



Informatics

Computersysteme

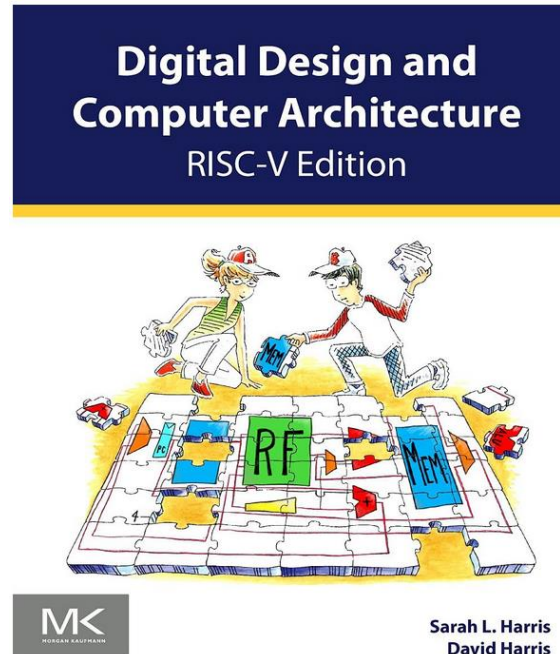
Building Blocks

Markus Bader

06.03.2025

Introduction

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



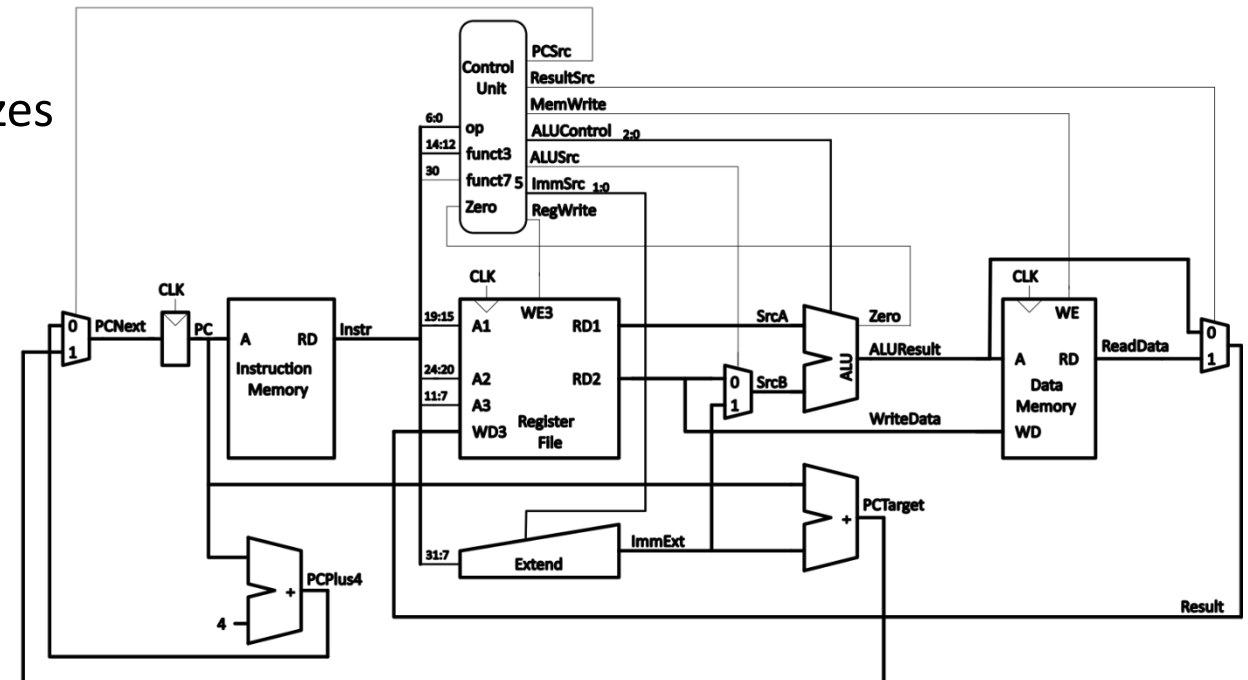
Application Software	<code>>"hello world!"</code>
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

**Digital Design and Computer Architecture,
RISC-V Edition: RISC-V Edition**

Computer Systems

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

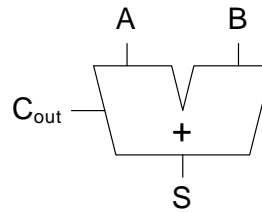


Adder

1-Bit Adders

- Half Adder (HF)
 - t_{HF} ... time/delay to compute outputs
- Full Adder (FA)
 - t_{FA} ... time/delay to compute outputs

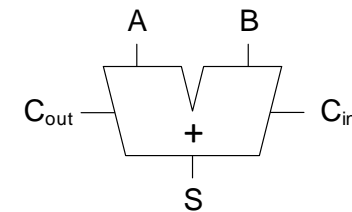
Half Adder (HA)



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

S =
C_{out} =

Full Adder (FA)

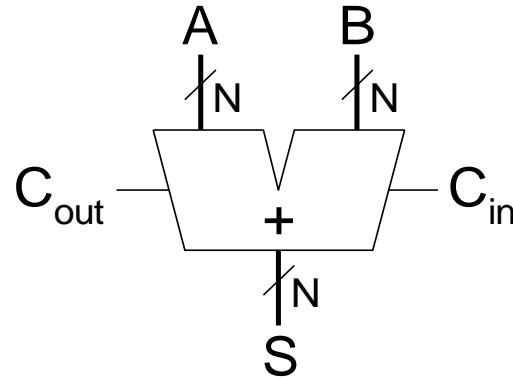


C _{in}	A	B	C _{out}	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

S =
C_{out} =

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

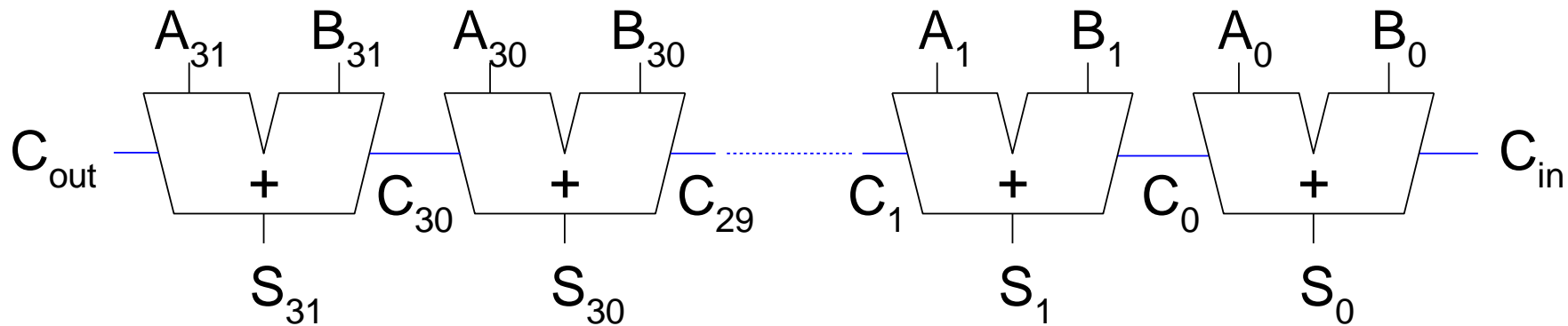


Ripple Carry Addition

DDCA Ch5 - Part 3: Ripple Carry Adders <https://youtu.be/mSwJvYz8nOQ?si=JSPAAs5-yZapKihf>

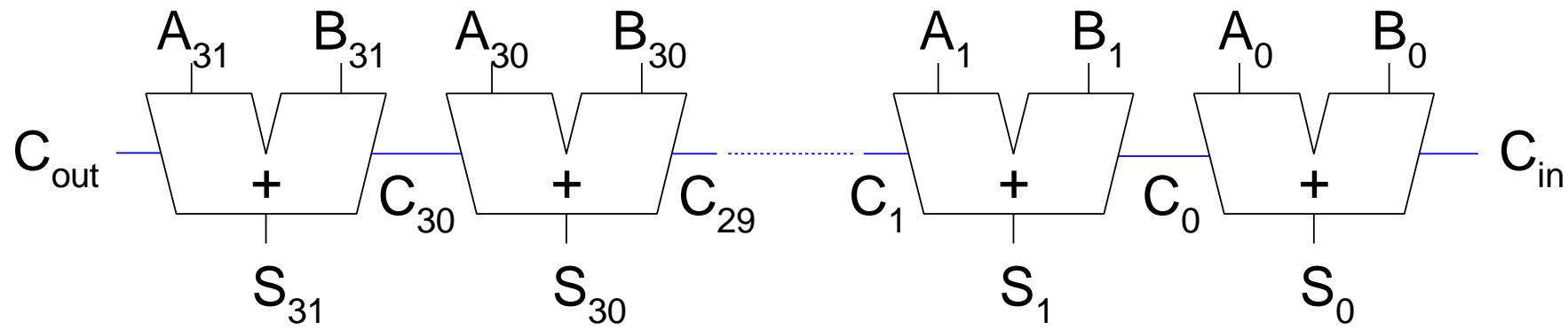
Ripple-Carry Adder

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



Ripple-Carry Adder Delay

- $t_{ripple} = Nt_{FA}$
 - t_{FA} is the delay of a 1 bit full adder



Carry Lookahead Addition

DDCA Ch5: Part 4 - Carry Lookahead Adders <https://youtu.be/aD-qA-jEKV0?si=gLit6-VCBnjTKSgG>

Carry-Lookahead Adder

- Compute 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
 - 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.

$$G_i = A_i B_i$$

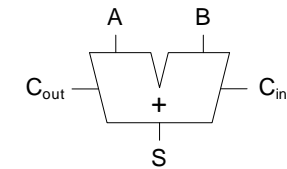
- Propagate: Column i will propagate a carry in to the carry out if A_i or B_i is 1.

$$P_i = A_i + B_i$$

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

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

Full Adder
(FA)



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

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

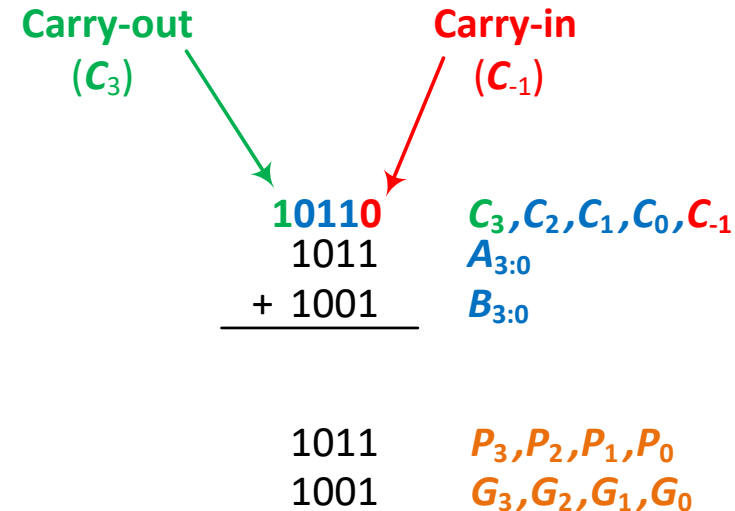
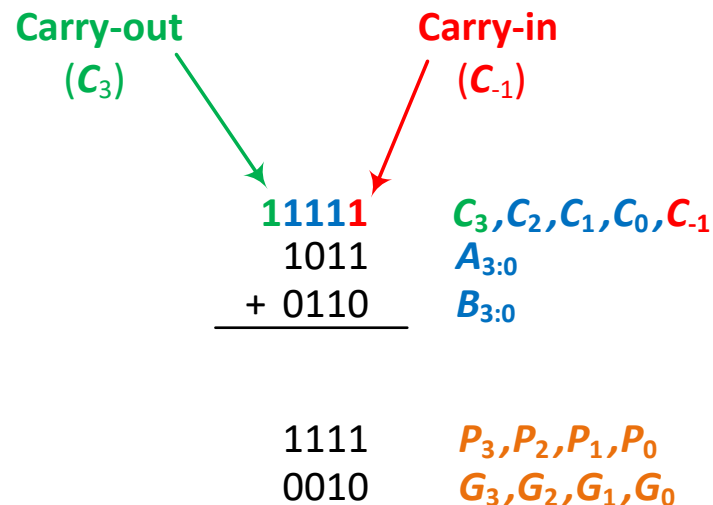
- Column propagate: $P_i = A_i + B_i$

- Column generate: $G_i = A_i B_i$

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

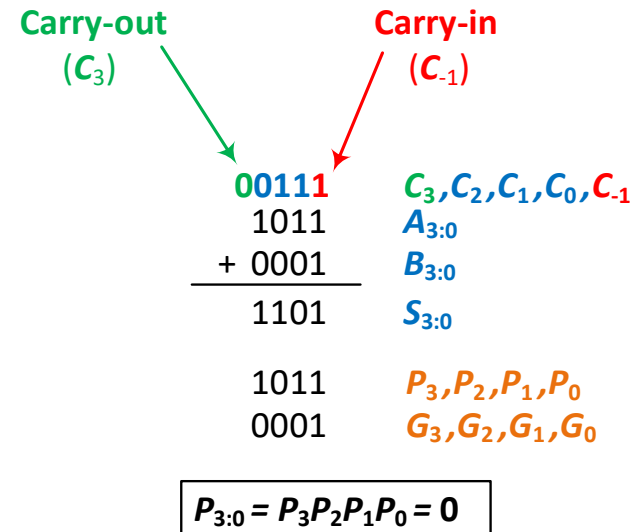
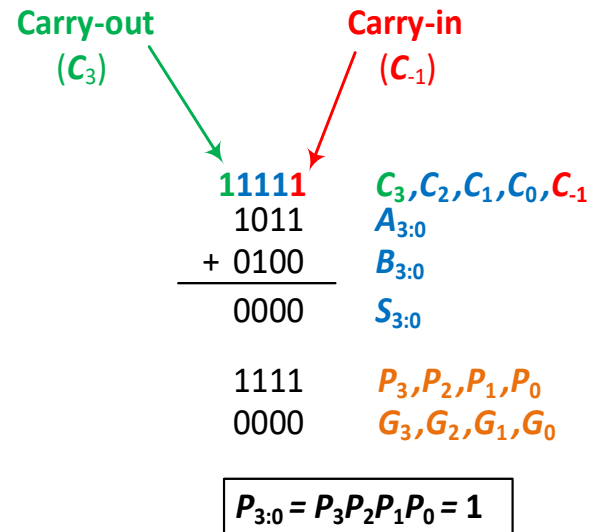
- **Examples:** Column propagate and generate signals:

- Column propagate: $P_i = A_i + B_i$
- Column generate: $G_i = A_i B_i$
 $C_i = G_i + P_i C_{i-1}$



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)
- Two Examples with 4-bit blocks



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 ...

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

- Three Examples with 4-bit blocks

Carry-out
(C_3)
↓

10011	$C_3, C_2, C_1, C_0, C_{-1}$
1001	$A_{3:0}$
+ 1100	$B_{3:0}$
0110	$S_{3:0}$
1101	P_3, P_2, P_1, P_0
1000	G_3, G_2, G_1, G_0

$G_{3:0} = 1$

Carry-out
(C_3)
↓

11000	$C_3, C_2, C_1, C_0, C_{-1}$
1110	$A_{3:0}$
+ 0100	$B_{3:0}$
0010	$S_{3:0}$
1110	P_3, P_2, P_1, P_0
0100	G_3, G_2, G_1, G_0

$G_{3:0} = 1$

Carry-out
(C_3)
↓

01101	$C_3, C_2, C_1, C_0, C_{-1}$
0110	$A_{3:0}$
+ 0010	$B_{3:0}$
1000	$S_{3:0}$
0110	P_3, P_2, P_1, P_0
0010	G_3, G_2, G_1, G_0

$G_{3:0} = 0$

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)

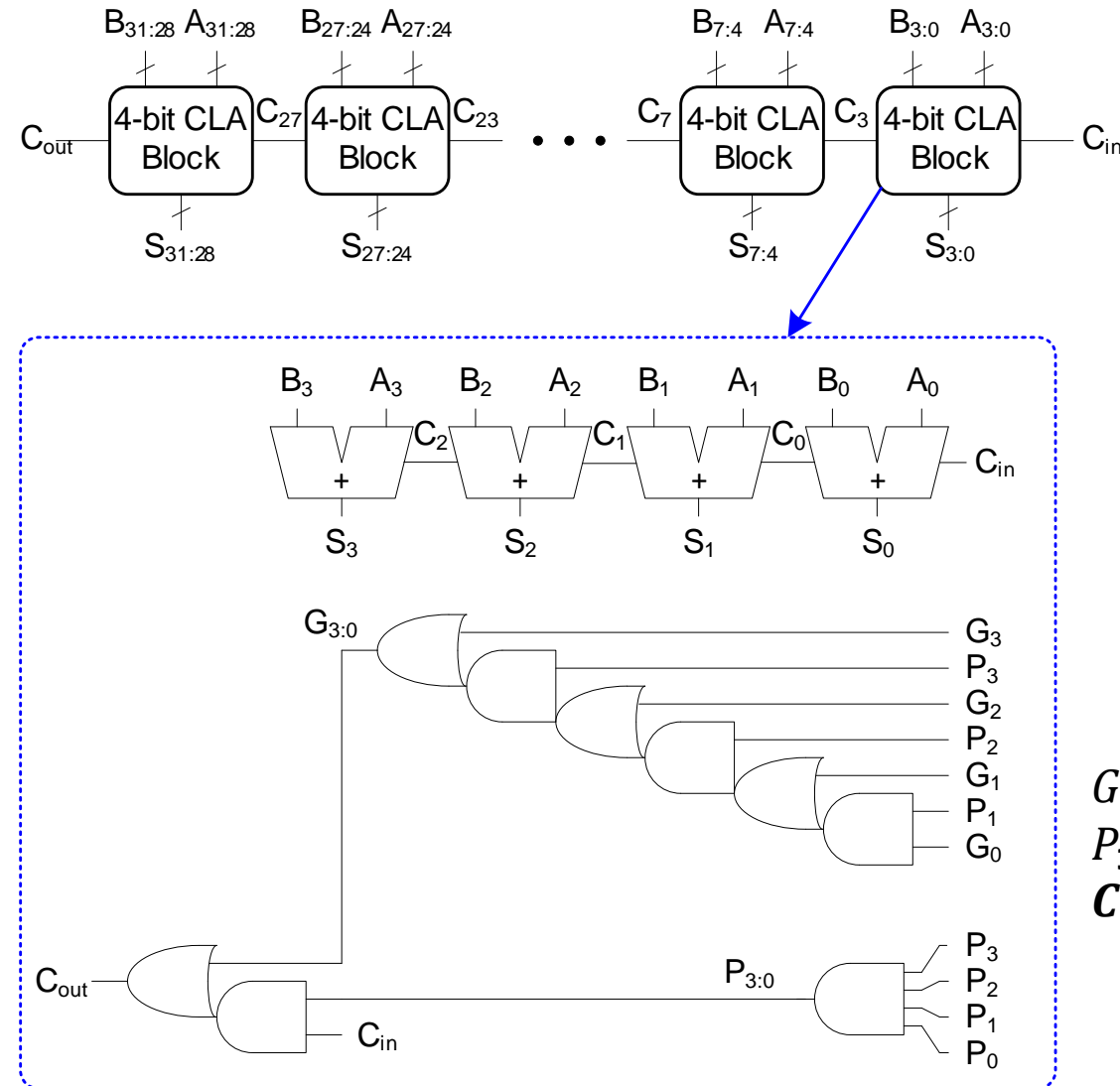
- A carry is generated:
 - in column 3, or
 - in column 2 and propagated through column 3, or
 - in column 1 and propagated through columns 2 and 3, or
 - in column 0 and propagated through columns 1-3

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

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

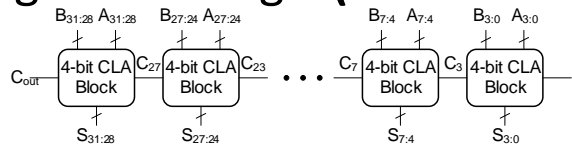
$$C_3 = G_{3:0} + P_{3:0} C_{-1}$$

32-bit Carry-Lookahead Addition with 4-bit Blocks

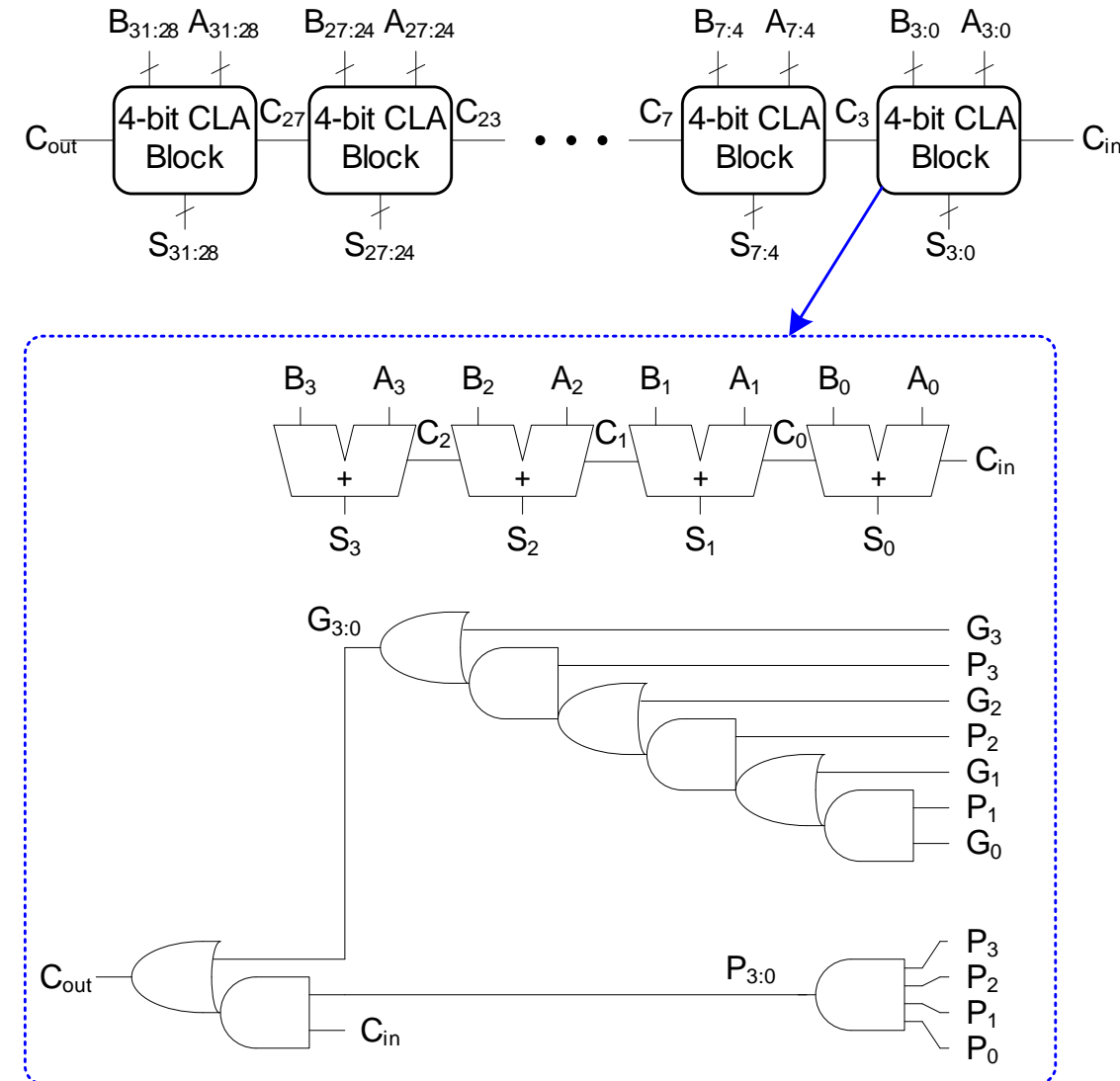
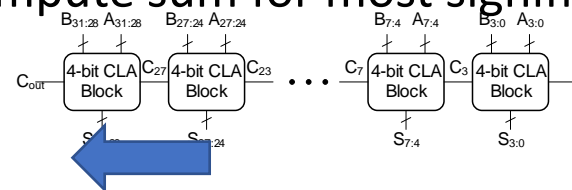


32-bit Carry-Lookahead Addition with 4-bit Blocks

- Step 1: Compute G_i and P_i for all columns
 - $G_i = A_i B_i$
 - $P_i = A_i + B_i$
- Step 2: Compute G and P for k -bit blocks
 - $P_{3:0} = P_3 P_2 P_1 P_0$
 - $G_{3:0} = G_3 + P_3[G_2 + P_2(G_1 + P_1 G_0)]$
- 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



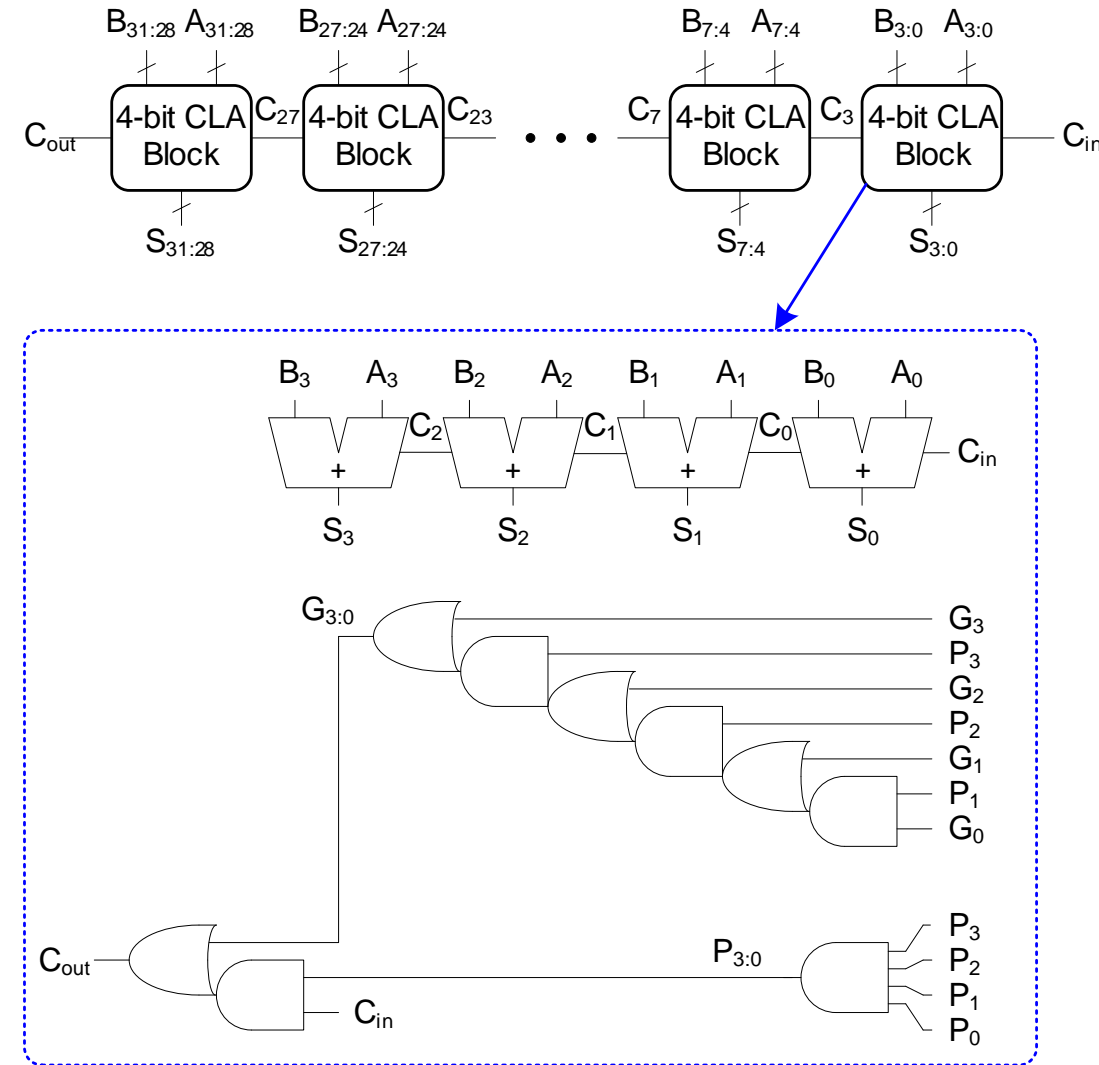
32-bit Carry-Lookahead Addition (CLA) with 4-bit Blocks

For N -bit CLA with k -bit blocks:

- t_{CLA} ... time/delay to compute outputs

$$t_{CLA} = t_{PG} + t_{PG_{Block}} + \left(\frac{N}{k} - 1 \right) t_{AND \& OR} + kt_{FA}$$

- 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 $N > 16$



Prefix Addition

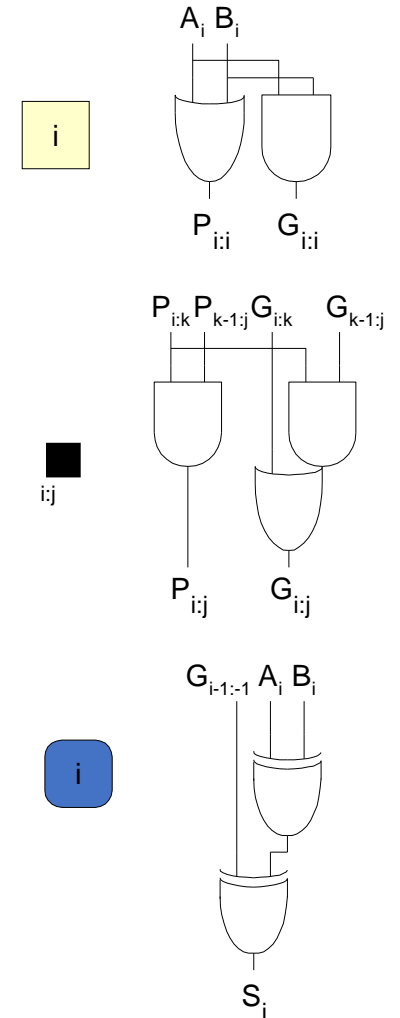
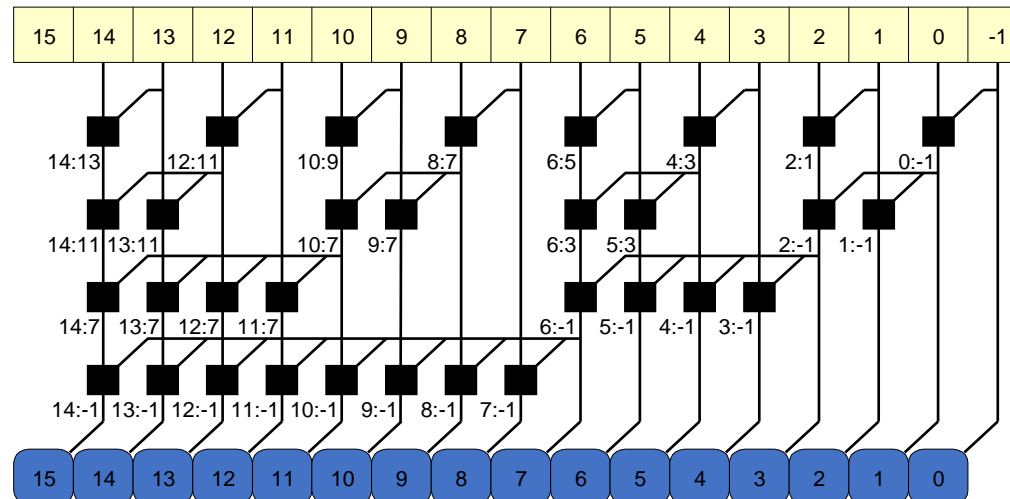
DDCA Ch5 - Part 5: Prefix Adders <https://youtu.be/Ue5CjG31E-c?si=hdY1Fi5auqO0FPCD>

Prefix Adder

- Computes carry in (C_{i-1}) for each column, then computes sum:

$$S_i = (A_i \oplus B_i) \oplus C_{i-1}$$

- It computes C_{i-1} by:
 - Computing G and P for 1-, 2-, 4-, 8-bit blocks, etc. until all G_i (carry in) known
 - $G_i = C_i$
- $\log_2 N$ stages



- Carry out either **generated** in a column or **propagated** from a previous column.
- Column -1 holds C_{in} , so
 - $G_{-1} = C_{in}, P_{-1} = X$ (*not used*)
- Carry in to column i = carry out of column $i - 1$:
 - $C_{i-1} = G_{i-1:-1}$
 - $G_{i-1:-1} \dots$ generate signal spanning columns $i - 1$ to -1
- Sum equation:
 - $S_i = (A_i \oplus B_i) \oplus C_{i-1}$
- **Goal:** Quickly compute $G_{0:-1}, G_{1:-1}, G_{2:-1}, G_{3:-1}, G_{4:-1}, G_{5:-1}, \dots$
 (called prefixes) ($= C_0, C_1, C_2, C_3, C_4, C_5, \dots$)

- Generate and propagate signals for a block spanning bits $i:j$
 - $G_{i:j} = G_{i:k} + P_{i:k}G_{k-1:j}$
 - $P_{i:j} = P_{i:k}P_{k-1:j}$
- In words:
 - Generate: block $i:j$ will generate a carry if:
 - upper part ($i:k$) generates a carry ($G_{i:k}$) or
 - upper part ($i:k$) propagates a carry ($P_{i:k}$) generated in lower part ($k-1:j$) ($G_{k-1:j}$)
 - Propagate: block $i:j$ will propagate a carry if both the upper and lower parts propagate the carry ($P_{i:k} \text{ AND } P_{k-1:j}$)

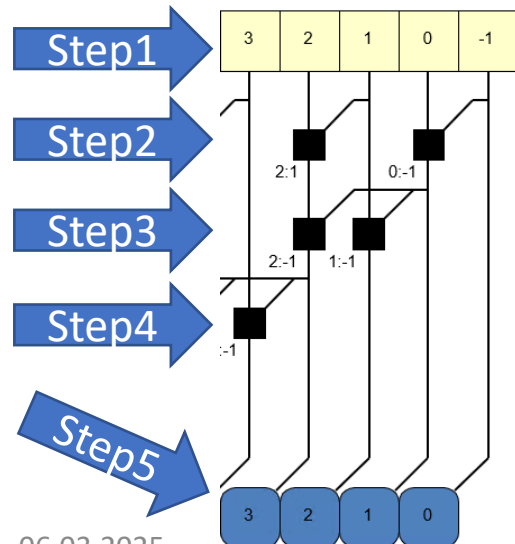
Prefix Adder Example

Step 1. Calculate P's and G's for **1-bit block**

$$\begin{array}{r} \overline{3} \overline{2} \overline{1} \overline{0} \overline{-1} \\ 1010 \\ + 0111 \\ \hline \end{array} \quad \begin{array}{l} \text{Column \#} \\ A_{3:0} \\ B_{3:0} \end{array}$$

$$\begin{array}{r} 1111 \\ 00101 \\ \hline \end{array} \quad \begin{array}{l} P_{3:3}, P_{2:2}, P_{1:1}, P_{0:0} \\ G_{3:3}, G_{2:2}, G_{1:1}, G_{0:0}, G_{-1:-1} \end{array}$$

$\uparrow C_{in}$ $\uparrow C_{in}$



Step 2. Calculate P's and G's for **2-bit blocks**

$$\begin{array}{r} \overline{3} \overline{2} \overline{1} \overline{0} \overline{-1} \\ 1010 \\ + 0111 \\ \hline \end{array} \quad \begin{array}{l} \text{Column \#} \\ A_{3:0} \\ B_{3:0} \end{array}$$

$$P_{L:R} = P_L \cdot P_R$$

$$G_{L:R} = G_L + P_L G_R$$

0:-1 Block:

$$P_{0:-1} = X$$

$$G_{0:-1} = G_{0:0} + P_{0:0} G_{-1:-1}$$

$$= 0 + 1 \cdot 1$$

$$= 1$$

2:1 Block:

$$P_{2:1} = P_{2:2} P_{1:1} = 1 \cdot 1$$

$$G_{2:1} = G_{2:2} + P_{2:2} G_{1:1}$$

$$= 0 + 1 \cdot 1$$

$$= 1$$

Step 3. Calculate P's and G's for **4-bit blocks**

$$\begin{array}{r} \overline{3} \overline{2} \overline{1} \overline{0} \overline{-1} \\ 1010 \\ + 0111 \\ \hline \end{array} \quad \begin{array}{l} \text{Column \#} \\ A_{3:0} \\ B_{3:0} \end{array}$$

$$P_{L:R} = P_L \cdot P_R$$

$$G_{L:R} = G_L + P_L G_R$$

2:-1 Block:

$$P_{2:-1} = X$$

$$G_{2:-1} = G_{2:1} + P_{2:1} G_{0:-1}$$

$$= 1 + 1 \cdot 1$$

$$= 1$$

We calculate 3-bit spans too:

1:-1 Block:

$$P_{1:-1} = X$$

$$G_{1:-1} = G_{1:1} + P_{1:1} G_{0:-1}$$

$$= 1 + 1 \cdot 1$$

$$= 1$$

Step 4. Continue to calculate P's and G's for larger blocks (8-bit, 16-bit, etc.)

Step 5. Use prefixes to calculate sums

$$C_{-1} = G_{-1:-1} = 1$$

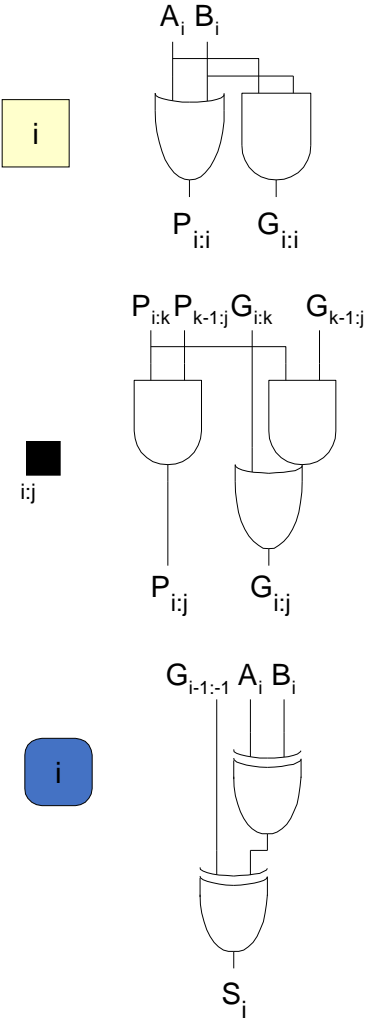
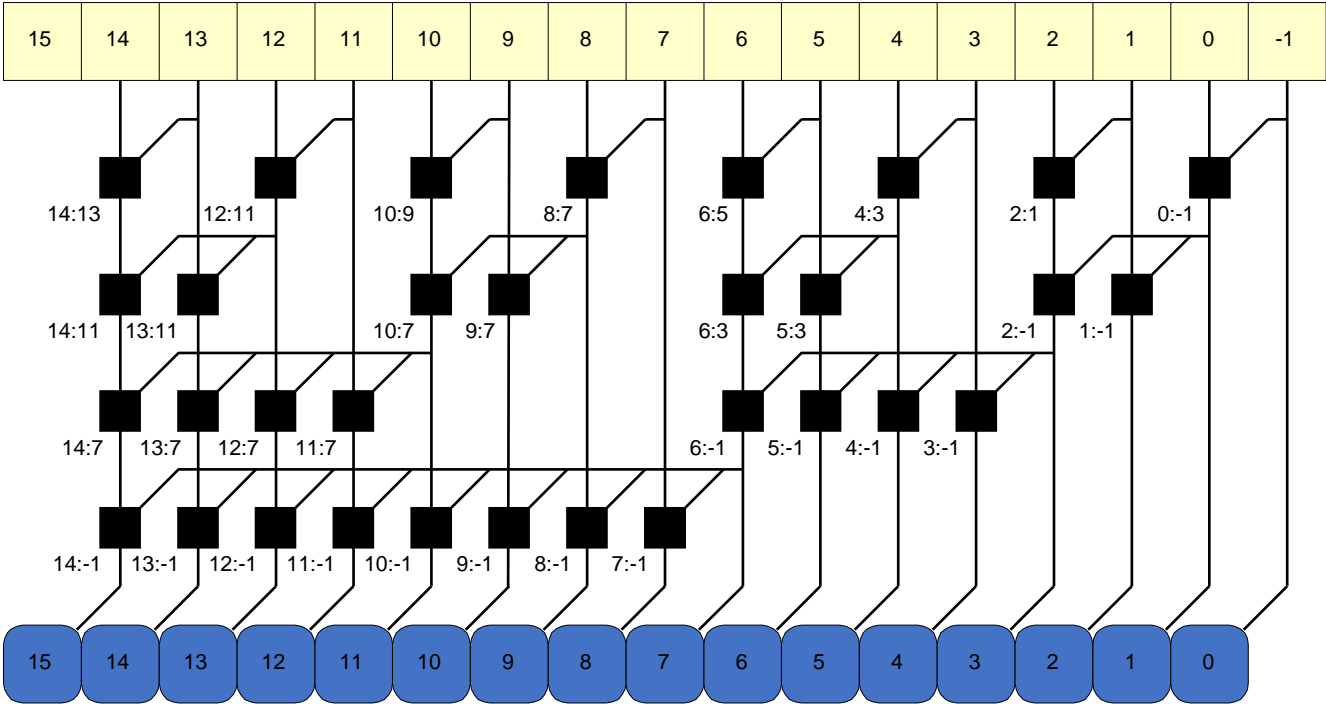
$$C_0 = G_{0:-1} = 1$$

$$C_1 = G_{1:-1} = 1$$

$$C_2 = G_{2:-1} = 1$$

$$S_i = A_i \oplus A_i \oplus C_{i-1}$$

Prefix Adder Schematic



Prefix Adder (PA) Delay

For N -bit PA with k -bit blocks:

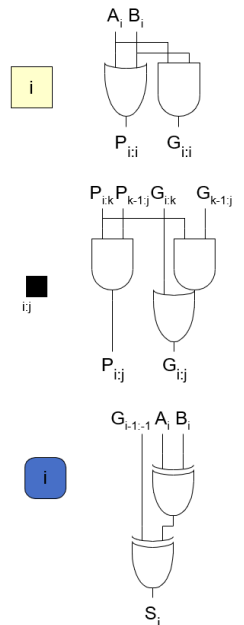
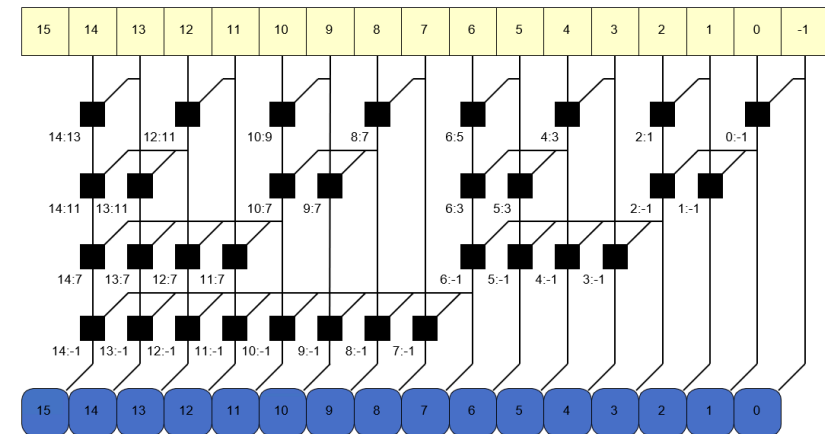
- t_{PA} ... time/delay to compute outputs

$$t_{PA} = t_{PG} + \log_2 N(t_{PG_{Prefix}}) + t_{xor}$$

- t_{PG} ... delay to produce P_i, G_i (AND or OR gate)
- $t_{PG_{Prefix}}$... delay of black prefix cell (AND-OR gate)

• Question: why not

~~$$t_{PA} = t_{PG} + \log_2 N(t_{PG_{Prefix}}) + 2t_{xor}$$~~



Adder Delay Comparisons

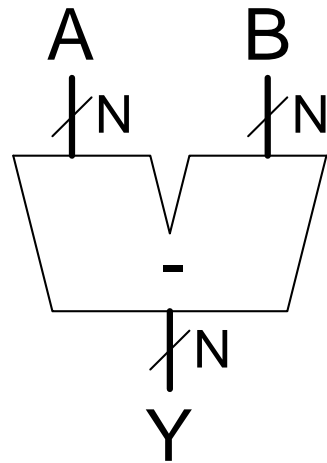
- Compare the delay of: 32-bit ripple-carry, CLA, and prefix adders
- CLA has 4-bit blocks
- 2-input gate delay = 100 ps; full adder delay = 300 ps
- $t_{ripple} = Nt_{FA}$
$$= 32(300 \text{ ps}) = \mathbf{9.6 \text{ ns}}$$
- $t_{CLA} = t_{PG} + t_{PG_{Block}} + \left(\frac{N}{k} - 1\right) t_{AND\&OR} + kt_{FA}$
$$= [100 + 600 + (7)200 + 4(300)] \text{ ps} = \mathbf{3.3 \text{ ns}}$$
- $t_{PA} = t_{PG} + \log_2 N \left(t_{PG_{Prefix}} \right) + t_{XOR}$
$$= [100 + \log_2 32(200) + 100] \text{ ps} = \mathbf{1.2 \text{ ns}}$$

Subtractors & Comparators

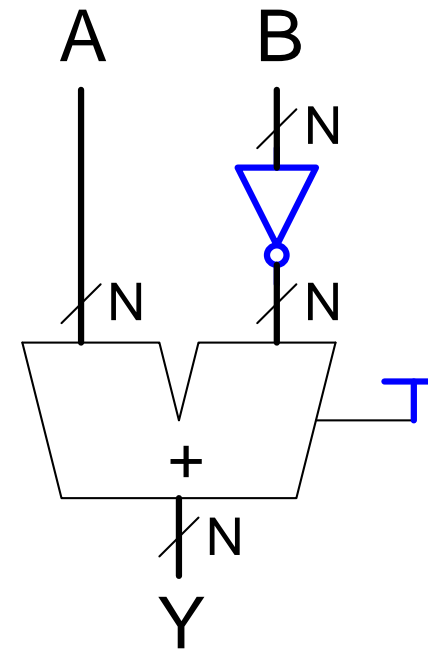
DDCA Ch5 - Part 6: Subtractors & Comparators <https://youtu.be/ZbZ33tj-ncg?si=K-82X-BYVB-5omYP>

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

Symbol



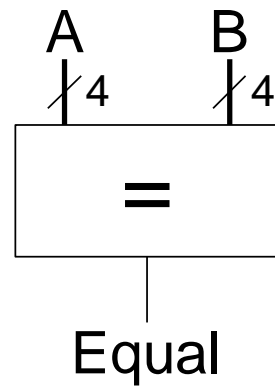
Implementation



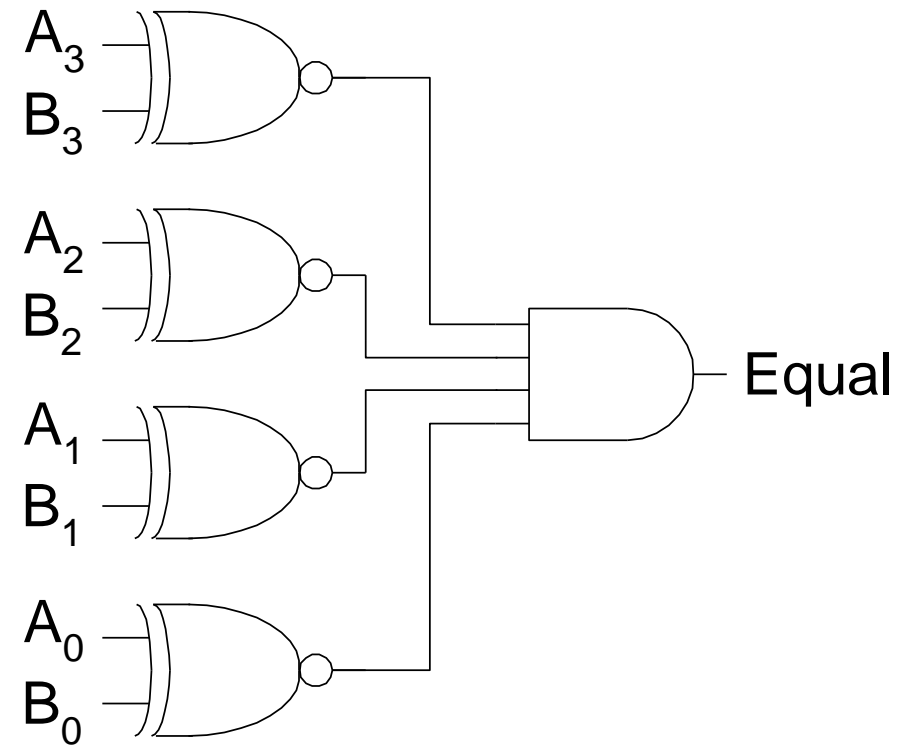
Comparator: Equality

$A == B$

Symbol



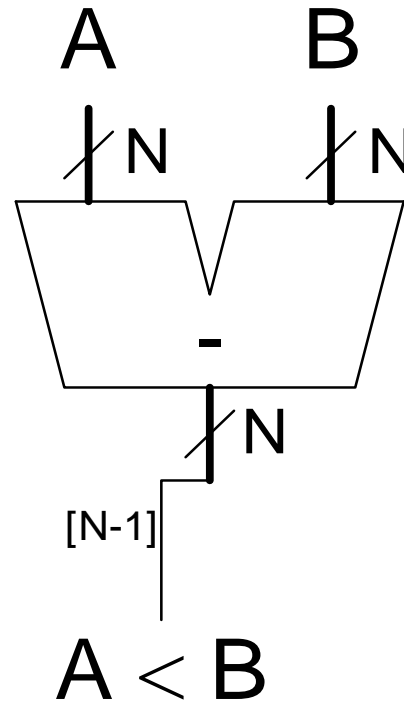
Implementation



Comparator: Signed Less Than

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

Beware of overflow



ALU: Arithmetic Logic Unit

DDCA Ch5 - Part 7: ALUs <https://youtu.be/WXMf0y4NoBw?si=hLaFhTAifuG7liVy>

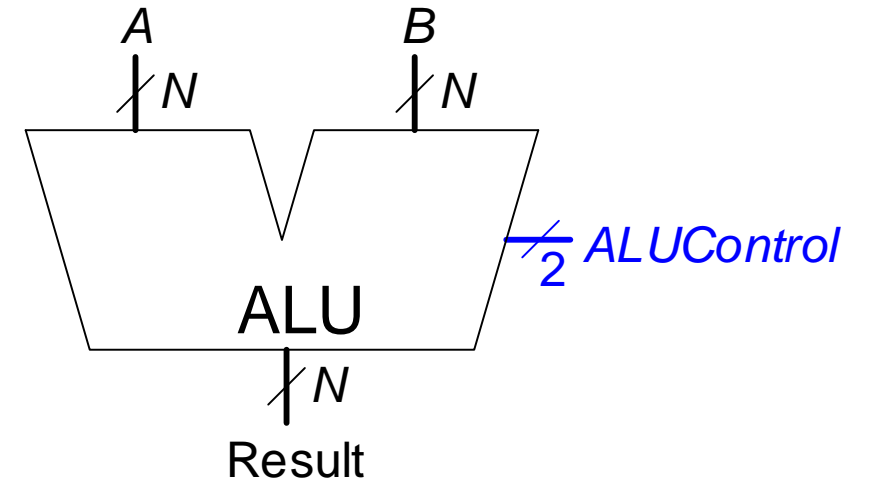
ALU: Arithmetic Logic Unit

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 - OR

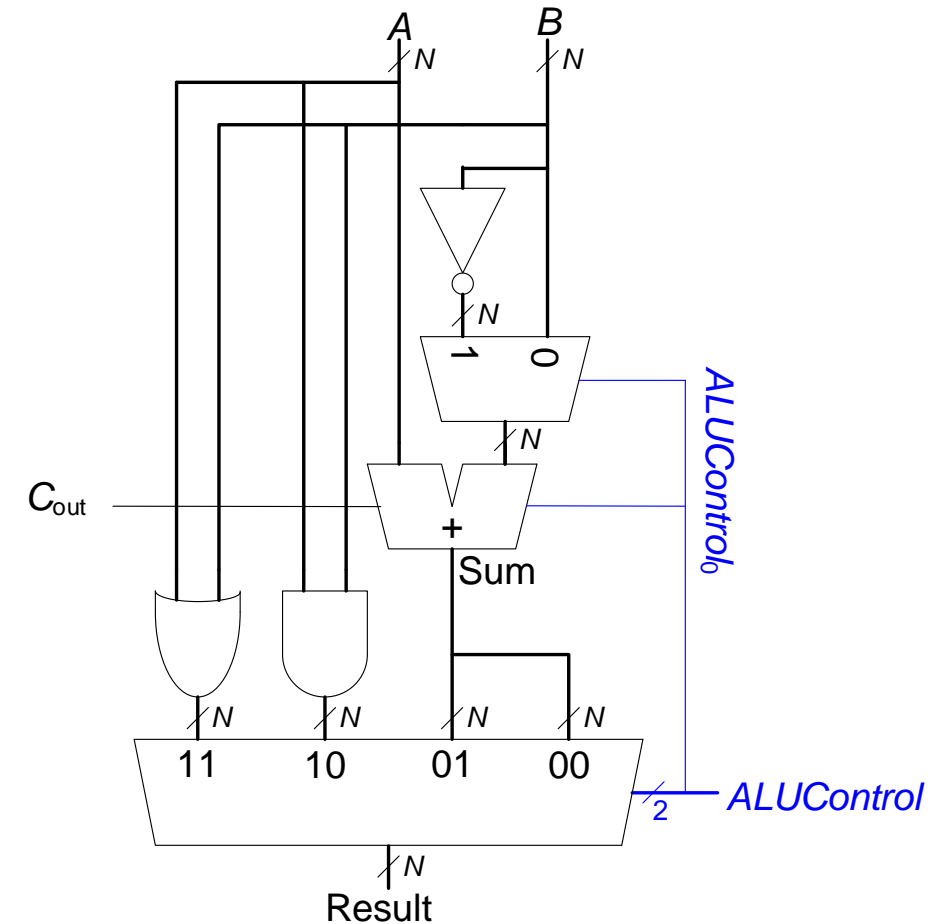
ALUControl _{1:0}	Function
00	Add
01	Subtract
10	AND
11	OR

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 - Add

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

Example: Perform $A + B$

ALUControl_{1:0} = 00

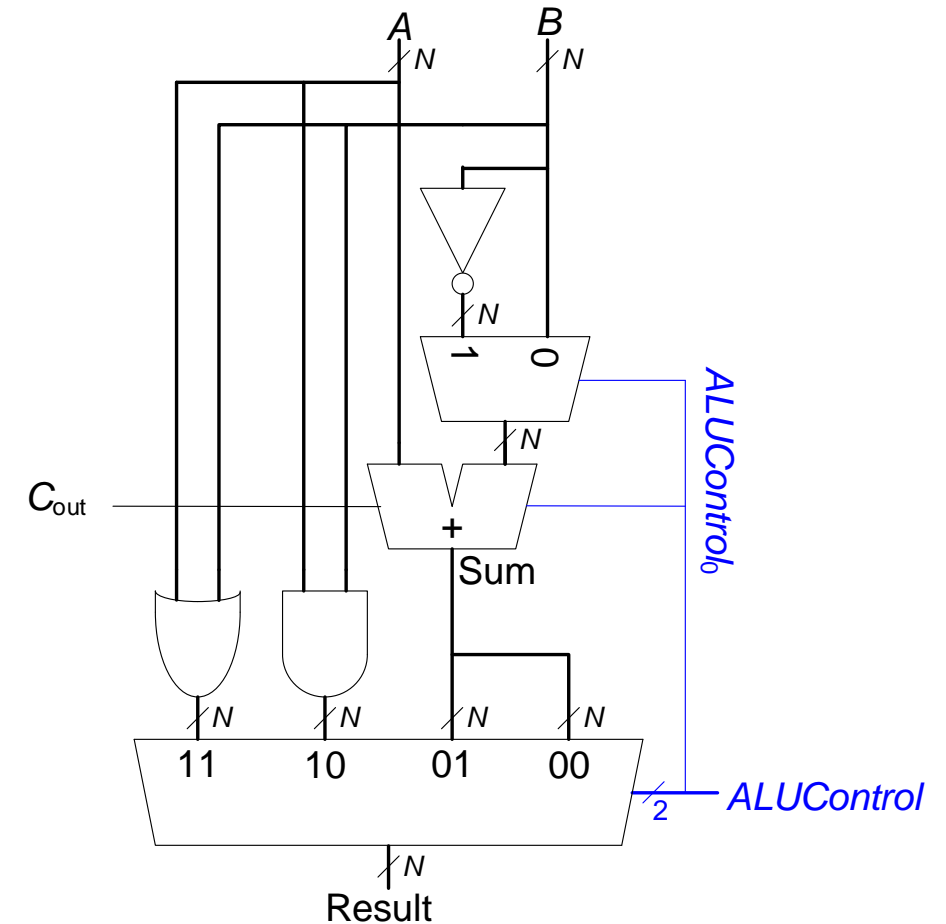
ALUControl₀ = 0, so:

C_{in} to adder = 0

2nd input to adder is B

Mux selects Sum as Result, so

Result = $A + B$



ALU: Arithmetic Logic Unit - Subtract

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

Example: Perform $A - B$

ALUControl_{1:0} = 01

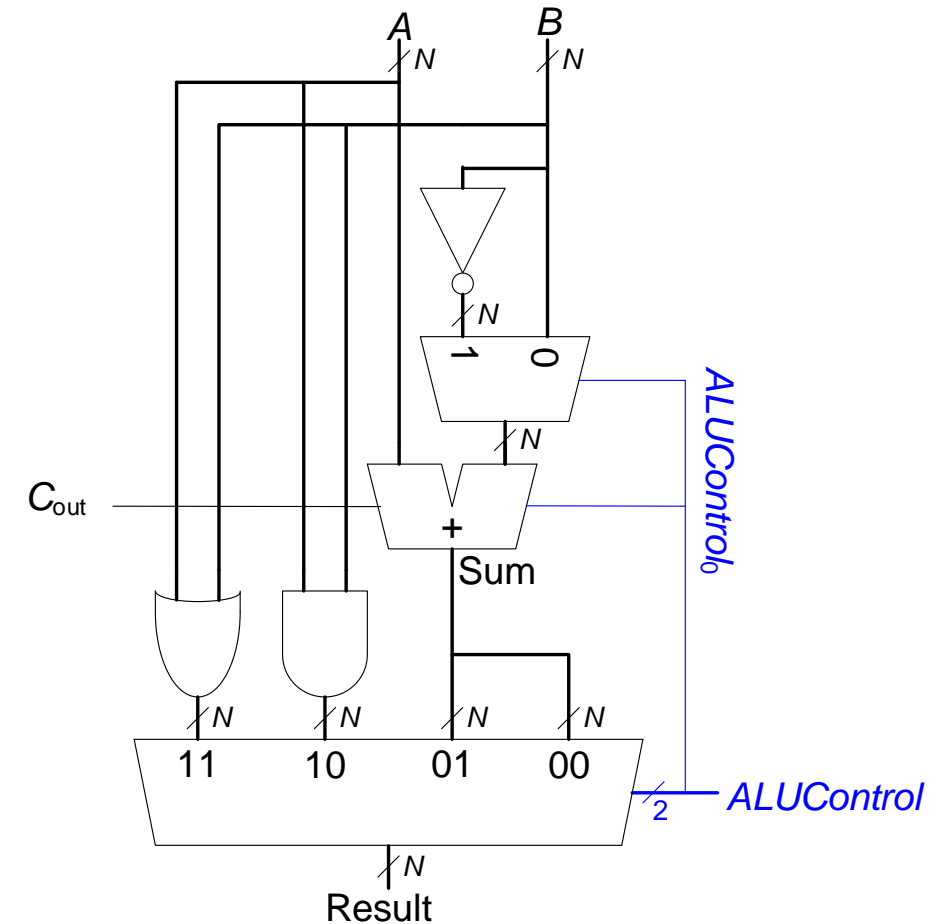
ALUControl₀ = 1, so:

C_{in} to adder = 1

2nd input to adder is inverse of B

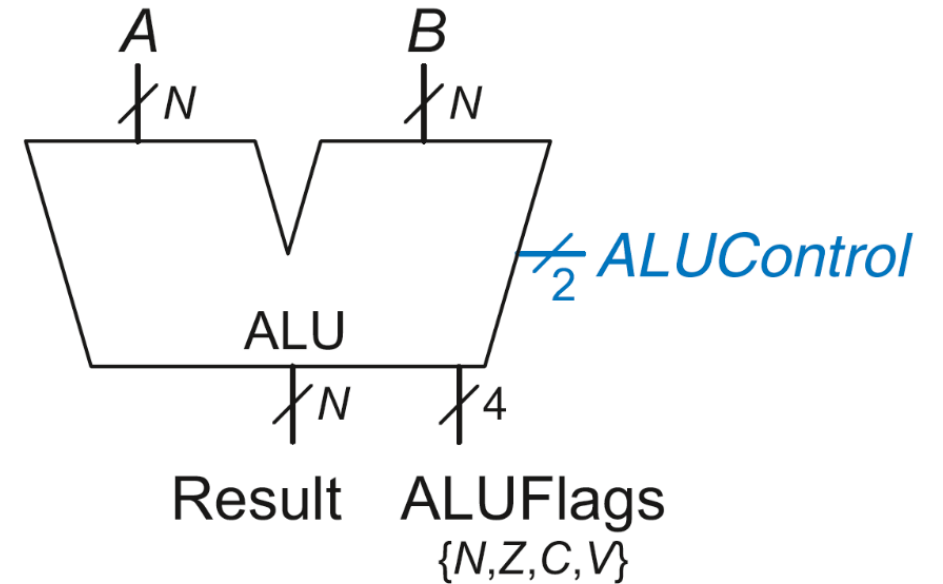
Mux selects Sum as Result, so

Result = $A + \text{not } B + 1$

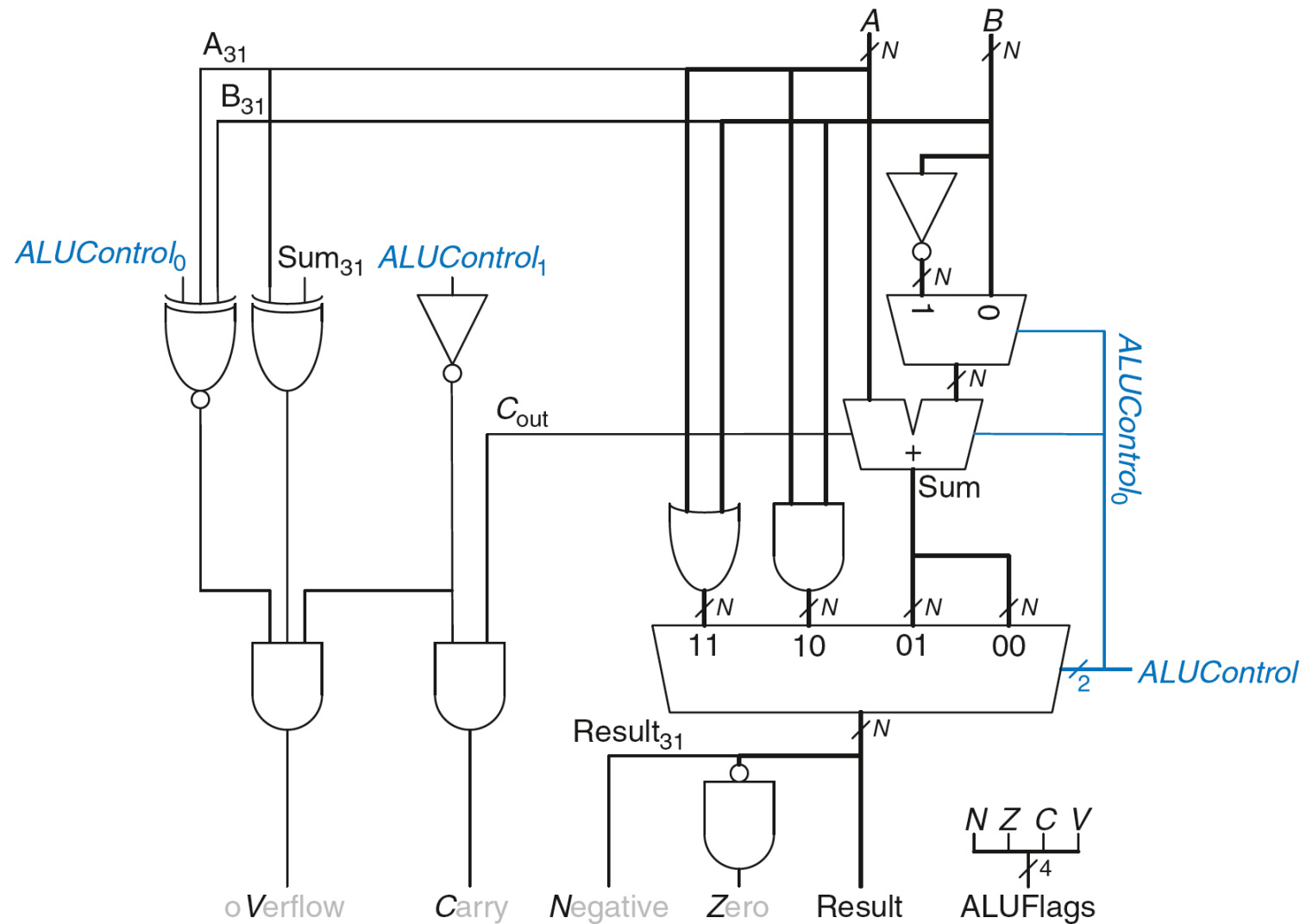


ALU with Status Flags

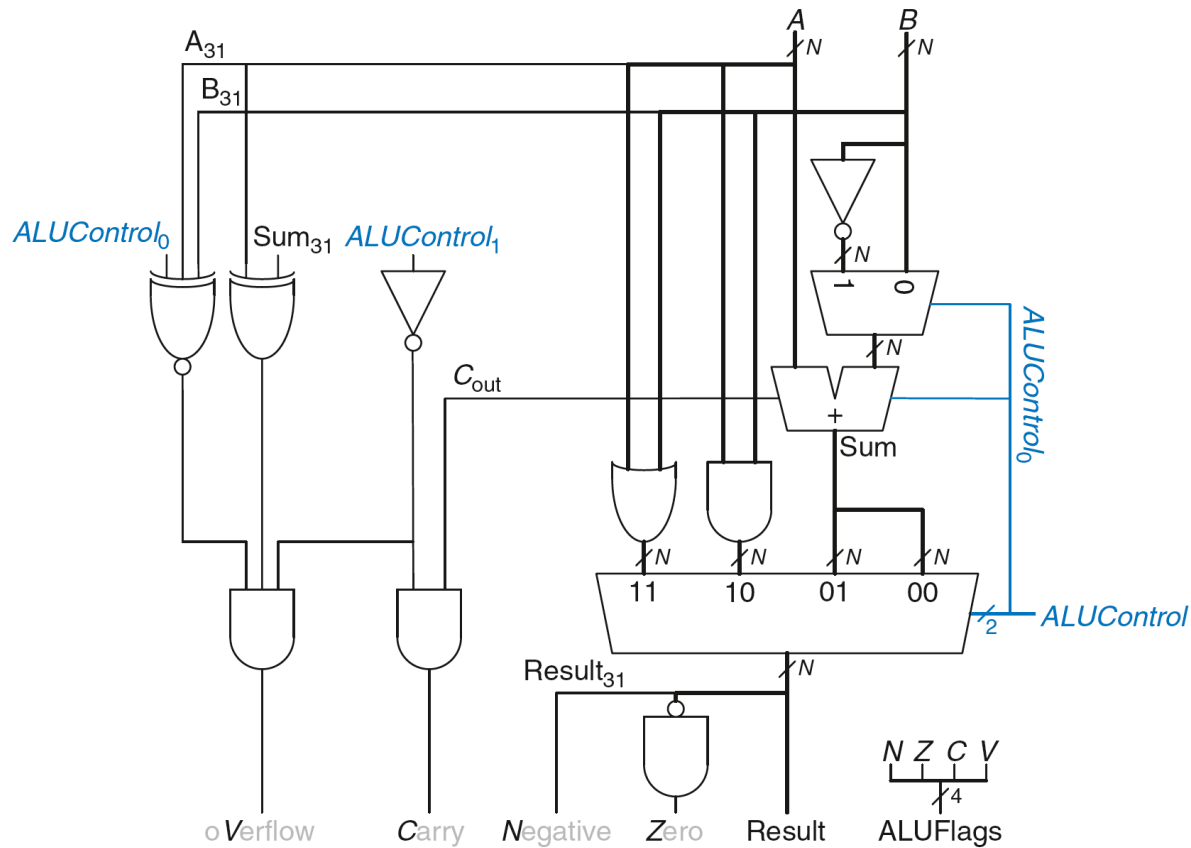
Flag	Description
<i>N</i>	Result is N egative
<i>Z</i>	Result is Z ero
<i>C</i>	Adder produces C arry out
<i>V</i>	Adder o V erflowed



ALU with Status Flags



ALU with Status Flags: Negative

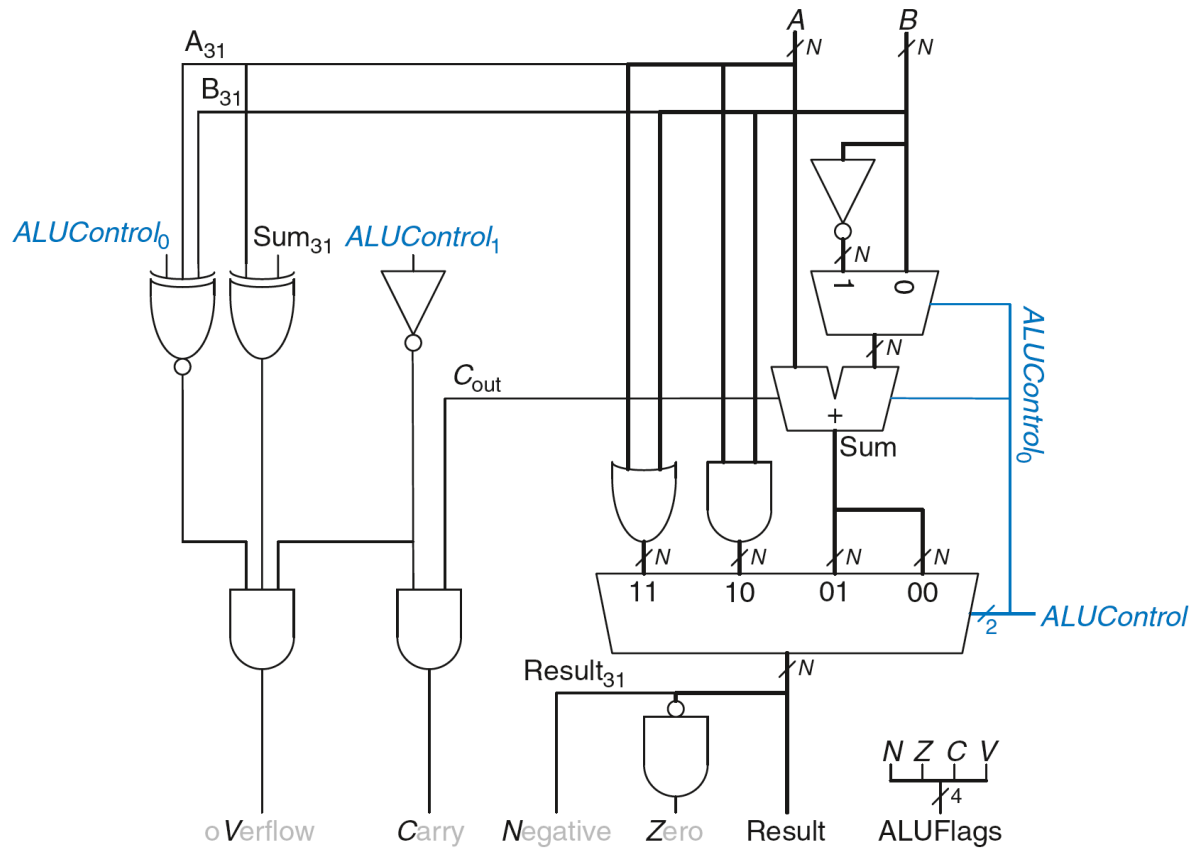


$N = 1$ if:

- Result is negative

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

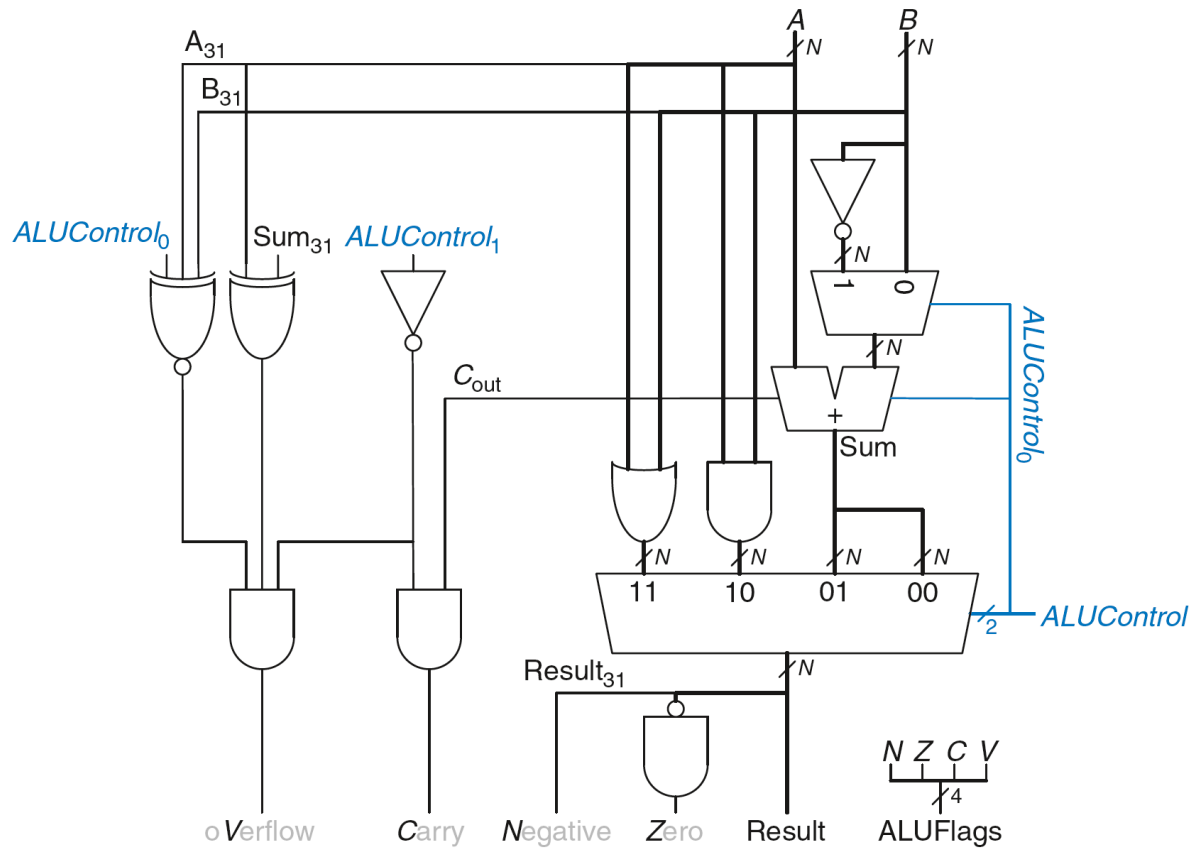
ALU with Status Flags: Zero



$Z = 1$ if:

- all of the bits of Result are 0

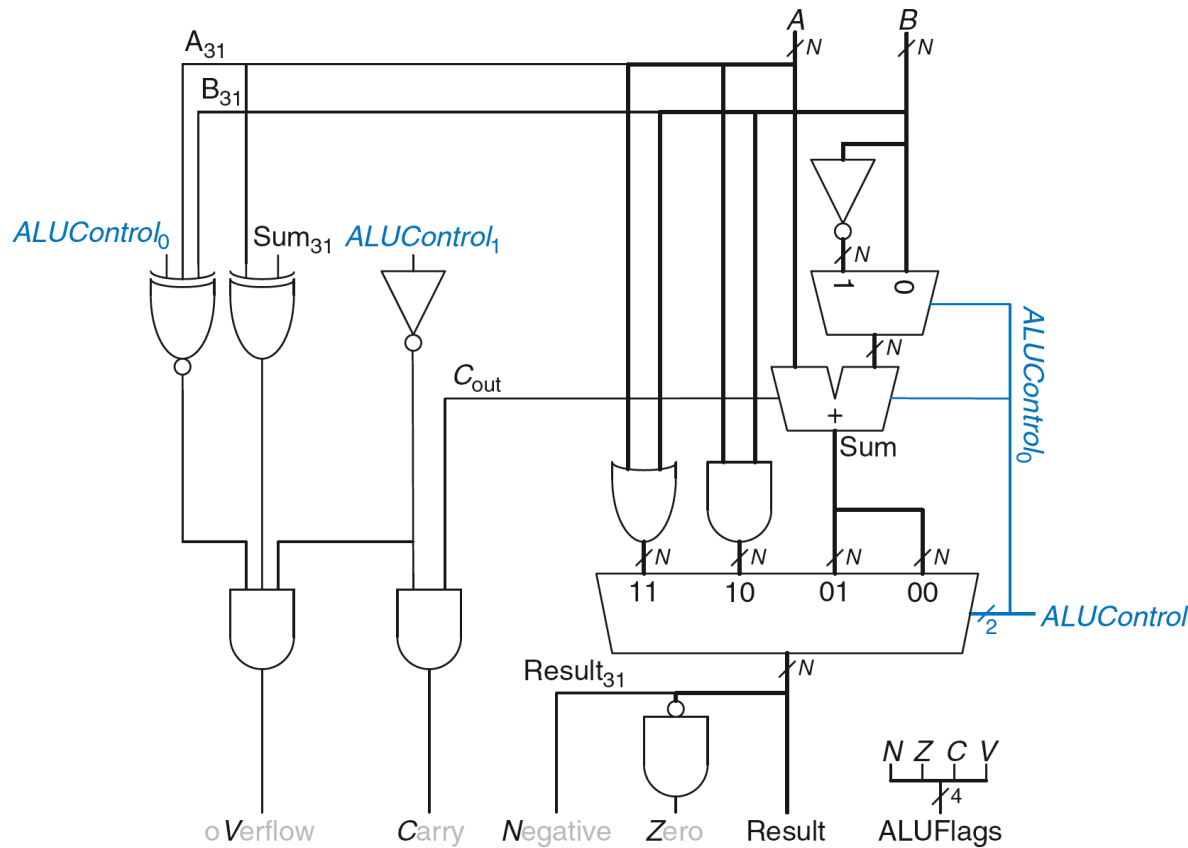
ALU with Status Flags: Carry



C = 1 if:

- C_{out} of Adder is 1
AND
- ALU is adding or subtracting
(ALUControl is 00 or 01)

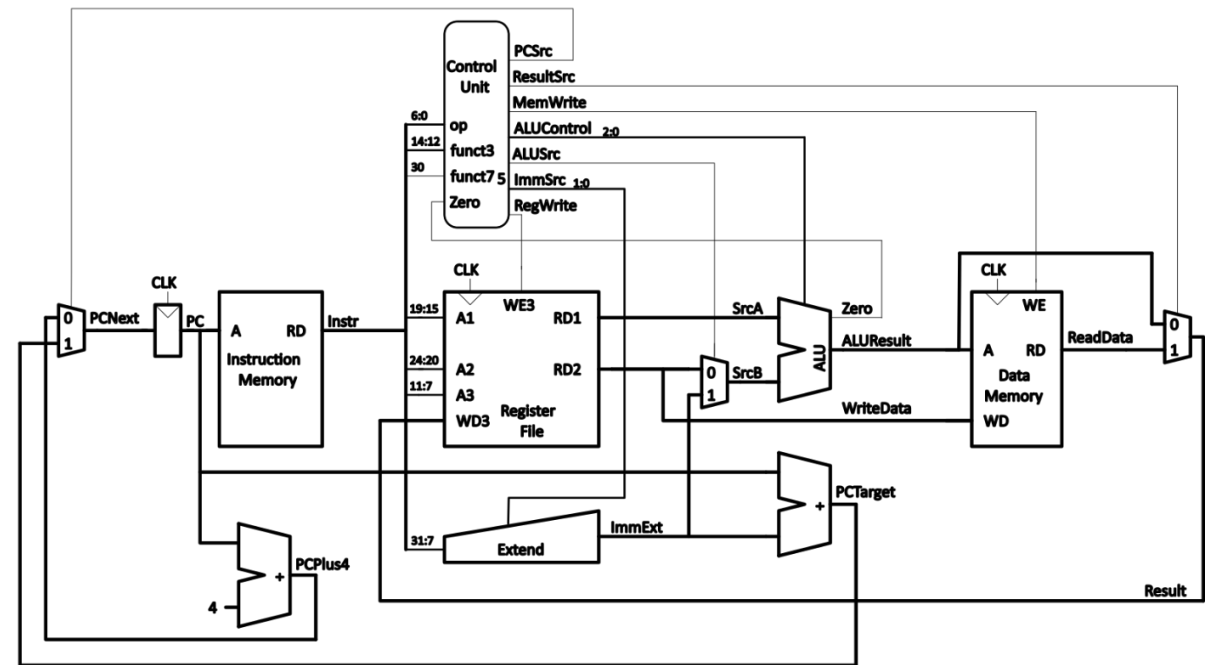
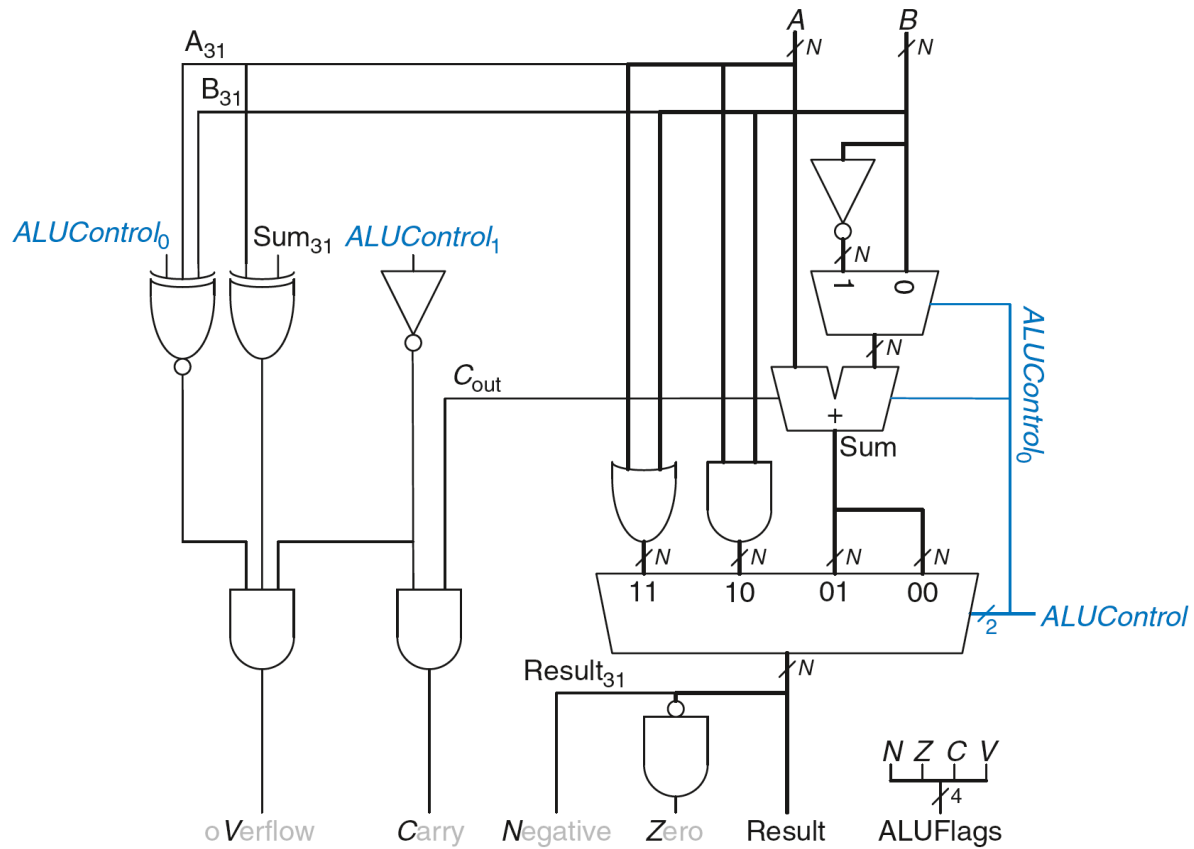
ALU with Status Flags: oVerflow



$V = 1$ if:

- ALU is performing addition or subtraction
(ALUControl1 = 0)
AND
- A and Sum have opposite signs
AND
- A and B have same signs for addition
(ALUControl0 = 0)
OR
- A and B have different signs for subtraction
(ALUControl0 = 1)

ALU with Status Flags: Why??



Comparison based on Flags

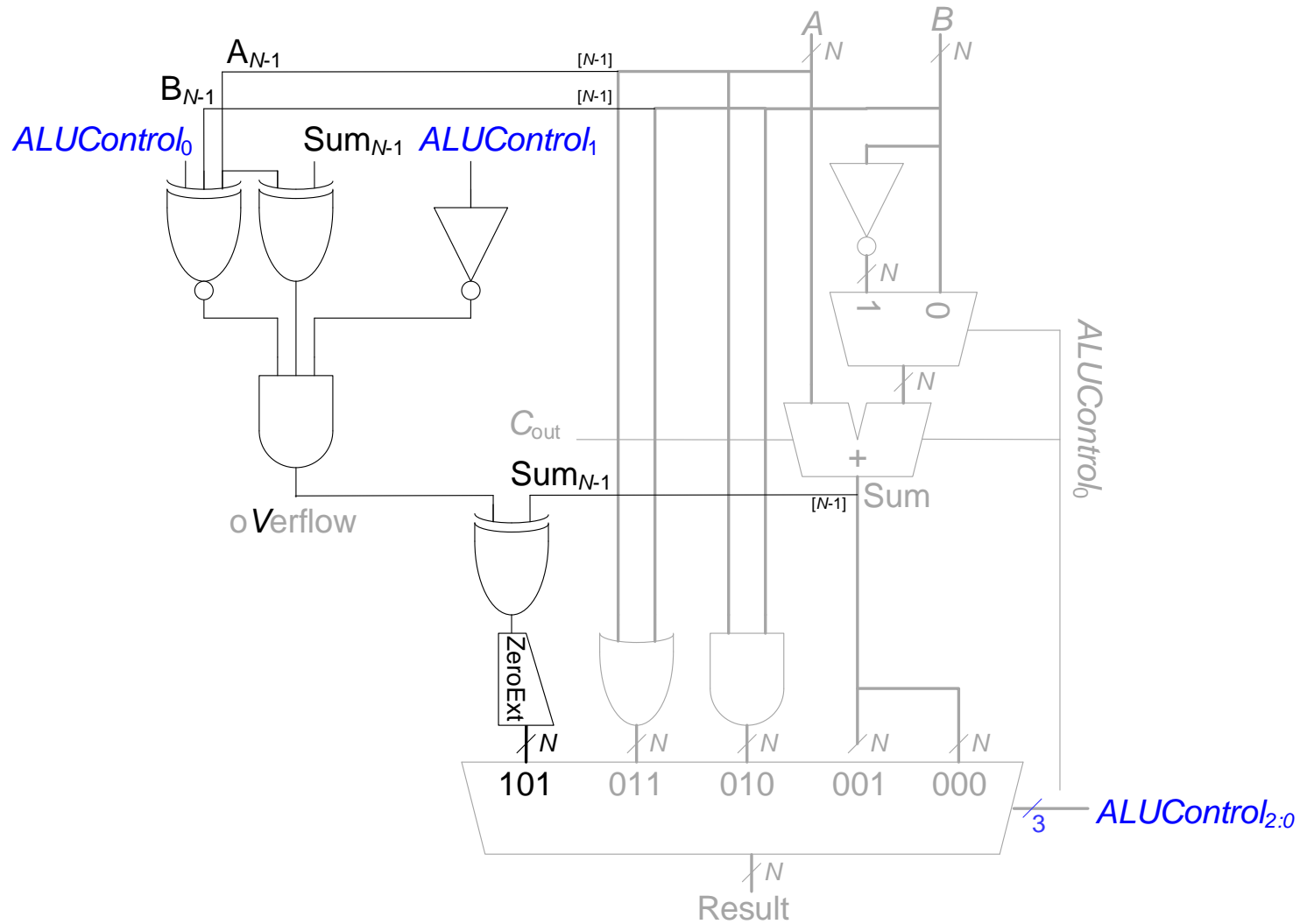
Compare by subtracting and checking flags

Different for signed and unsigned

Comparison	Signed	Unsigned
==	Z	Z
!=	\bar{Z}	\bar{Z}
<	$N \oplus V$	\bar{C}
<=	$Z \vee (N \oplus V)$	$Z \vee \bar{C}$
>	$\bar{Z} \wedge (\overline{N \oplus V})$	$\bar{Z} \wedge C$
>=	$\overline{(N \oplus V)}$	C

- **Set Less Than** (also called Set if Less Than)
 - **Sets lsb** of result **if $A < B$**
 - Result = 0000...001 if $A < B$
 - Result = 0000...000 otherwise
 - Comes in signed and unsigned flavors
- **XOR**
 - Result = $A \text{ XOR } B$

Fixing Overflow Error in SLT Logic



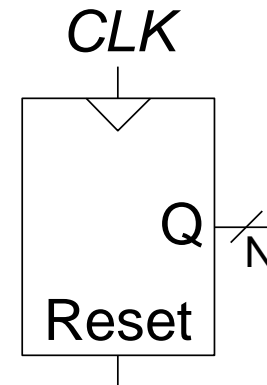
Counters & Shift Registers

DDCA Ch5 - Part 12: Counters & Shift Registers <https://www.youtube.com/watch?v=okczOaycfqk>

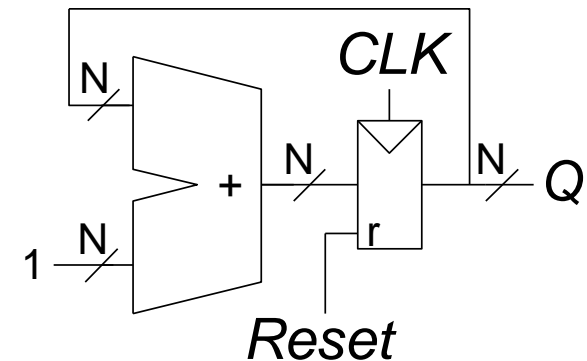
Counters

- 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



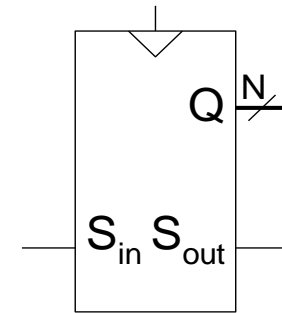
Divide-by-2^N Counter

- Most significant bit of an N -bit counter toggles every 2^N cycles.
 - $f_{out} = \frac{f_{clk}}{2^N}$
- Useful for slowing a clock.
 - E.g.: blink an LED
- Example: 50 MHz clock, 24-bit counter
 - $50\text{ MHz} = 50000000\text{ Hz}$
 - $2^{24} = 16777216$
 - $\frac{50000000\text{ Hz}}{16777216} = 2.98\text{ Hz}$

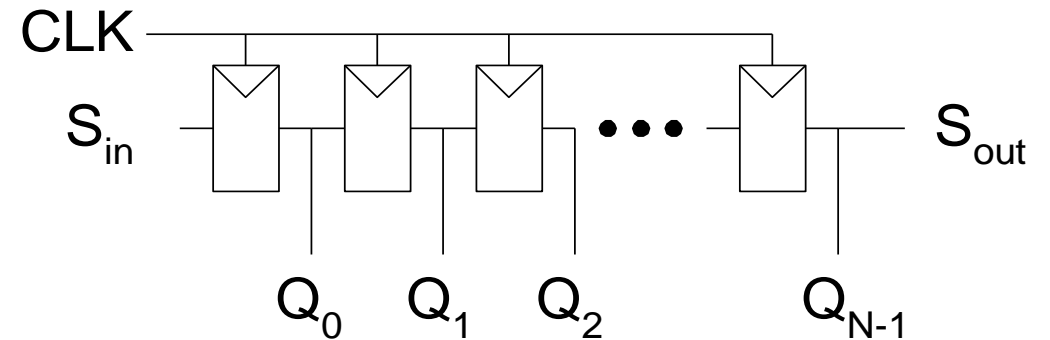
Shift Registers

- 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

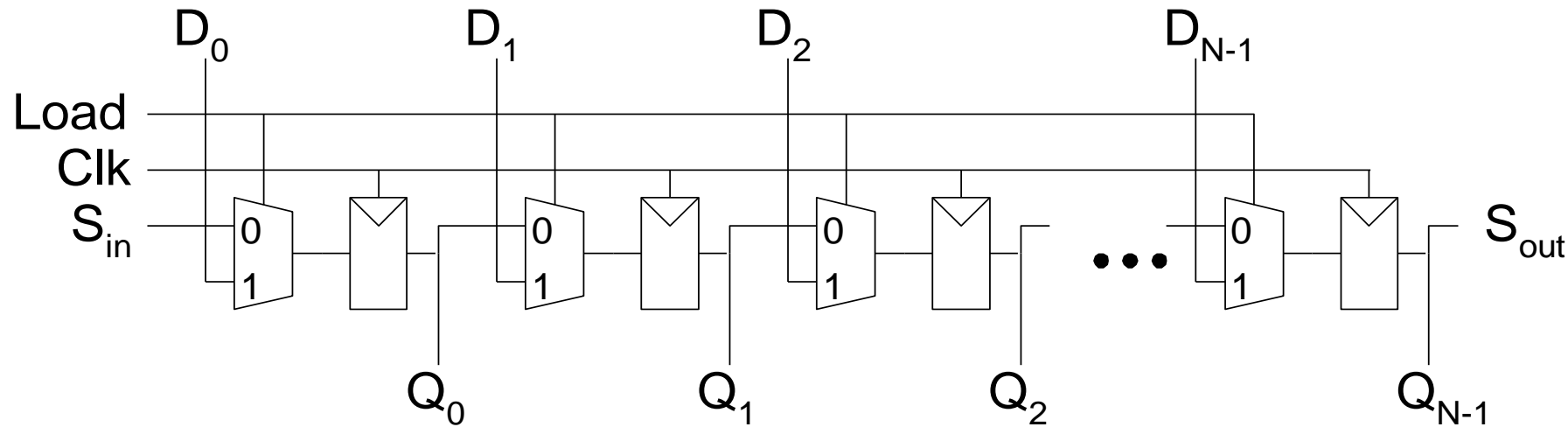


Implementation



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})

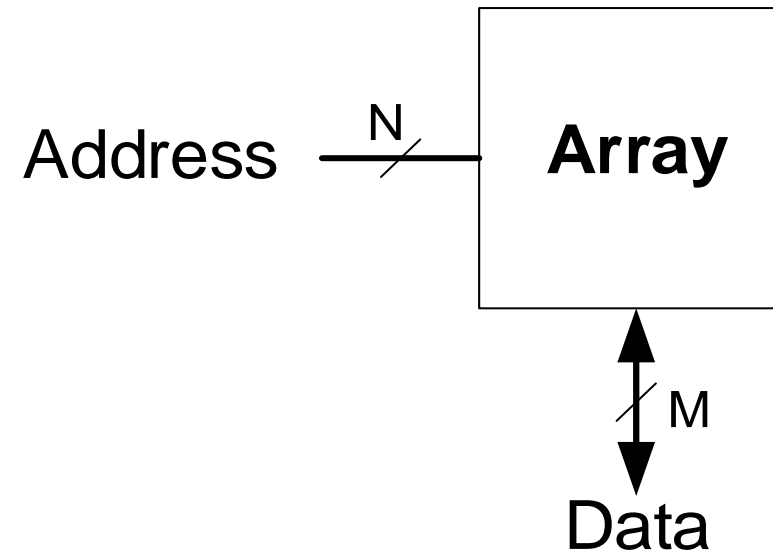


Memory

DDCA Ch5 - Part 13: Memory Introduction <https://www.youtube.com/watch?v=x2NfNfMbIJE>

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

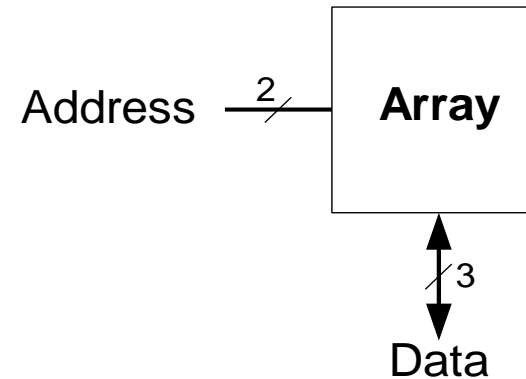
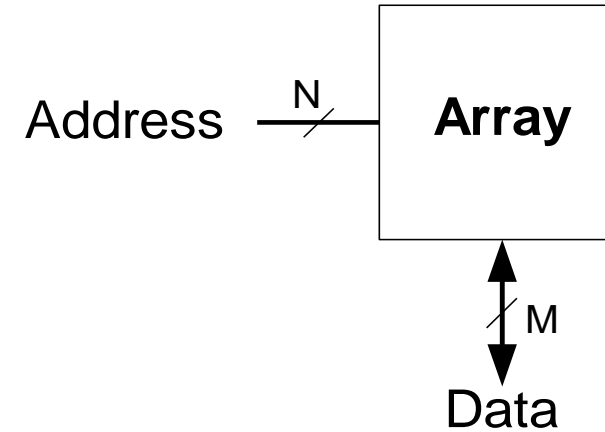
- 2-dimensional array of bit cells
- Each bit cell stores one bit
- N address bits and M data bits:
 - 2^N rows and M columns
 - Depth: number of rows (number of words)
 - Width: number of columns (size of word)
 - Array size: depth \times width = $2^N \times M$

- Example

$N = 2, M = 3,$

Array Size?

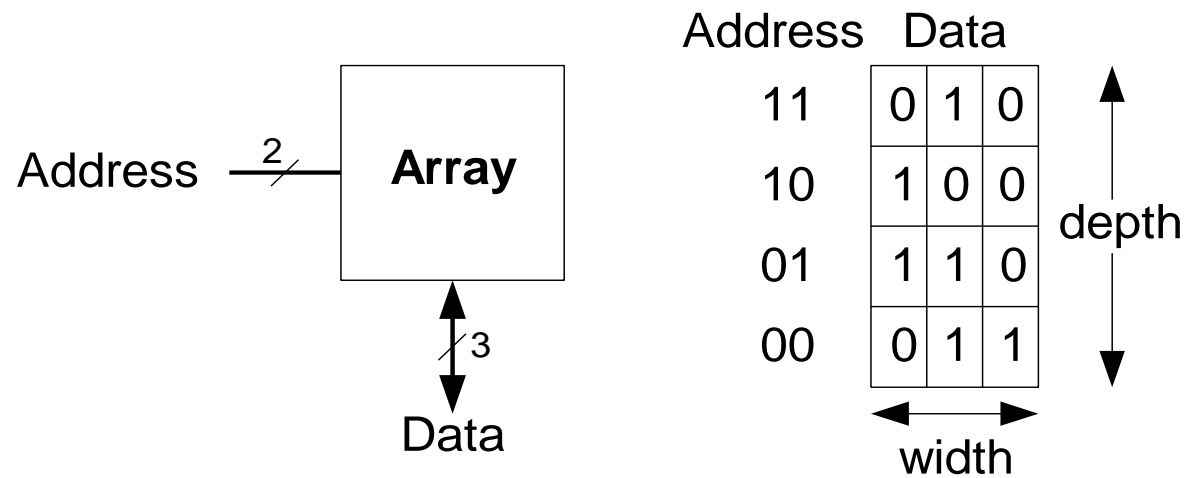
$2^2 \times 3 = 12$ bits

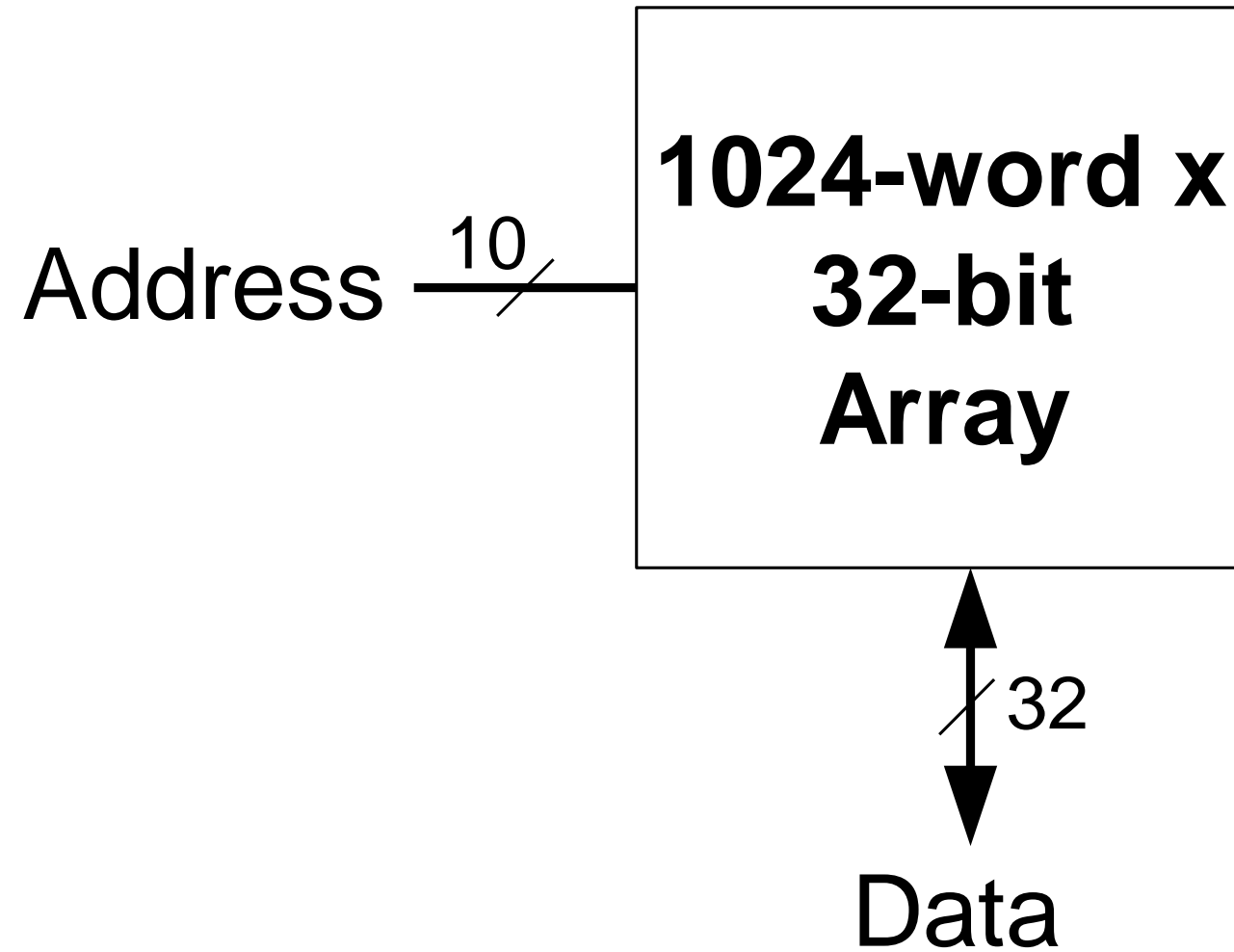


Address	Data			
11	0	1	0	depth ↑ ↓
10	1	0	0	
01	1	1	0	
00	0	1	1	
	width ← →			

Memory Array Example

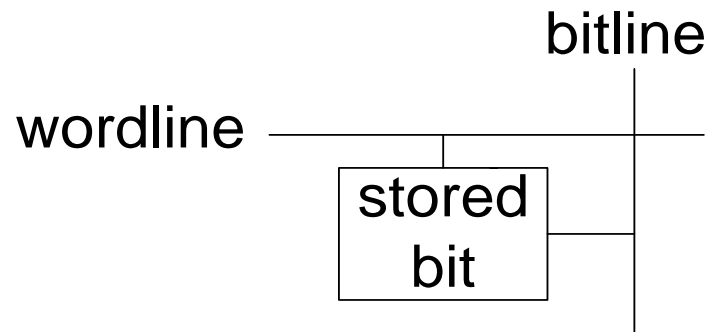
- $2^2 \times 3$ -bit array
- **Number of words:** 4
- **Word size:** 3-bits
- For example, the 3-bit word stored at address 10 is 100





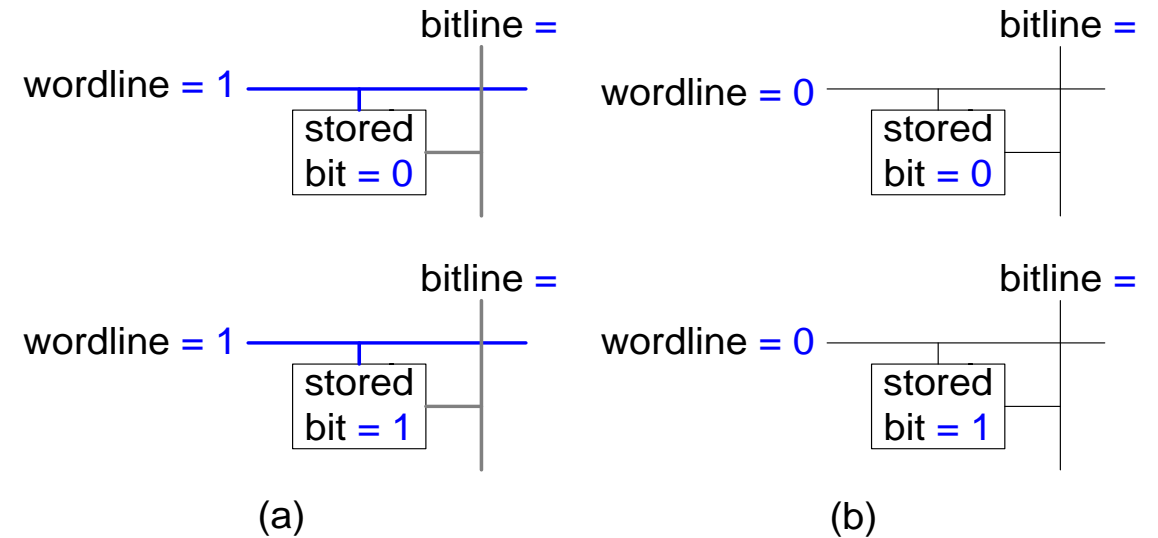
Memory Array Bit Cells

- One Bit Cell



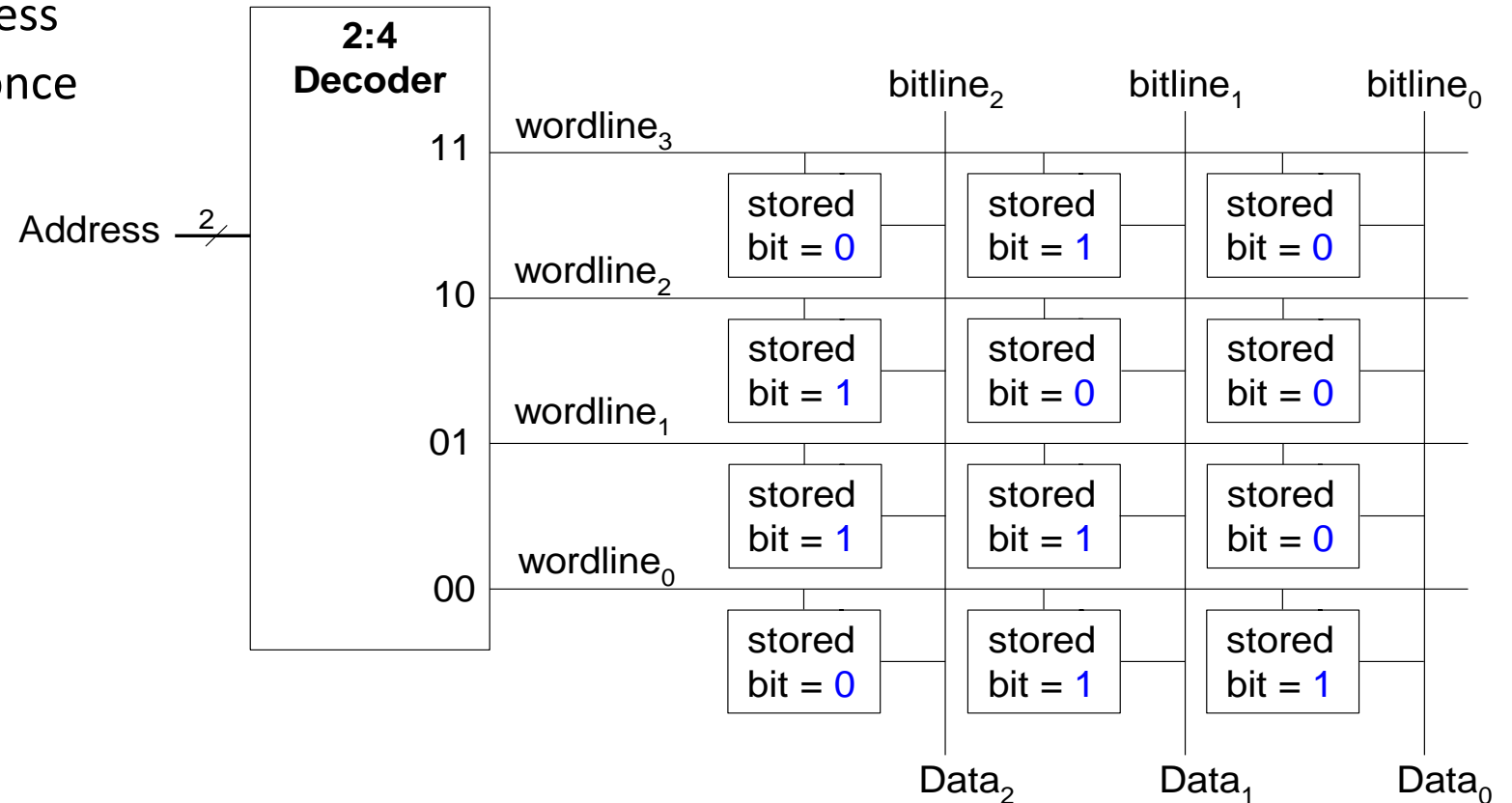
- Reading a One Bit Cell

- Open Collector ?
- Tri-State ?



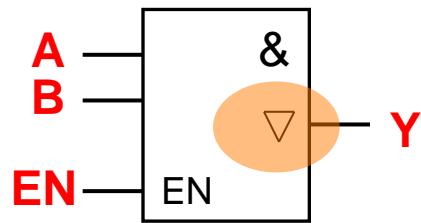
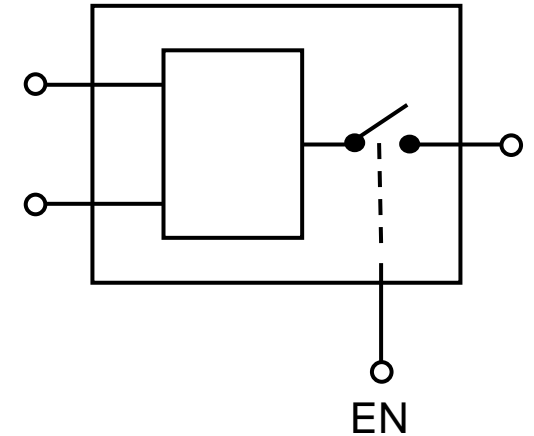
Memory Array

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



Connecting Outputs / Tri-State

- Two logical States
 - HIGH, LOW
- Three physical States
 - HIGH (H), LOW (L), OPEN (Z)
- OPEN-State is normally reachable via ENABLE-Signal



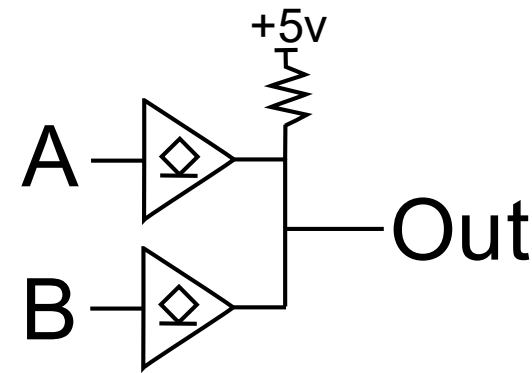
A	B	EN	Y
L	L	H	L
L	H	H	L
H	L	H	L
H	H	H	H
x	x	L	Z

Connecting Outputs / Open-Collector

- Wired Logic

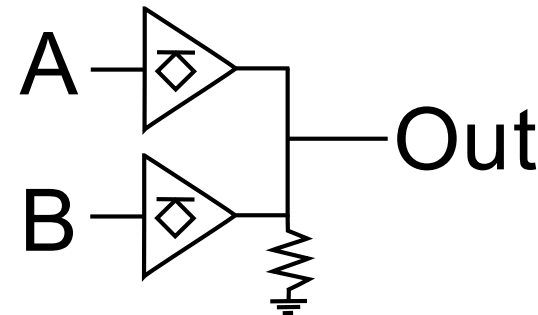
In this example, 5V is considered HIGH (true), and 0V is LOW (false). This gate can be easily extended with more inputs.

- Active-high wired AND connection



Inputs		Output
A	B	A AND B
LOW	LOW	LOW
LOW	HIGH	LOW
HIGH	LOW	LOW
HIGH	HIGH	HIGH

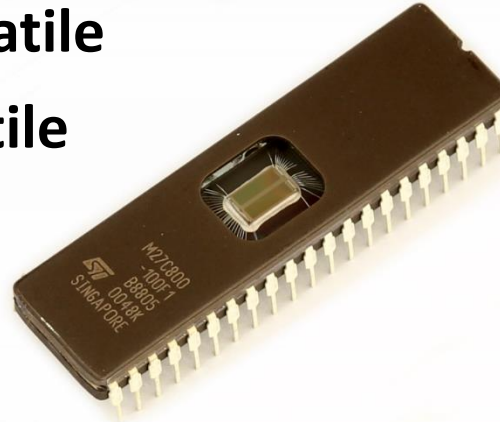
- Active-high wired OR connection



Inputs		Output
A	B	A OR B
LOW	LOW	LOW
LOW	HIGH	HIGH
HIGH	LOW	HIGH
HIGH	HIGH	HIGH

Types of Memory

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



Kanaanitische Sprache:
"Möge dieser Stoßzahn die Läuse in Haar und Bart ausrotten", steht auf dem Fundstück. Es wird auf etwa 1700 Jahre vor Christus datiert.
<https://www.tagesschau.de/wissen/forschung/israel-laeusekamm-elfenbein-inschrift-ieruslaem-101.html>

A500 512Kb RAM-Speicherkarte
mit Echtzeit-Uhr für Commodore
Amiga 500

RAM (Random Access Memory)

DDCA Ch5 - Part 14: RAM <https://www.youtube.com/watch?v=CqZW44iWwYk>

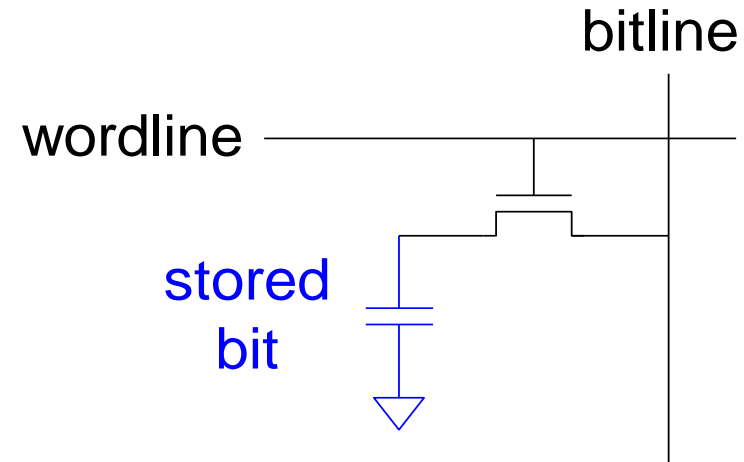
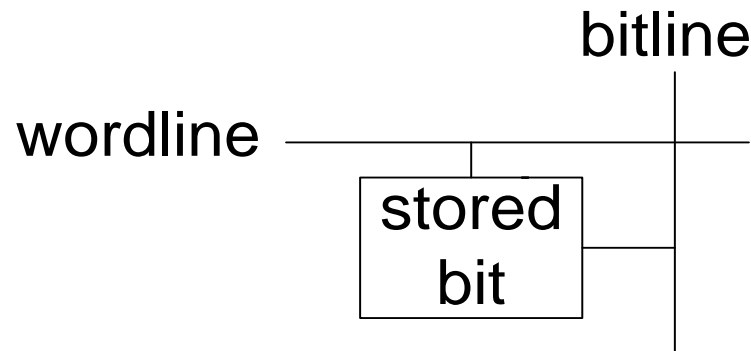
- **DRAM** (Dynamic random access memory)
- **SRAM** (Static random access memory)
- Differ in how they store data:
 - DRAM uses a capacitor
 - SRAM uses cross-coupled inverters

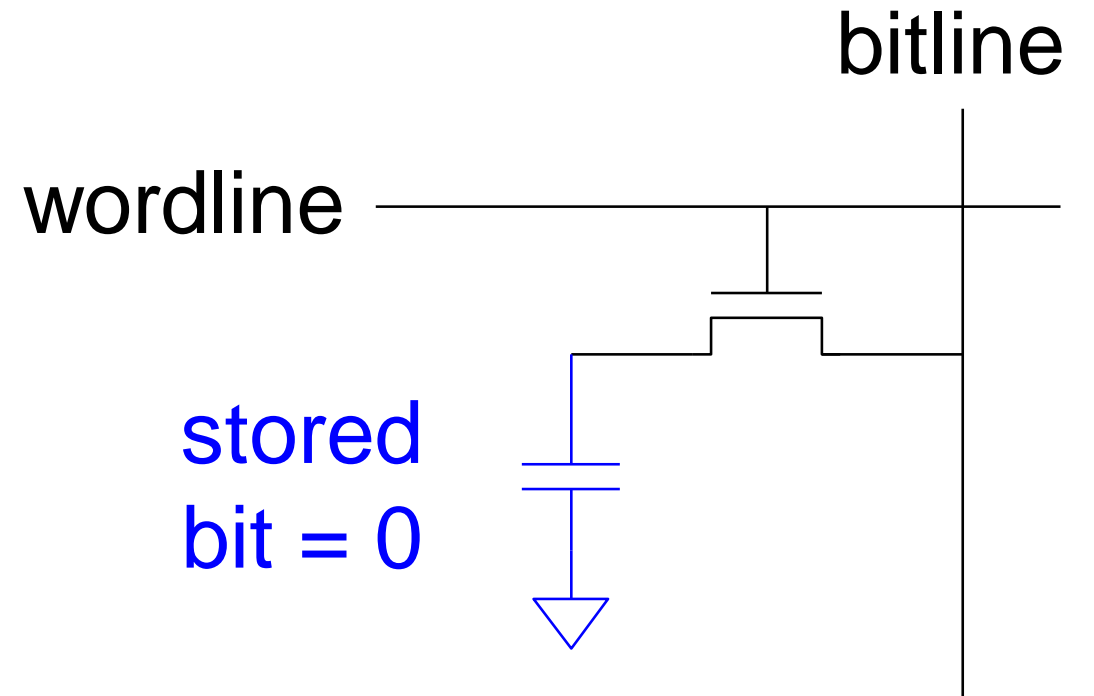
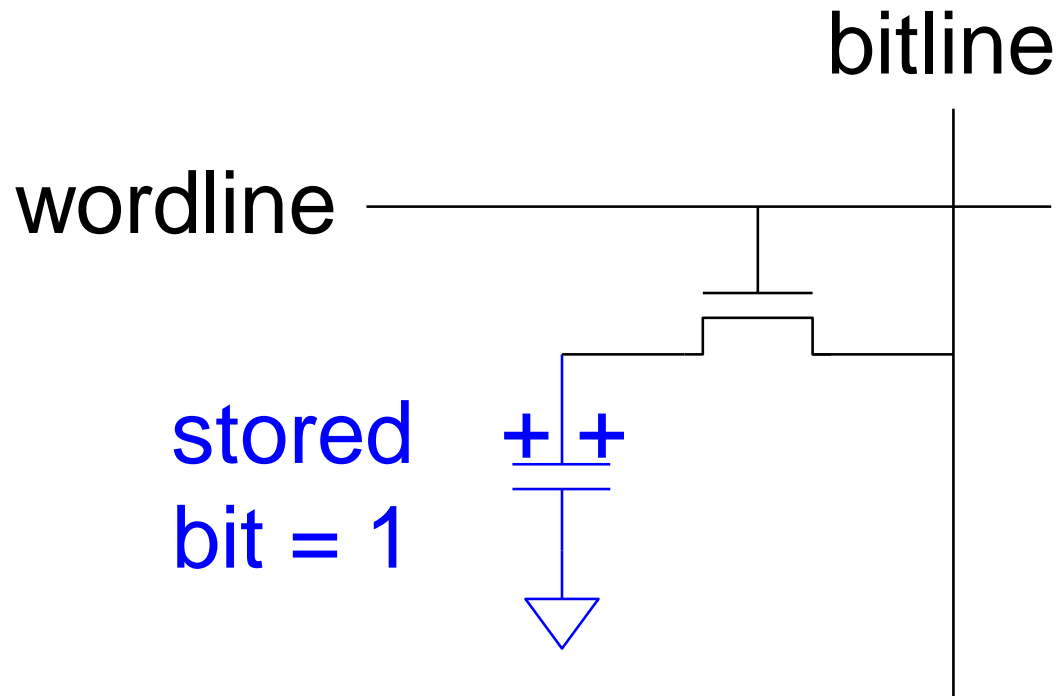
Robert Dennard, 1932 -

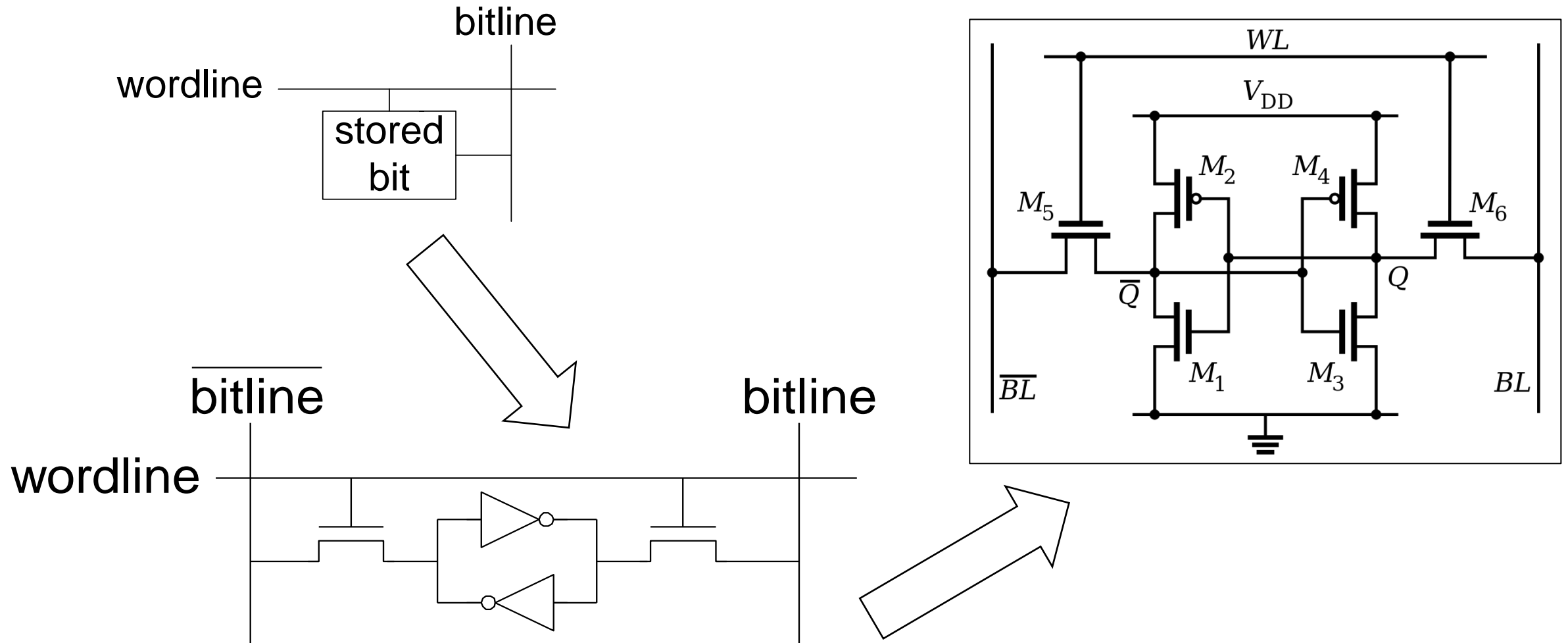
- Invented DRAM in 1966 at IBM
- Others were skeptical that the idea would work
- By the mid-1970's DRAM in virtually all computers



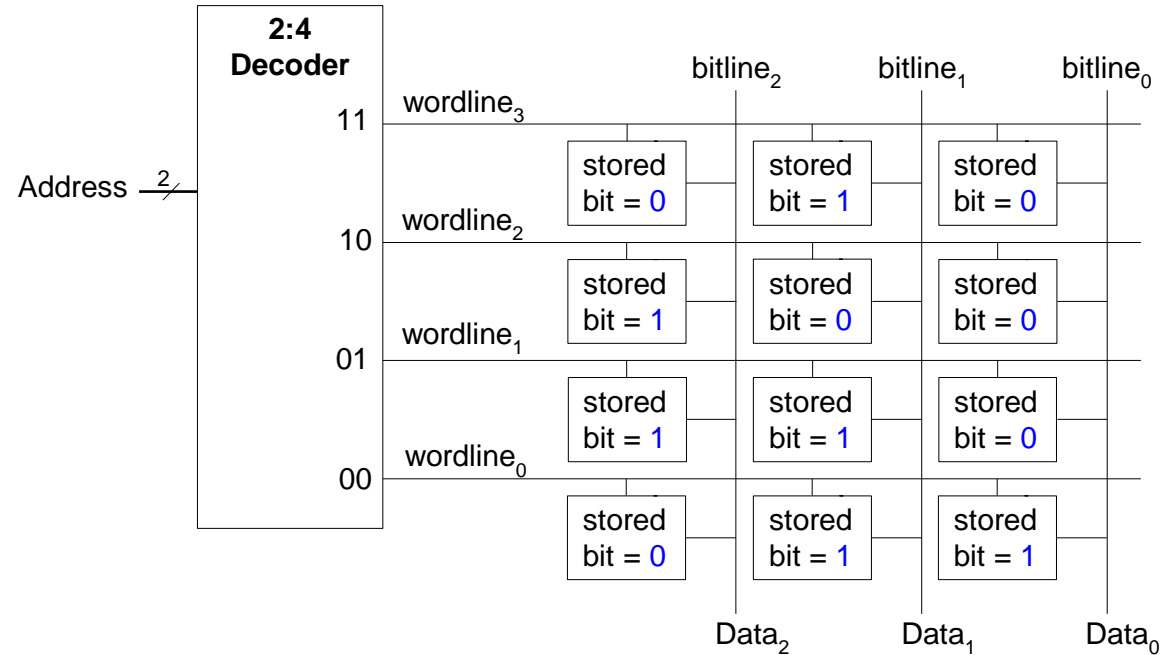
- Data bits stored on **capacitor**
- Dynamic because the value needs to be **refreshed** (rewritten) **periodically** and after read:
 - Charge leakage from the capacitor degrades the value
 - Reading destroys the stored value



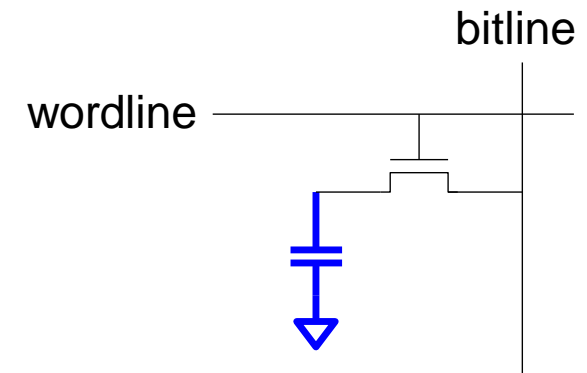




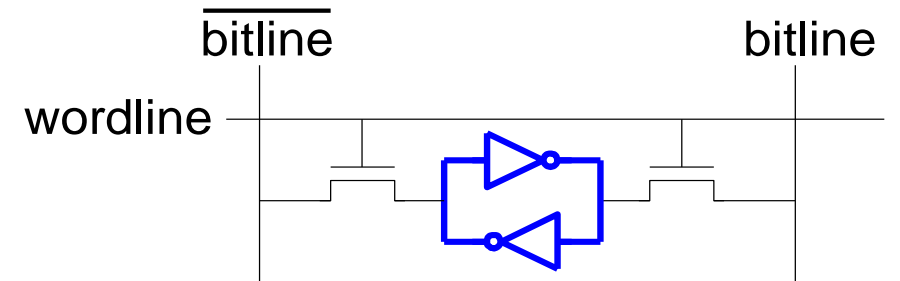
Memory Arrays Review



- DRAM



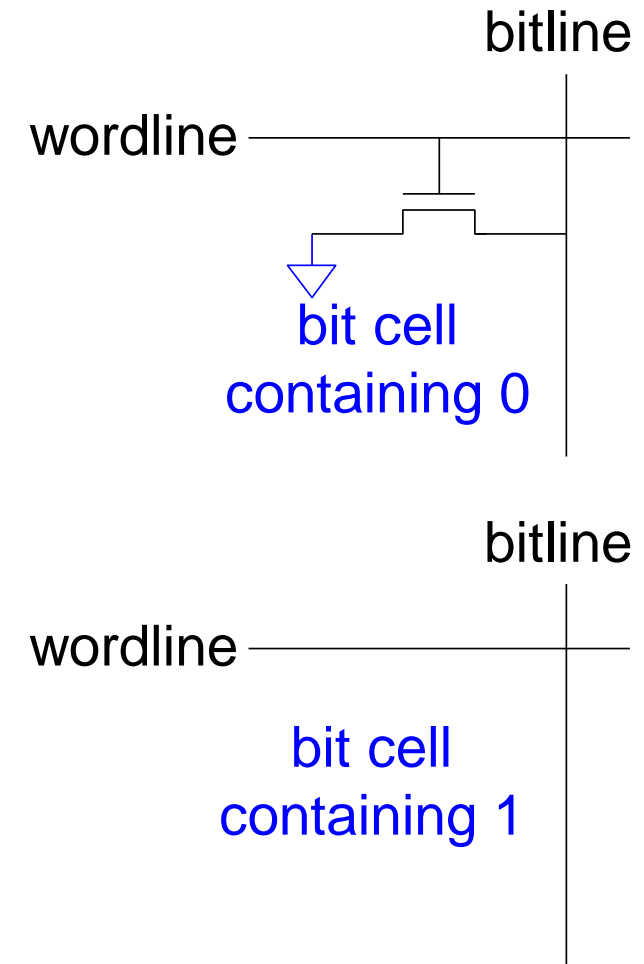
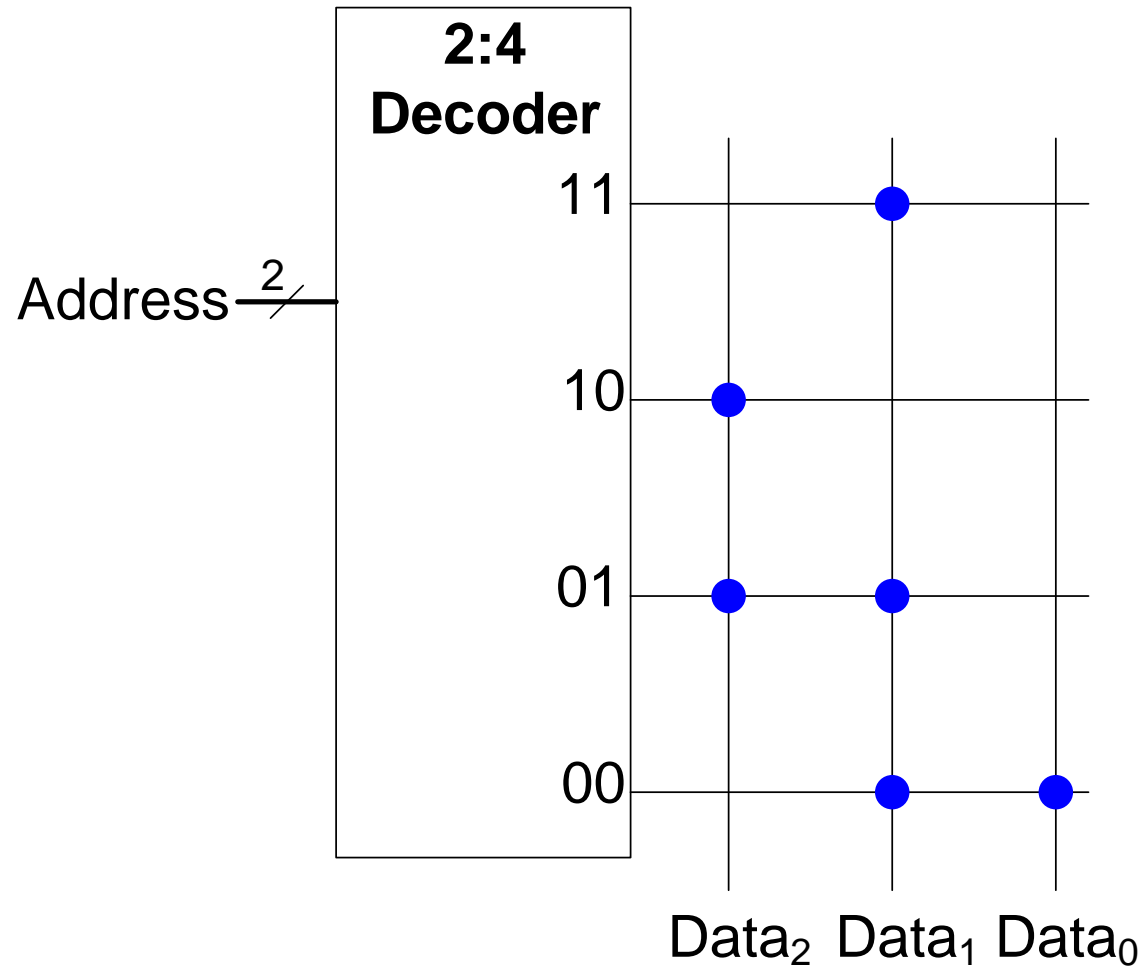
- SRAM



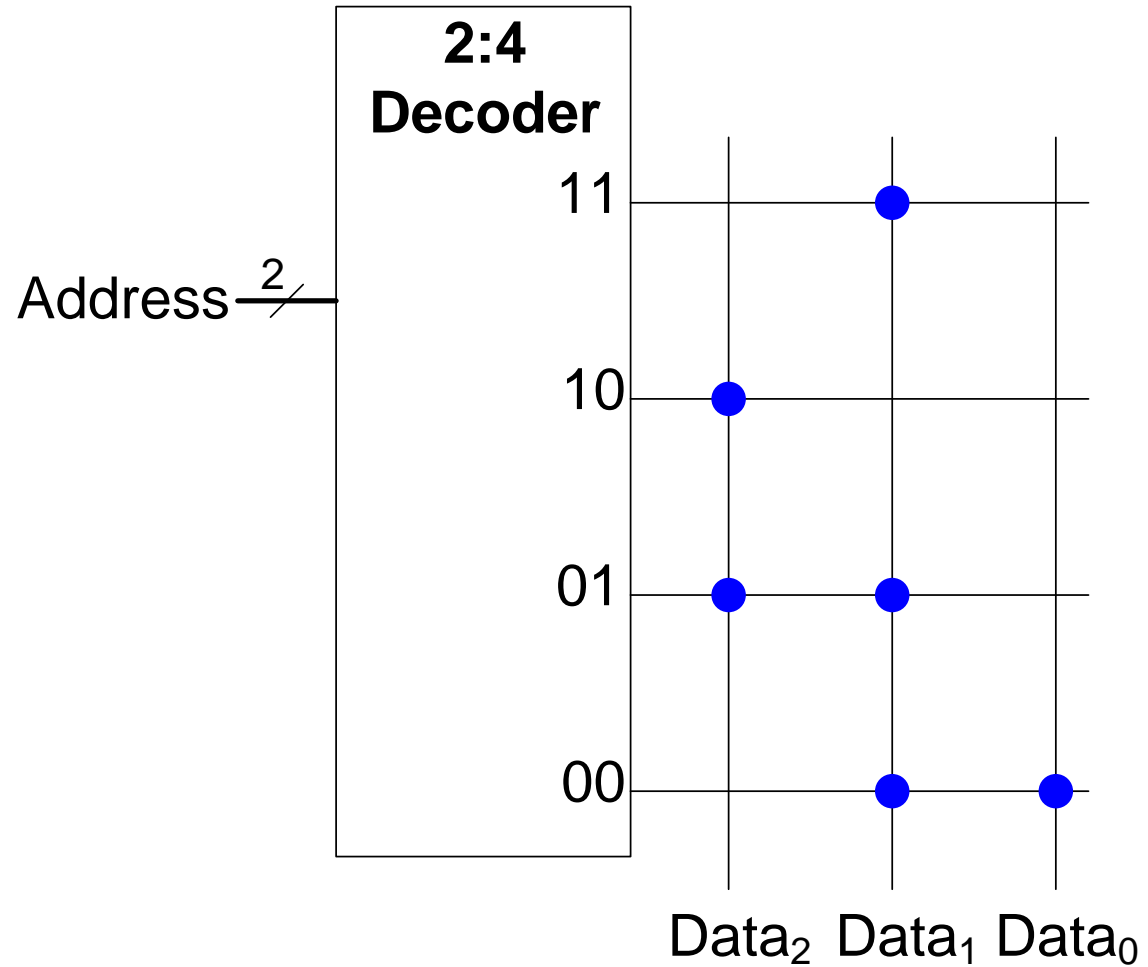
ROMs (Read Only Memories)

DDCA Ch5 - Part 15: ROMs (Read Only Memories) <https://www.youtube.com/watch?v=KBLery-6LKU>

ROM: Dot Notation



ROM Storage

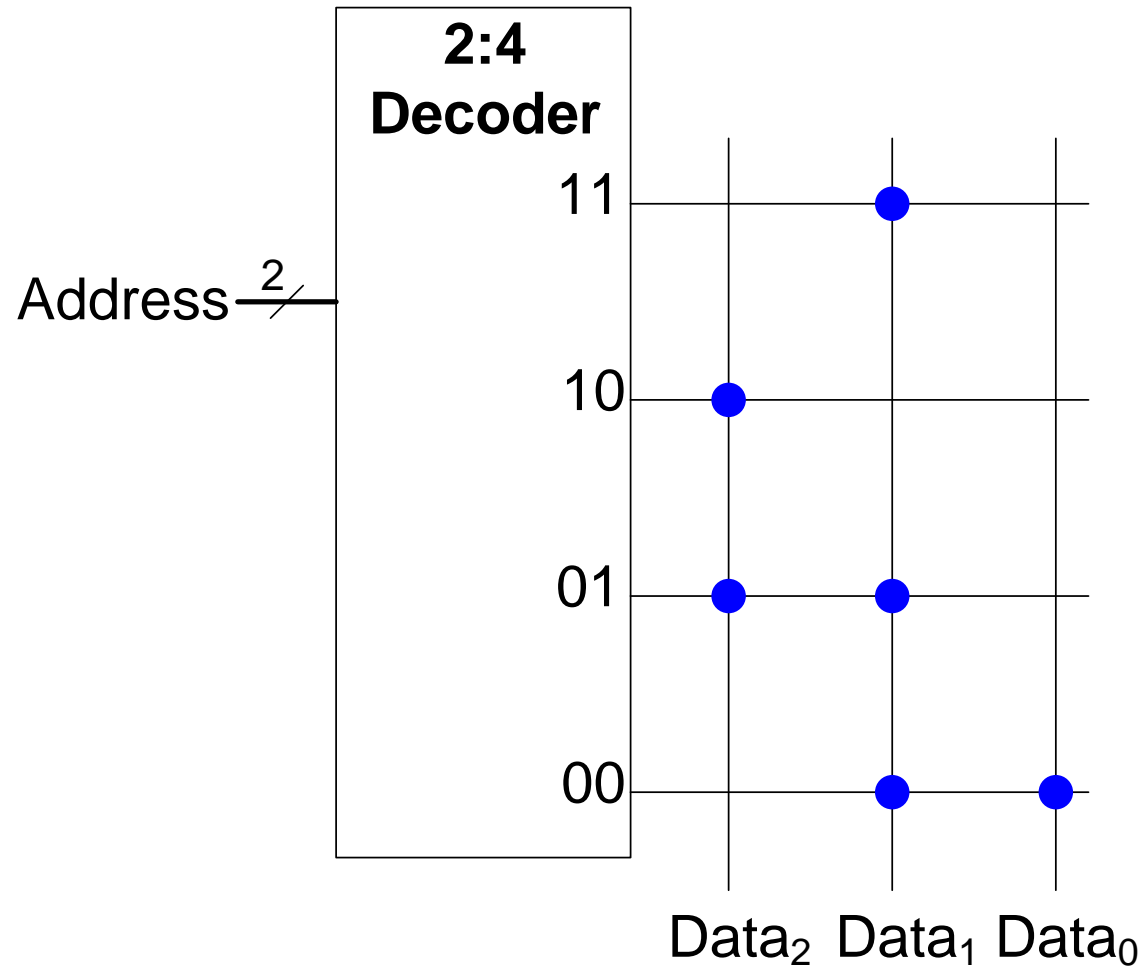


Address	Data			depth ↑ ↓
11	0	1	0	
10	1	0	0	
01	1	1	0	
00	0	1	1	
			width ←→	

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

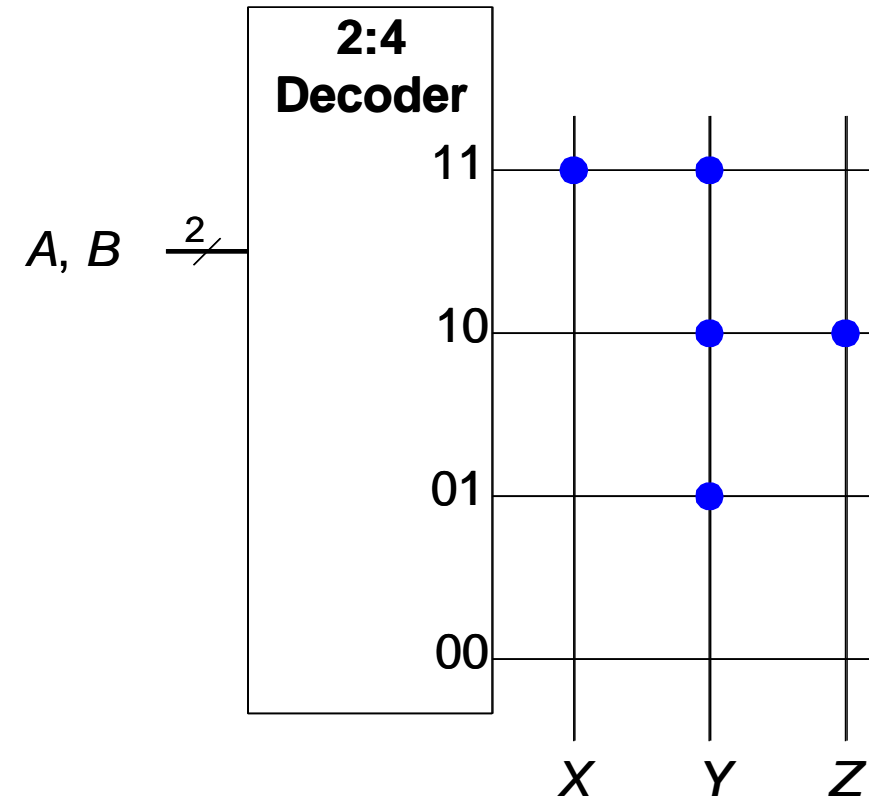




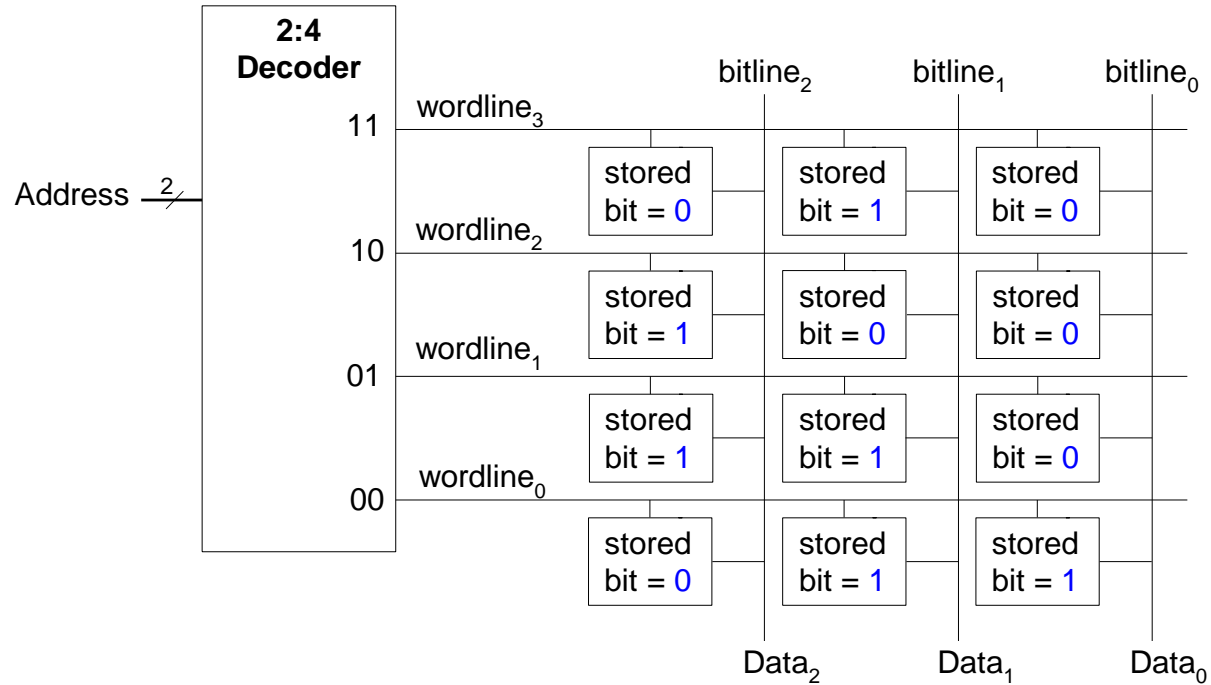
- $Data_2 = A_1 \oplus A_0$
- $Data_1 = \overline{A_1} + A_0$
- $Data_0 = \overline{A_1 + A_0}$

Example: Logic with ROMs

- Implement the following logic functions using a $2^2 \times 3$ -bit ROM:
- $X = AB$
- $Y = A + B$
- $Z = A \bar{B}$

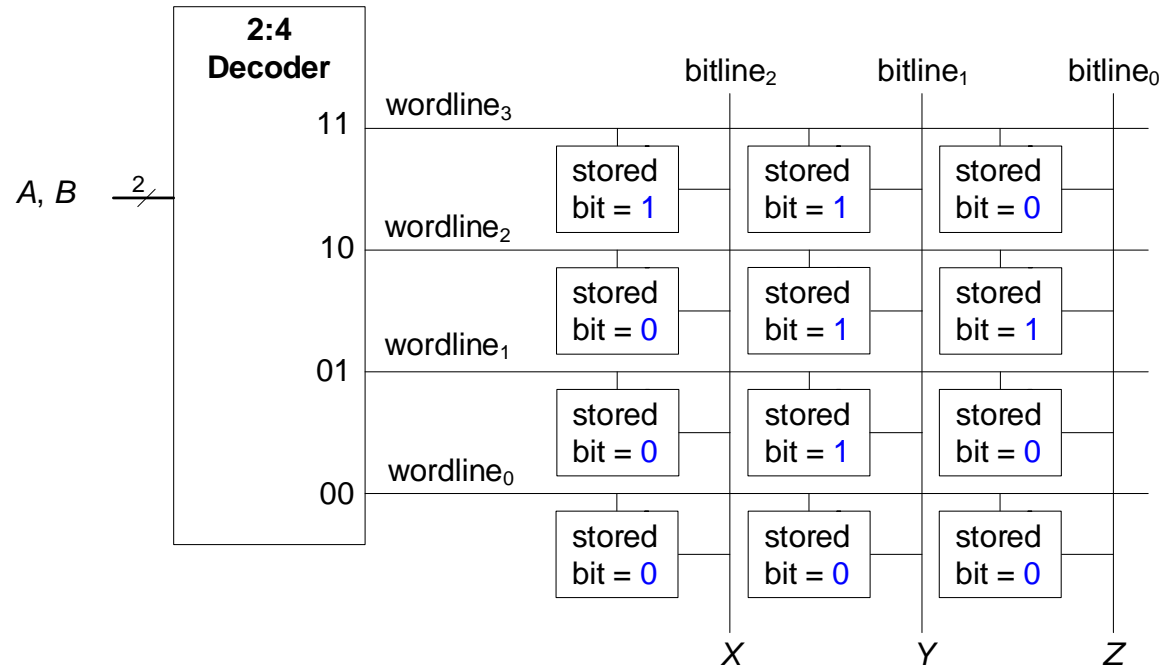


Logic with Any Memory Array



- $Data_2 = A_1 \oplus A_0$
- $Data_1 = \overline{A_1} + A_0$
- $Data_0 = \overline{A_1 + A_0}$

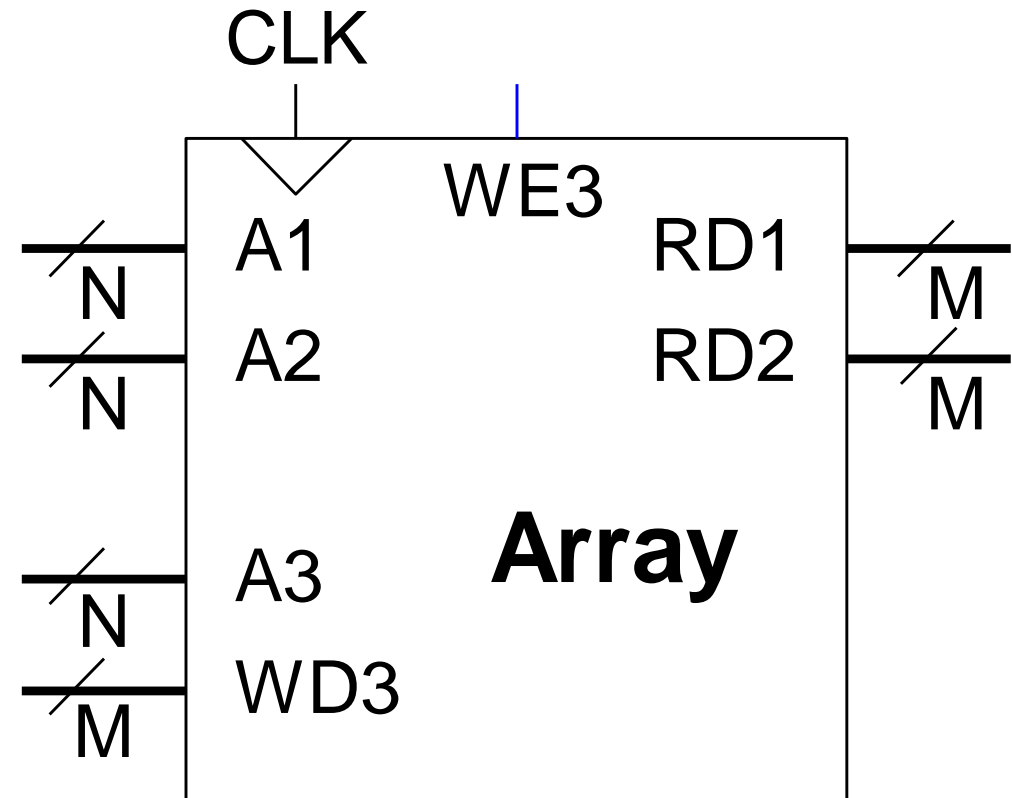
Logic with Memory Arrays



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

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



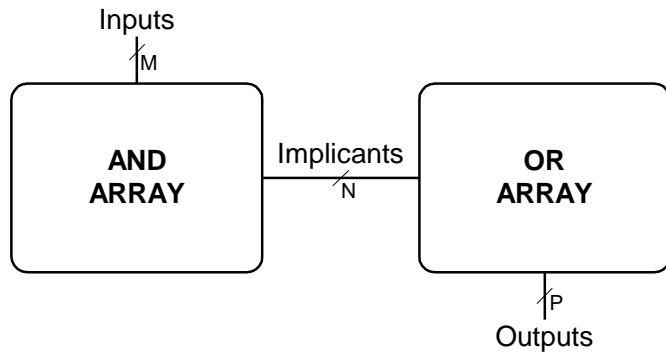
Logic Arrays: PLAs & FPGAs

DDCA Ch5 - Part 17: Logic Arrays https://www.youtube.com/watch?v=fP-oQ7vz_4c

Logic Arrays: PLAs & FPGAs

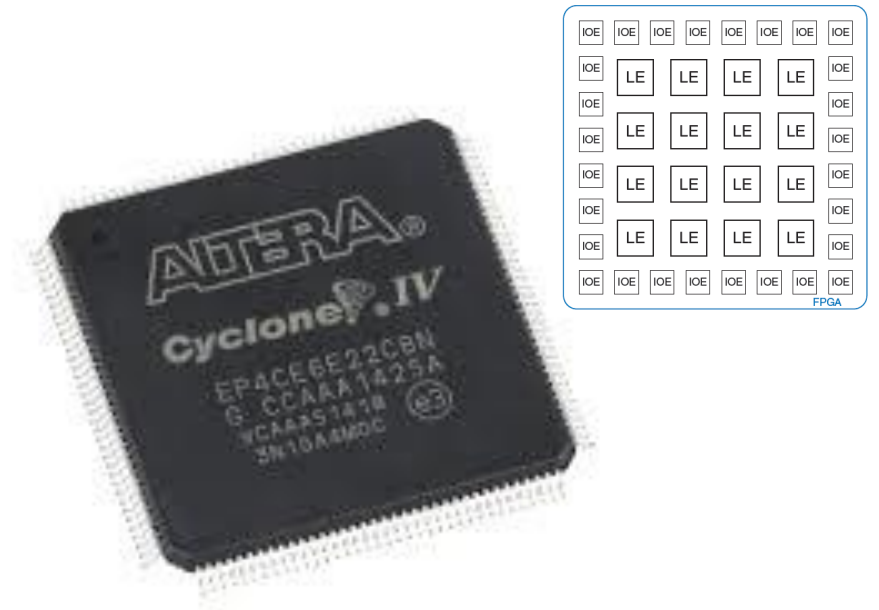
- **PLAs** (Programmable logic arrays)

- 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

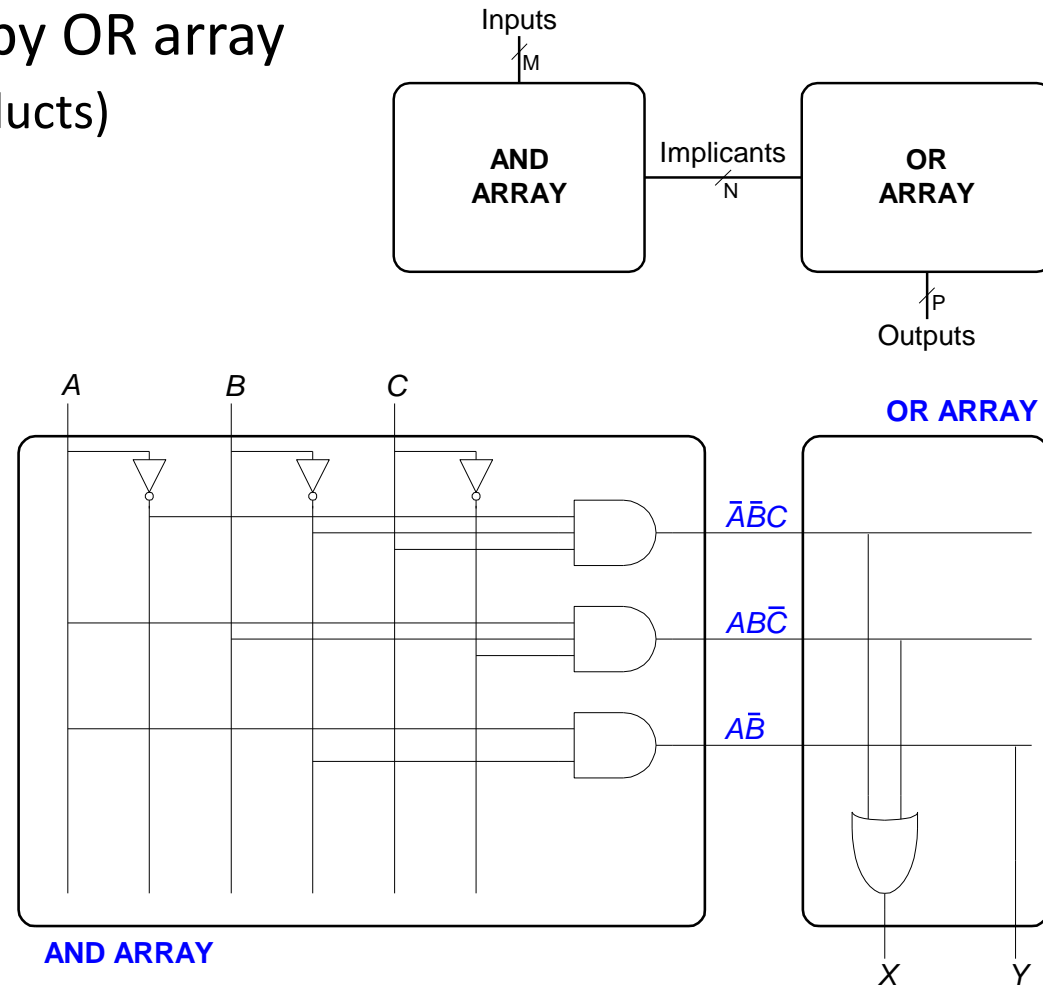


PLAs: Programmable Logic Arrays

- AND array followed by OR array
 - → SOP (sum of products)

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

- $Y = A\bar{B}$

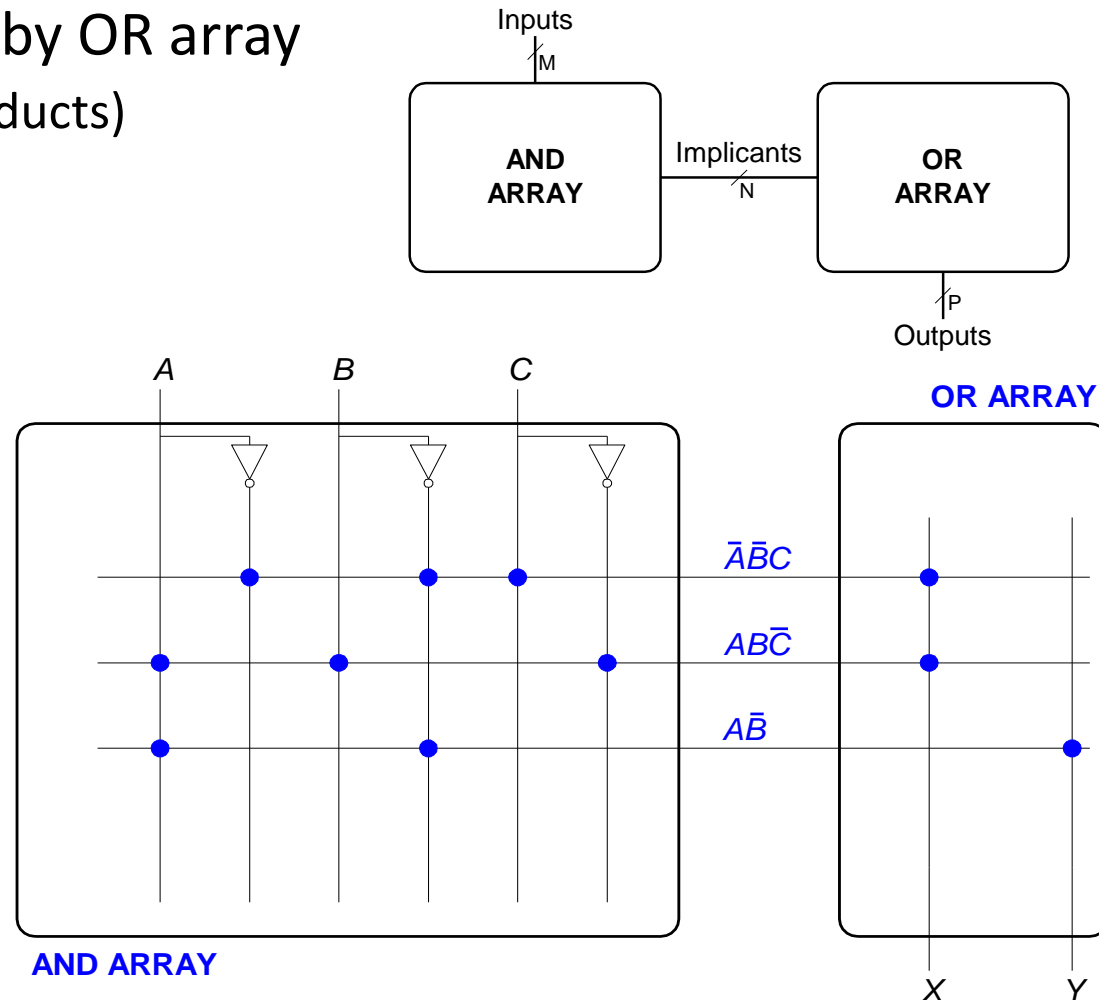


PLAs: Programmable Logic Arrays

- AND array followed by OR array
 - \rightarrow SOP (sum of products)

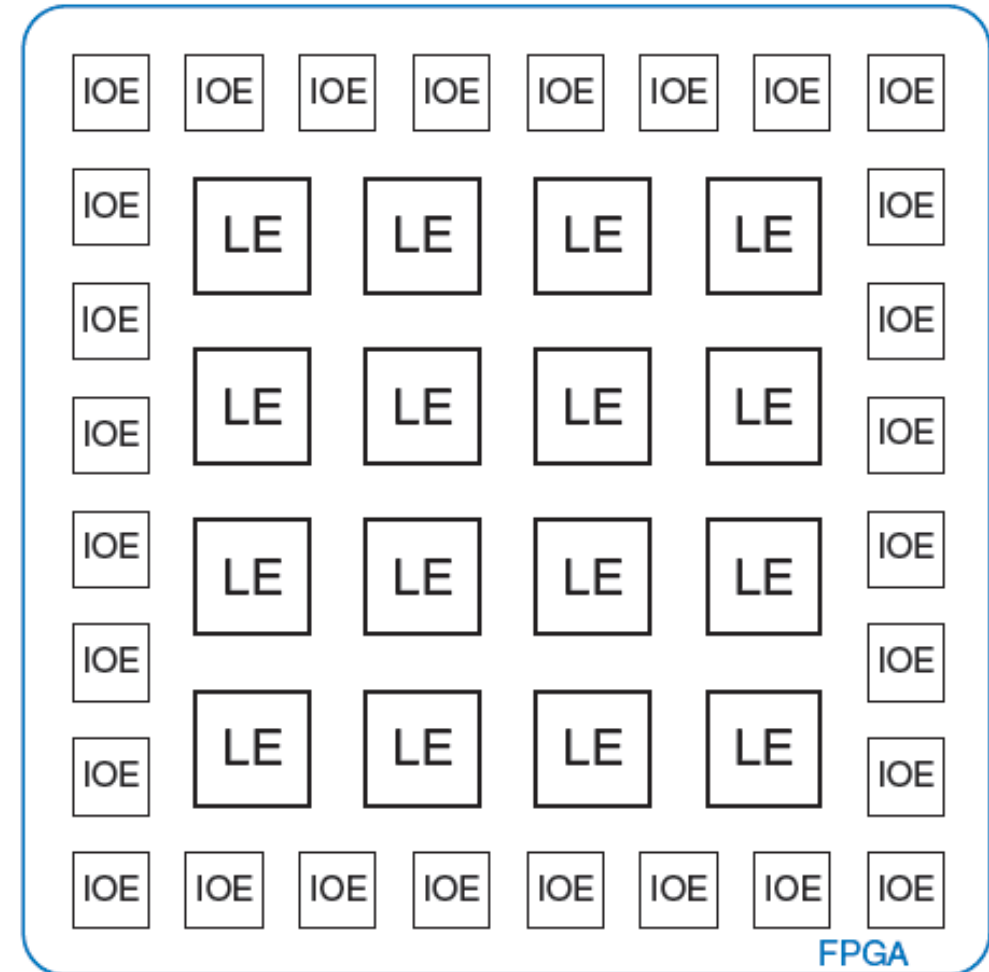
- $X = \bar{A}\bar{B}C + AB\bar{C}$

- $Y = A\bar{B}$



FPGAs: Field Programmable Gate Arrays

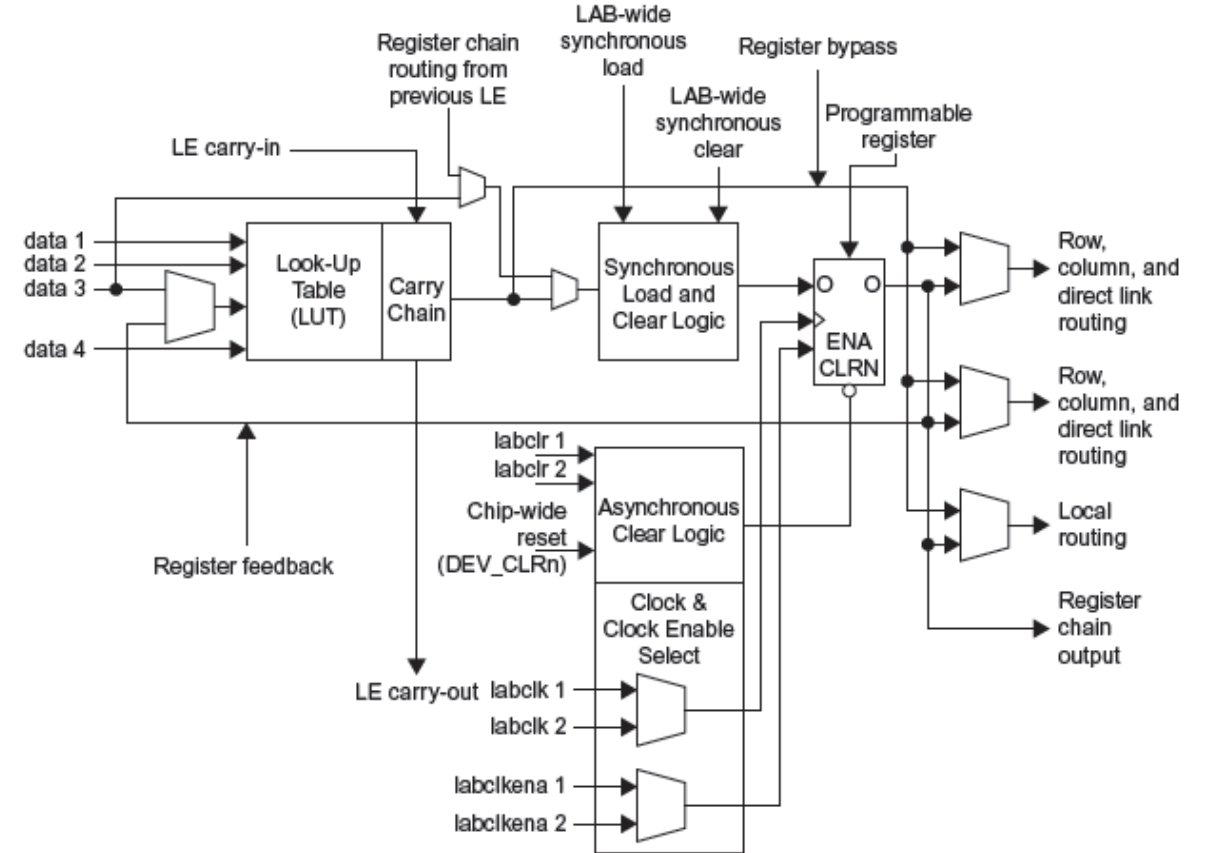
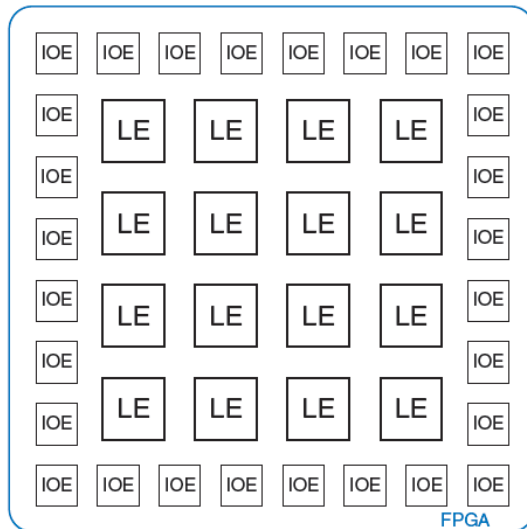
- FPGAs are composed of:
 - **LEs** (Logic elements): perform logic
 - **IOEs** (Input/output elements): interface with outside world
 - **Programmable interconnection**: connect LEs and IOEs
 - Some FPGAs include other building blocks such as multipliers and RAMs



LE: Logic Element

LEs are composed of:

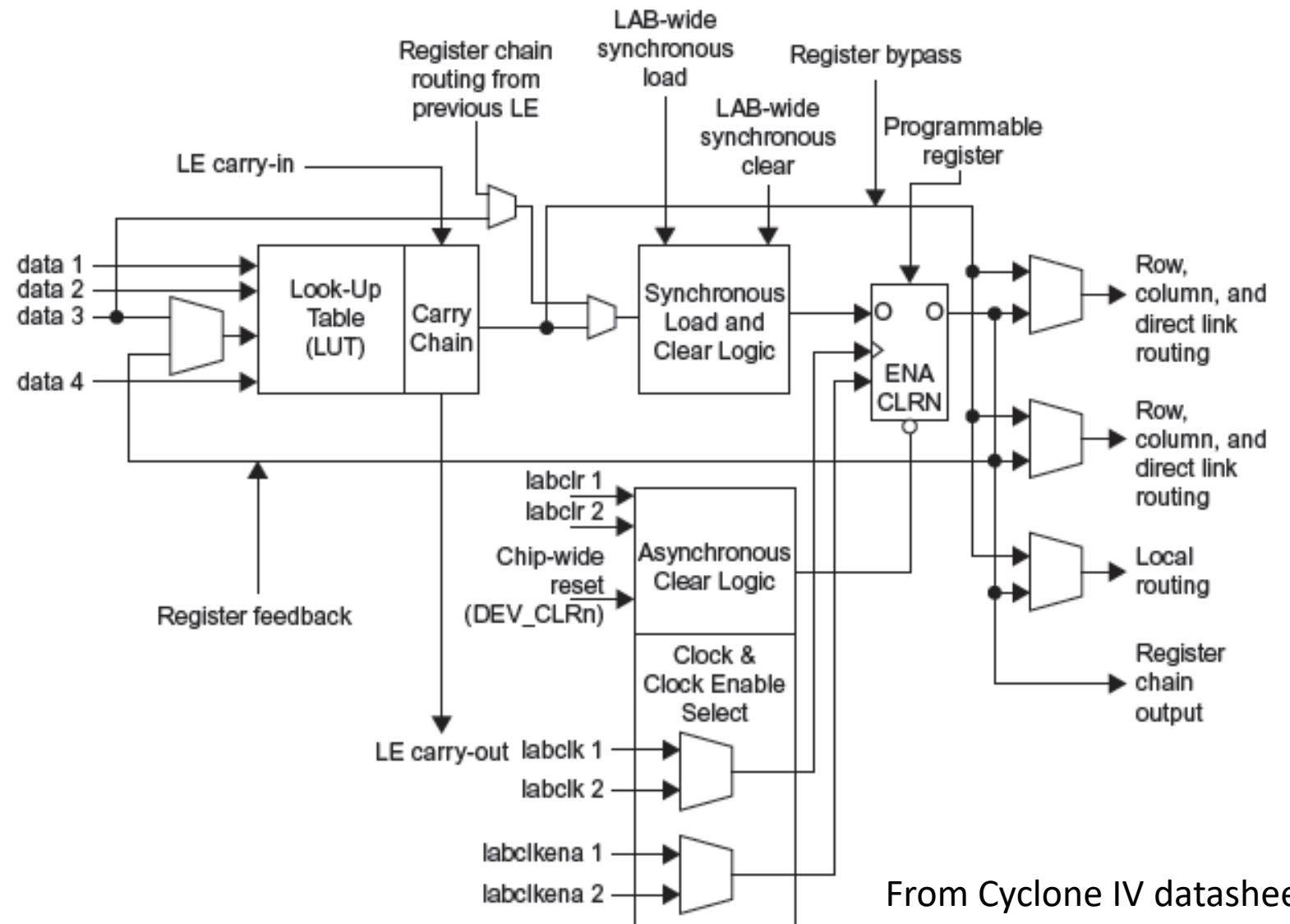
- **LUTs** (lookup tables): perform combinational logic
- **Flip-flops**: perform sequential logic
- **Multiplexers**: connect LUTs and flip-flops



Altera Cyclone IV LE

The Altera Cyclone IV LE has:

- 1 four-input **LUT**
- 1 **registered output**
- 1 **combinational output**

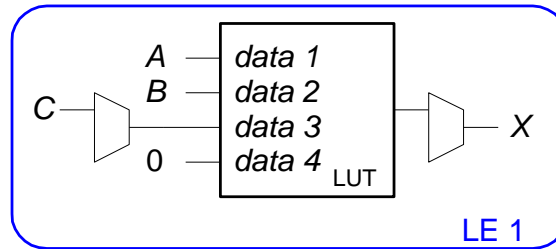


From Cyclone IV datasheet

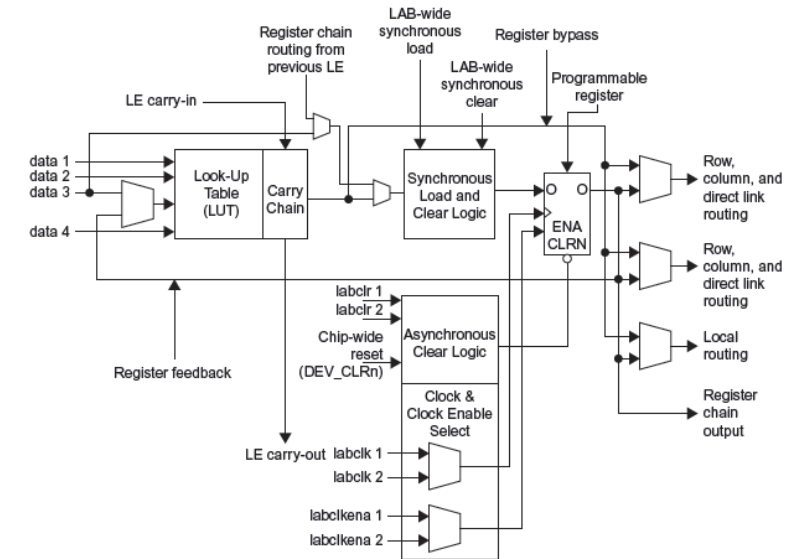
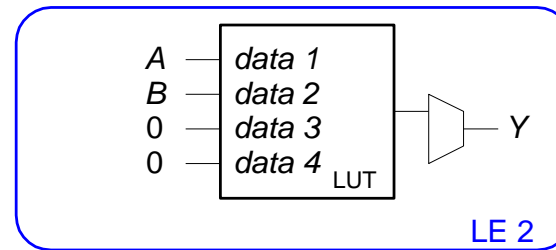
LE Configuration Example

- Show how to configure a Cyclone IV LE to perform the following functions:
- $X = \bar{A}\bar{B}C + AB\bar{C}$
- $Y = A\bar{B}$

(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



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

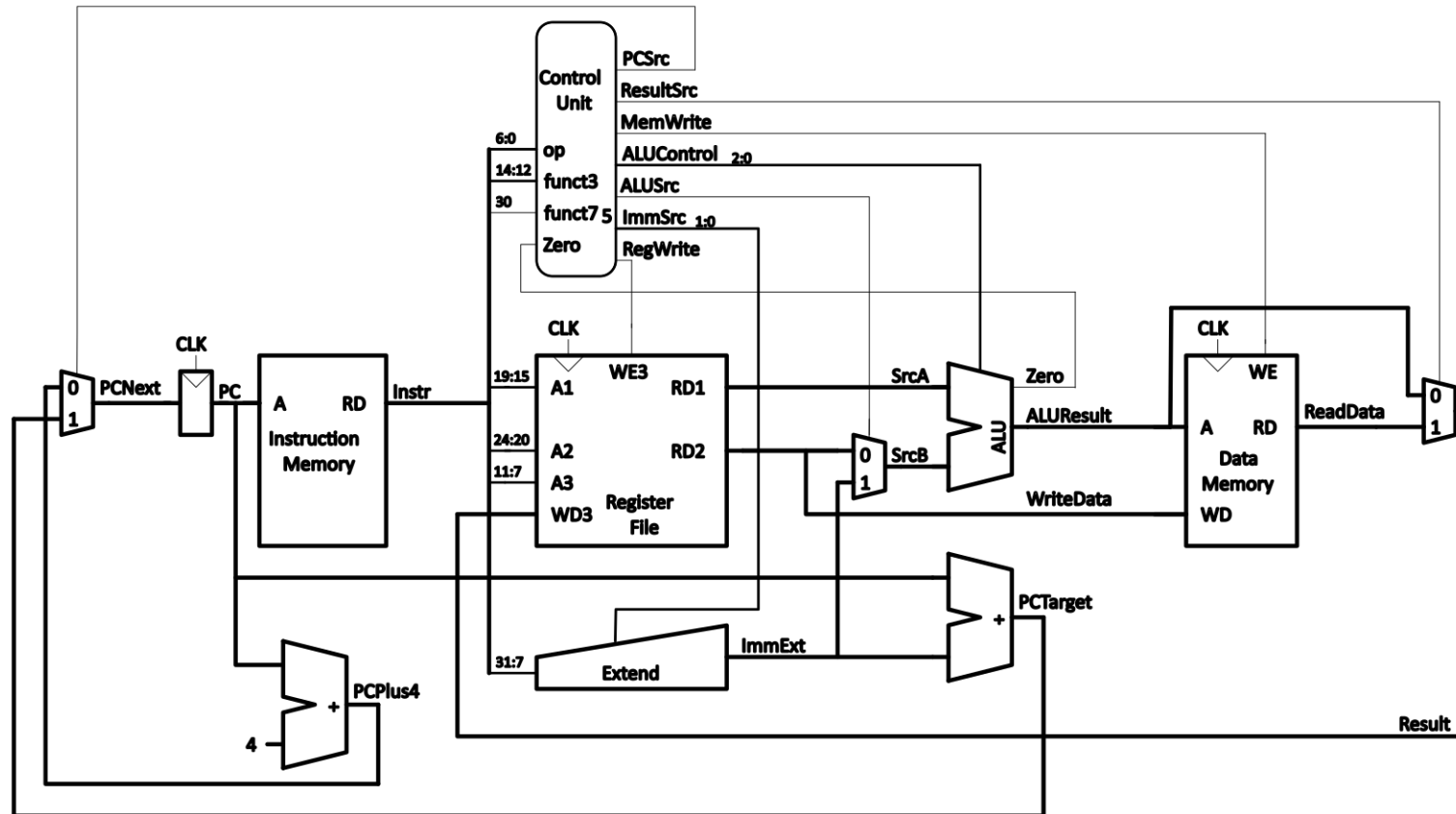
Solution:

- 5 LEs
- One LE can hold a bit of state and the next state logic, which is a function of 4 variables (2 inputs, 2 bits of state)
- One LE can compute a bit of output, which is a function of 2 variables (2 bits of state)
- Thus 2 LEs are needed for state and 3 LEs for outputs

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

Now we know all parts



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.