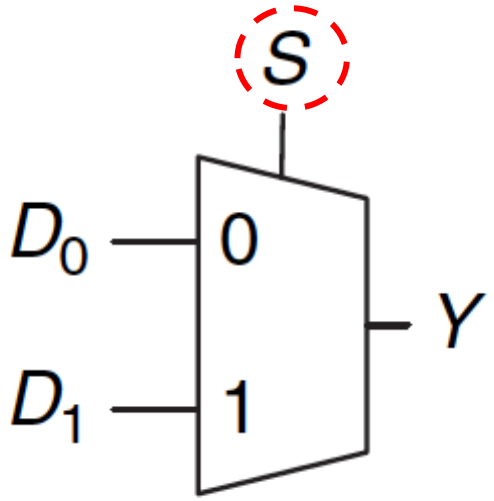
Lesson 3

Multiplexer (MUX)

Multiplexer (MUX), or Selector

- Selects one of the *N* inputs to connect it to the output
  - based on the value of a $\log_2 N$-bit control input called select
- Example: 2-to-1 MUX

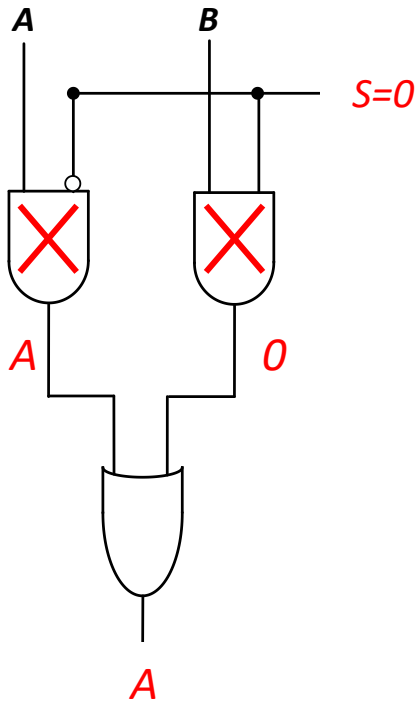
S	*Y*
0	D_0
1	D_1



S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Multiplexer (MUX), or Selector (II)

- Selects one of the *N* inputs to connect it to the output
  - based on the value of a $\log_2 N$-bit control input called select
- Example: 2-to-1 MUX
- S=1
  - A AND 0 = 0
  - B AND 1 = B
  - B OR 0 = B

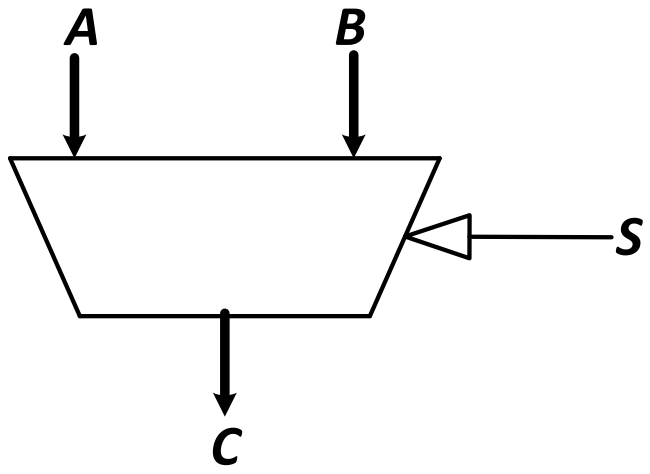


- S=0
  - A AND 1 = A
  - B AND 0 = 0
  - A OR 0 = A

Multiplexer (MUX), or Selector (III)

- The output C is always connected to either the input A or the input B
  - Output value depends on the value of the select line S

S	C
0	A
1	B



Example (10 min):

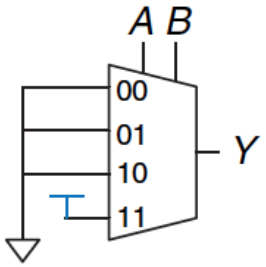
- Draw the schematic for a 4-input (4:1) MUX
  - Gate level: as a combination of basic AND, OR, NOT gates and simulate it in the logic.ly
  - Module level: As a combination of 2-input (2:1) MUXes

Aside: Logic Using Multiplexers

- Multiplexers can be used as lookup tables to perform logic functions

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

$Y = AB$



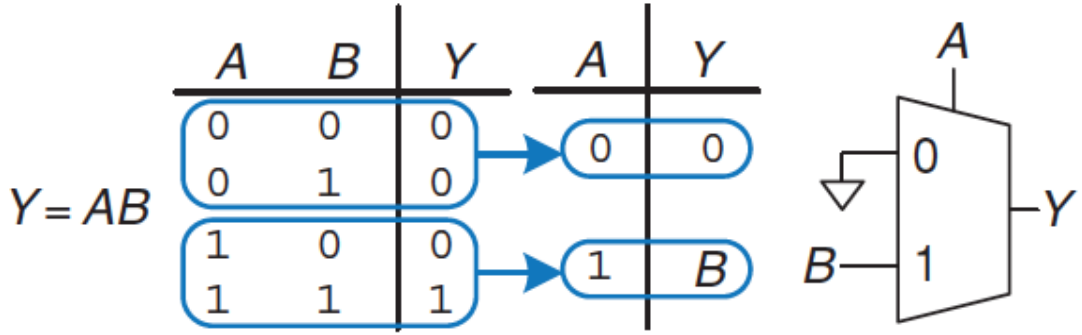


Figure 2.59 4:1 multiplexer implementation of two-input AND function

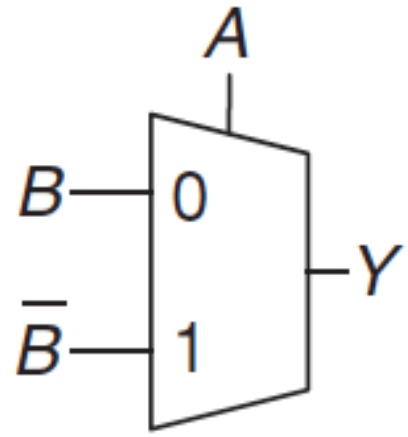
Aside: Logic Using Multiplexers (II)

- Multiplexers can be used as lookup tables to perform logic functions

$Y = A \oplus B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

A	Y
0	B
1	$\overline{B}$

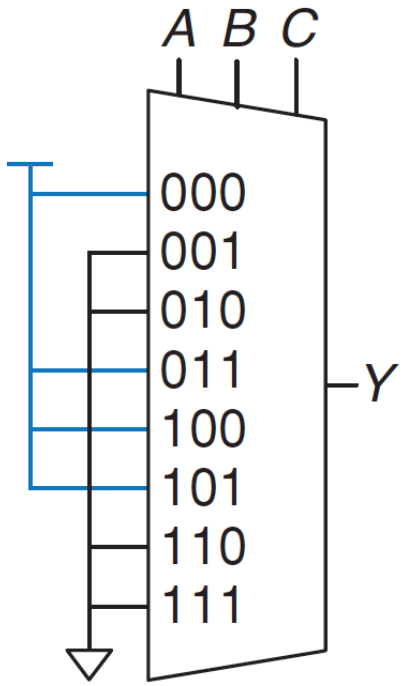


Aside: Logic Using Multiplexers (III)

- Multiplexers can be used as lookup tables to perform logic functions

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

$$Y = A\overline{B} + \overline{B}\overline{C} + \overline{A}BC$$



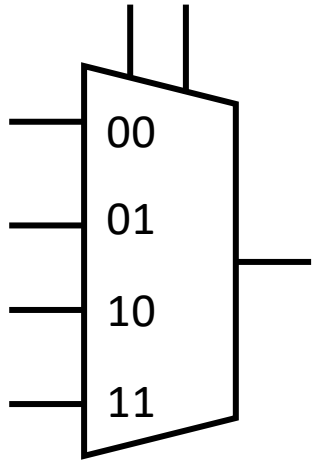
3X1 Mux

Aside: Logic Using Multiplexers (III)

- How to implement the same logic by 2X1 MUX?

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

$$Y = A\overline{B} + \overline{B}\overline{C} + \overline{A}BC$$



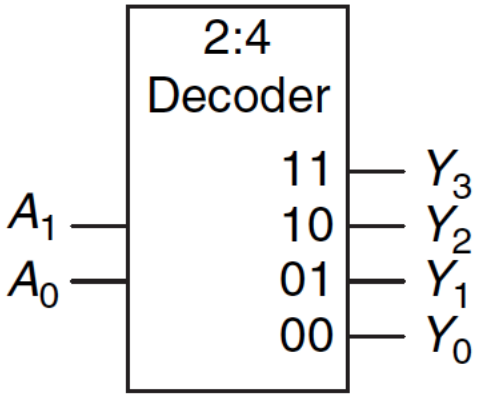
2X1 Mux

Decoder

Decoder

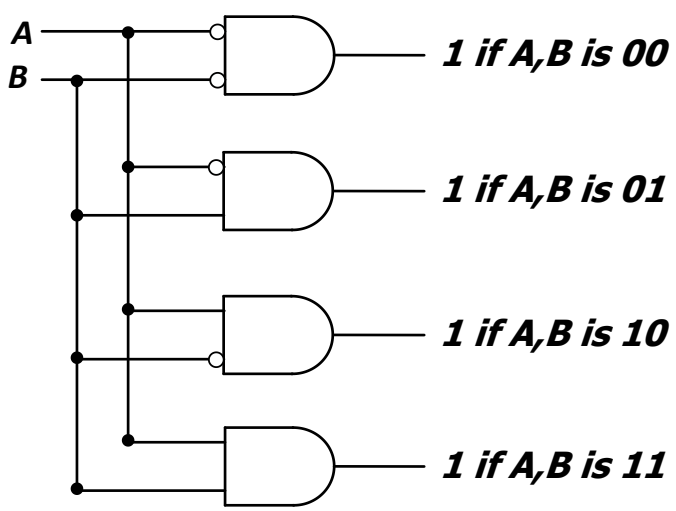
- "Input pattern detector"
- n inputs and 2^n outputs
- Exactly one of the outputs is 1 and all the rest are 0s
- The one output that is logically 1 is the output corresponding to the input pattern that the logic circuit is expected to detect
- Example: 2-to-4 decoder

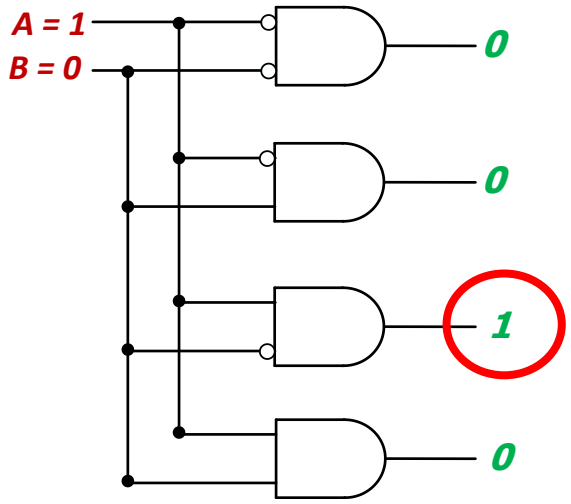
A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



Decoder (I)

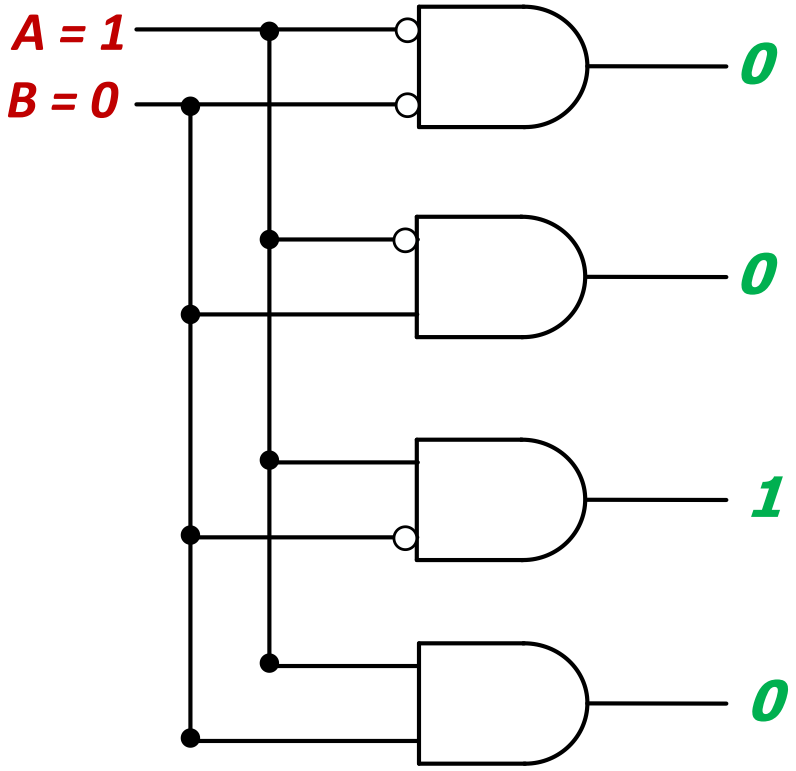
- n inputs and 2^n outputs
- Exactly one of the outputs is 1 and all the rest are 0s
- The one output that is logically 1 is the output corresponding to the input pattern that the logic circuit is expected to detect





Decoder (II)

- The decoder is useful in determining how to interpret a bit pattern
  - **It could be the address of a row in DRAM, that the processor intends to read from**
  - **It could be an instruction in the program and the processor has to decide what action to do! (based on *instruction opcode*)**



Full Adder

Full Adder (I)

- **Binary addition**
  - Similar to decimal addition
  - From right to left
  - One column at a time
  - One sum and one carry bit

- Truth table of binary addition on one column of bits within two n-bit operands

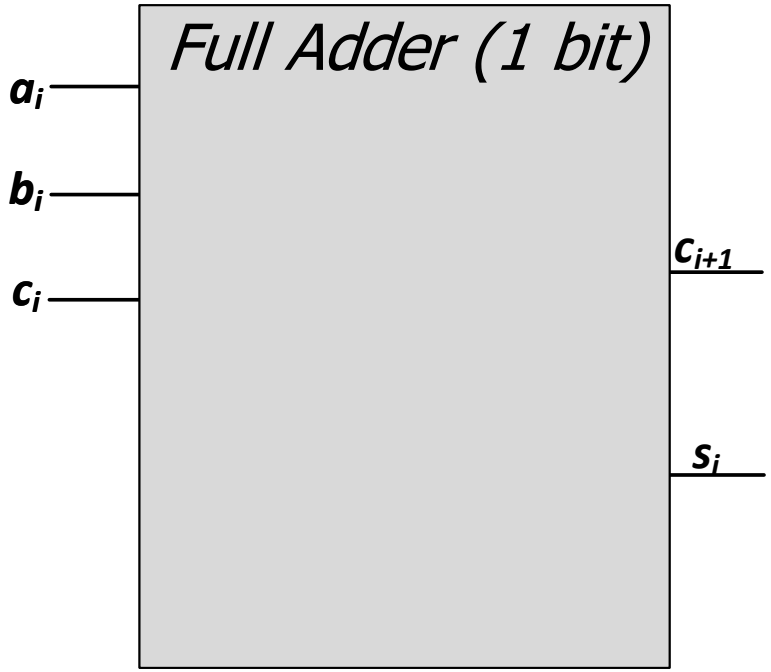
$$\begin{array}{r} a_{n-1}a_{n-2}\ldots a_1a_0 \\ b_{n-1}b_{n-2}\ldots b_1b_0 \\ C_n\ C_{n-1}\quad\ldots\quad C_1 \\ \hline S_{n-1}\quad\ldots\quad S_1S_0 \end{array}$$

a_i	b_i	$carry_i$	$carry_{i+1}$	S_i
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

Full Adder (II)

- **Binary addition**
  - N 1-bit additions
  - **SOP of 1-bit addition**

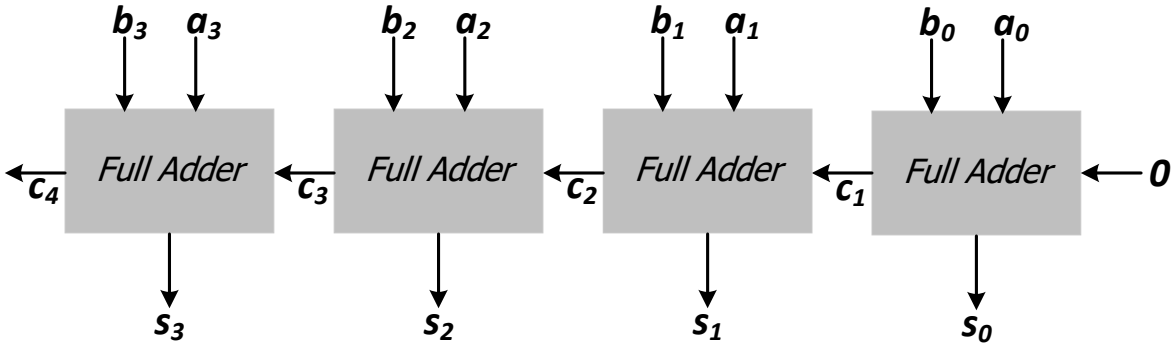
$$\begin{array}{r} a_{n-1}a_{n-2}\ldots a_1a_0 \\ b_{n-1}b_{n-2}\ldots b_1b_0 \\ C_n\ C_{n-1}\quad\ldots\quad C_1 \\ \hline S_{n-1}\quad\ldots\quad S_1S_0 \end{array} \downarrow$$



a_i	b_i	$carry_i$	$carry_{i+1}$	S_i
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

4-Bit Adder from Full Adders

- Creating a **4-bit adder** out of 1-bit full adders
  - To add two 4-bit binary numbers A and B

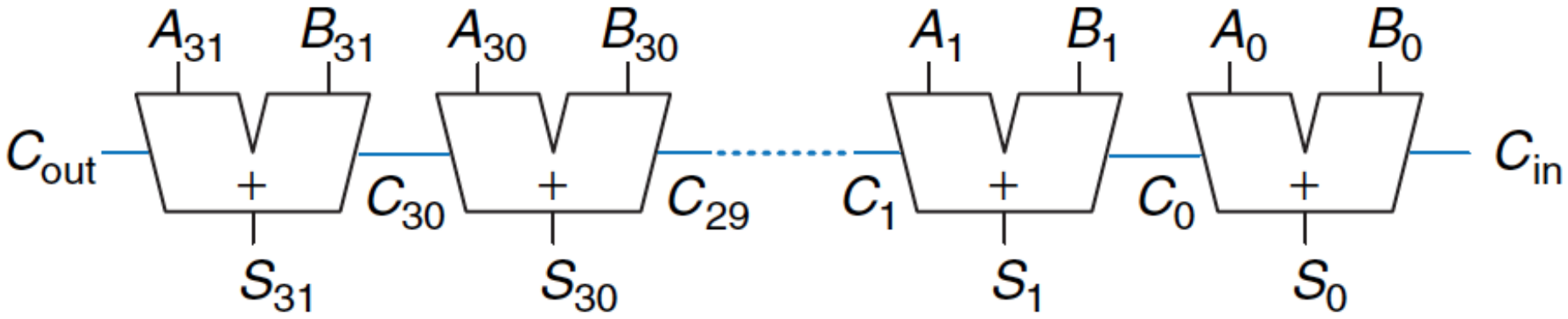


$$\begin{array}{rcccc} & a_3 & a_2 & a_1 & a_0 \\ + & b_3 & b_2 & b_1 & b_0 \\ \hline c_4 & c_3 & c_2 & c_1 & \\ \hline & s_3 & s_2 & s_1 & s_0 \end{array}$$

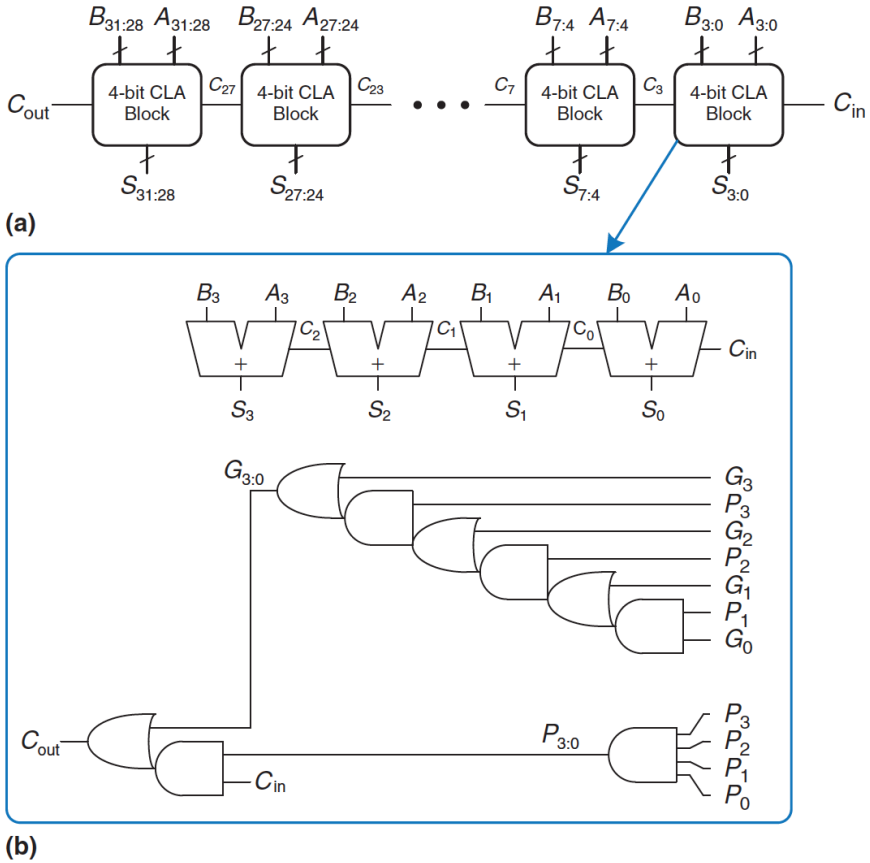
$$\begin{array}{rcccc} & 1 & 0 & 1 & 1 \\ + & 1 & 0 & 0 & 1 \\ \hline 1 & 0 & 1 & 1 & \\ \hline & 0 & 1 & 0 & 0 \end{array}$$

Adder Design: Ripple Carry Adder

- Delay propagation problem



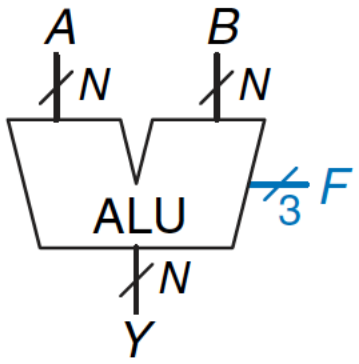
Adder Design: Carry Lookahead Adder



ALU (Arithmetic Logic Unit)

ALU (Arithmetic Logic Unit)

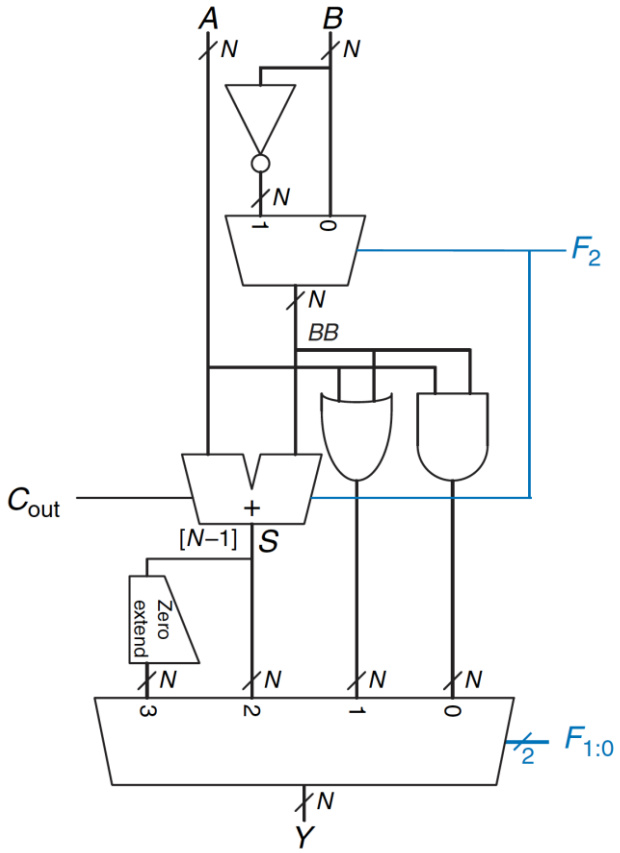
- Combines a variety of arithmetic and logical operations into a single unit (that performs only one function at a time)
- Usually denoted with this symbol:



$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT

Example ALU (Arithmetic Logic Unit)

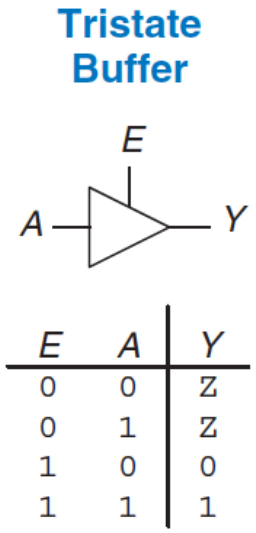
$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A – B
111	SLT



Tri-State Buffer

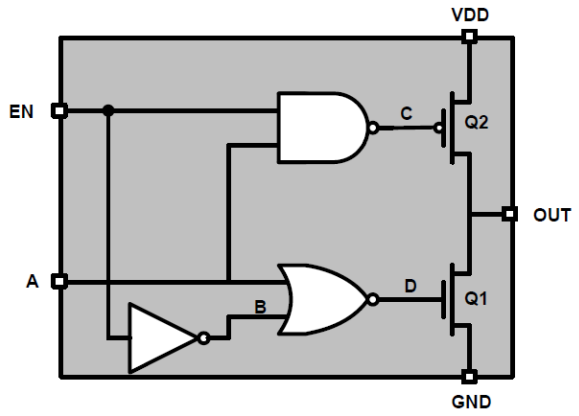
Tri-State Buffer

- A tri-state buffer enables gating of different signals onto a wire
- Floating signal (Z): Signal that is not driven by any circuit
  - Open circuit, floating wire



E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

Figure 2.40 Tristate buffer

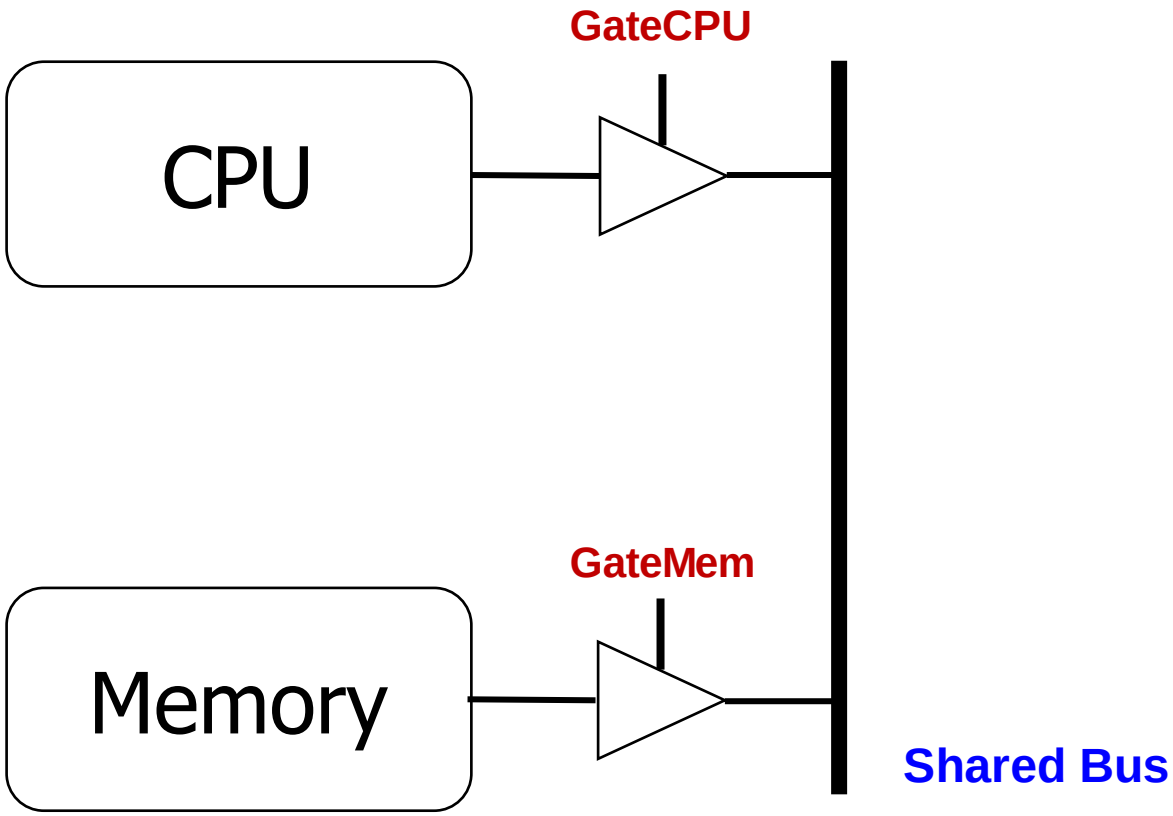


EN	A	B	C	D	Q1	Q2	OUT
L	L	H	H	L	off	off	HI-Z
L	H	H	H	L	off	off	HI-Z
H	L	L	H	H	on	off	L
H	H	L	L	L	off	on	H

Example: Use of Tri-State Buffers

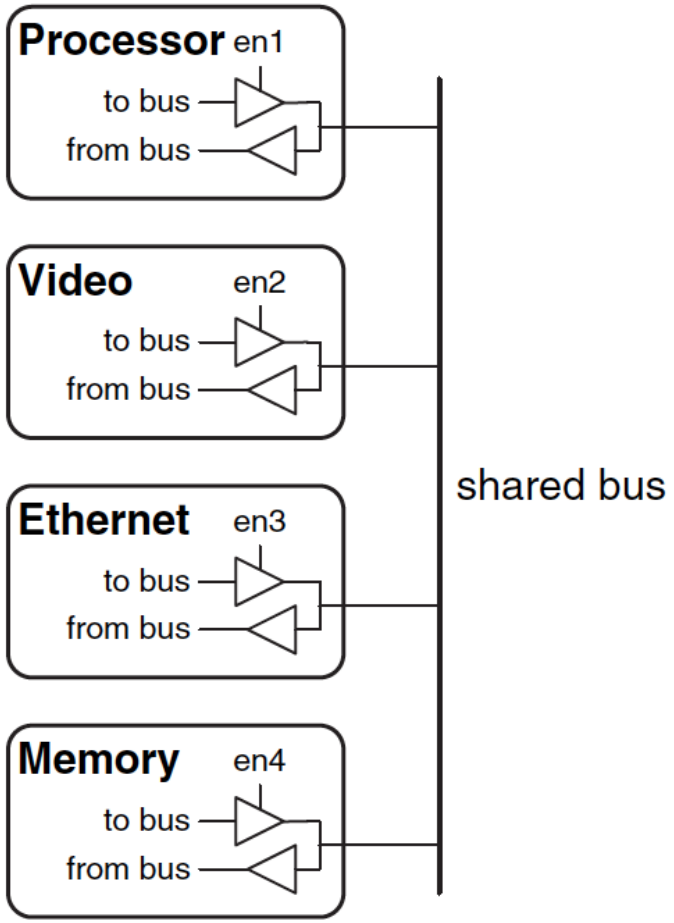
- Imagine a wire connecting the CPU and memory
  - At any time only the CPU or the memory can place a value on the wire, both not both
  - You can have two tri-state buffers: one driven by CPU, the other memory; and ensure at most one is enabled at any time

Example Design with Tri-State Buffers

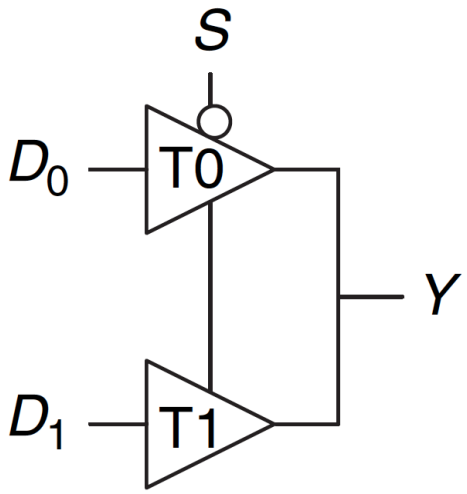


Another Example

- Shared Bus is a common line between peripherals
- All of the devices connected with Tri-State Buffers
- When a device use the shared bus all other buffers are disconnected.

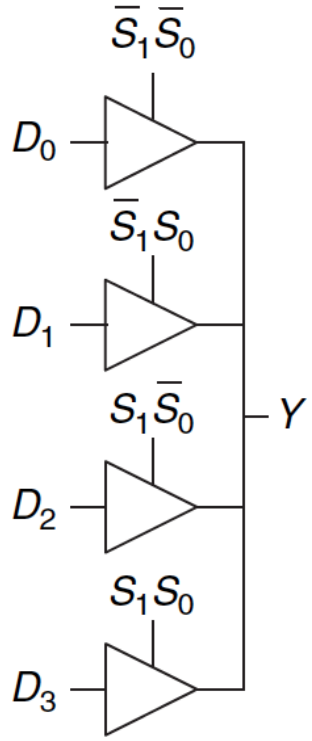


Multiplexer Using Tri-State Buffers



$$Y = D_0\overline{S} + D_1 S$$

Figure 2.56 **Multiplexer using tristate buffers**



Karnaugh Maps (K-Maps)

Complex Cases

- One example $\boldsymbol{Cout = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC}$

- Problem
  - Easy to see how to apply Uniting Theorem…
  - Hard to know if you applied it in all the right places…
  - …especially in a function of many more variables

- Question
  - Is there an easier way to find potential simplifications?
  - i.e., potential applications of Uniting Theorem…?

- Answer
  - Need an intrinsically *geometric* representation for Boolean f( )
  - Something we can draw, see…

Karnaugh Map

- Karnaugh Map (K-map) method
  - K-map is an alternative method of representing the truth table that helps visualize adjacencies in up to 6 dimensions
  - Physical adjacency ↔ Logical adjacency

2-variable K-map

A \ B	0	1
0	00	01
1	10	11

3-variable K-map

A \ BC	00	01	11	10
0	000	001	011	010
1	100	101	111	110

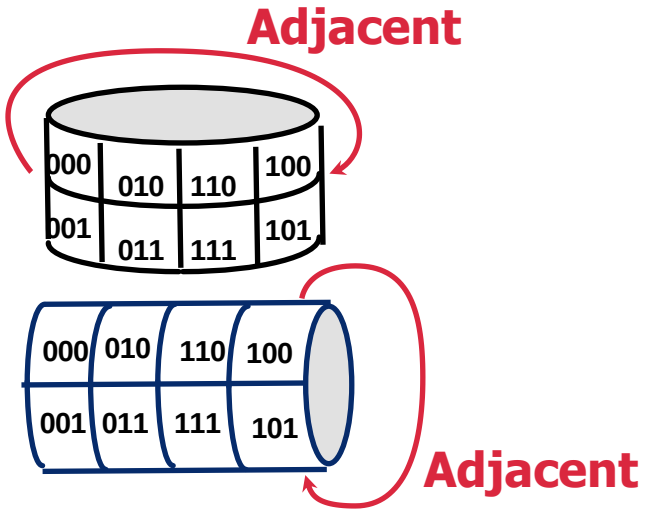
4-variable K-map

AB \ CD	00	01	11	10
00	0000	0001	0011	0010
01	0100	0101	0111	0110
11	1100	1101	1111	1110
10	1000	1001	1011	1010

Numbering Scheme: **00, 01, 11, 10 is called a "Gray Code" — only a *single bit (variable) changes* from one code word and the next code word**

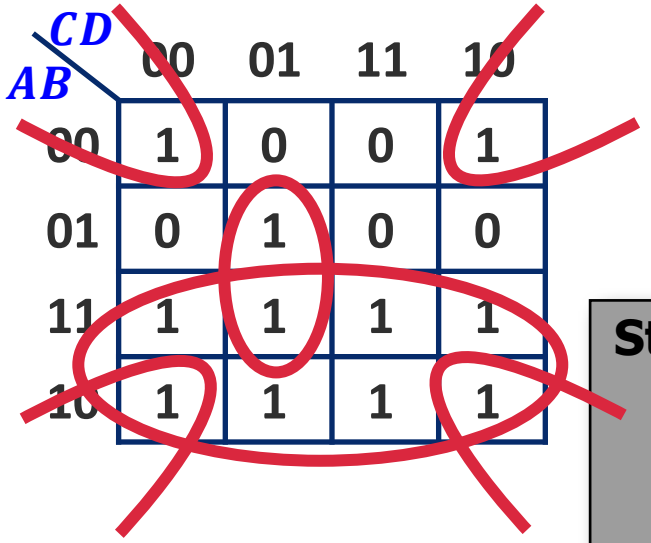
Karnaugh Map Methods

A \ BC	00	01	11	10
0	000	001	011	010
1	100	101	111	110



K-map adjacencies go "around the edges"
Wrap around from first to last column
Wrap around from top row to bottom row

K-map Cover - 4 Input Variables



$$F = A + \bar{B}\bar{D} + B\bar{C}D$$

Strategy for "circling" rectangles on Kmap:

Biggest "oops!" that people forget:

Logic Minimization Using K-Maps

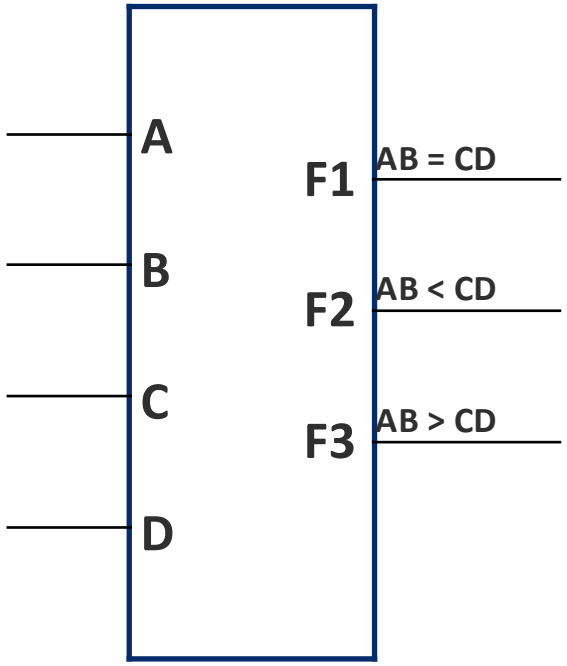
- Very simple guideline:
  - Circle all the rectangular blocks of 1's in the map, using the fewest possible number of circles
    - Each circle should be as large as possible
  - Read off the implicants that were circled

- More formally:
  - A Boolean equation is minimized when it is written as a sum of the fewest number of prime implicants
  - Each circle on the K-map represents an implicant
  - The largest possible circles are prime implicants

K-map Rules

- **What can be legally combined (circled) in the K-map?**
  - Rectangular groups of size 2^k for any integer k
  - Each cell has the same value (1, for now)
  - All values must be adjacent
    - Wrap-around edge is okay
- **How does a group become a term in an expression?**
  - Determine which literals are constant, and which vary across group
  - Eliminate varying literals, then AND the constant literals
    - constant 1 → use X, constant 0 → use $\overline{X}$
- **What is a good solution?**
  - Biggest groupings → eliminate more variables (literals) in each term
  - Fewest groupings → fewer terms (gates) all together
  - OR together all AND terms you create from individual groups

K-map Example: Two-bit Comparator



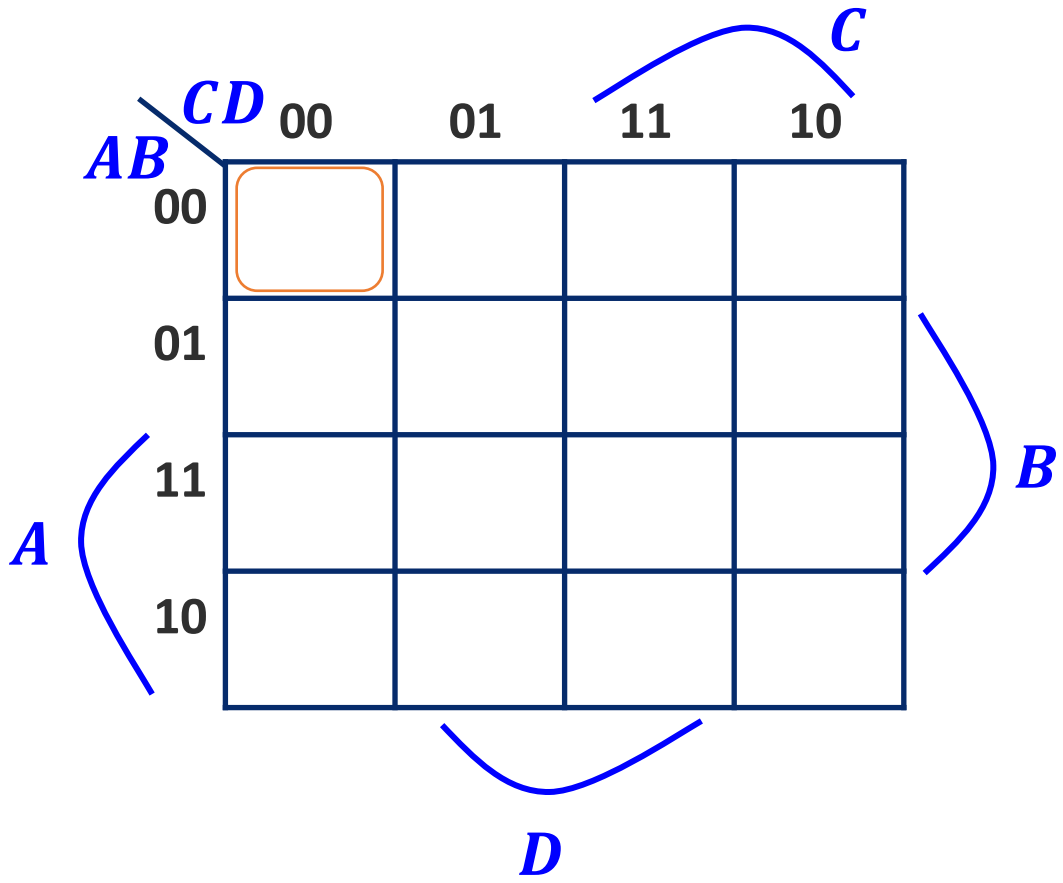
A	B	C	D	F1	F2	F3
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	1	0	0

Design Approach:

Write a 4-Variable K-map for each of the 3 output functions

K-map Example: Two-bit Comparator (2)

K-map for F1

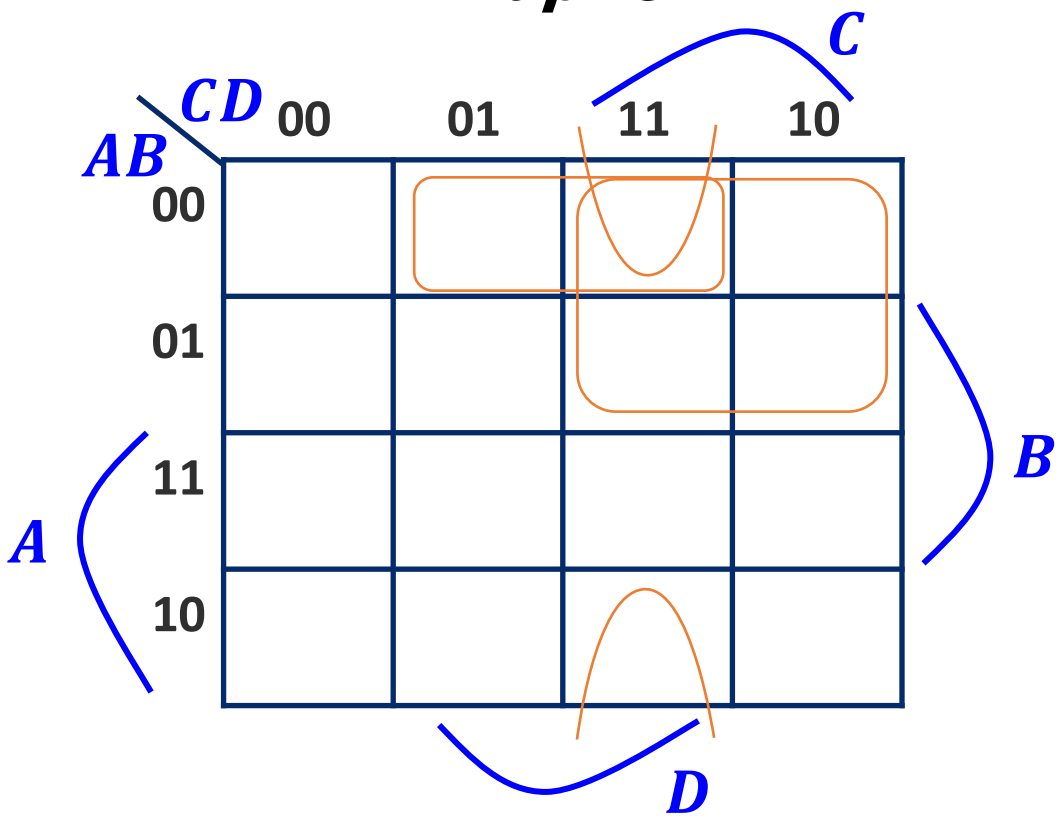


F1 =

A	B	C	D	F1	F2	F3
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	1	0	0

K-map Example: Two-bit Comparator (3)

K-map for F2



A	B	C	D	F1	F2	F3
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	1	0	0

F2 =

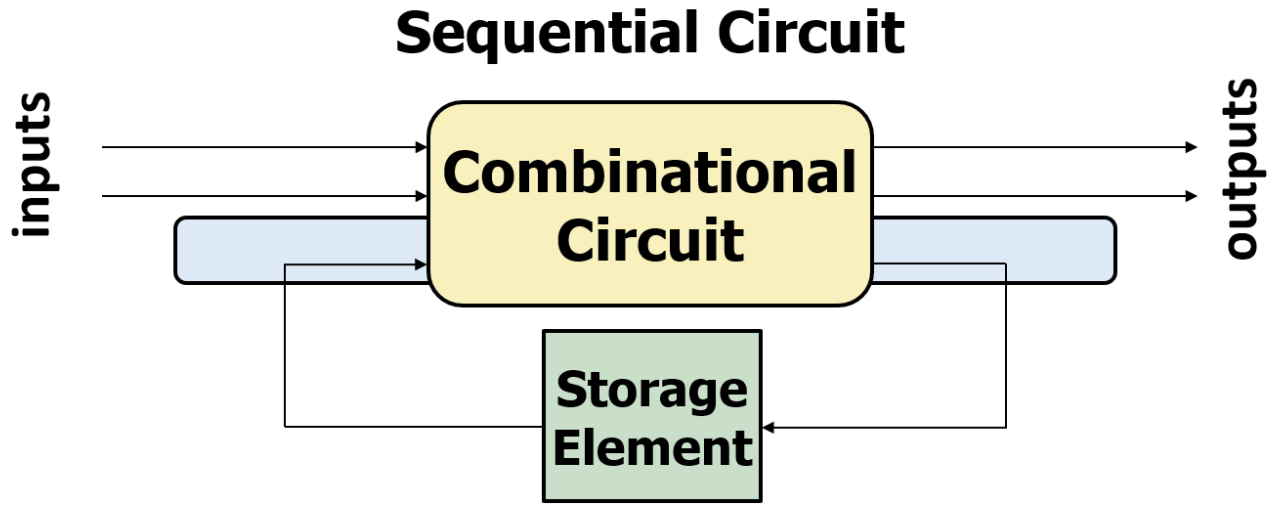
F3 = ? (Exercise for you)

Sequential Logic Circuits and Design

- Circuits that can store information
  - Cross-coupled inverter
  - R-S Latch
  - Gated D Latch
  - D Flip-Flop
  - Register

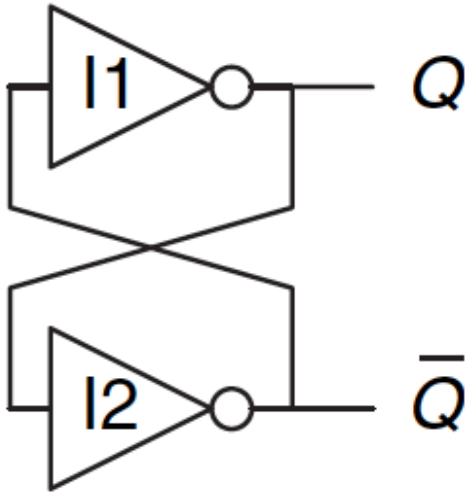
Introduction

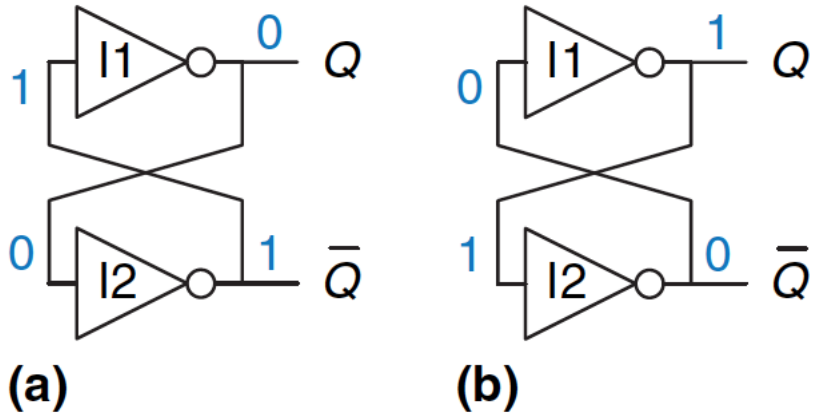
- Combinational circuit output depends **only** on current input
- We want circuits that produce output depending on **current** and **past** input values – circuits with **memory**
- How can we design a circuit that **stores information**?



Capturing Data

Basic Element: Cross-Coupled Inverters



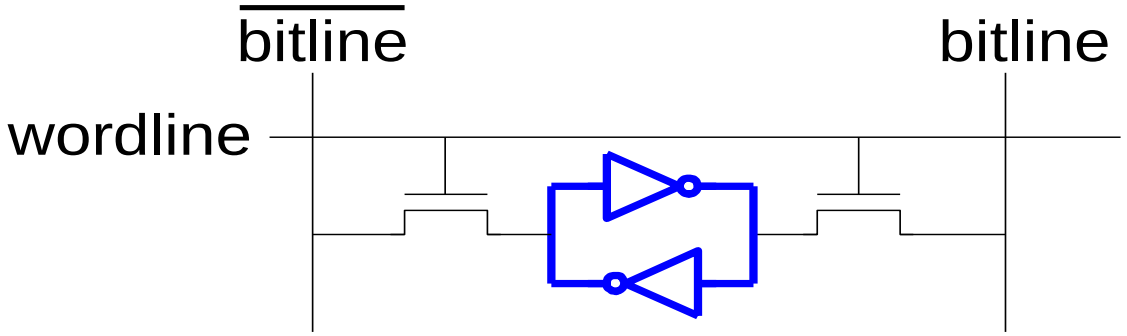


- Has two stable states: Q=1 or Q=0.
- Has a third possible "metastable" state with both outputs oscillating between 0 and 1 (we will see this later)
- Not useful without a *control mechanism* for setting Q

Image source: Harris and Harris, Digital Design and Computer Architecture, 2nd Ed., p.110.

More Realistic Storage Elements

- **Have a control mechanism for setting Q**
  - We will see the R-S latch soon
  - Let's look at an SRAM (static random access memory) cell first



SRAM cell

- We will get back to SRAM (and DRAM) later

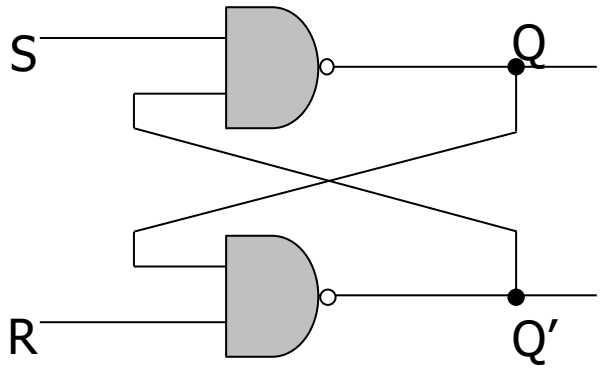
The Big Picture: Storage Elements

- Latches and Flip-Flops
  - Very fast, parallel access
  - Very expensive (one bit costs tens of transistors)
- Static RAM (SRAM)
  - Relatively fast, only one data word at a time
  - Expensive (one bit costs 6+ transistors)
- Dynamic RAM (DRAM)
  - Slower, one data word at a time, reading destroys content (refresh), needs special process for manufacturing
  - Cheap (one bit costs only one transistor plus one capacitor)
- Other storage technology (flash memory, hard disk, tape)
  - Much slower, access takes a long time, non-volatile
  - Very cheap

Basic Storage Element: The R-S Latch

The R-S (Reset-Set) Latch

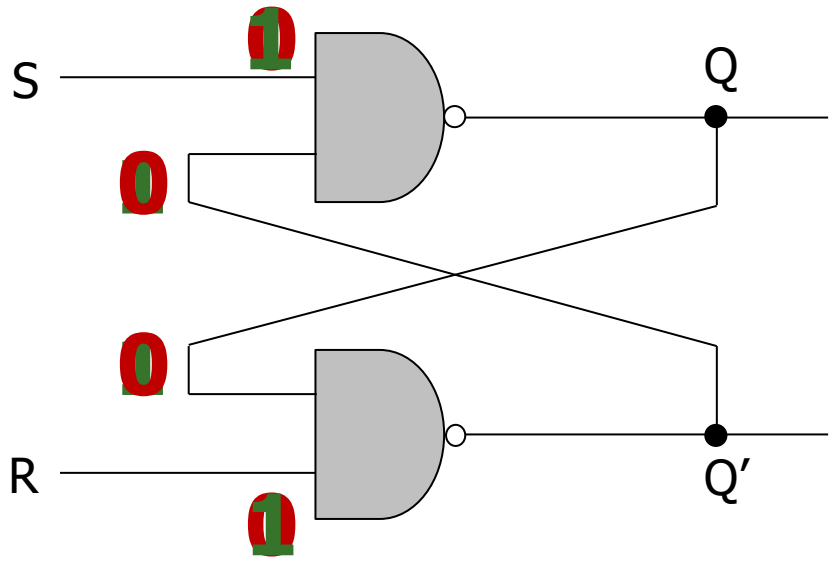
- Cross-coupled **NAND gates**
  - Data is stored at **Q** (inverse at **Q'**)
  - **S** and **R** are control inputs
    - In *quiescent* (*idle*) *state,* **both S and R are held at 1**
    - **S (set):** drive **S** to 0 (keeping **R** at 1) to change **Q** to 1
    - **R (reset):** drive **R** to 0 (keeping **S** at 1) to change **Q** to 0
- **S** and **R** should never **both** be 0 at the same time



Input		Output
R	S	Q
1	1	Q_{prev}
1	0	1
0	1	0
0	0	Forbidden

Why not R=S=0?

Input		Output
R	S	Q
1	1	Q_{prev}
1	0	1
0	1	0
0	0	Forbidden

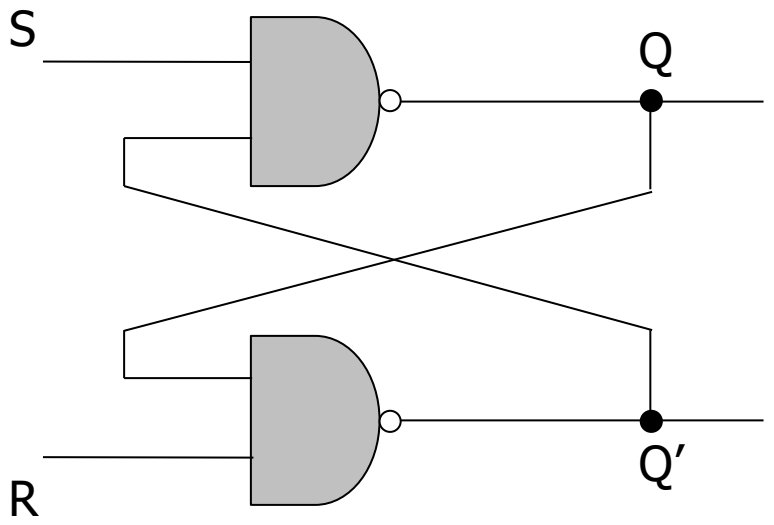


1. If **R=S=0, Q** and **Q'** will both settle to 1, which **breaks** our invariant that **Q** = !**Q'**
2. If **S** and **R** transition back to 1 at the same time, **Q** and **Q'** begin to oscillate between 1 and 0 because their final values depend on each other (**metastability**)
   - This eventually settles depending on **variation in the circuits**

The Gated D Latch

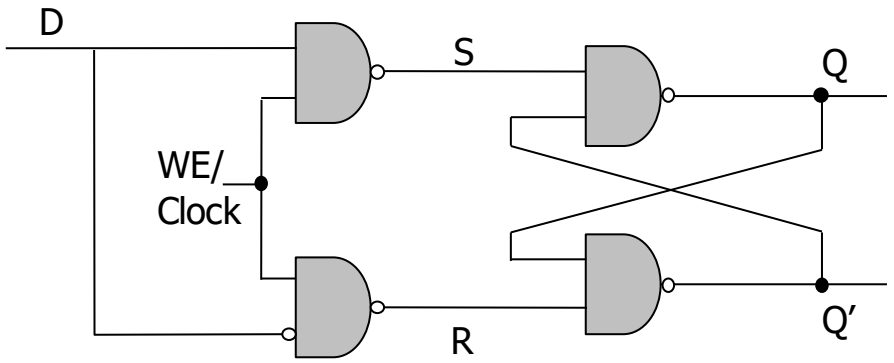
The Gated D Latch

- How do we **guarantee** correct operation of an R-S Latch?



The Gated D Latch

- How do we **guarantee** correct operation of an R-S Latch?
  - Reduce the number of states to three:
    - WE/Clock =1, D=1, Q=1
    - WE/Clock =1, D=0, Q=0
    - WE/Clock =0, D=X



Input		Output
WE	D	Q
0	0	Q_{prev}
0	1	Q_{prev}
1	0	0
1	1	1

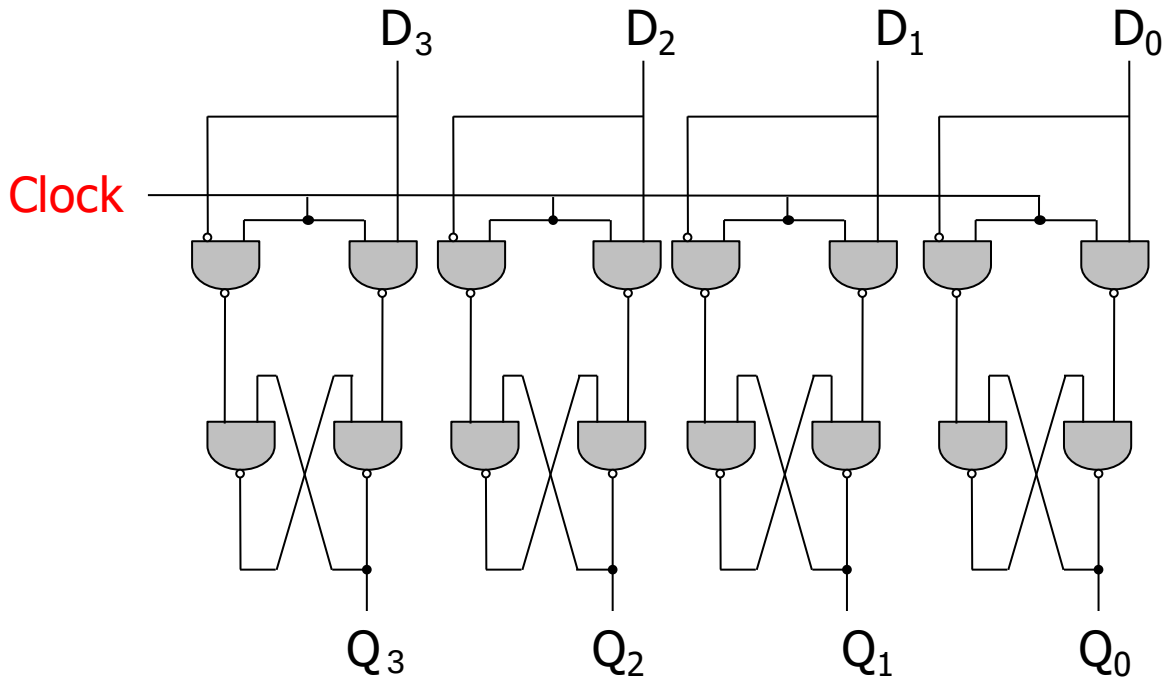
- **Q** takes the value of **D**, when **write enable (WE)** is set to 1
- **S** and **R** can never be 0 at the same time!

The Register

The Register

How can we use D latches to store **more** data?

- Use **more** D latches!
- A single Clock signal for all latches for simultaneous writes



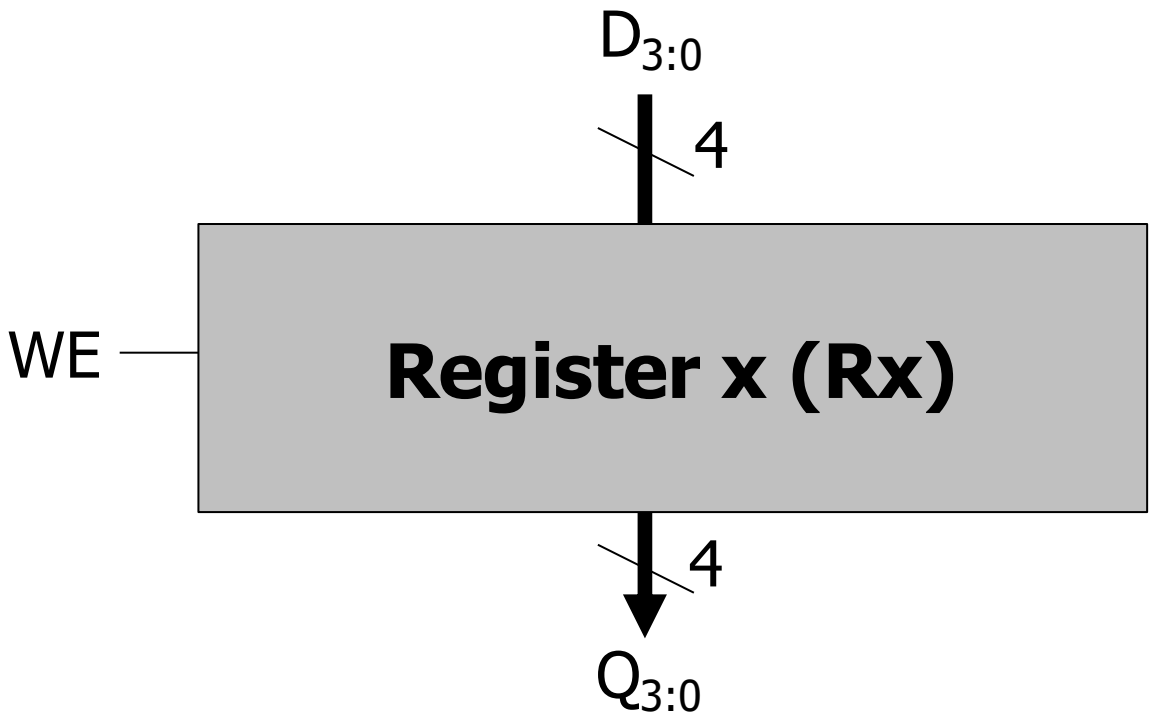
Here we have a **register,** or a structure that stores more than one bit and can be read from and written to

This **register** holds 4 bits, and its data is referenced as Q[3:0]

The Register

How can we use D latches to store **more** data?

- Use **more** D latches!
- A single Clock signal for all latches for simultaneous writes



Here we have a **register,** or a structure that stores more than one bit and can be read from and written to

This **register** holds 4 bits, and its data is referenced as Q[3:0]