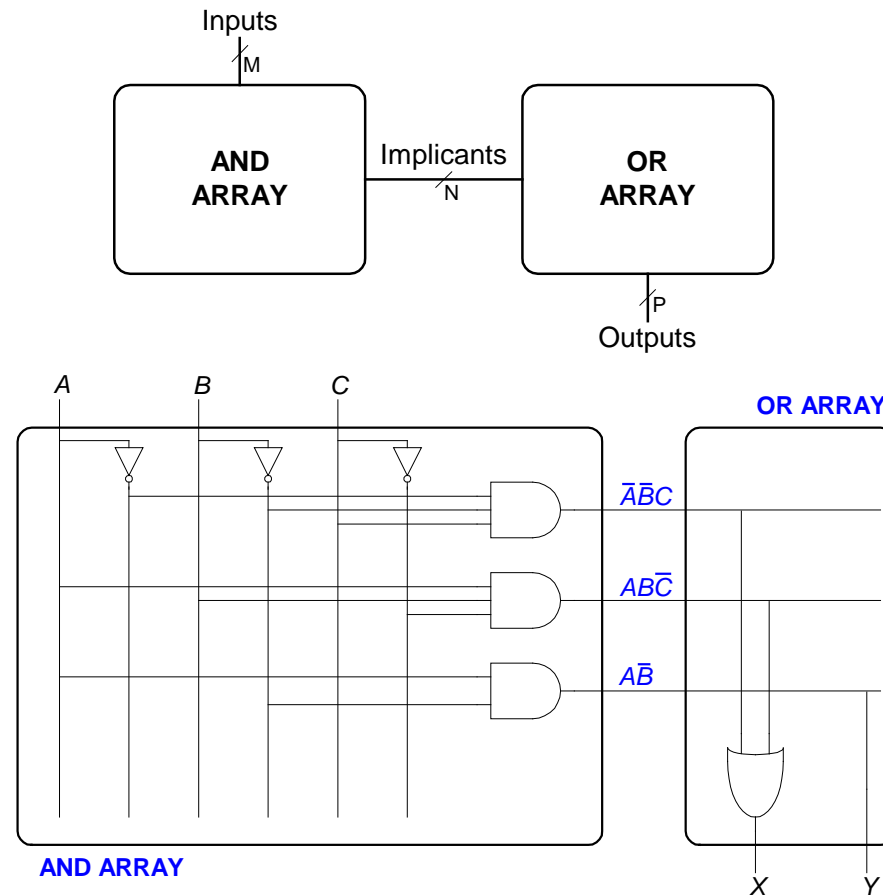
- Programmable gate arrays of which connections can be configured to perform any function
- Mass-produced in large quantities, so they are inexpensive
- Reconfigurable, allowing designs to be modified without replacing the hardware
  - Reconfigurability is valuable during development and is also useful in the field because a system can be upgraded by simply downloading the new configuration.

# Types of 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: Programmable Logic Arrays

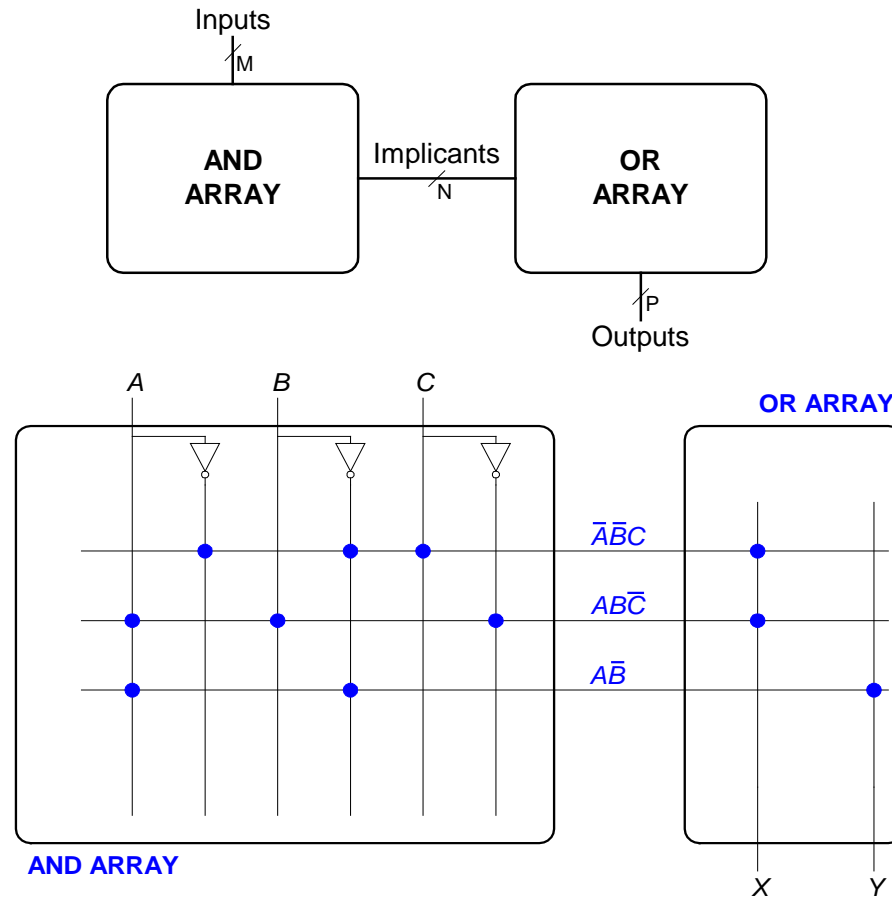
- $X = \overline{A}\overline{B}C + A\overline{B}C$
- $Y = A\overline{B}$





# PLAs: Dot Notation

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

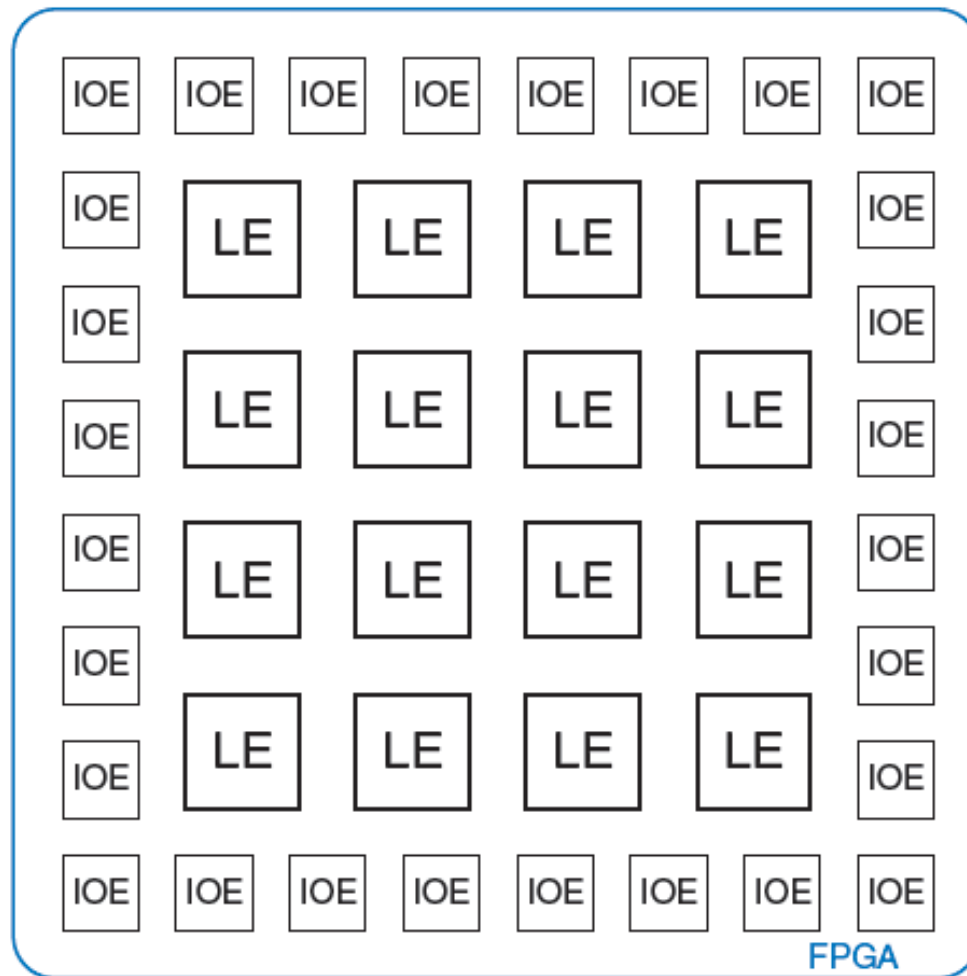


- Dots in the AND array: literals comprising an implicant
- Dots in the OR array: implicants being part of the output function

# FPGAs: Field Programmable Gate Arrays

- Composed of:
  - **LEs** (Logic elements): perform combinational or sequential functions
  - **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

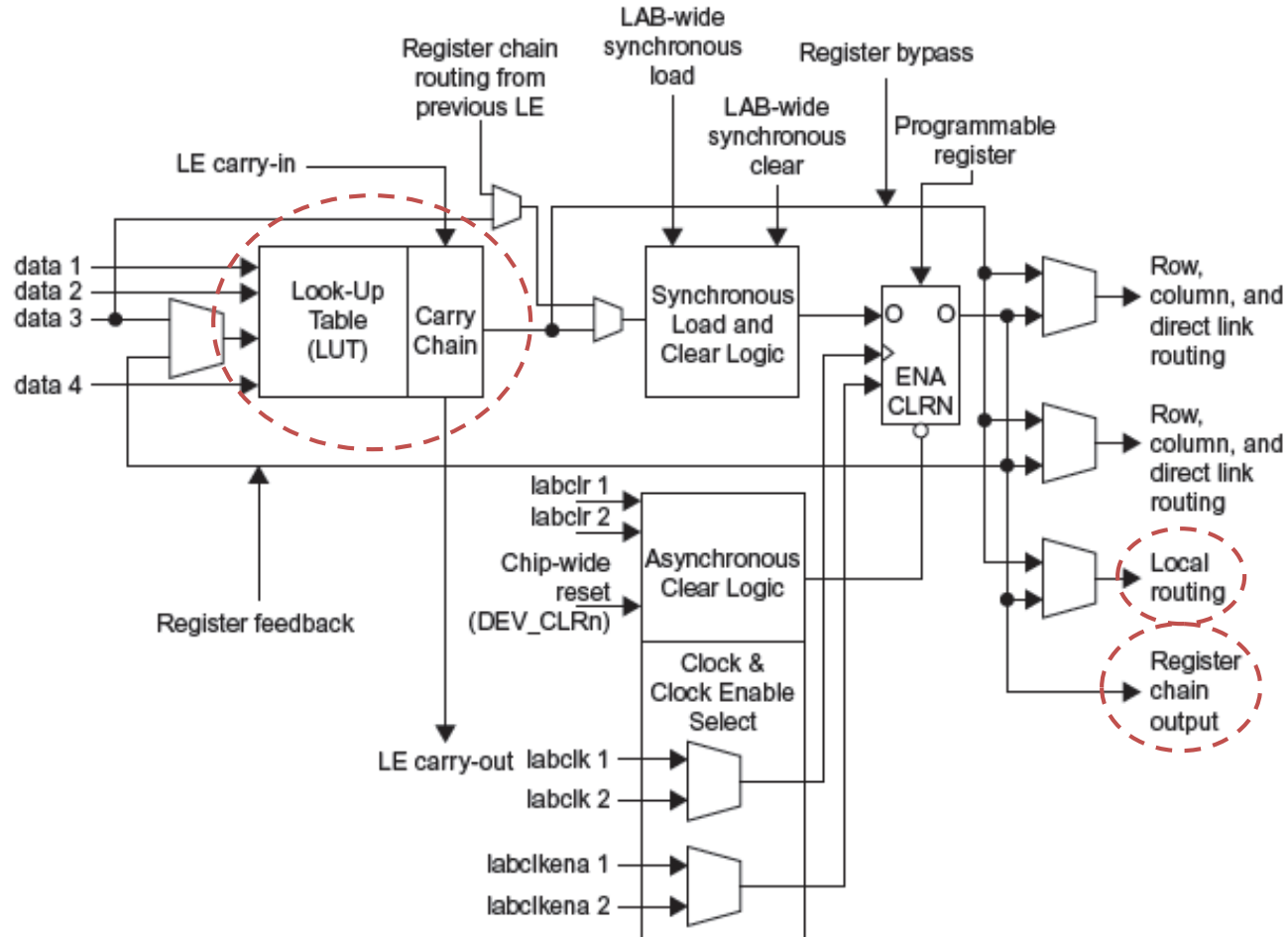
# General FPGA Layout



# 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



*From Cyclone IV datasheet*

# Altera Cyclone IV LE

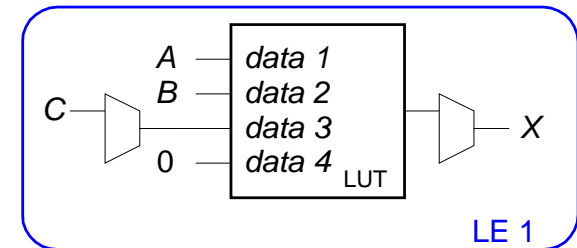
- **The Altera Cyclone IV LE has:**
  - 1 four-input **LUT**
  - 1 **registered output**
  - 1 **combinational output**

# LE Configuration Example

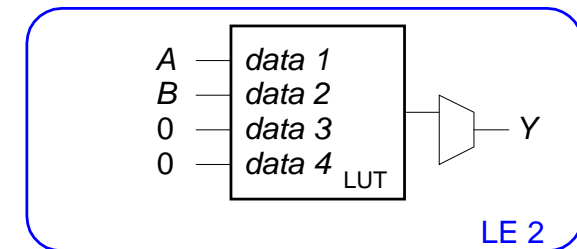
Show how to configure a Cyclone IV LE to perform the following functions:

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

(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



# Logic Elements Example 1

How many Cyclone IV LEs are required to build

$$Y = A1 \oplus A2 \oplus A3 \oplus A4 \oplus A5 \oplus A6$$

**Solution:**

2 LEs

First computes  $Y1 = A1 \oplus A2 \oplus A3 \oplus A4$  (function of 4 variables)

Second computes  $Y = Y1 \oplus A5 \oplus A6$  (function of 3 variables)



# Logic Elements Example 2

How many Cyclone IV LEs are required to build

32-bit 2:1 multiplexer

**Solution:**

32 LEs

A 1-bit mux is a function of 3 variables and fits in one LE

A 32-bit mux requires 32 copies

# Logic Elements Example 3

How many Cyclone IV LEs are required to build

Arbitrary FSM with 2 bits of state, 2 inputs, 3 outputs

**Solution:**

5 LEs

One LE can hold a bit of state and the next state logic, which is a function of 4 variables (2 bits of state, 2 inputs)

One LE can compute a bit of output, which is a function of at most 4 variables (2 bits of state, 2 inputs)

Thus 2 LEs are needed for state and 3 LEs for outputs

# About these Notes

**Digital Design and Computer Architecture Lecture Notes**

**© 2021 Sarah Harris and David Harris**

**These notes may be used and modified for educational and/or non-commercial purposes so long as the source is attributed.**