Digital Design & Computer Arch.

Lecture 3: Combinational Logic II and Sequential Logic

Prof. Onur Mutlu

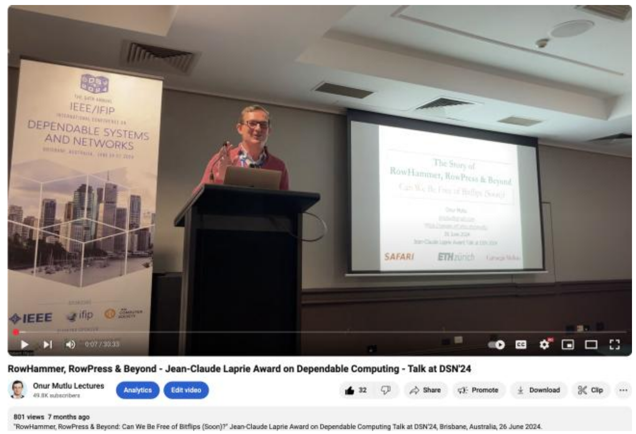
ETH Zürich

Spring 2026

26 February 2026

Extra Credit Assignment: Talk Analysis

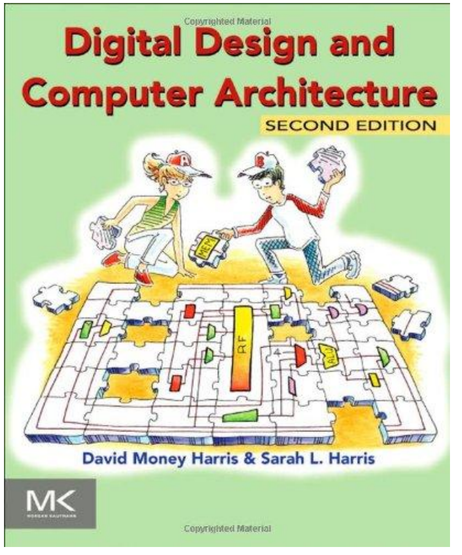
- The Story of RowHammer, RowPress & Beyond
- **Watch and analyze this short lecture (30 minutes)**
  - https://www.youtube.com/watch?v=U1EcqXlclKU (June 2024)

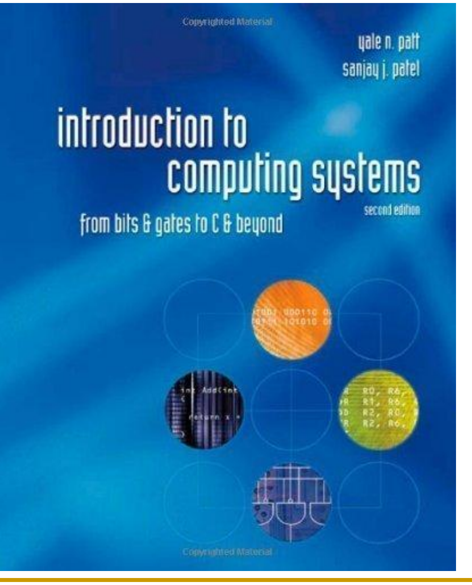


- **Assignment – for 1% extra credit**
  - **Write a good 1-page individualized summary (no AI use)**
    - What are your key takeaways? What did you learn?
    - What surprised you about the content presented? What excited you?
    - What do you think solutions should be like?
    - Submit your summary to Moodle – deadline March 21

Reading Assignments for This/Next Week

- Chapters 1-2 in Harris & Harris





- Chapters 1,2,3 in Patt and Patel

- Supplementary Lecture Slides on Binary Numbers

Reading Assignments for This/Next Week

- This week
  - Introduction
    - P&P Chapters 1 & 2 + H&H Chapter 1
  - Combinational Logic
    - P&P Chapter 3 until 3.3 + H&H Chapter 2

- Next week
  - Hardware Description Languages and Verilog
    - H&H Chapter 4 until 4.3 and 4.5
  - Sequential Logic
    - P&P Chapter 3.4 until end + H&H Chapter 3 in full

- Within 2-3 weeks, we will be done with
  - **P&P Chapters 1-3 + H&H Chapters 1-4**

What Will We Learn Today?

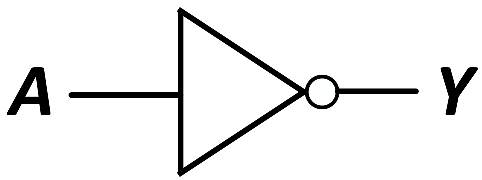
- Complete Combinational Logic
- Start Sequential Logic

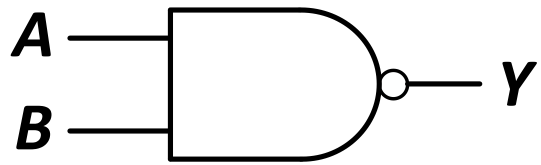
Combinational Logic II

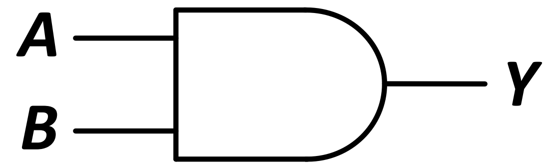
Combinational Logic Lecture Outline

- Building blocks of modern computers
  - Transistors
  - Logic gates
- Combinational logic circuits
- Boolean algebra
- Using Boolean algebra to represent combinational circuits
- Basic combinational logic blocks
- Simplifying combinational logic circuits

CMOS NOT, NAND, AND Gates



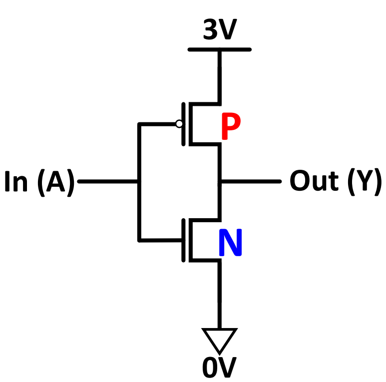


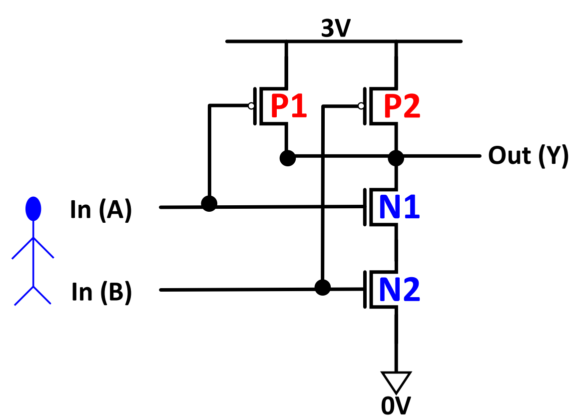


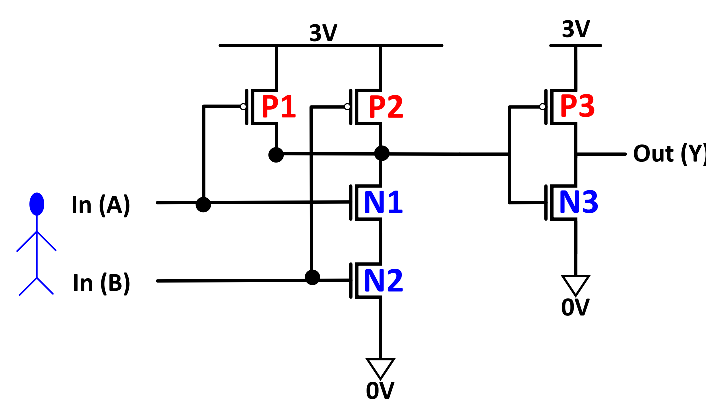
A	Y
0	1
1	0

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

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







Combinational Logic Circuits

Boolean Logic Equations

Using Boolean Equations to Represent a Logic Circuit

Recall: Boolean Equations Enable Us To…

- Represent the function of a combinational logic block
  - Functional Specification

- Methodically transform the function into simpler functions
  - which lead to different hardware realizations
  - Logic Minimization or Logic Simplification
  - We can automate this process → Computer-Aided Design or Electronic Design Automation

- Different Boolean expressions lead to different logic gate implementations
  → Different hardware area, cost, latency, energy properties

Recall: Standardized SOP and POS Forms

- Enable a single, universally-agreed-on way of representing a Boolean function starting from its truth table
  - Also called "canonical representations"

- Sum of Products (SOP) form

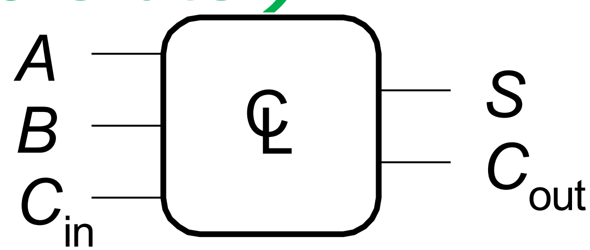
- Product of Sums (POS) form

Recall: Logic Simplification (Minimization)

- Using Boolean Algebra, we can simplify the SOP or POS form of any function in a methodical way

- Starting with the canonical SOP or POS form enables convenience and automation
  - Truth table → SOP/POS form → Boolean Simplification Rules

- ***Example (full 1-bit adder – more later):***



$$S = F(A, B, C_{in})$$
$$C_{out} = G(A, B, C_{in})$$

$$S = A \oplus B \oplus C_{in}$$ 3-input XOR

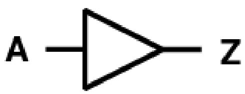
$$C_{out} = AB + AC_{in} + BC_{in}$$

3-input majority

Combinational Building Blocks used in Modern Computers

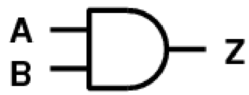
Recall: Common Logic Gates

Buffer



A	Z
0	0
1	1

AND



A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

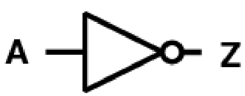
OR

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

XOR

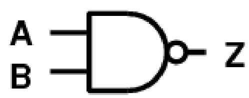
A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

Inverter



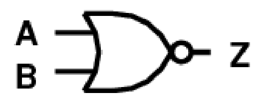
A	Z
0	1
1	0

NAND



A	B	Z
0	0	1
0	1	1
1	0	1
1	1	0

NOR



A	B	Z
0	0	1
0	1	0
1	0	0
1	1	0

XNOR

A	B	Z
0	0	1
0	1	0
1	0	0
1	1	1

We Will Cover Many Building Blocks

- Basic logic gates (AND, OR, NOT, NAND, NOR, XOR)
- Decoder
- Multiplexer
- Full Adder
- Programmable Logic Array (PLA)
- Comparator
- Arithmetic Logic Unit (ALU)
- Tri-State Buffer

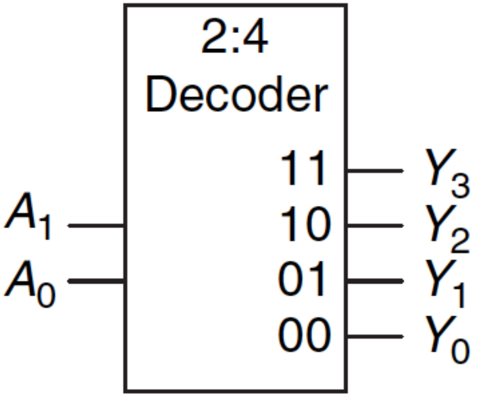
- Standard form representations: SOP & POS
- Logic simplification via Boolean Algebra
- Logical completeness

Decoder

Recall: Decoder

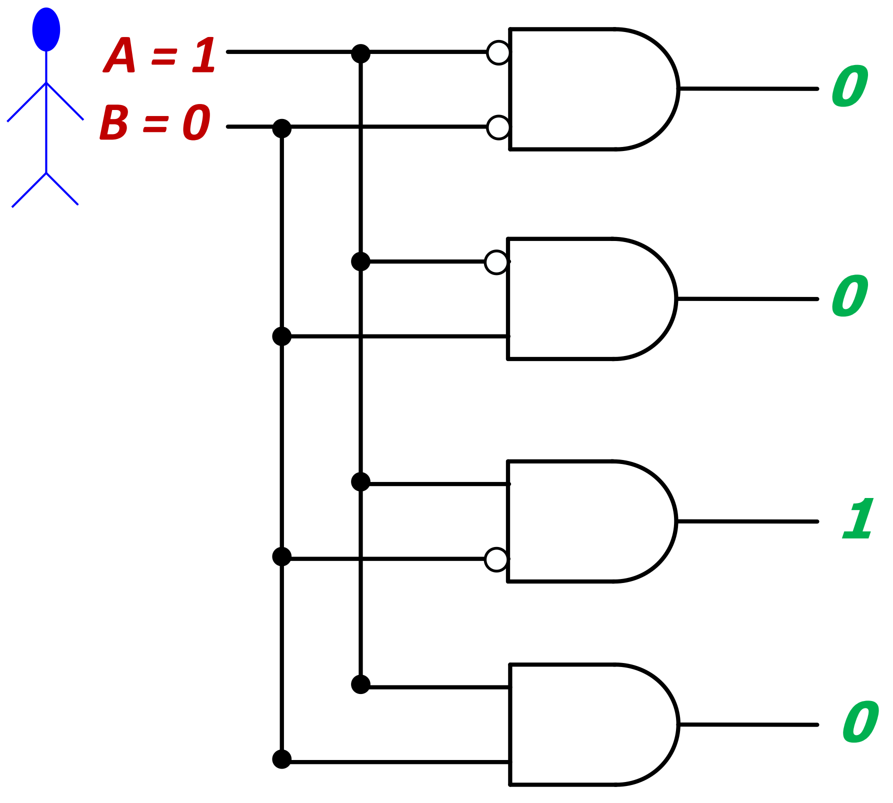
- "Input pattern detector"
- n inputs and 2^n outputs
- Exactly one of the outputs is 1 and all the rest are 0s
- The output that is logically 1 is the output corresponding to the input pattern that the logic circuit is expected to detect
- Example: 2-to-4 decoder

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



Recall: Decoder (II)

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

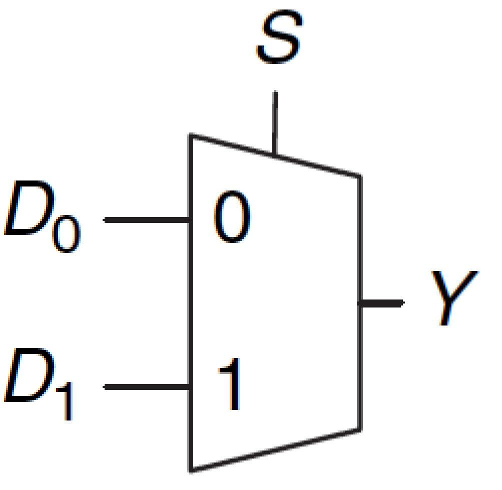


Multiplexer (MUX)

Recall: Multiplexer (MUX), or Selector

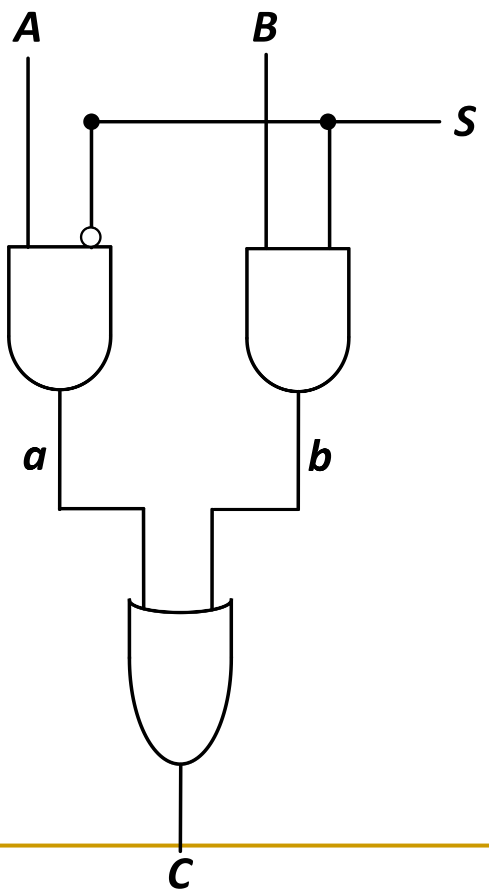
- Selects one of the *N* inputs to connect it to the output
  - based on the value of a $\log_2 N$-bit control input called select
- Example: 2-to-1 MUX

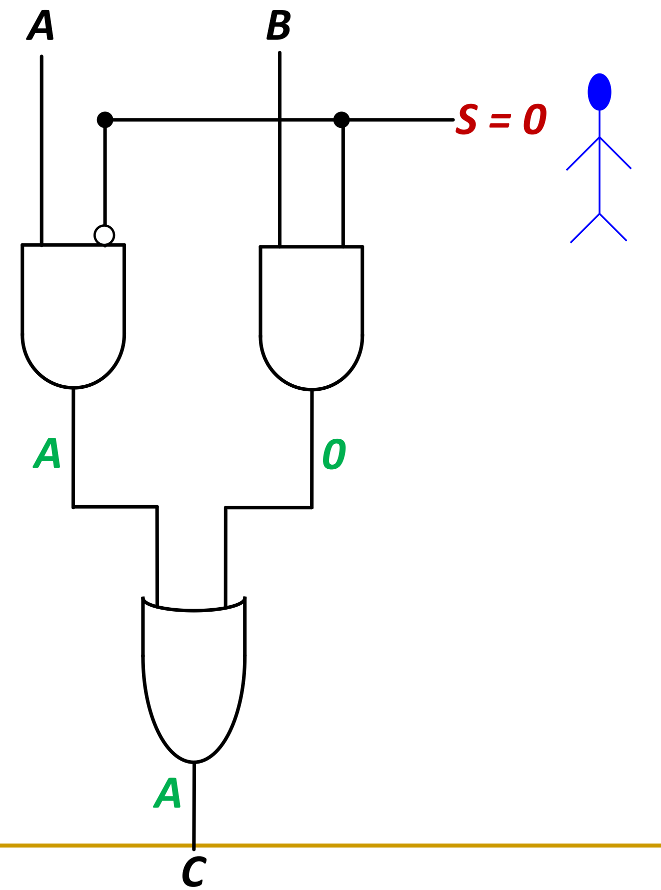
S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



Recall: 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

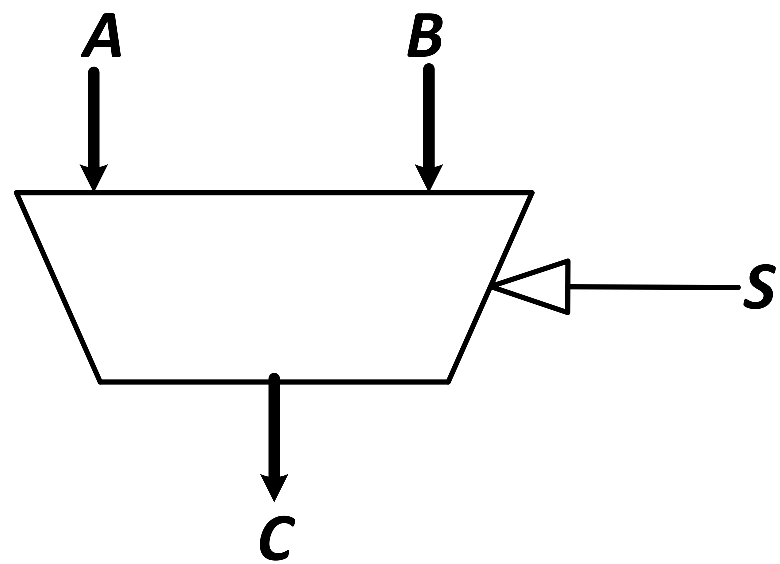




Recall: Multiplexer (MUX), or Selector (III)

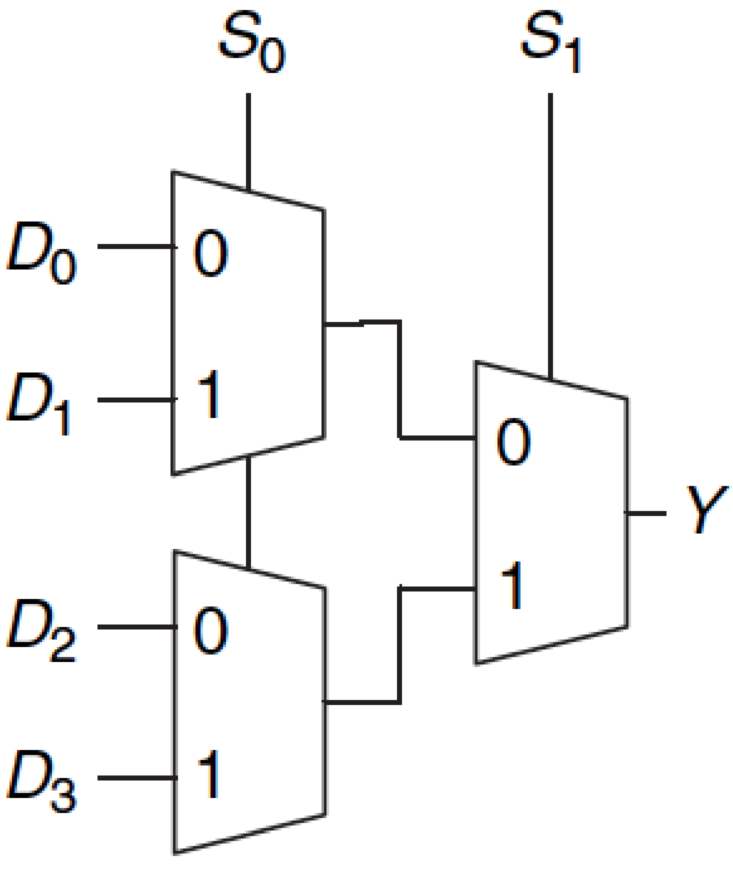
- The output C is always connected to either the input A or the input B
  - Output value depends on the value of the select line S

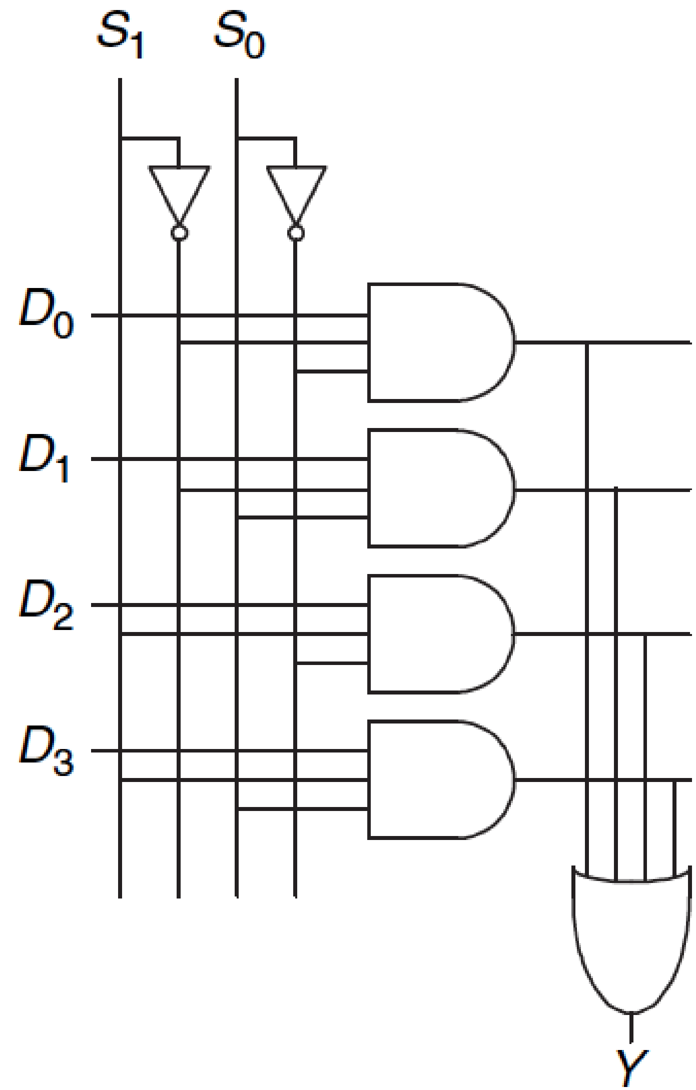
S	C
0	A
1	B



- Your task: Draw the schematic for an 4-input (4:1) MUX
  - Gate level: as a combination of basic AND, OR, NOT gates
  - Module level: As a combination of 2-input (2:1) MUXes

Recall: A 4-to-1 Multiplexer





Recall: 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

$Y = AB$

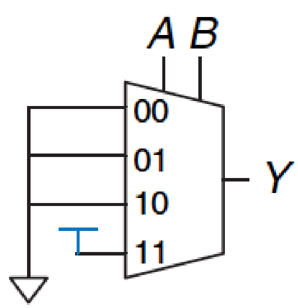
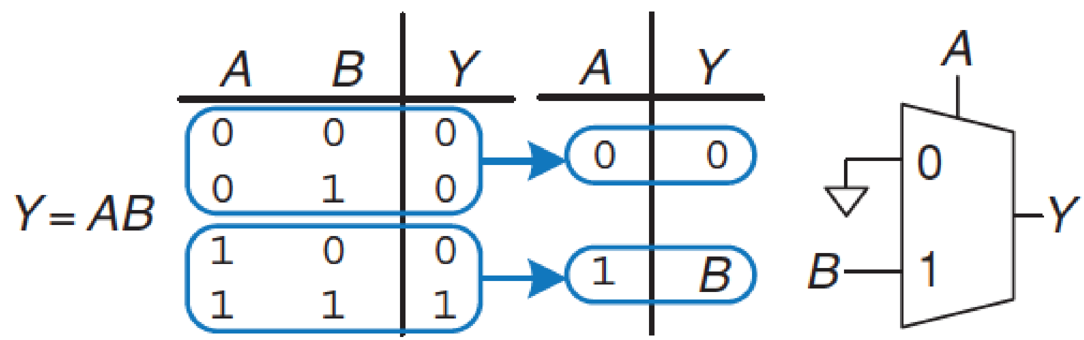


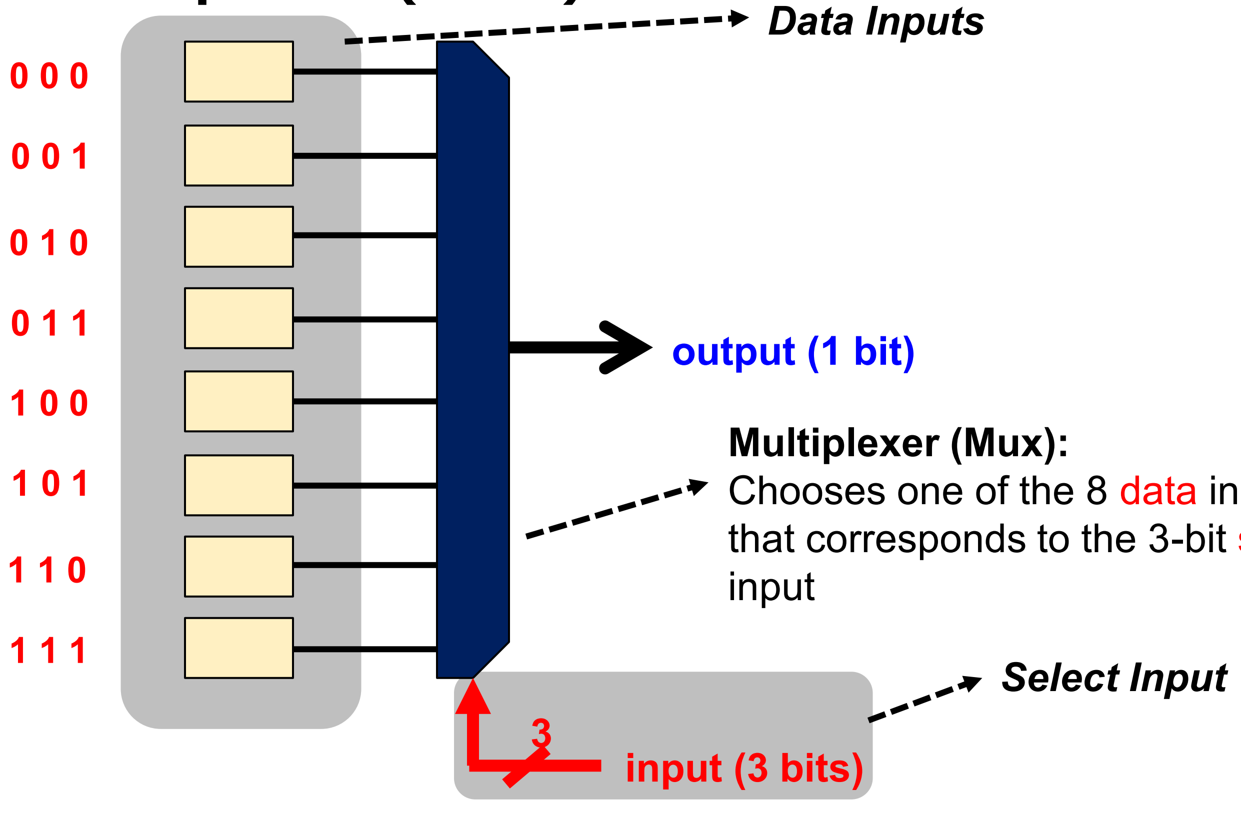
Figure 2.59 4:1 multiplexer implementation of two-input AND function



Idea: Formulate the truth table as a multiplexer

Recall: 8-Input Lookup Table (LUT)

- **3-bit input LUT (3-LUT)**



3-LUT can implement
any 3-bit input function

Recall: An Example of Programming a LUT

- A function that outputs '1' when there are **at least two '1's in a 3-bit input**

In C:

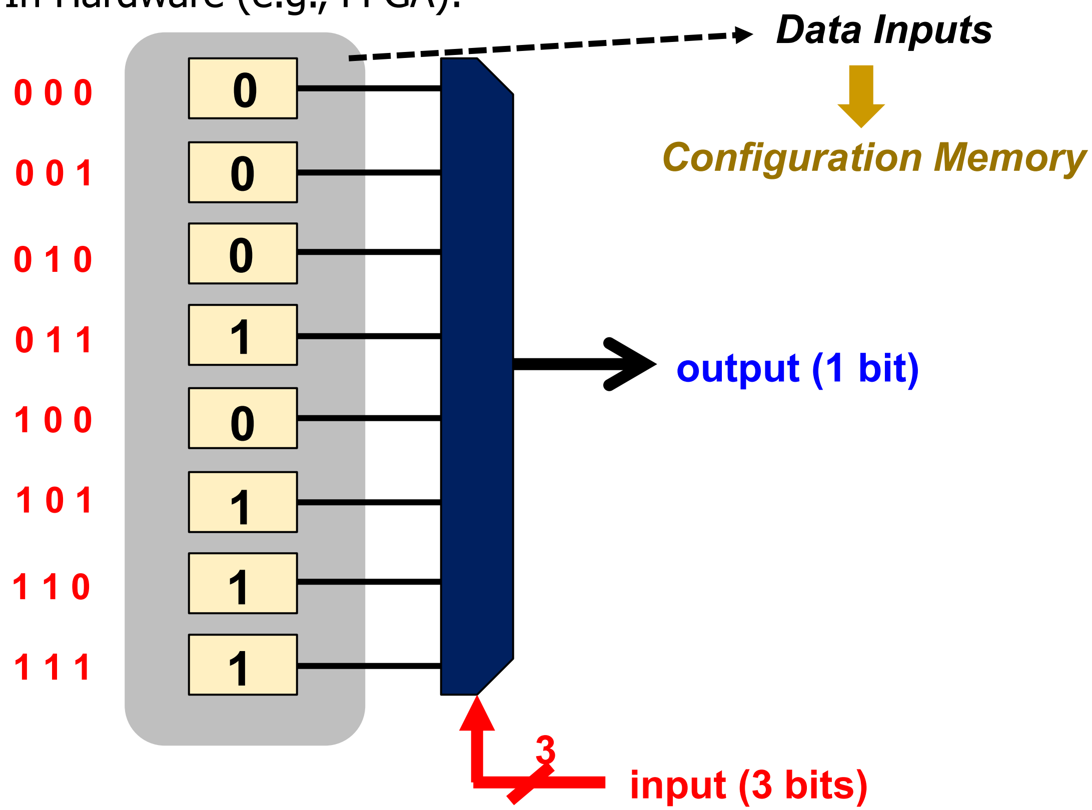
```
int count = 0;
for(int i = 0; i < 3; i++) {
    count += input & 1;
    input = input >> 1;
}

if(count > 1) return 1;

return 0;
```

```
switch(input){
   case 0:
   case 1:
   case 2:
   case 4:
      return 0;
   default:
      return 1;}
```

In Hardware (e.g., FPGA):



Recall: Aside: Logic Using Decoders (I)

- Decoders can be combined with OR gates to build logic functions.

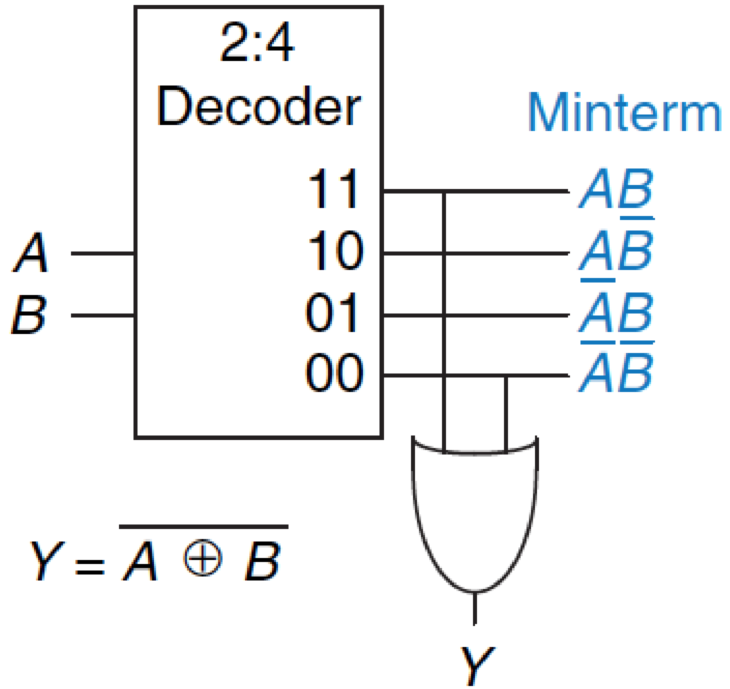


Figure 2.65 **Logic function using decoder**

Read H&H Chapter 2.8

We Will Cover Many Building Blocks

- Basic logic gates (AND, OR, NOT, NAND, NOR, XOR)
- Decoder
- Multiplexer
- Full Adder
- Programmable Logic Array (PLA)
- Comparator
- Arithmetic Logic Unit (ALU)
- Tri-State Buffer

- Standard form representations: SOP & POS
- Logic simplification via Boolean Algebra
- Logical completeness

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

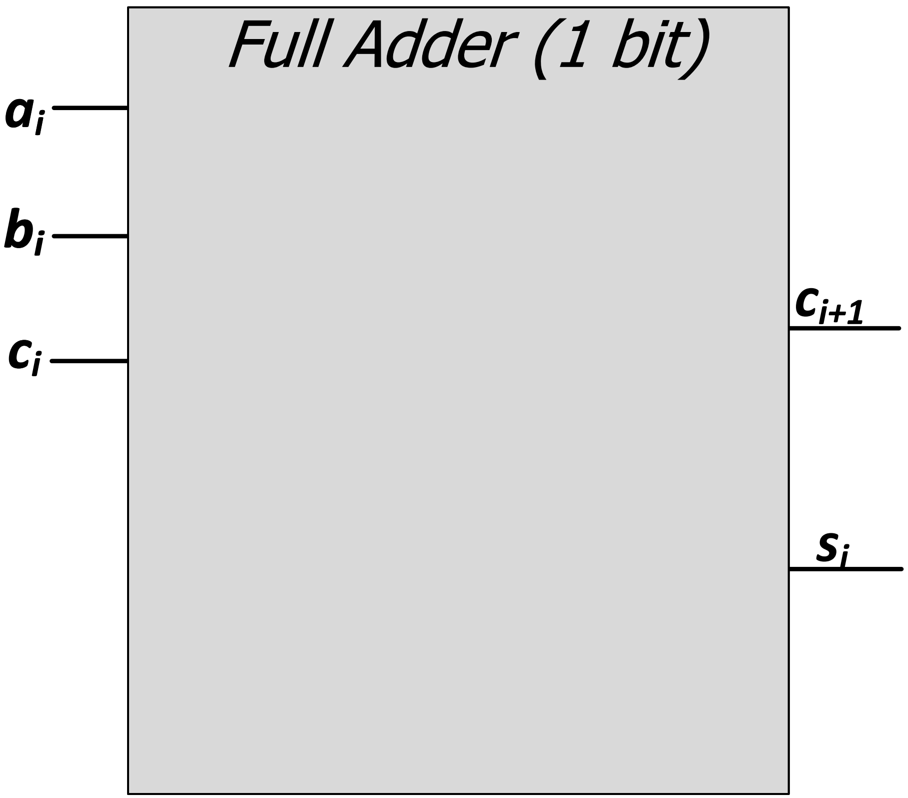
$$\begin{array}{r} a_{n-1}a_{n-2}\ldots a_1a_0 \\ b_{n-1}b_{n-2}\ldots b_1b_0 \\ C_n\ C_{n-1}\quad \ldots\quad C_1\ \ \\ \hline S_{n-1}\quad \ldots\quad S_1S_0 \end{array}$$

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

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**

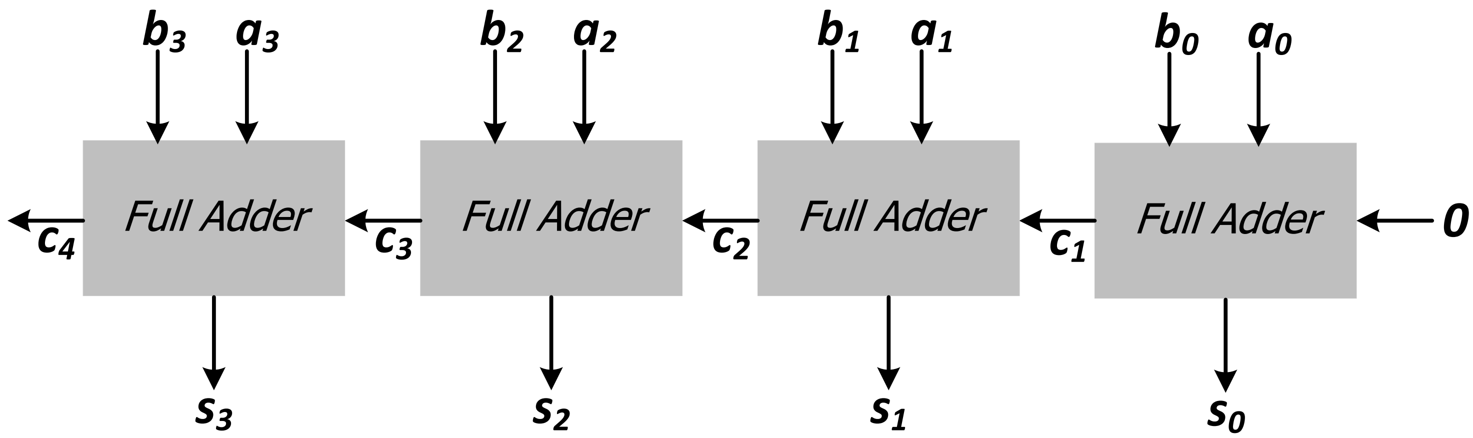


$$
\begin{array}{r}
a_{n-1}a_{n-2}\ldots a_1a_0 \\
b_{n-1}b_{n-2}\ldots b_1b_0 \\
C_n\,C_{n-1}\quad\ldots\quad C_1 \\
\hline
S_{n-1}\quad\ldots\quad S_1S_0
\end{array}
$$

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
			MAJ	XOR

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



$$
\begin{array}{rcccc}
 & a_3 & a_2 & a_1 & a_0 \\
+ & b_3 & b_2 & b_1 & b_0 \\
c_4 & c_3 & c_2 & c_1 & \\
\hline
 & s_3 & s_2 & s_1 & s_0
\end{array}
\qquad
\begin{array}{rcccc}
 & 1 & 0 & 1 & 1 \\
+ & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & \\
\hline
 & 0 & 1 & 0 & 0
\end{array}
$$

Adder Design: Ripple Carry Adder

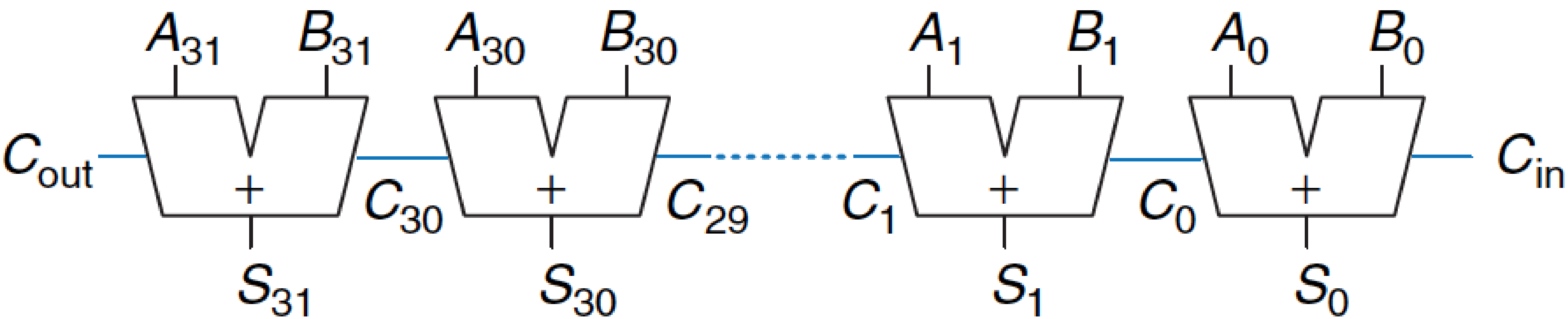
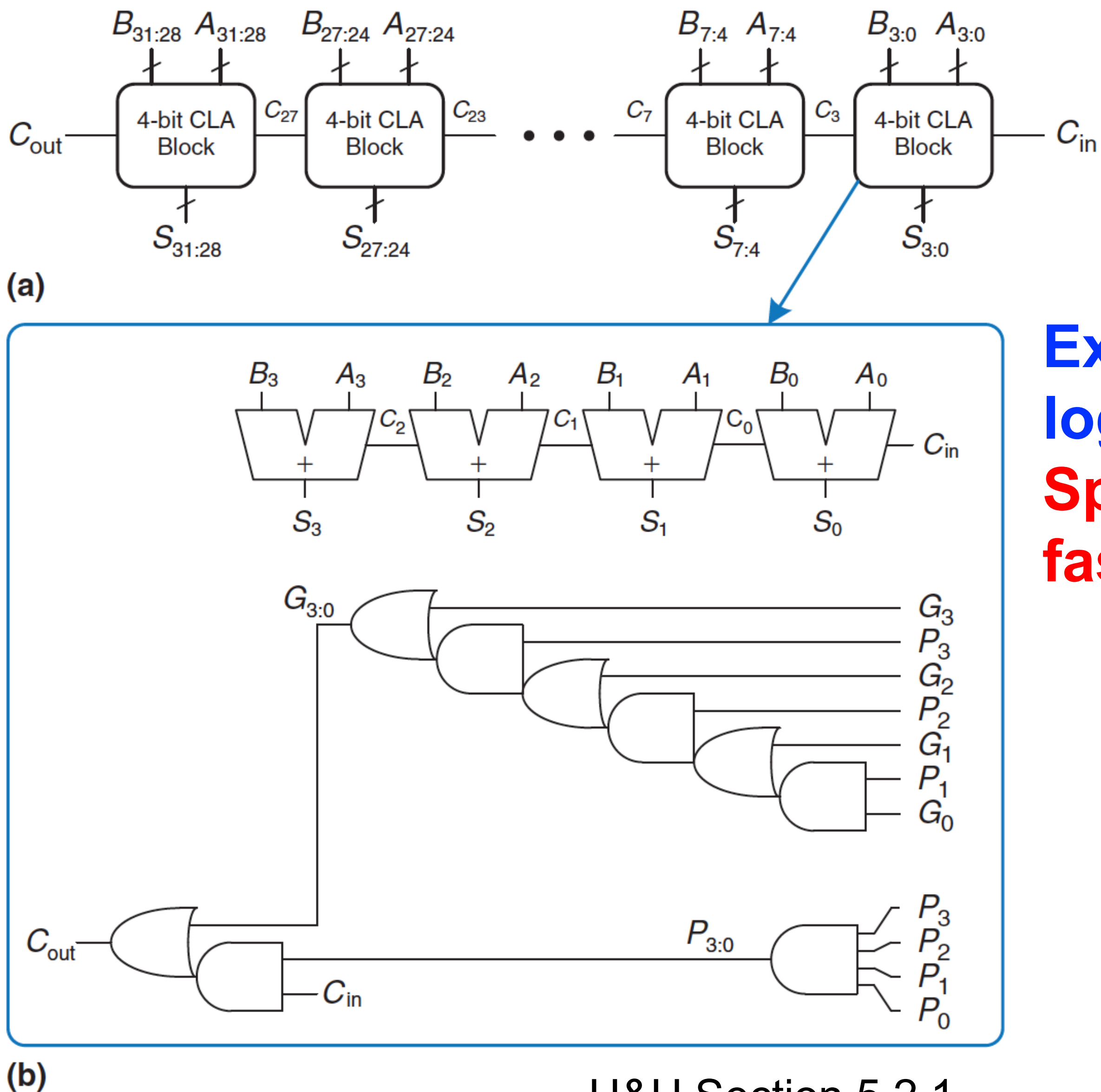


Figure 5.5 **32-bit ripple-carry adder**

H&H Section 5.2.1

Adder Design: Carry Lookahead Adder



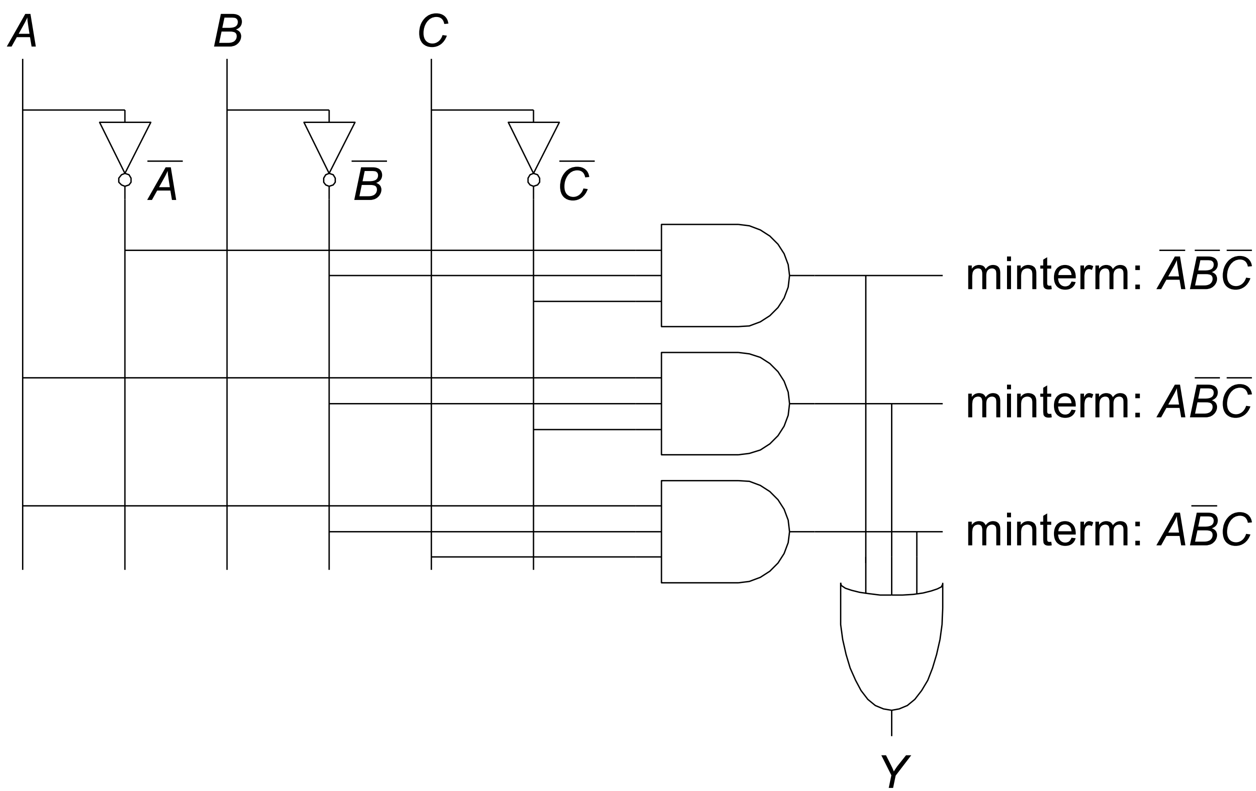
Example of logic specialization:
Specialized logic for fast carry calculation

H&H Section 5.2.1

Programmable Logic Array (PLA)

PLA: Recall: SOP Form

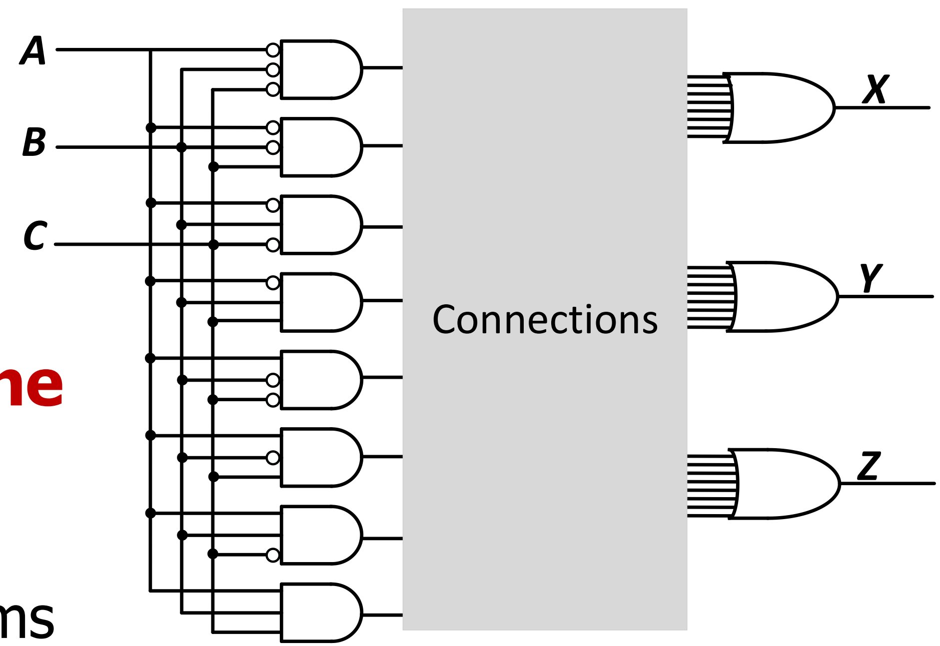
- **SOP (sum-of-products) leads to two-level logic**
- Example: $Y = (\overline{A} \cdot \overline{B} \cdot \overline{C}) + (A \cdot \overline{B} \cdot \overline{C}) + (A \cdot \overline{B} \cdot C)$



A PLA enables the two-level SOP implementation of **any** N-input M-output function

The Programmable Logic Array (PLA)

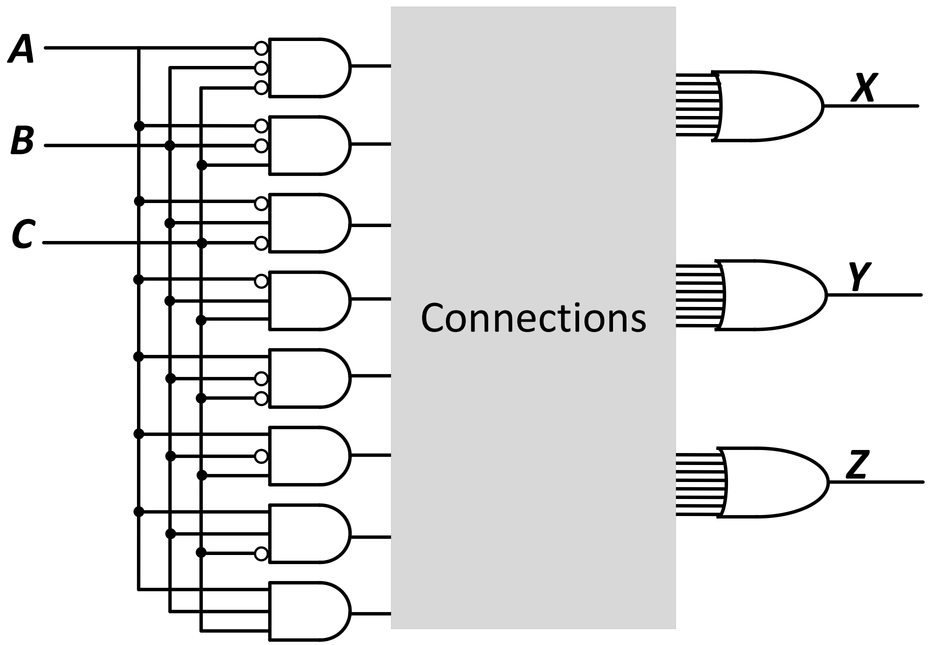
- The below logic structure is a very common building block for implementing any collection of logic functions one wishes to
- An array of AND gates followed by an array of OR gates
- **How do we determine the number of AND gates?**
  - **Remember SOP:** the number of possible minterms
  - For an n-input logic function, we need a PLA with 2^n n-input AND gates
- **How do we determine the number of OR gates?** The number of output columns in the truth table



A PLA enables the two-level SOP implementation of **any** N-input M-output function

The Programmable Logic Array (PLA)

- How do we implement a logic function?
  - Connect the output of an AND gate to the input of an OR gate if the corresponding minterm is included in the SOP
  - This is a simple programmable logic construct

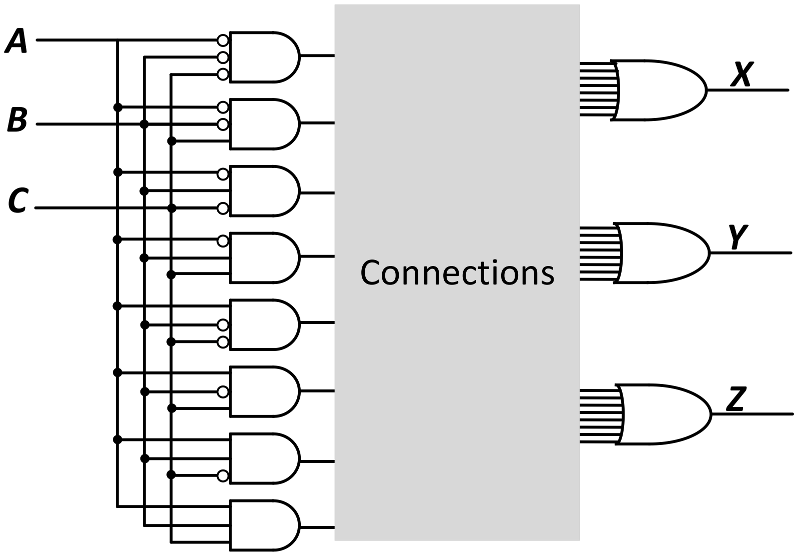


- **Programming a PLA**: we program the connections from AND gate outputs to OR gate inputs to implement a desired logic function

- Is there any other type of programmable logic?
  - Yes! An FPGA…
  - An FPGA uses more advanced structures, as we see in the labs

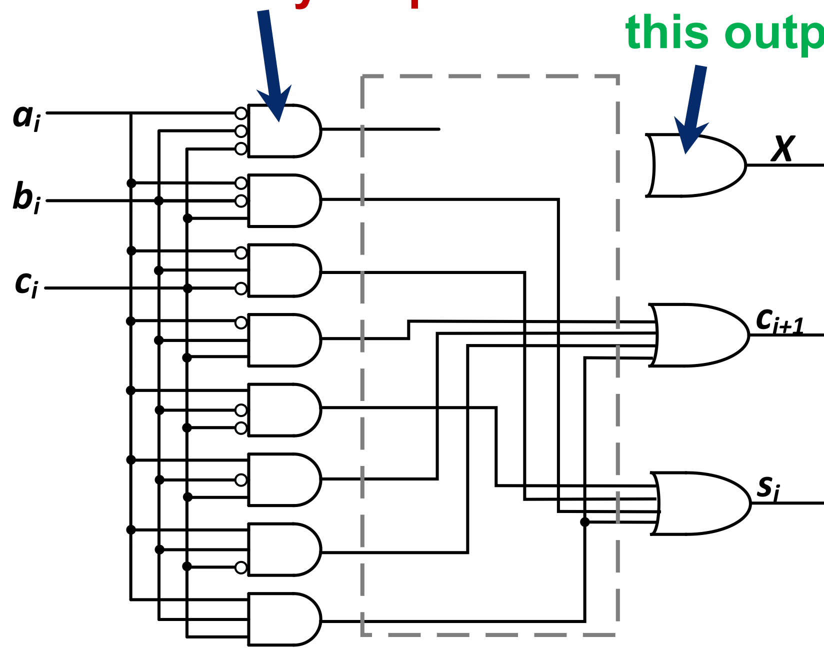
A PLA enables the two-level SOP implementation of **any** N-input M-output function

Implementing a Full Adder Using a PLA



This input should not be connected to any outputs

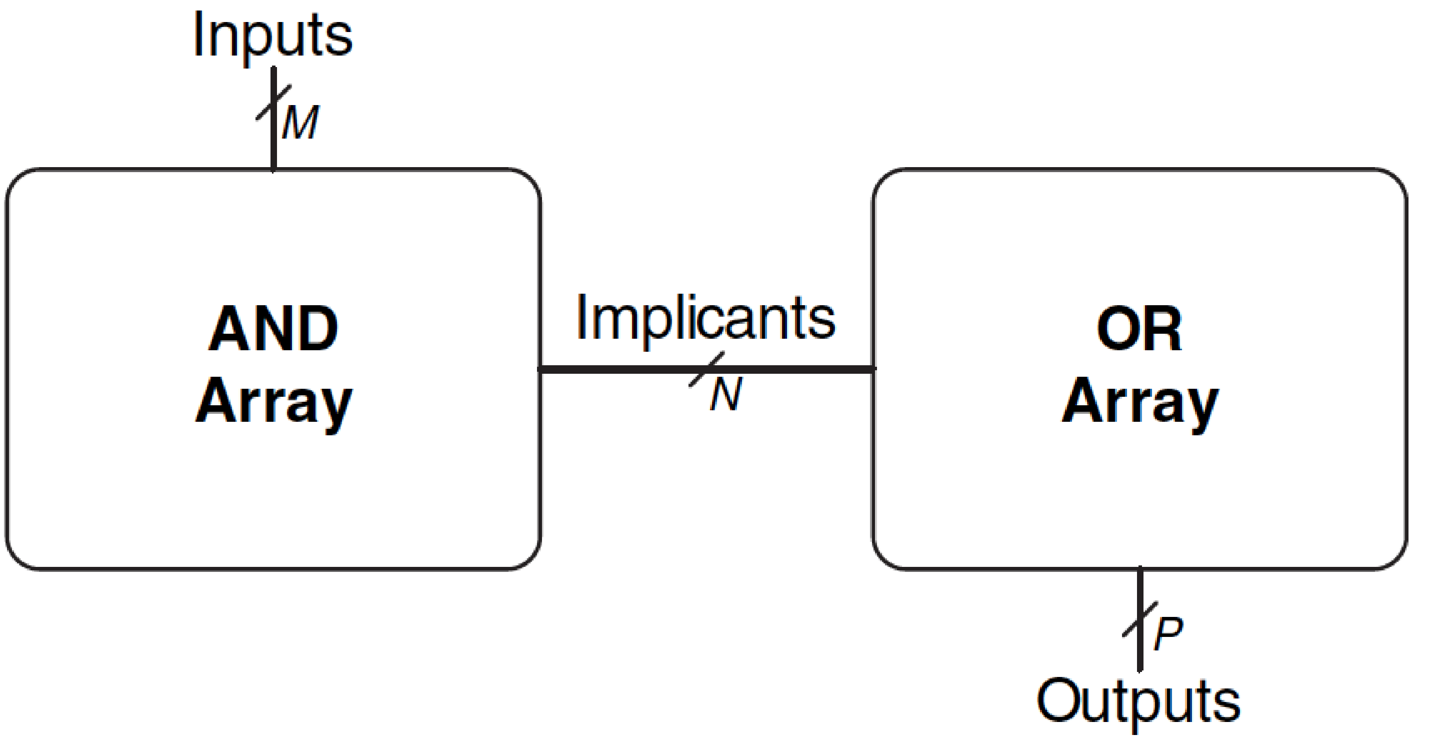
We do not need this output



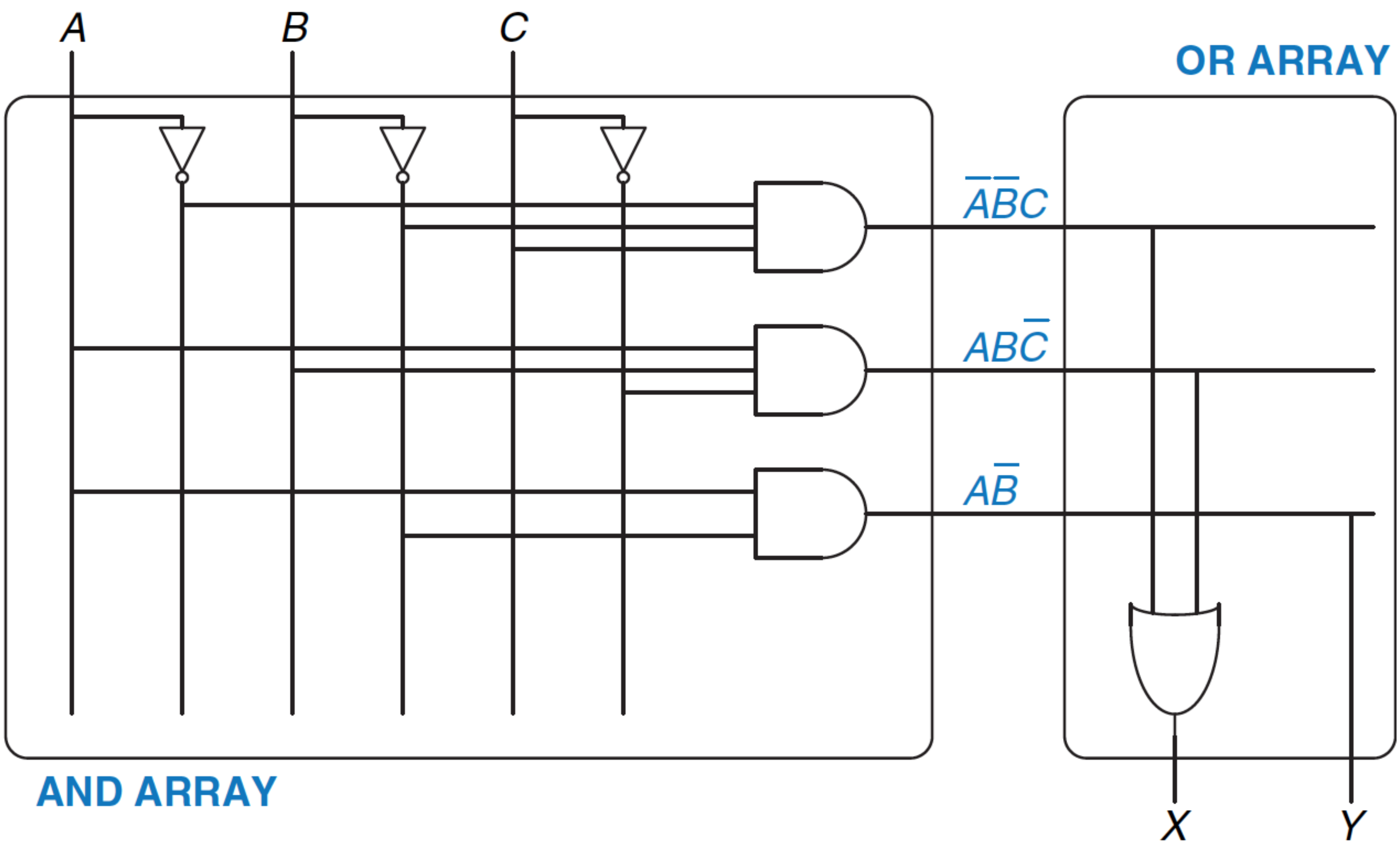
Truth table of a full adder

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

PLA Example (I)

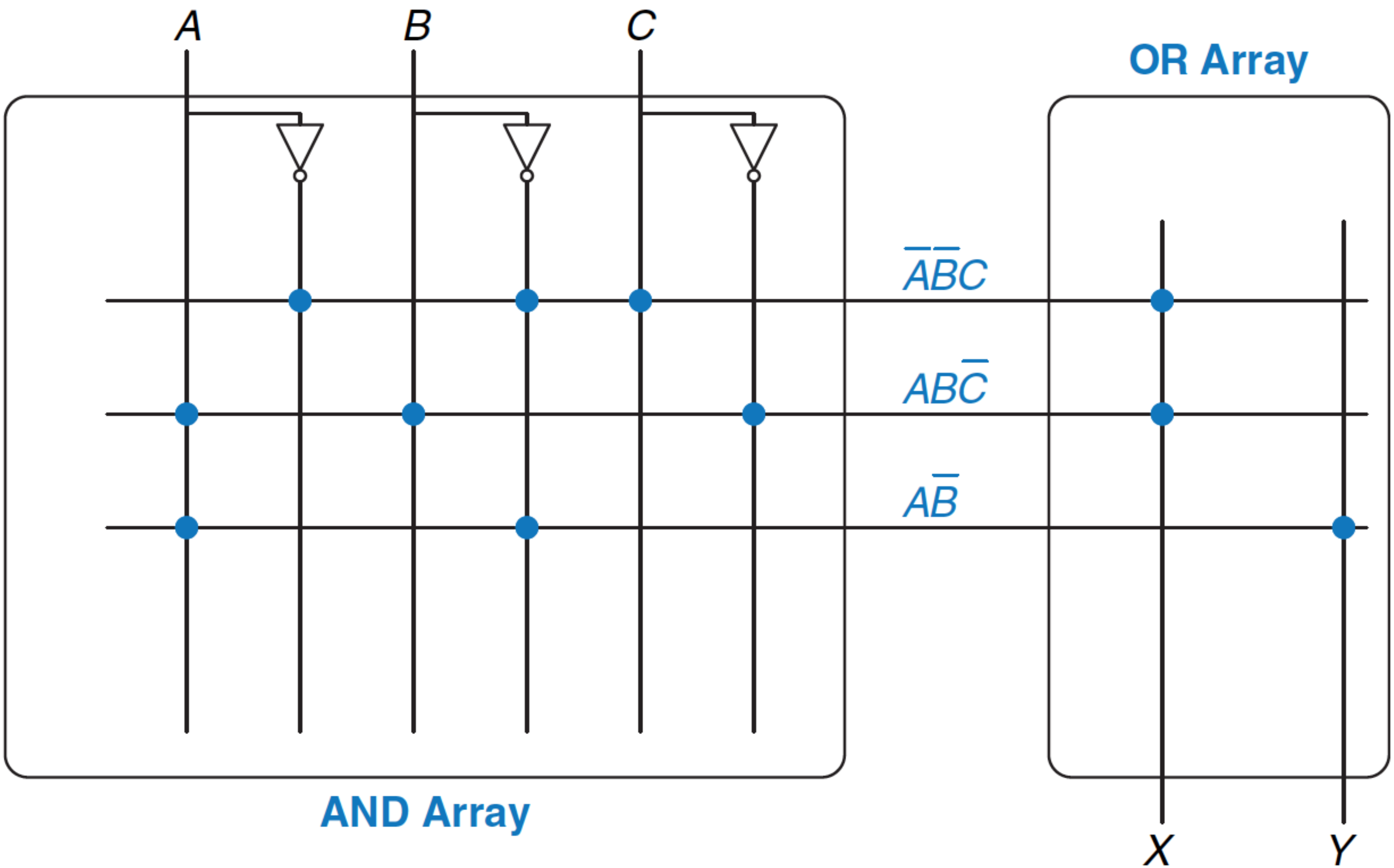


PLA Example Function (II)



Read H&H Chapter 5.6.1

PLA Example Function (III)



Read H&H Chapter 5.6.1

Logical Completeness

Logical (Functional) Completeness

- Any logic function we wish to implement could be accomplished with a PLA
  - PLA consists of only AND gates, OR gates, and inverters
  - We just have to program connections based on SOP of the intended logic function

- The set of gates {AND, OR, NOT} is logically complete because we can build a circuit to carry out the specification of any truth table we wish, without using any other kind of gate

- NAND is also logically complete. So is NOR.
  - Your task: Prove this.

More Combinational Blocks

More Combinational Building Blocks

- H&H Chapter 2 in full
  - Required Reading
  - E.g., see Tri-state Buffer and Z values in Section 2.6

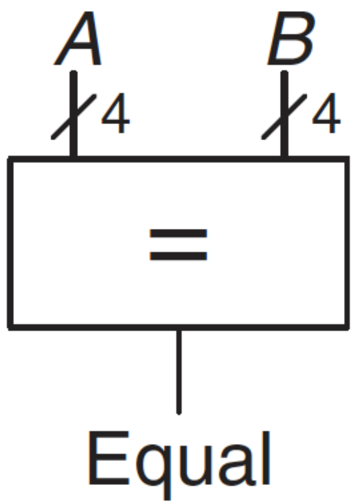
- H&H Chapter 5
  - Will be required reading soon

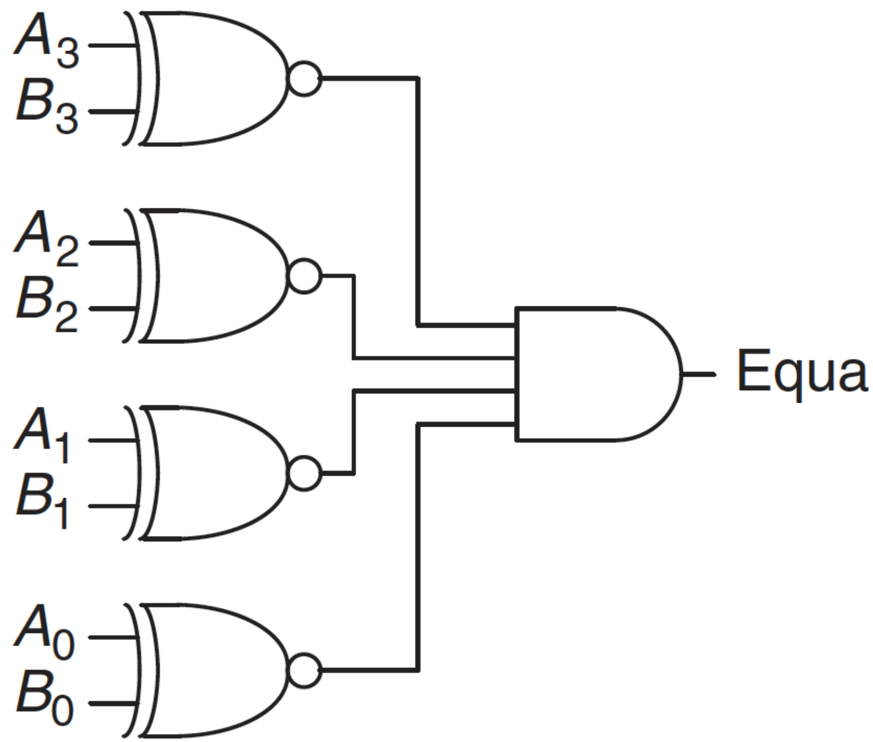
- You will benefit greatly by reading the "combinational" parts of Chapter 5 soon.
  - Sections 5.1 and 5.2
  - E.g., Adder, Subtractor, Comparator, Shifter/Rotator, Multiplier, Divider

Comparator

Equality Checker (Compare if Equal)

- Checks if two N-input values are exactly the same
- Example: 4-bit Comparator





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:

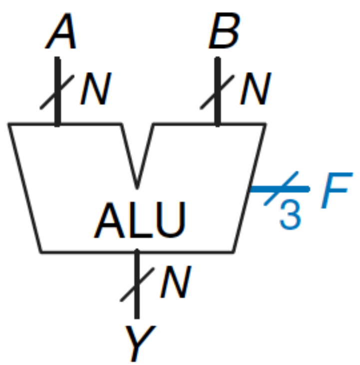


Figure 5.14 ALU symbol

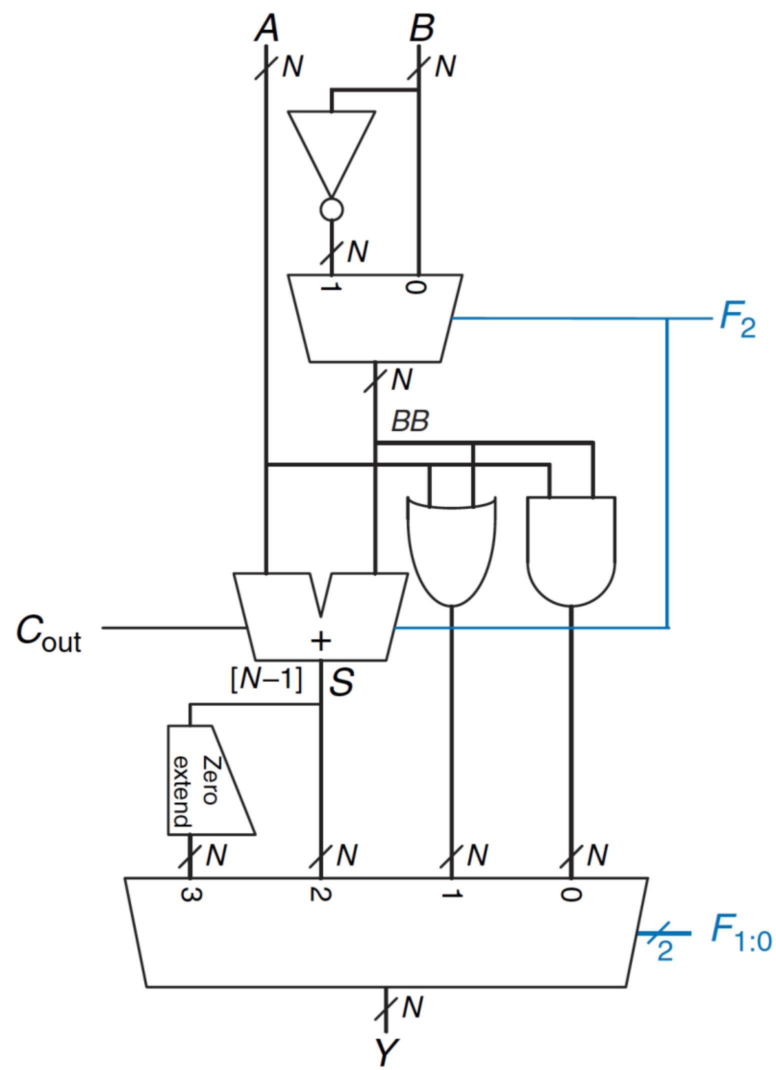
Table 5.1 ALU operations

$F_{2:0}$	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)

Table 5.1 ALU operations

$F_{2:0}$	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



More Combinational Building Blocks

- See H&H Chapter 5.2 for
  - Subtractor (using 2's Complement Representation)
  - Shifter and Rotator
  - Multiplier
  - Divider
  - ...

More Combinational Building Blocks

- H&H Chapter 2 in full
  - Required Reading
  - E.g., see Tri-state Buffer and Z values in Section 2.6

- H&H Chapter 5
  - Will be required reading soon

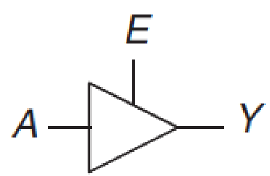
- You will benefit greatly by reading the "combinational" parts of Chapter 5 soon.
  - Sections 5.1 and 5.2
  - E.g., Adder, Subtractor, Comparator, Shifter/Rotator, Multiplier, Divider

Tri-State Buffer

Tri-State Buffer

- A tri-state buffer enables gating of different signals onto a wire

Tristate Buffer



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

Figure 2.40 Tristate buffer

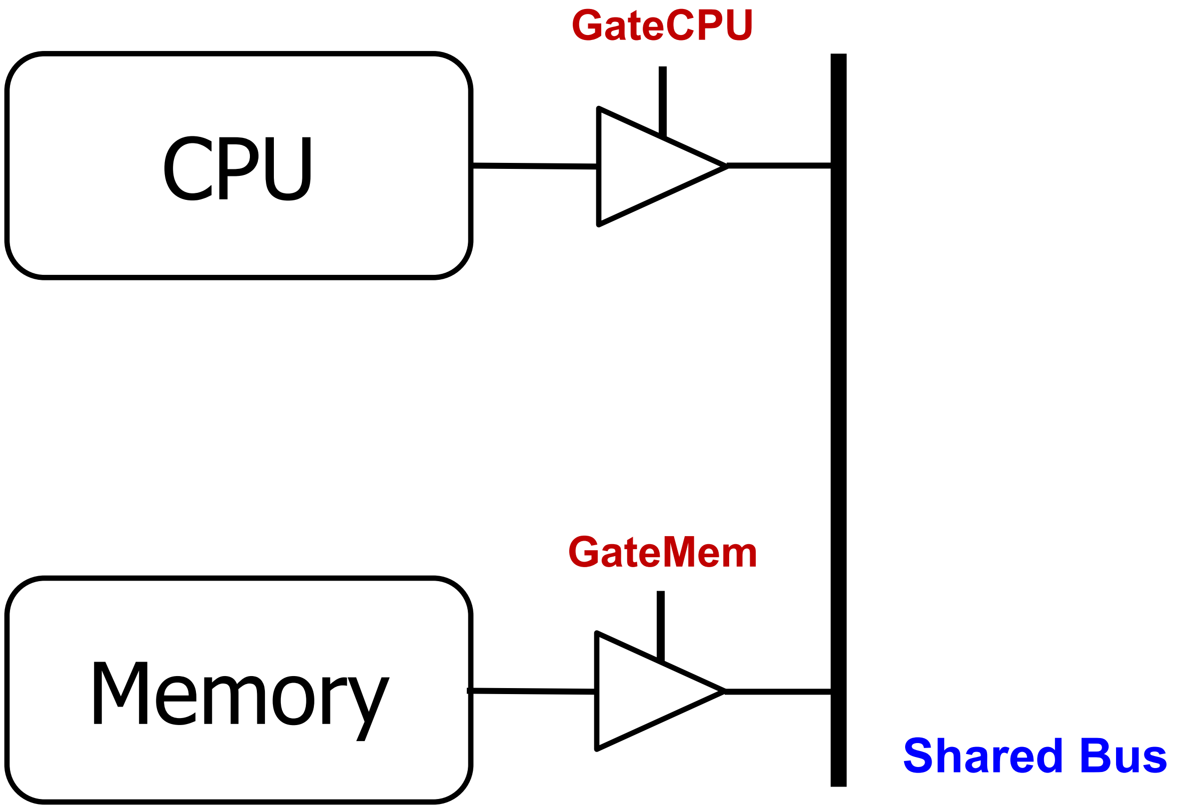
A tri-state buffer acts like a switch

- Floating signal (Z): Signal that is not driven by any circuit
  - Open circuit, floating wire

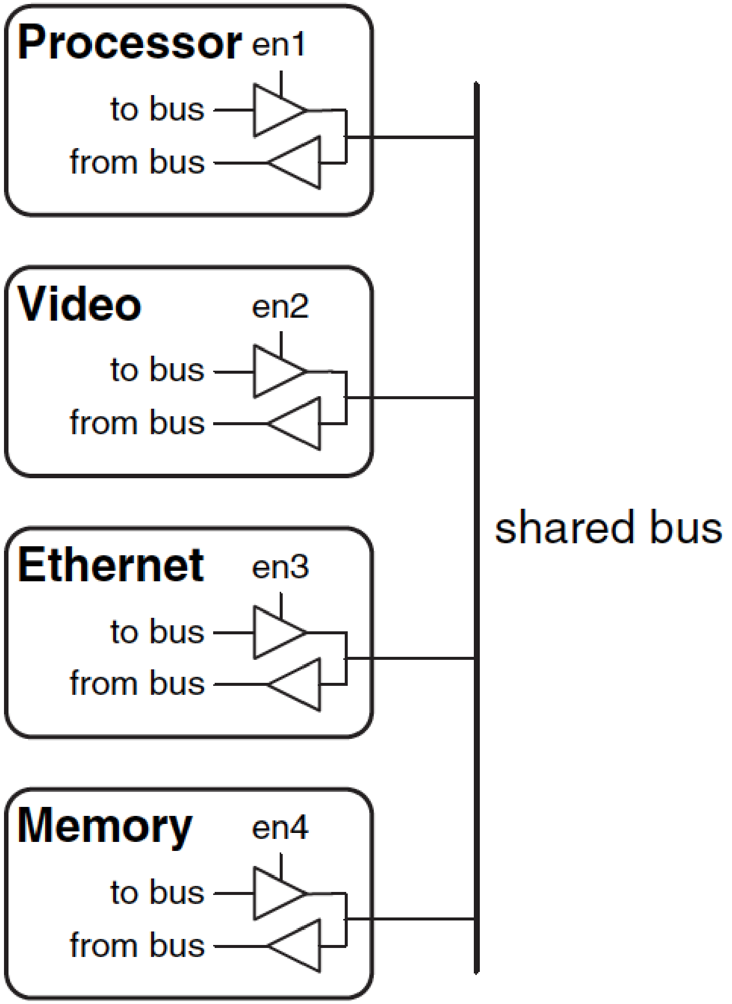
Example: Use of Tri-State Buffers

- 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

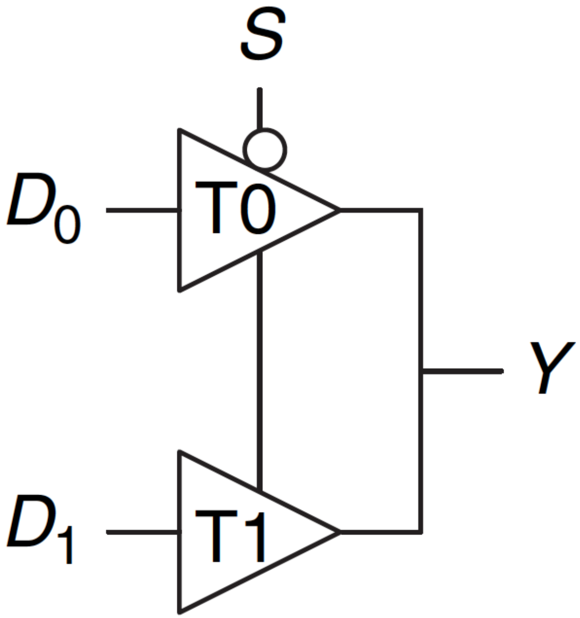
Example Design with Tri-State Buffers



Another Example



Multiplexer Using Tri-State Buffers



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

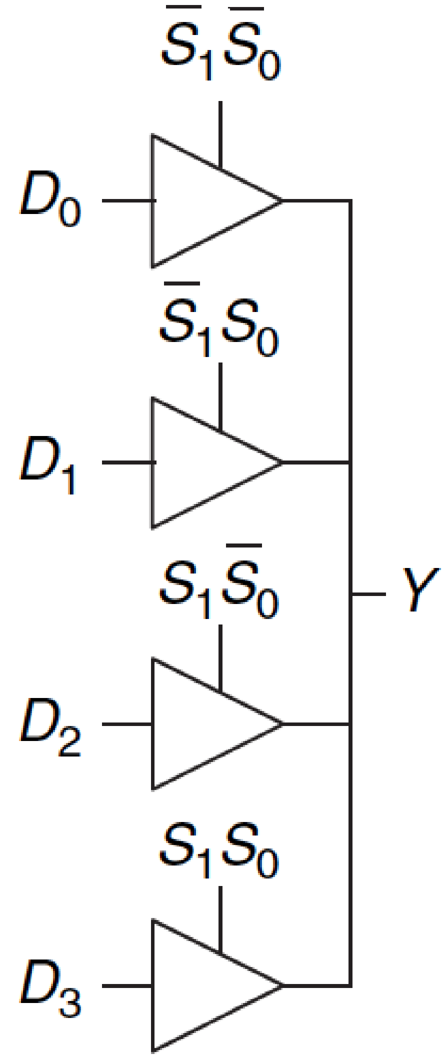
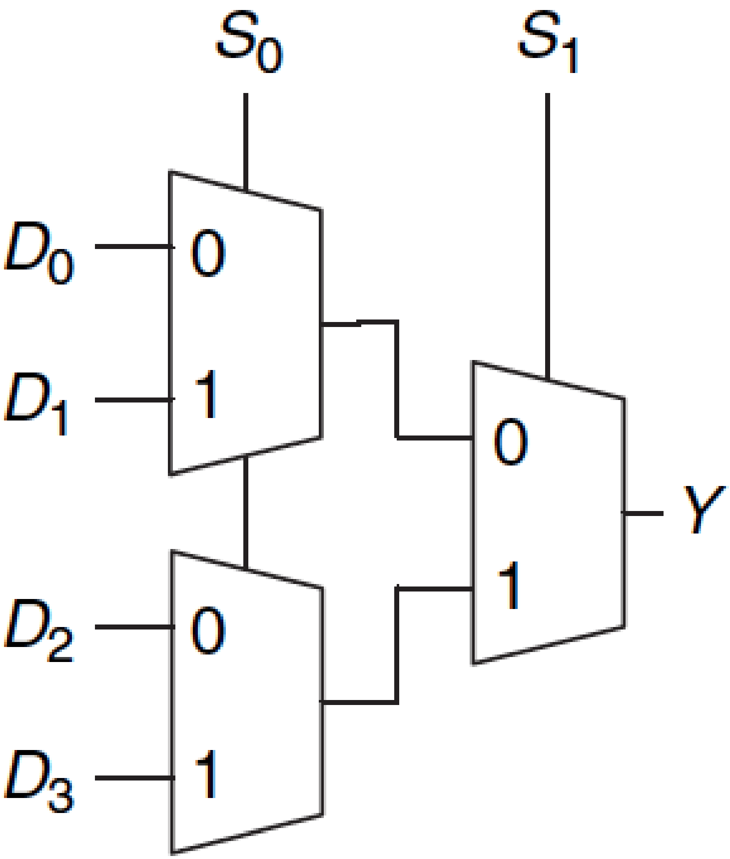
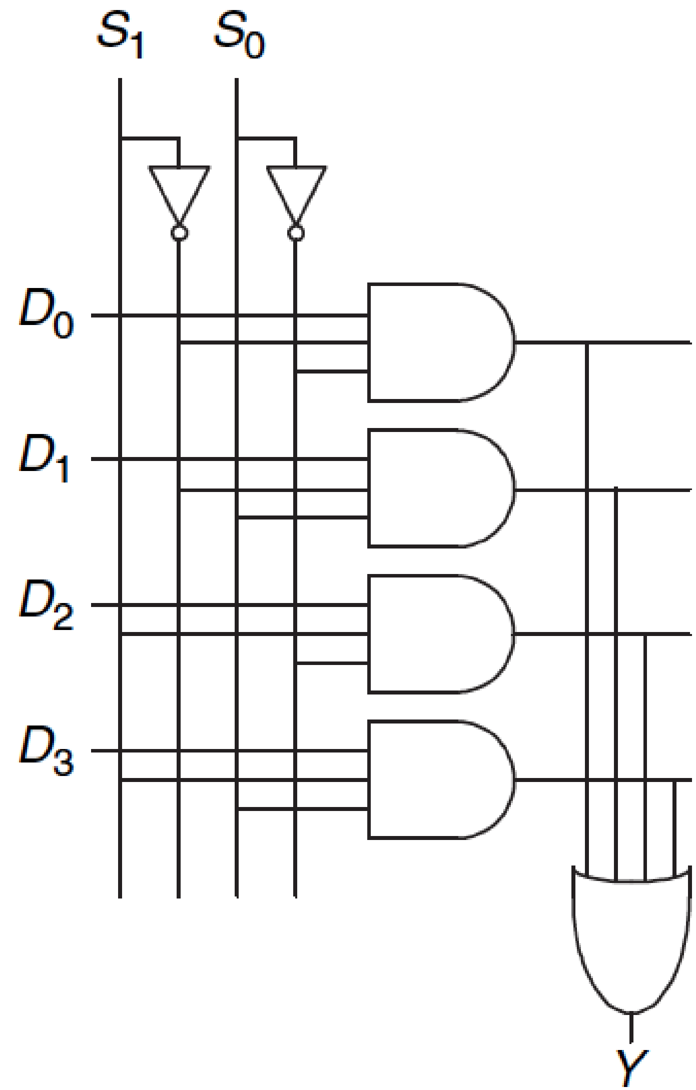


Figure 2.56 Multiplexer using tristate buffers

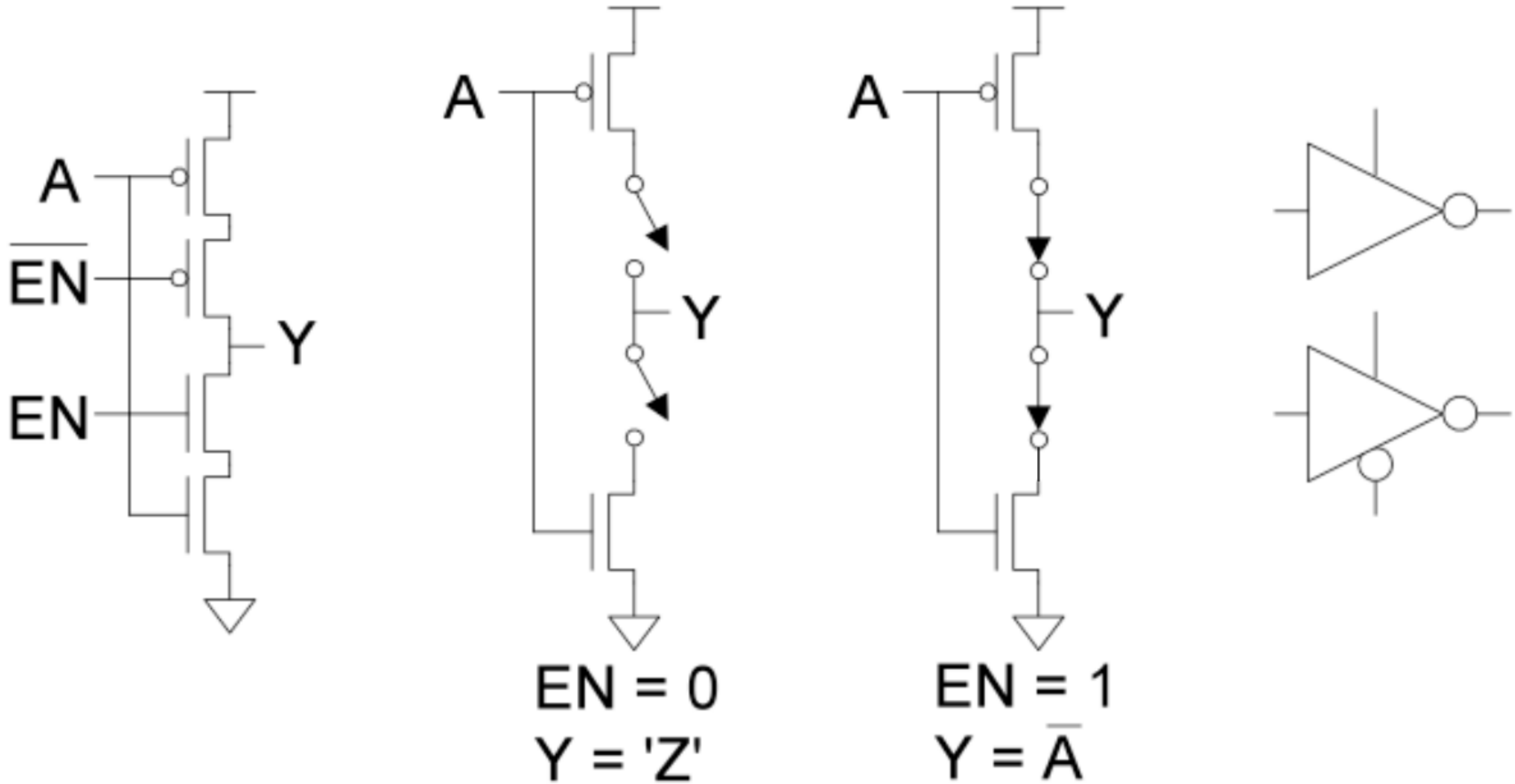
Recall: A 4-to-1 Multiplexer





Digging Deeper: Tri-State Buffer in CMOS

- How do you implement a Tri-State Buffer using transistors?



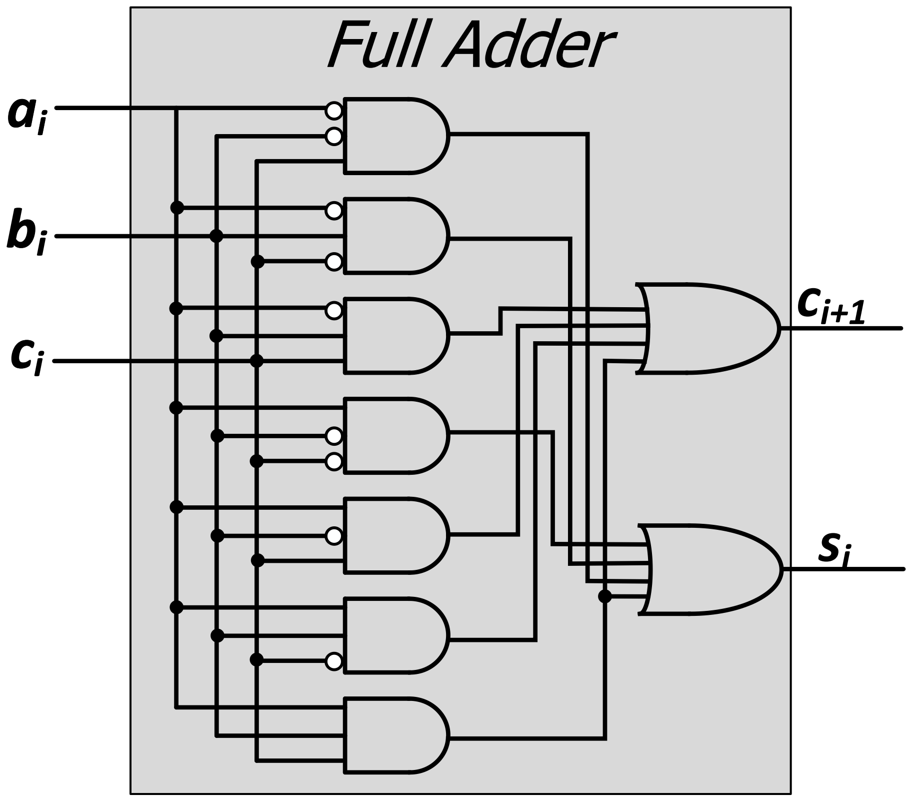
We Covered Combinational Logic Blocks

- Basic logic gates (AND, OR, NOT, NAND, NOR, XOR)
- Decoder
- Multiplexer
- Full Adder
- Programmable Logic Array (PLA)
- Comparator
- Arithmetic Logic Unit (ALU)
- Tri-State Buffer

- Standard form representations: SOP & POS
- Logic simplification via Boolean Algebra
- Logical completeness

Logic Simplification using Boolean Algebra Rules

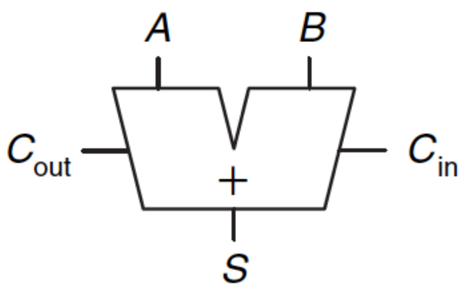
Recall: Full Adder in SOP Form Logic



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

Goal: Simplified Full Adder

Full Adder



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

$S = A \oplus B \oplus C_{in}$ 3-input XOR

$C_{out} = AB + AC_{in} + BC_{in}$ 3-input majority

How do we simplify Boolean logic?

How do we automate simplification?

Quick Recap on Logic Simplification

- The original Boolean expression (i.e., logic circuit) may not be optimal (e.g., in number of terms or logic gates)

$$F = \sim A(A + B) + (B + AA)(A + \sim B)$$

- Can we reduce a given Boolean expression to an equivalent expression with fewer terms?

$$F = A + B$$

- The goal of logic simplification:
  - Reduce the number of gates/inputs
  - Reduce implementation cost (and potentially latency & power)

A basis for what the automated design tools are doing today

Logic Simplification

- Systematic techniques for simplifications
  - amenable to automation

Key Tool: The Uniting Theorem — $F = A\overline{B} + AB$

A	B	F
0	0	0
0	1	0
1	0	1
1	1	1

$$F = A\overline{B} + AB = A(\overline{B} + B) = A(1) = A$$

B's value changes within the rows where F==1 ("ON set")

A's value does NOT change within the ON-set rows

If an input (B) can change without changing the output, that input value is not needed

→ *B is eliminated, A remains*

A	B	G
0	0	1
0	1	0
1	0	1
1	1	0

$$G = \overline{A}\overline{B} + A\overline{B} = (\overline{A} + A)\overline{B} = \overline{B}$$

B's value stays the same within the ON-set rows

A's value changes within the ON-set rows

→ *A is eliminated, B remains*

Logic Simplification

- Systematic techniques for simplifications
  - amenable to automation

Key Tool: The Uniting Theorem — $F = A\overline{B} + AB$

A	B	F
0	0	0
0		
1		
1	1	1

$$F = A\overline{B} + AB = A(\overline{B} + B) = A(1) = A$$

Essence of Simplification:

Find two-element subsets of the ON-set where only one variable changes its value. This single varying variable *can be eliminated!*

value is not needed

→ *B is eliminated, A remains*

A	B	G
0	0	1
0	1	0
1	0	1
1	1	0

$$G = \overline{A}\,\overline{B} + A\overline{B} = (\overline{A} + A)\overline{B} = \overline{B}$$

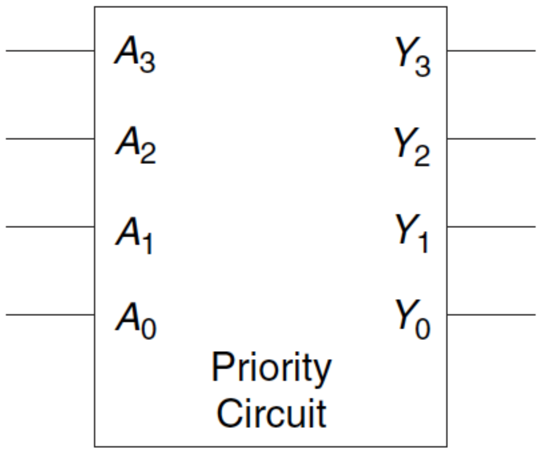
B's value stays the same within the ON-set rows

A's value changes within the ON-set rows

→ *A is eliminated, B remains*

Logic Simplification Example: Priority Circuit

- Priority Circuit
  - Inputs: "Requestors" with priority levels
  - Outputs: "Grant" signal for each requestor
  - Example 4-bit priority circuit
  - Real life example: Imagine a bus requested by 4 processors



A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

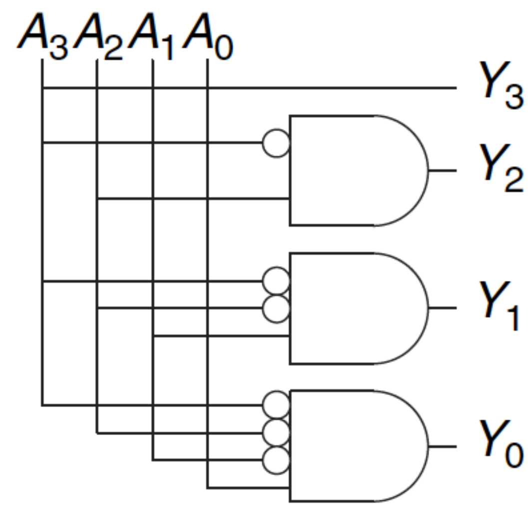
Simplified Priority Circuit

- Priority Circuit
  - Inputs: "Requestors" with priority levels
  - Outputs: "Grant" signal for each requestor
  - Example 4-bit priority circuit

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	0
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	0
1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0

A_3	A_2	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	X	0	0	1	0
0	1	X	X	0	1	0	0
1	X	X	X	1	0	0	0

Figure 2.29 Priority circuit truth table with don't cares (X's)



X (Don't Care) means *I don't care what the value of this input is*

Logic Simplification: Karnaugh Maps (K-Maps)

Karnaugh Maps are Fun…

- A pictorial way of minimizing circuits by visualizing opportunities for simplification
- They are for you to **study on your own**…
  - We may cover them later if time permits (very unlikely)

- See backup slides
- Read H&H Section 2.7
- Watch videos of Lectures 5 and 6 from 2019 DDCA course:
  - https://youtu.be/0ks0PeaOUjE?list=PL5Q2soXY2Zi8J58xLKBNFQFHRO3GrXxA9&t=4570
  - https://youtu.be/ozs18ARNG6s?list=PL5Q2soXY2Zi8J58xLKBNFQFHRO3GrXxA9&t=220

We Are Done with Combinational Logic

- Building blocks of modern computers
  - Transistors
  - Logic gates
- Combinational logic circuits
- Boolean algebra
- Using Boolean algebra to represent combinational circuits
- Basic combinational logic blocks
- Simplifying combinational logic circuits

Sequential Logic

Agenda for Today and Tomorrow

- Today
  - Start (and finish) Sequential Logic
- Tomorrow & Next week
  - Hardware Description Languages and Verilog
    - Combinational Logic
    - Sequential Logic
  - Timing and Verification

Sequential Logic Circuits and Design

What We Will Learn Today

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

- **Sequential logic circuits**
  - State & Clock
  - Asynchronous vs. Synchronous

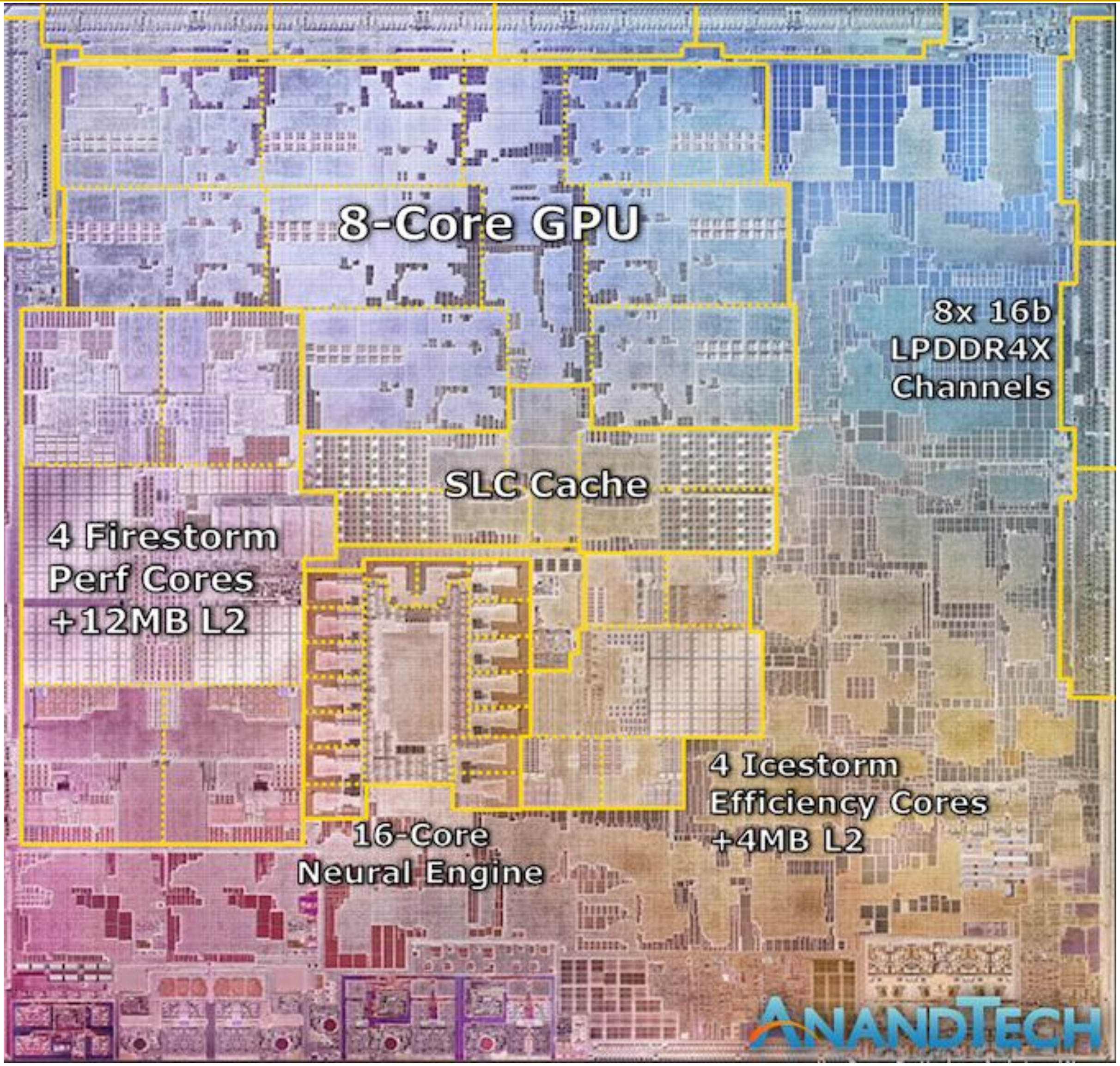
- **Finite State Machines (FSM)**
  - How to design FSMs

Readings

- Combinational Logic
  - P&P Chapter 3 until 3.3 + H&H Chapter 2
- Sequential Logic
  - P&P Chapter 3.4 until end + H&H Chapter 3 in full
- Hardware Description Languages and Verilog
  - H&H Chapter 4 in full
- Timing and Verification
  - H&H Chapters 2.9 and 3.5 + (start Chapter 5)

- By the end of next week, make sure you are done with
  - **P&P Chapters 1-3 + H&H Chapters 1-4**

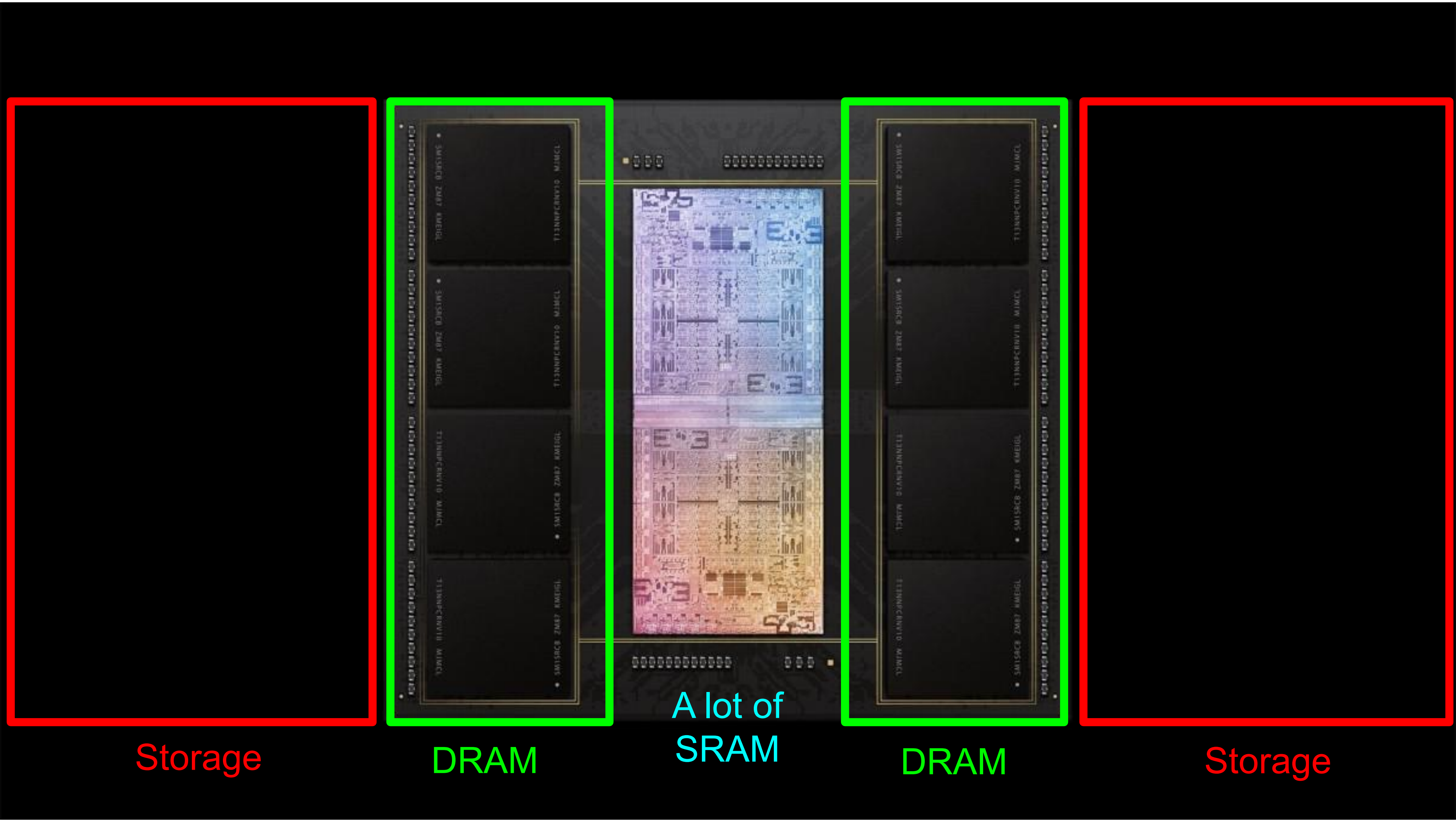
No Real Computer Can Work w/o Memory



Apple M1, 2021

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested

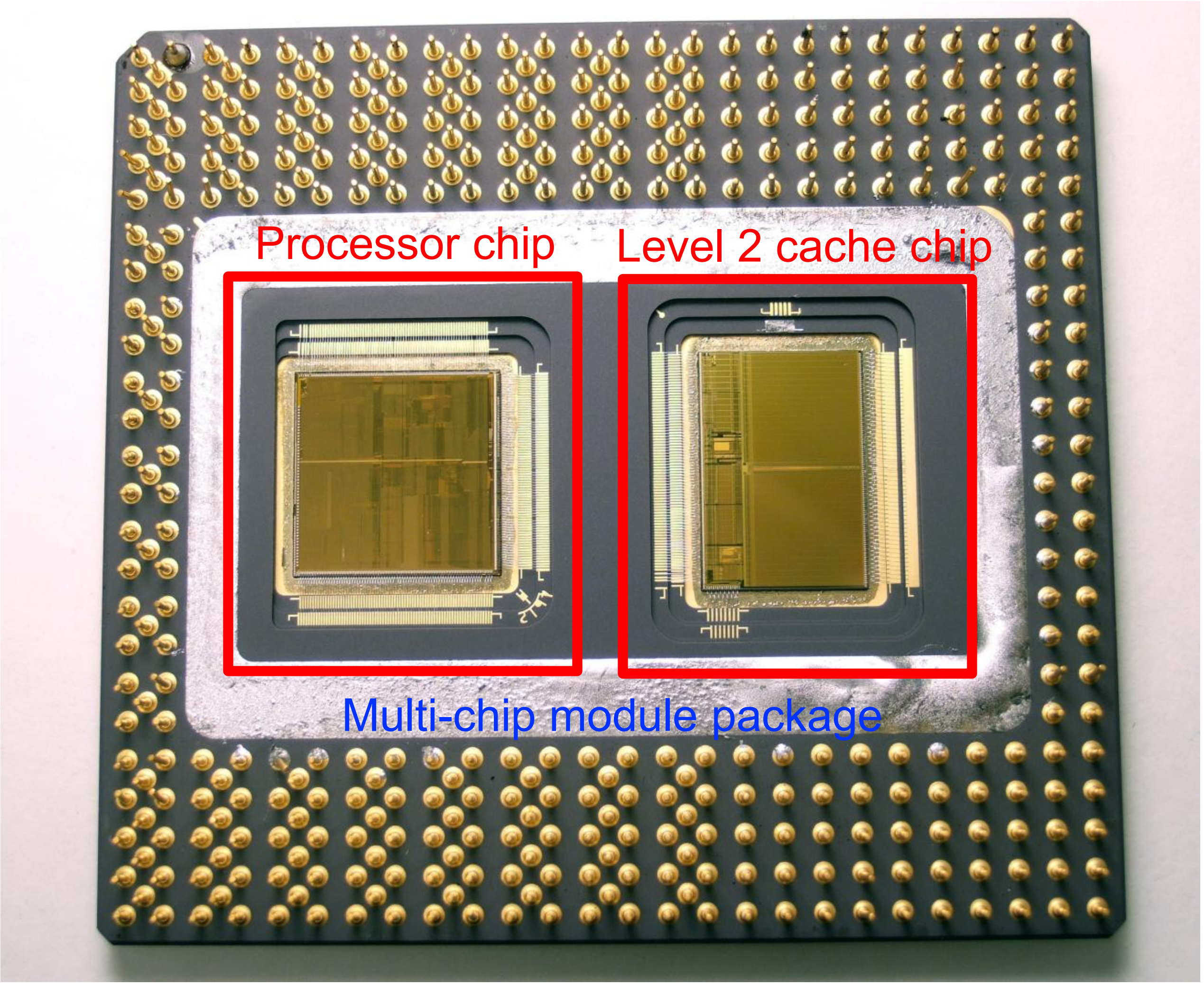
A Large Fraction of Modern Systems is Memory



Apple M1 Ultra System (2022)

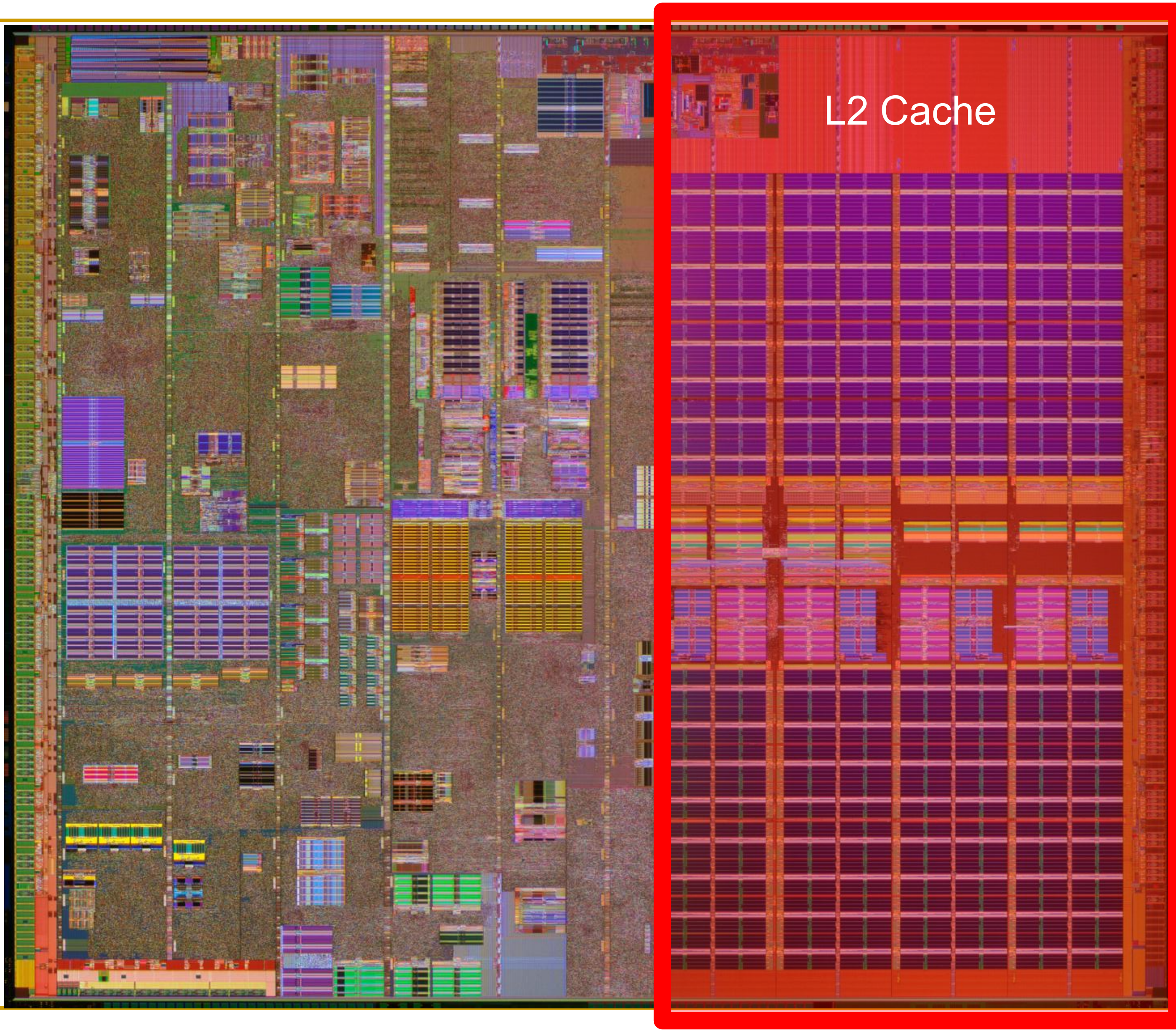
https://www.gsmarena.com/apple_announces_m1_ultra_with_20core_cpu_and_64core_gpu-news-53481.php

A Large Fraction of Modern Systems is Memory



Intel Pentium Pro, 1995

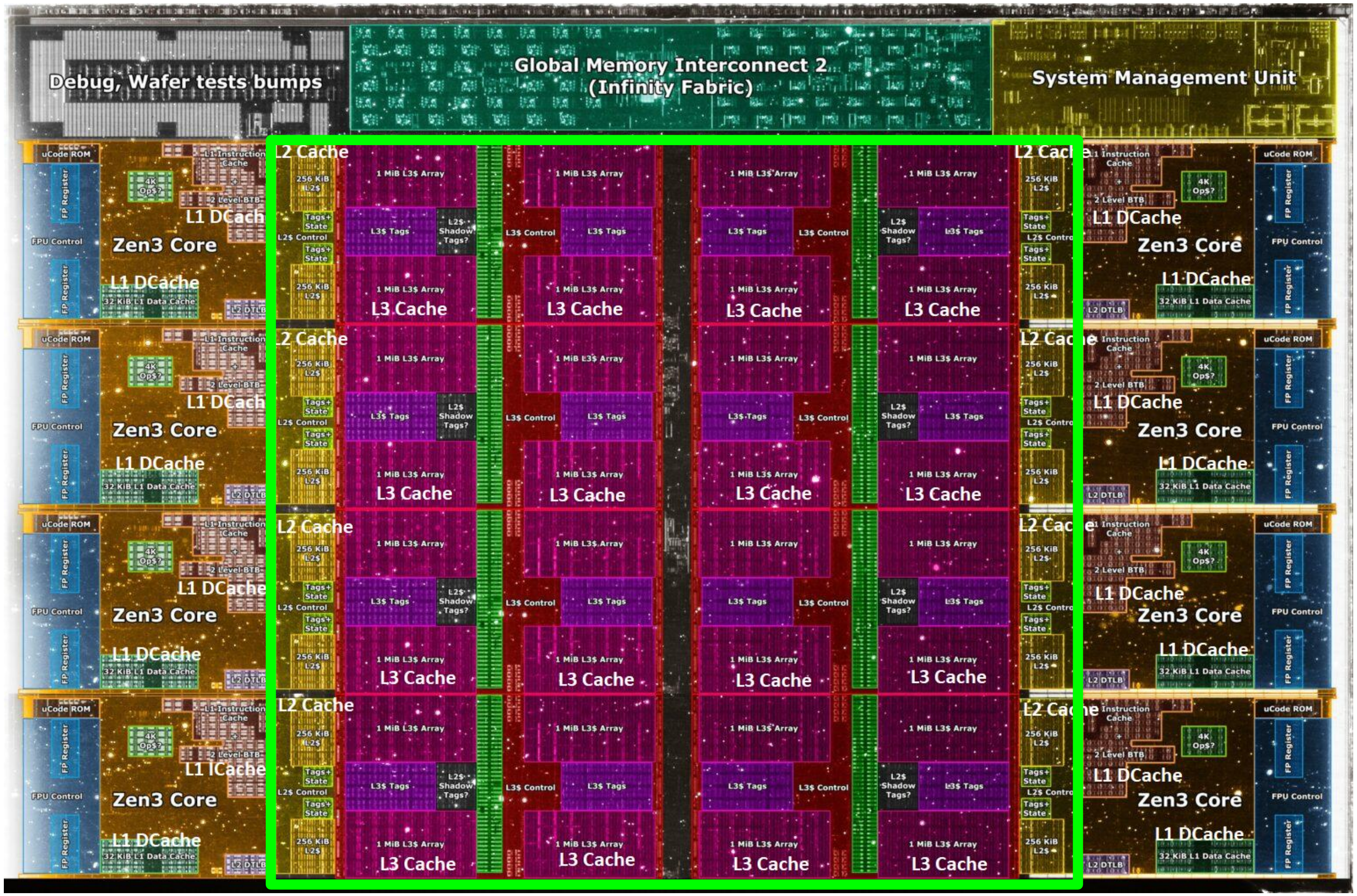
A Large Fraction of Modern Systems is Memory



https://download.intel.com/newsroom/kits/40thanniversary/gallery/images/Pentium_4_6xx-die.jpg

Intel Pentium 4, 2000

A Large Fraction of Modern Systems is Memory



AMD Ryzen 5000, 2020

Core Count:
8 cores/16 threads

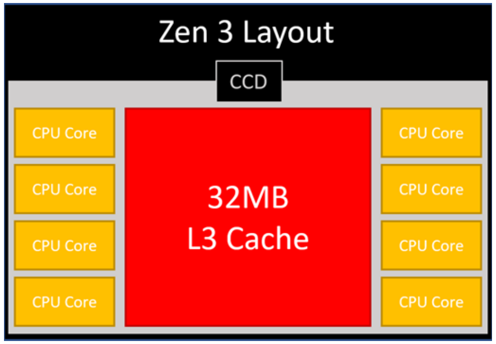
L1 Caches:
32 KB per core

L2 Caches:
512 KB per core

L3 Cache:
32 MB shared

https://wccftech.com/amd-ryzen-5000-zen-3-vermeer-undressed-high-res-die-shots-close-ups-pictured-detailed/

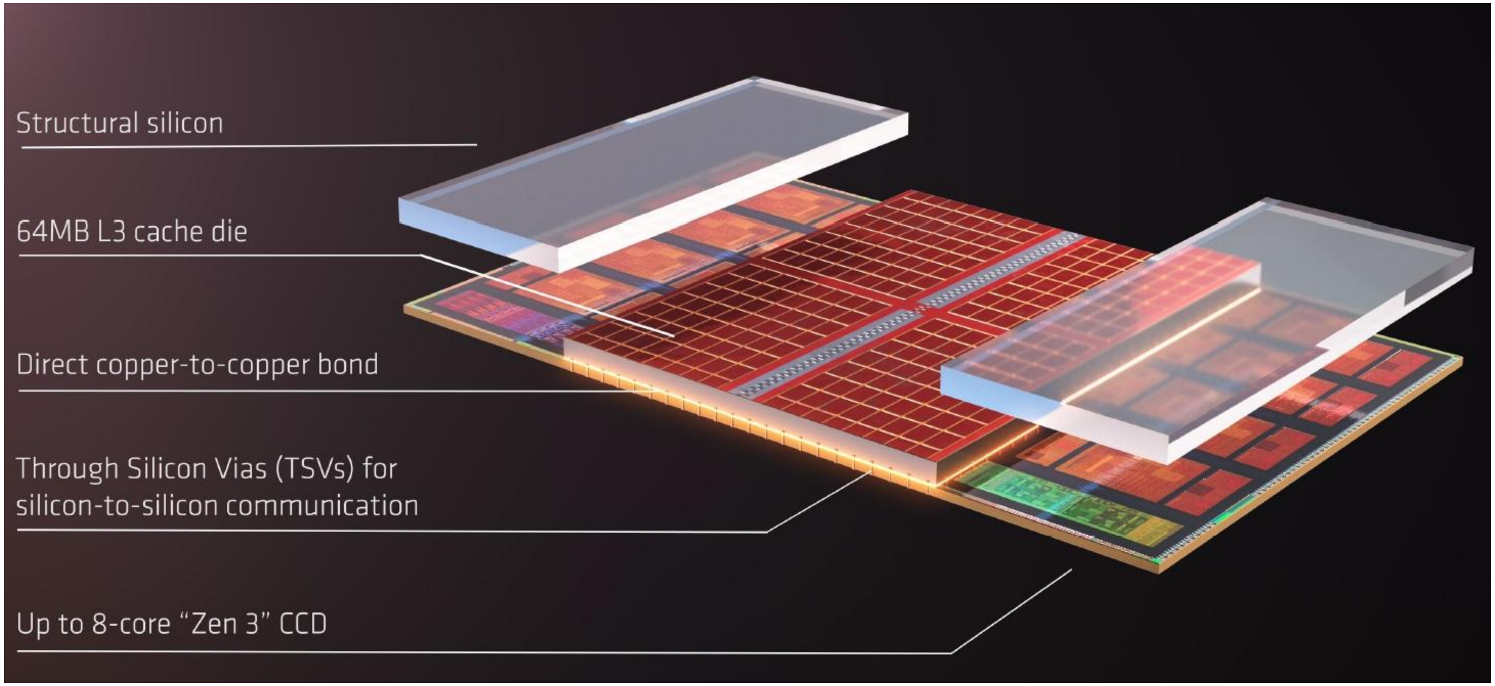
Adding Even More Memory in 3D (2021)



https://community.microcenter.com/discussion/5134/comparing-zen-3-to-zen-2

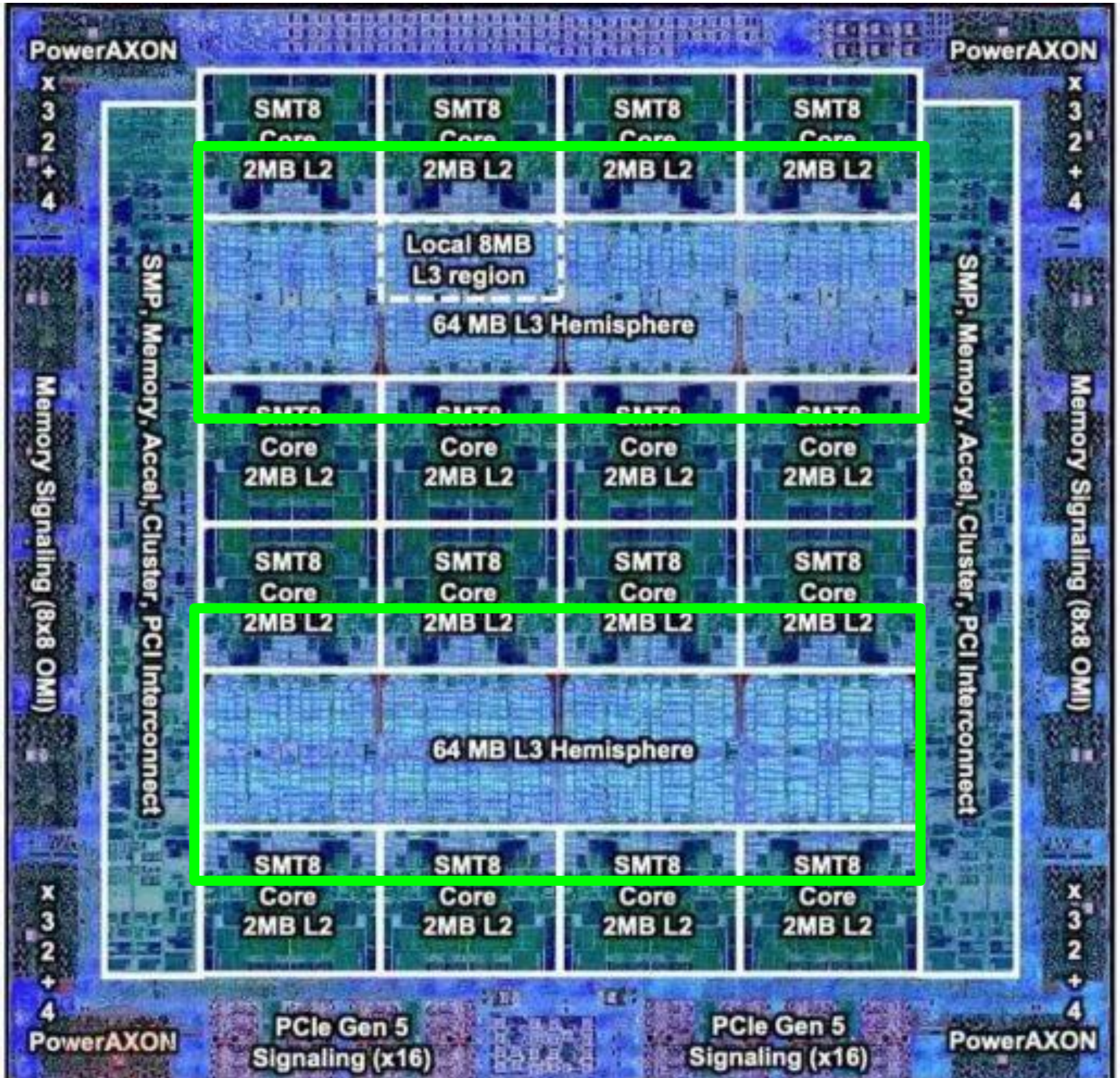
AMD increases the L3 size of their 8-core Zen 3 processors from 32 MB to 96 MB

Additional 64 MB L3 cache die stacked on top of the processor die
- Connected using Through Silicon Vias (TSVs)
- Total of 96 MB L3 cache



https://youtu.be/gqAYMx34euU
https://www.tech-critter.com/amd-keynote-computex-2021/

A Large Fraction of Modern Systems is Memory



IBM POWER10, 2020

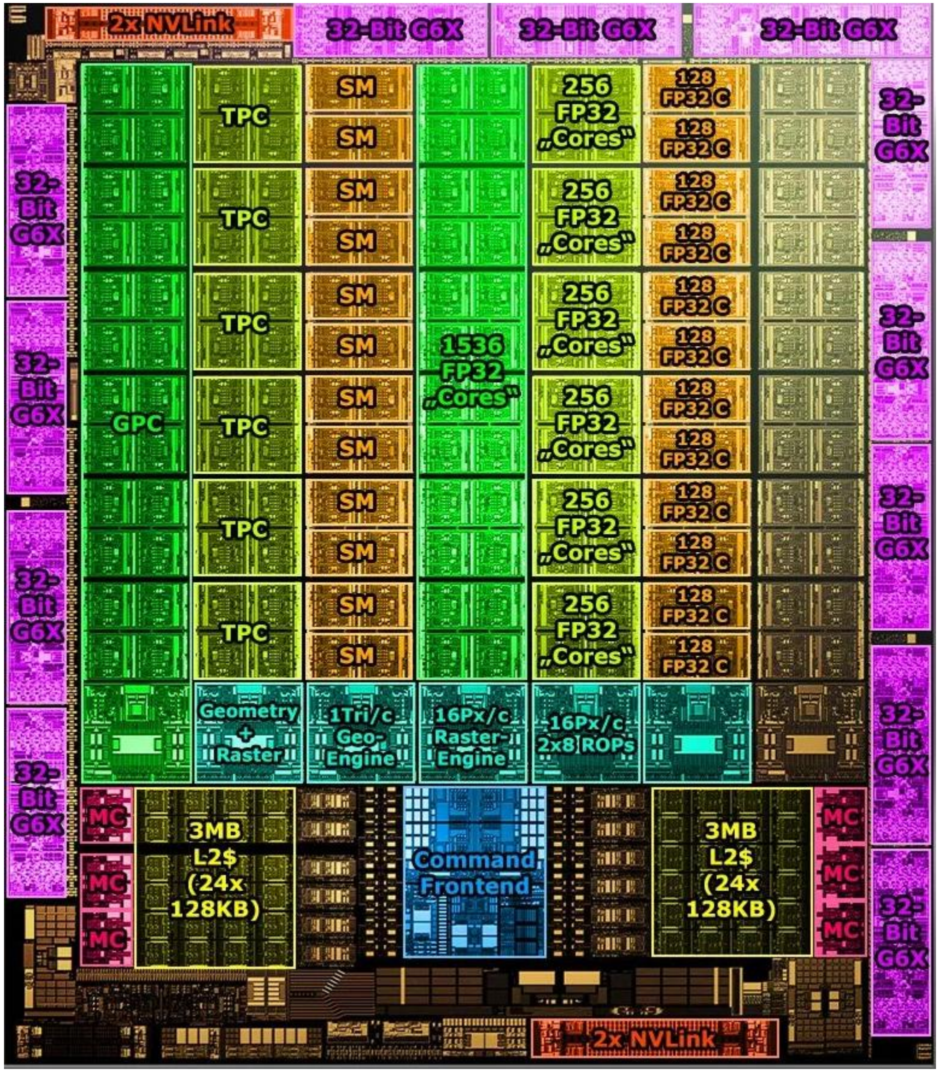
Cores:
15-16 cores,
8 threads/core

L2 Caches:
2 MB per core

L3 Cache:
120 MB shared

https://www.it-techblog.de/ibm-power10-prozessor-mehr-speicher-mehr-tempo-mehr-sicherheit/09/2020/

A Large Fraction of Modern Systems is Memory



Nvidia Ampere, 2020

Cores:

128 Streaming Multiprocessors

L1 Cache or Scratchpad:

192KB per SM

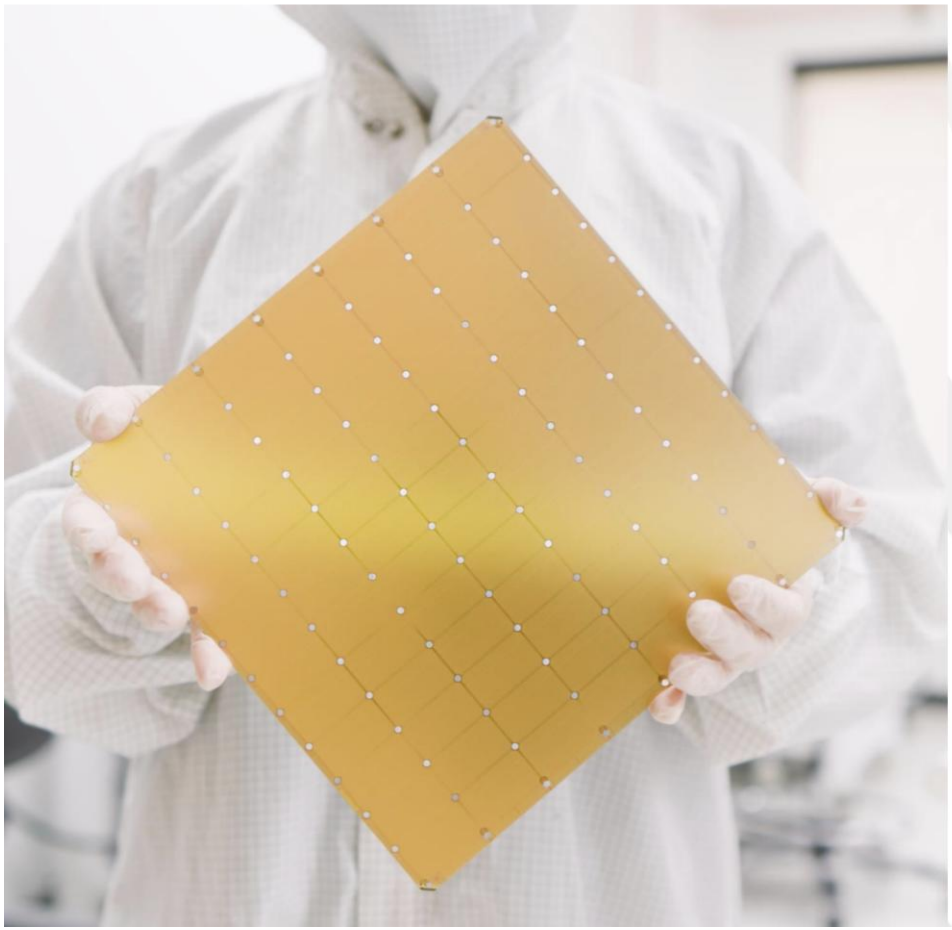
Can be used as L1 Cache and/or Scratchpad

L2 Cache:

40 MB shared

https://www.tomshardware.com/news/infrared-photographer-photos-nvidia-ga102-ampere-silicon

Cerebras's Wafer Scale Engine-3 (2023)



Cerebras Wafer-Scale Engine

The largest chip ever produced

46,225 mm² silicon

4 trillion transistors

900,000 AI cores

125 Petaflops of AI compute

44 Gigabytes of on-chip memory

21 PByte/s memory bandwidth

214 Pbit/s fabric bandwidth

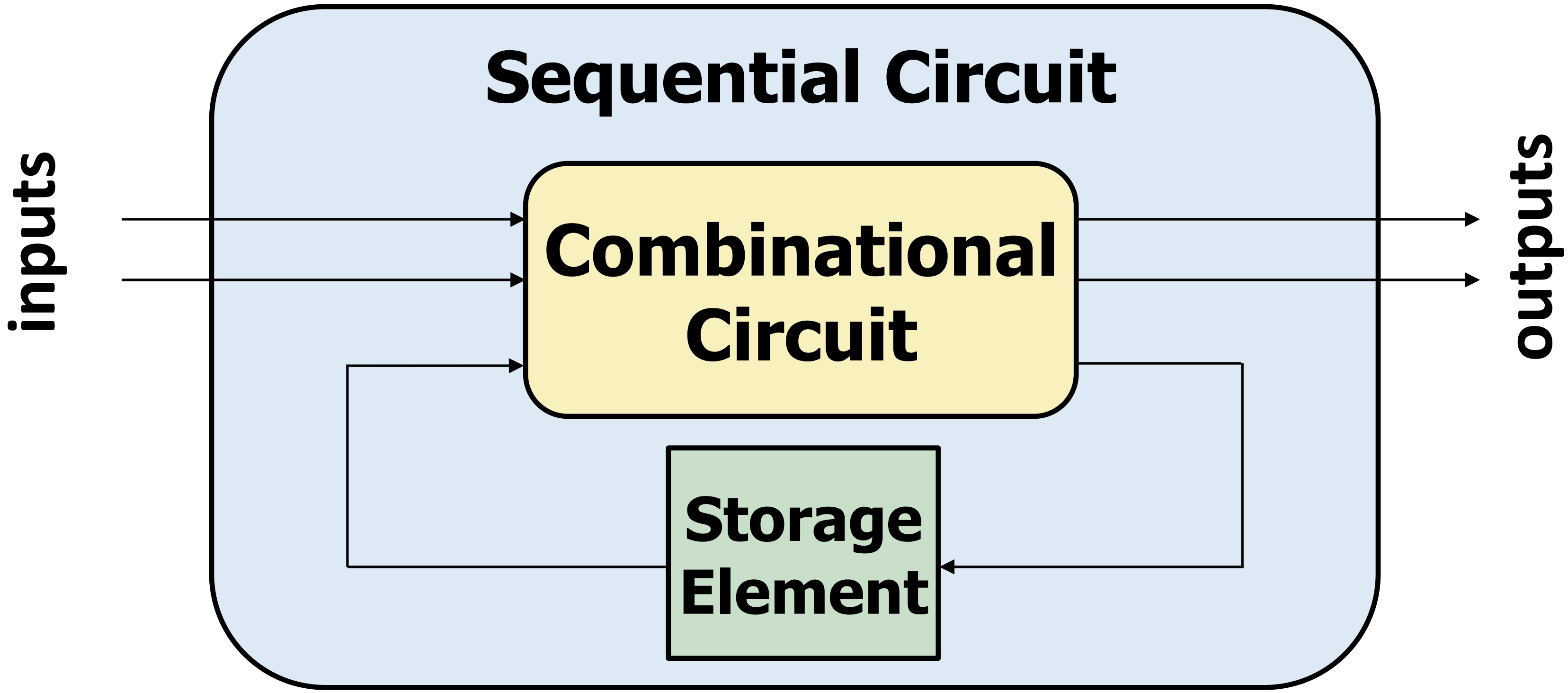
5nm TSMC process

Wafer-Scale AI: Enabling Unprecedented AI Compute Performance, Hot Chips 2024

Circuits That Can Store Information

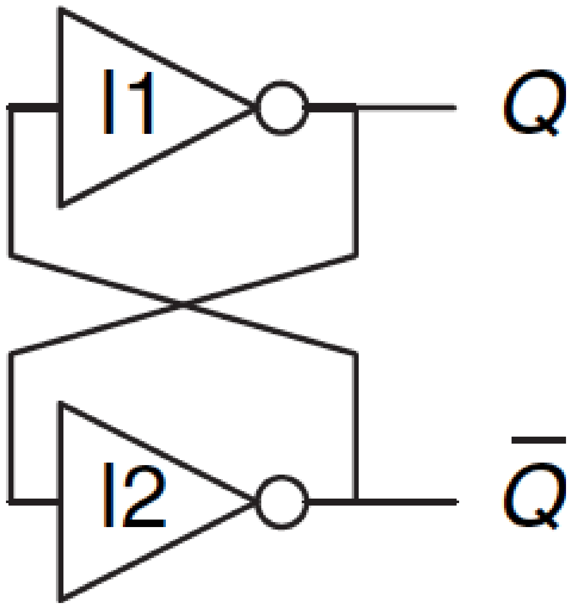
Introduction

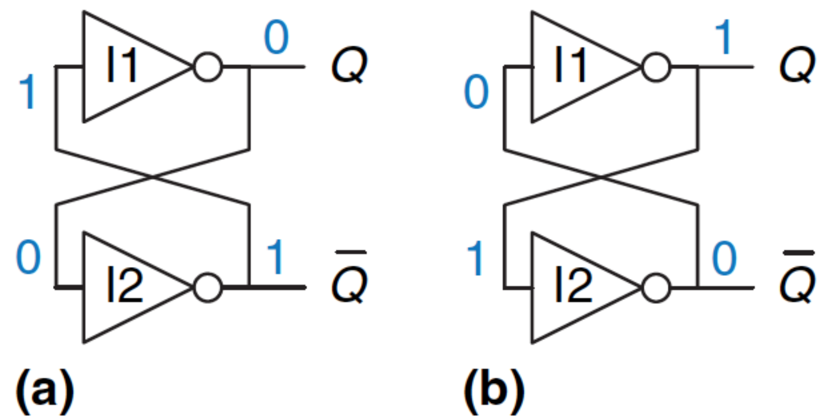
- Combinational circuit output depends **only** on current input
- We want circuits that produce output depending on **current** and **past** input values – circuits with **memory**
- How can we design a circuit that **stores information**?



Capturing Data

Basic Element: Cross-Coupled Inverters



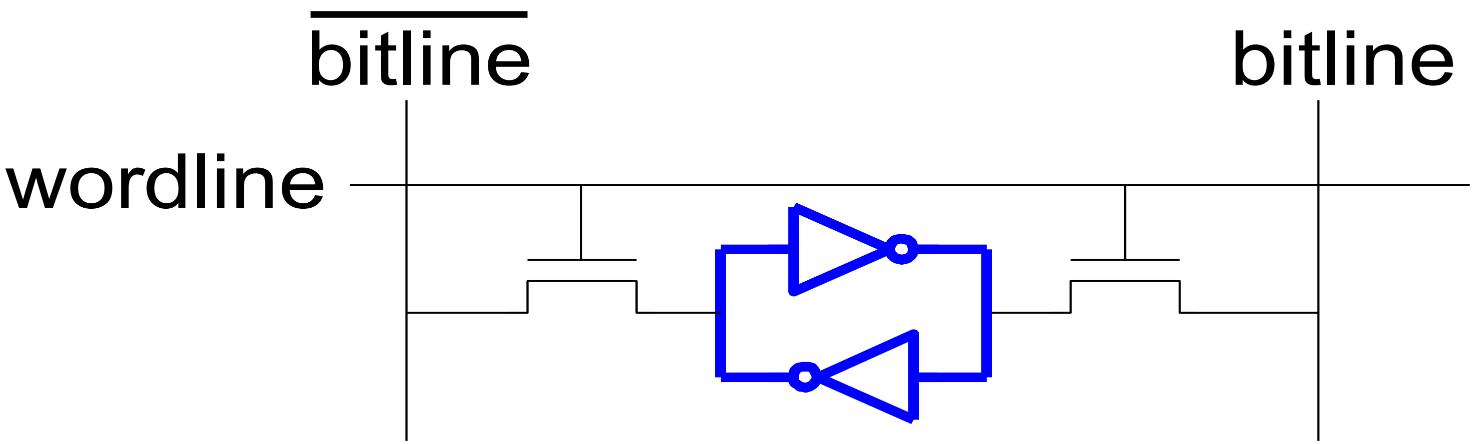


- Has two stable states: Q=1 or Q=0.
- Has a third possible "metastable" state with both outputs oscillating between 0 and 1 (we will see this later)
- Not useful without a *control mechanism* for setting Q

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

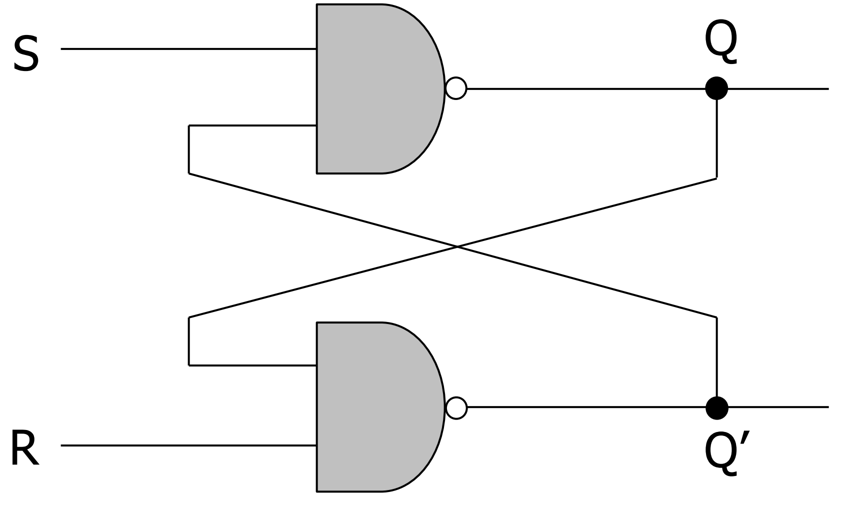
The Big Picture: Storage Elements

- Latches and Flip-Flops
  - Very fast, parallel access
  - Very expensive (one bit costs tens of transistors)
- Static RAM (SRAM)
  - Relatively fast
  - Expensive (one bit costs 6+ transistors)
- Dynamic RAM (DRAM)
  - Slower, 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

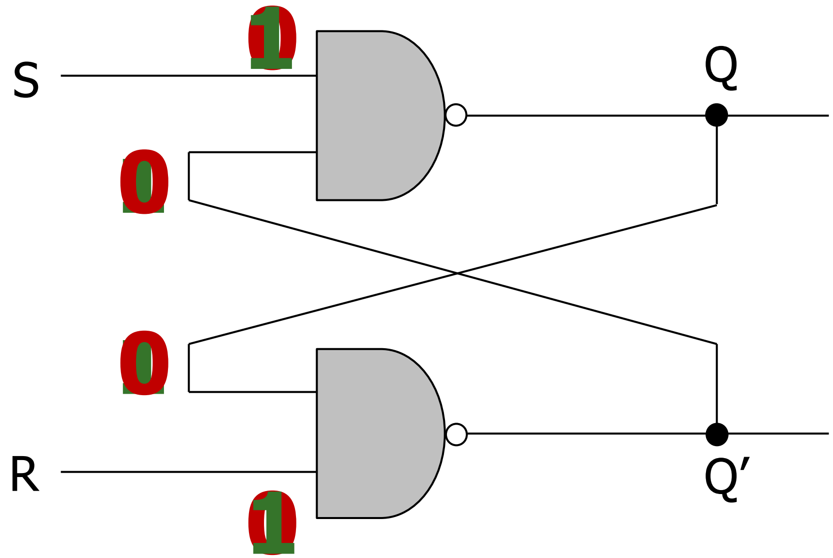
The R-S (Reset-Set) Latch

- 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 not **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?



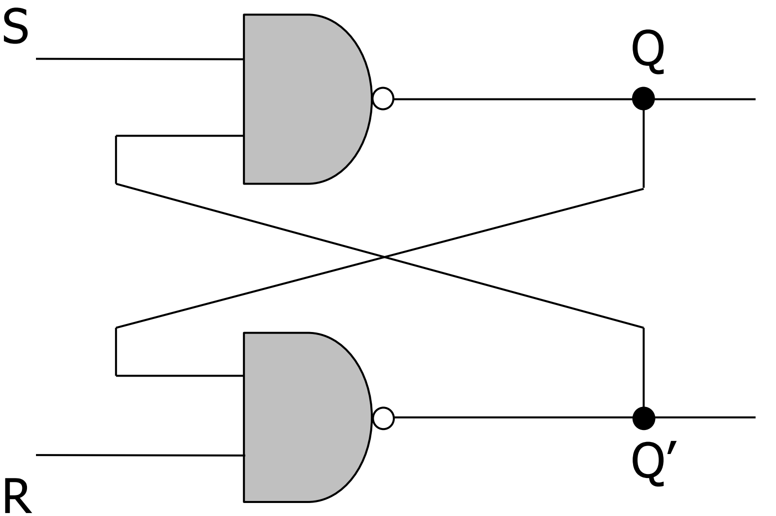
Input		Output
R	S	Q
1	1	Q_{prev}
1	0	1
0	1	0
0	0	Forbidden

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** (more on this in the **Timing Lecture**)

Gated D Latch

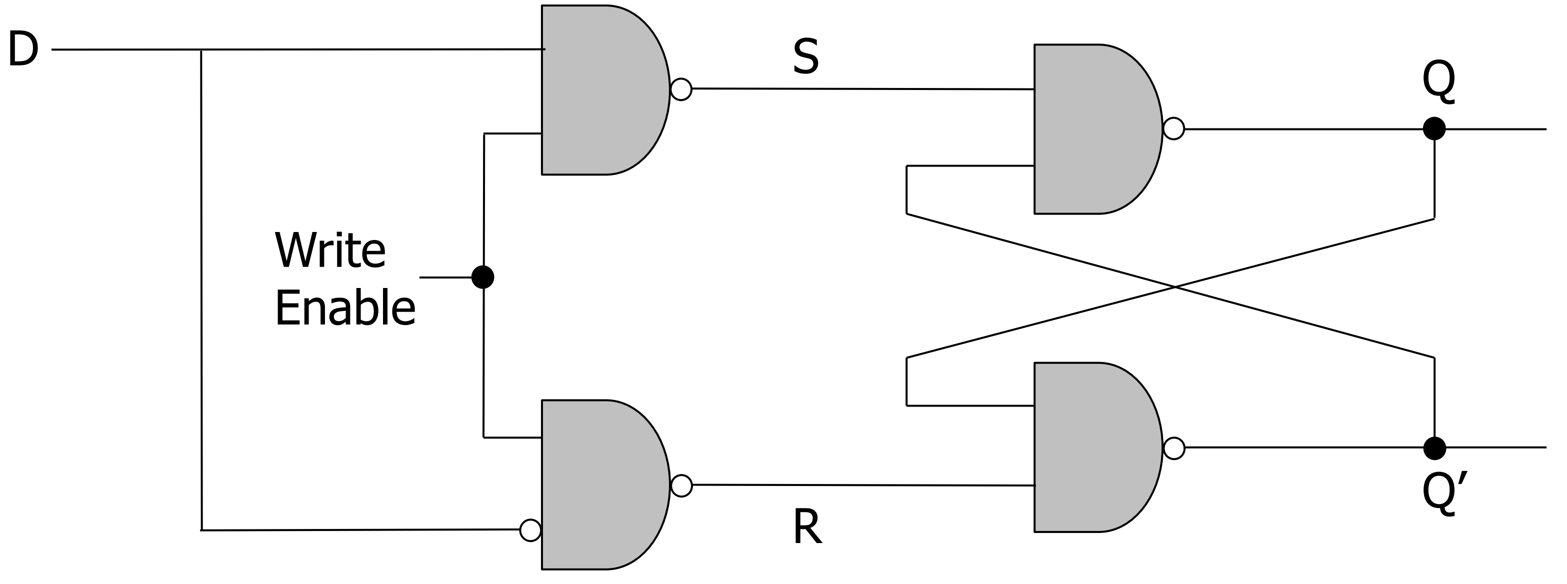
The Gated D Latch

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



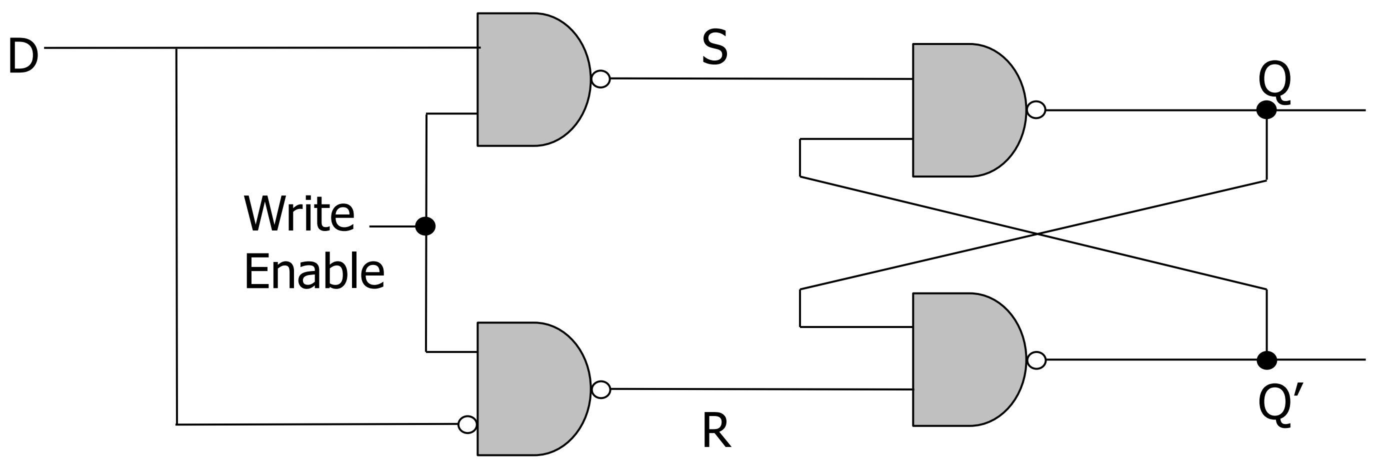
The Gated D Latch

- How do we **guarantee** correct operation of an R-S Latch?
  - Add two more NAND gates!



  - **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 Gated D Latch



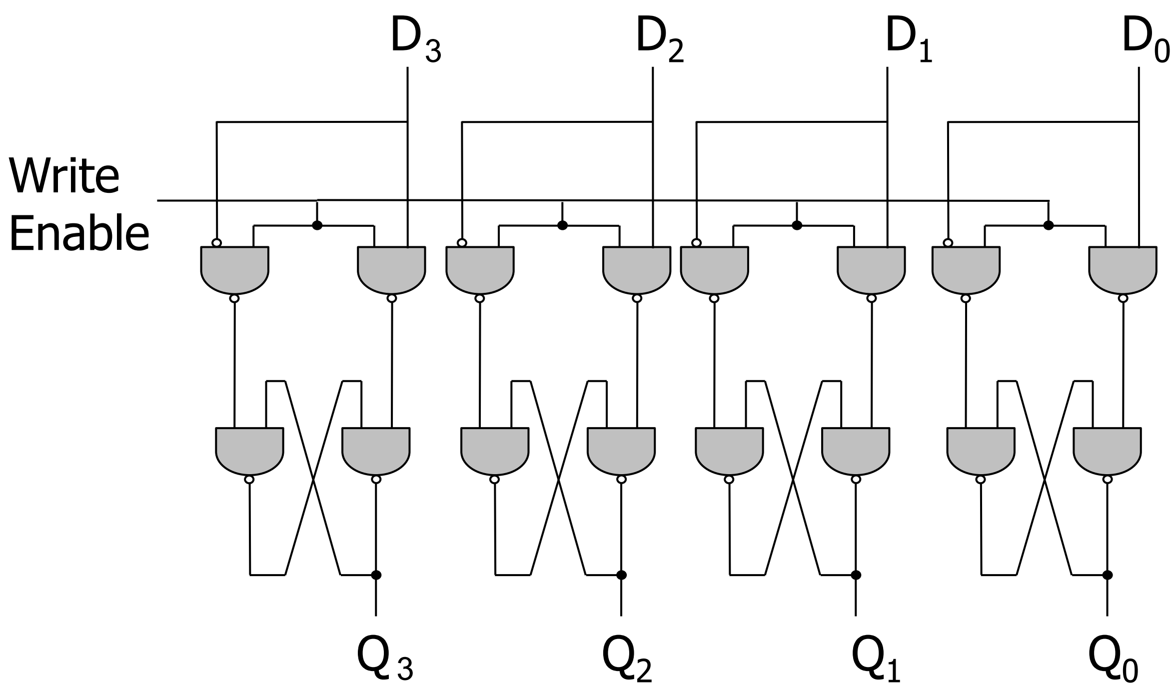
Input		Output
WE	D	Q
0	0	Q_{prev}
0	1	Q_{prev}
1	0	0
1	1	1

Register

The Register

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

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



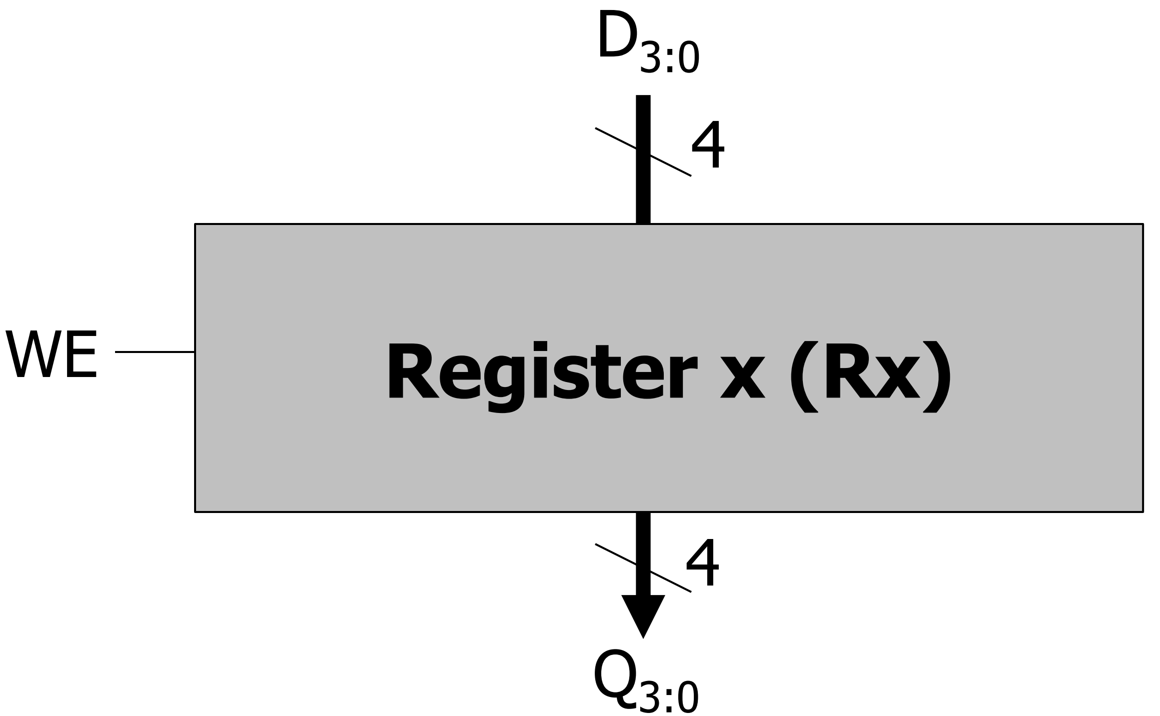
Here we have a **register,** or a structure that stores more than one bit and can be read from and written to

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 WE 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]

Memory

Memory

- **Memory** is comprised of locations that can be written to or read from. An example memory array with 4 locations:

Addr(00):	0100 1001	**Addr**(01):	0100 1011
Addr(10):	0010 0010	**Addr**(11):	1100 1001

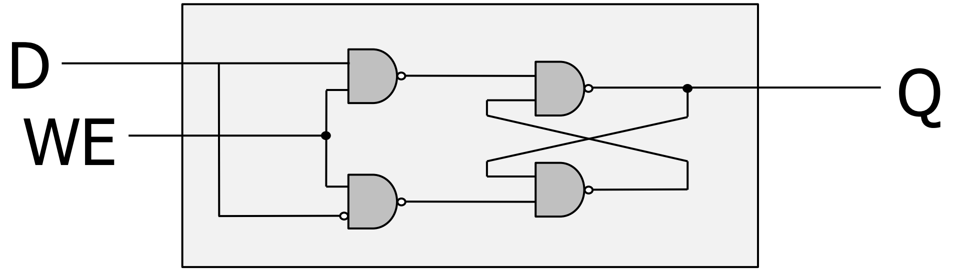
- Every unique location in memory is indexed with a unique **address.** 4 locations require 2 address bits (log[#locations]).
- **Addressability:** the number of **bits** of information stored **in each location**. This example: addressability is 8 bits.
- The entire set of **unique locations** in memory is referred to as the **address space.**
- Typical memory is **MUCH** larger (e.g., billions of locations)

Addressing Memory

Let's implement a simple memory array with:

- 3-bit addressability **&** address space size of 2 (total of 6 bits)

1 Bit



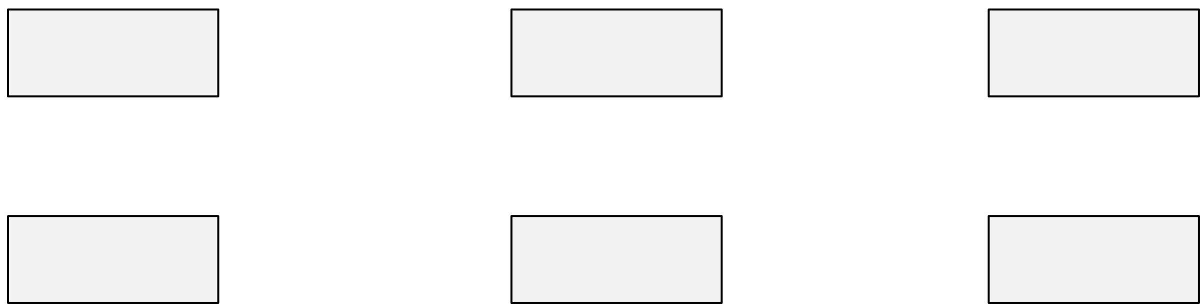
6-Bit Memory Array

Addr(0)	Bit_2	Bit_1	Bit_0
Addr(1)	Bit_2	Bit_1	Bit_0

Reading from Memory

How can we select an address to read?

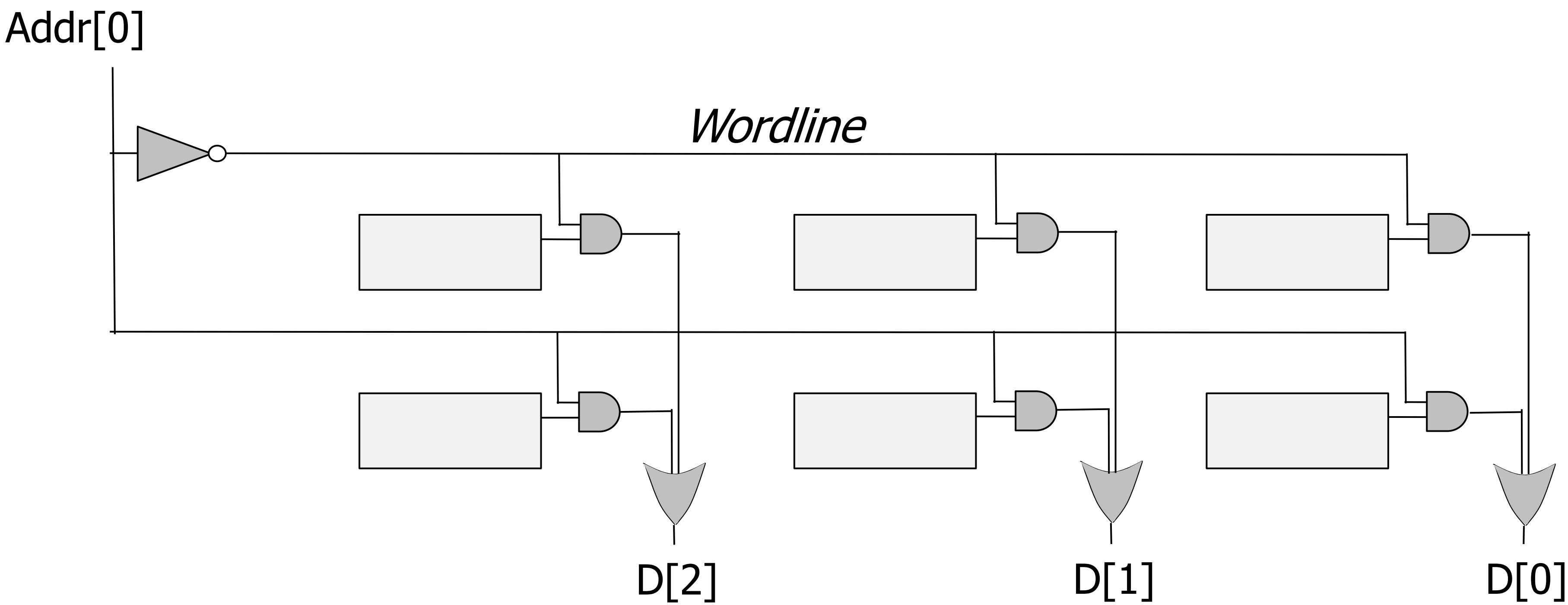
- Because there are 2 addresses, address size is log(2)=1 bit



Reading from Memory

How can we select an address to read?

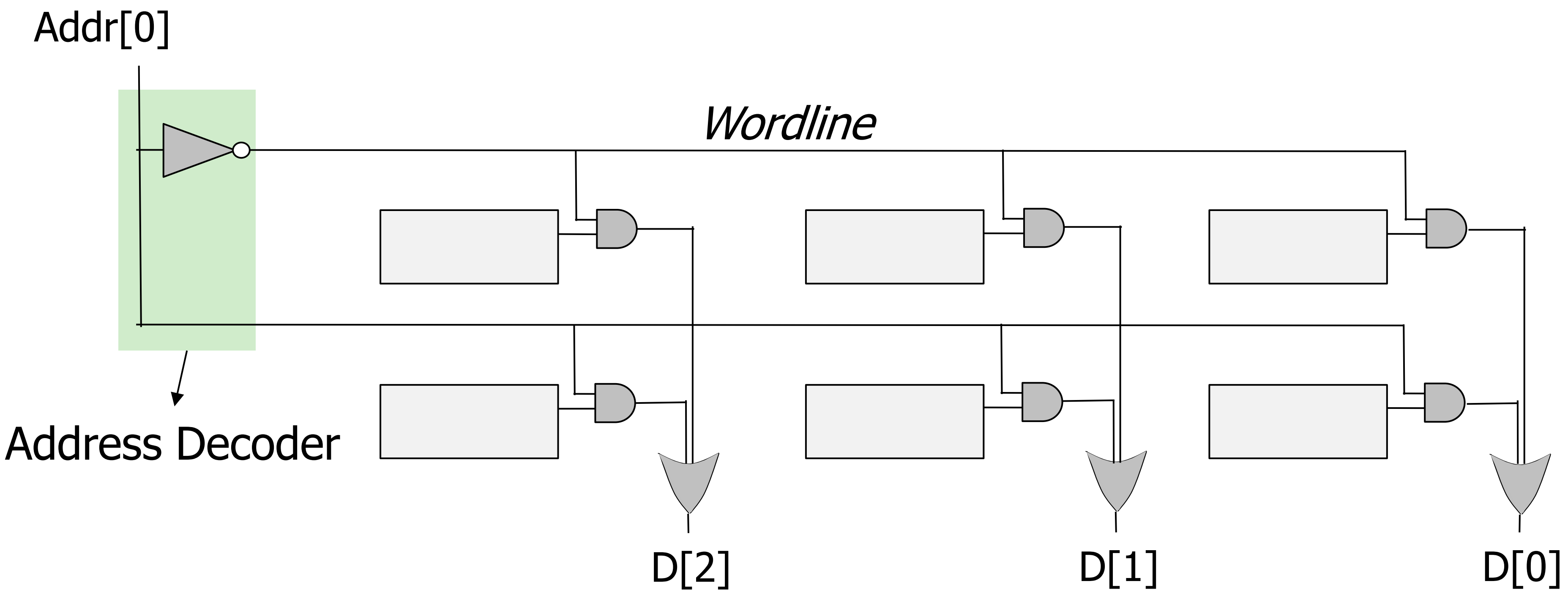
- Because there are 2 addresses, address size is log(2)=1 bit



Reading from Memory

How can we select an address to read?

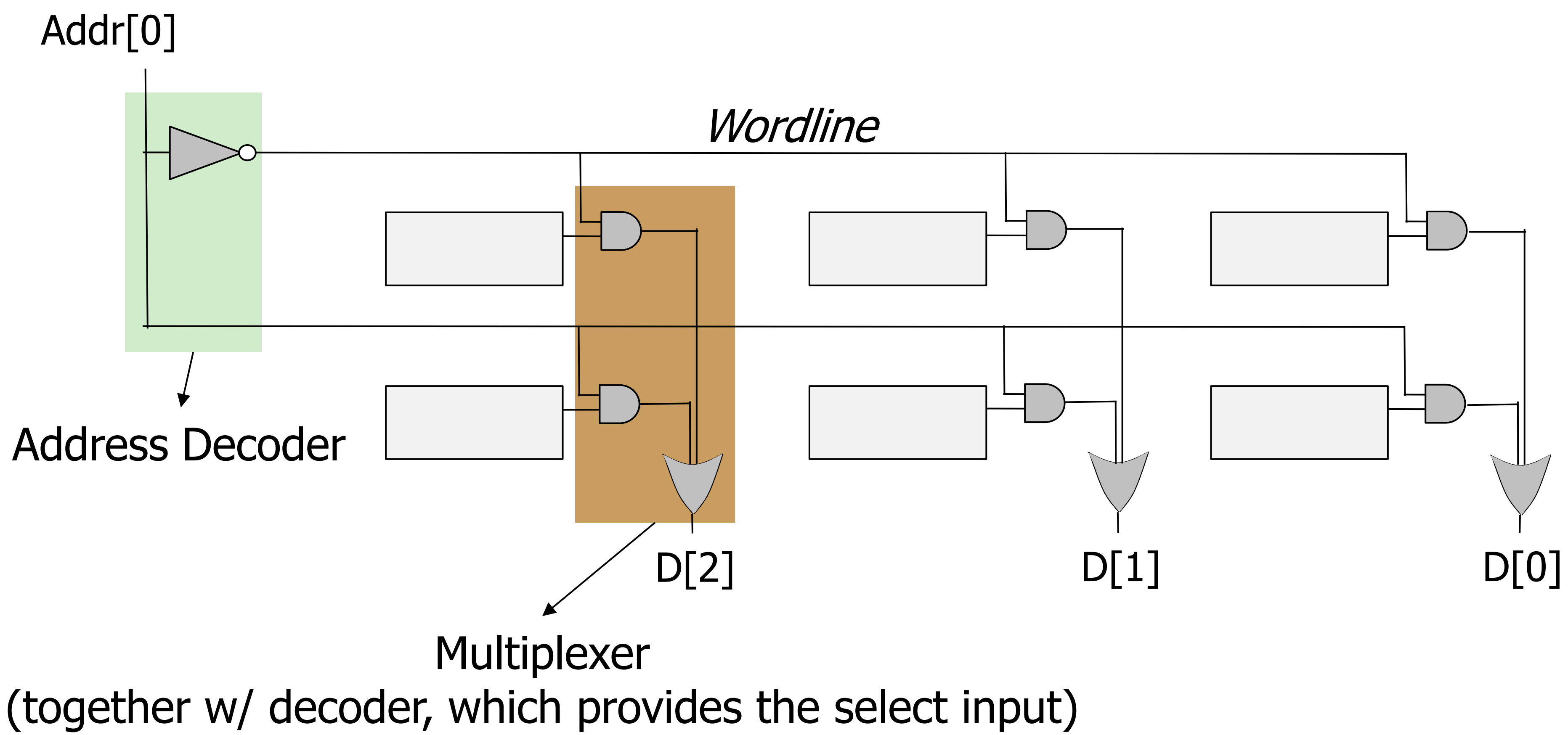
- Because there are 2 addresses, address size is log(2)=1 bit



Reading from Memory

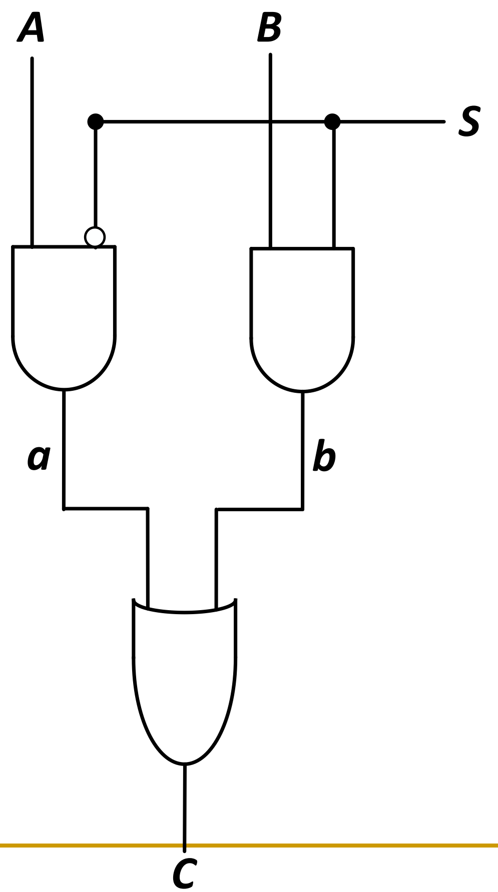
How can we select an address to read?

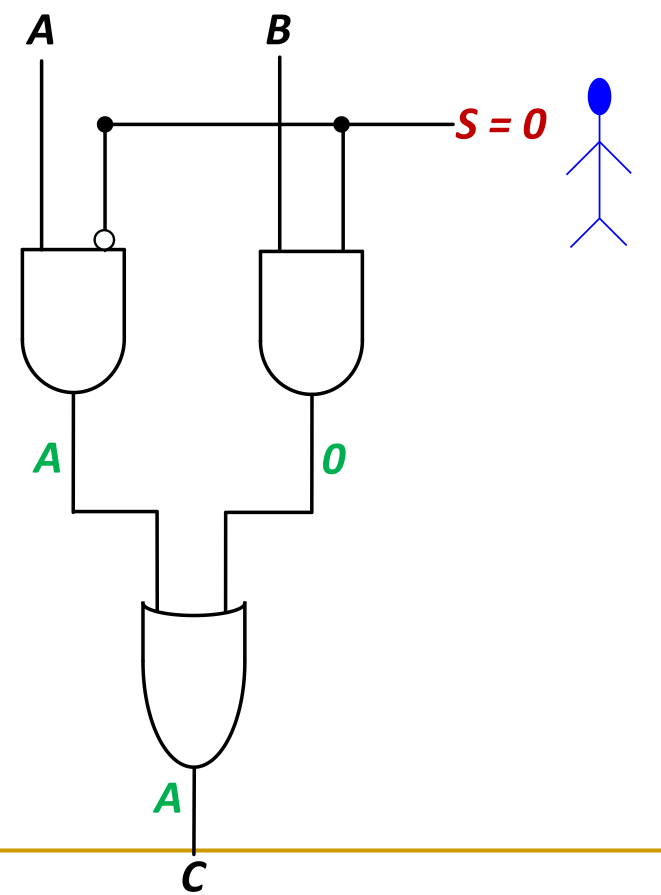
- Because there are 2 addresses, address size is log(2)=1 bit



Recall: Multiplexer (MUX), or Selector

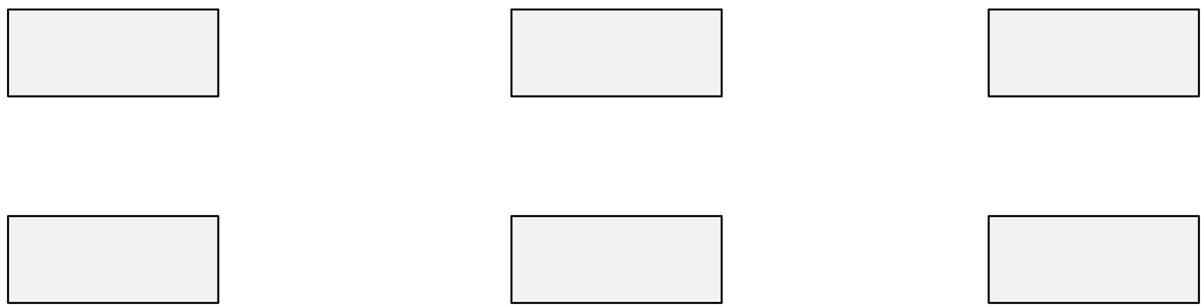
- Selects one of the *N* inputs to connect it to the output
  - based on the value of a $\log_2 N$-bit control input called select
- Example: 2-to-1 MUX





Writing to Memory

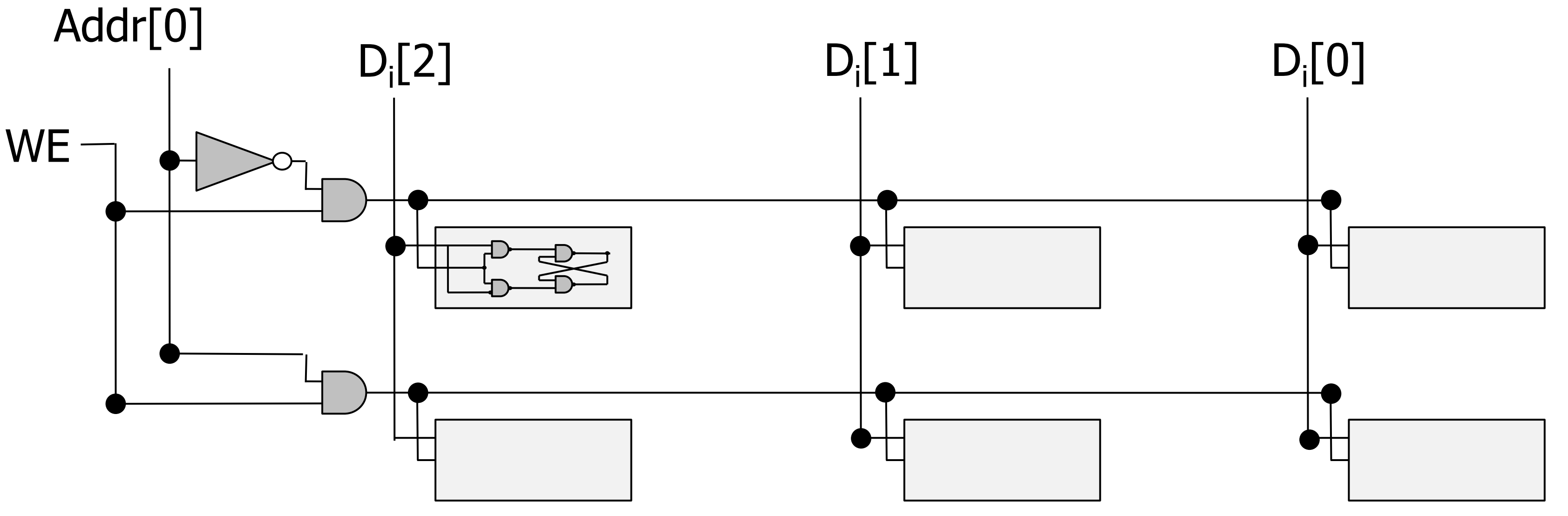
How can we select an address and write to it?



Writing to Memory

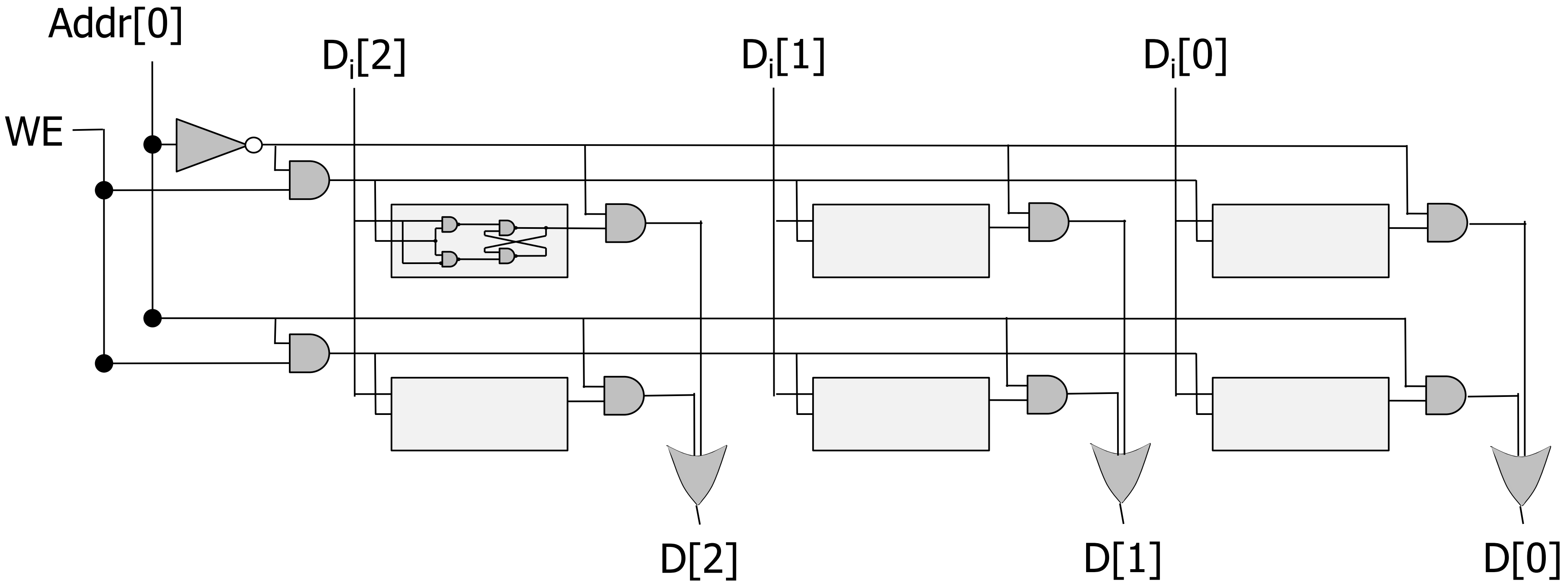
How can we select an address and write to it?

- Input is indicated with D_i

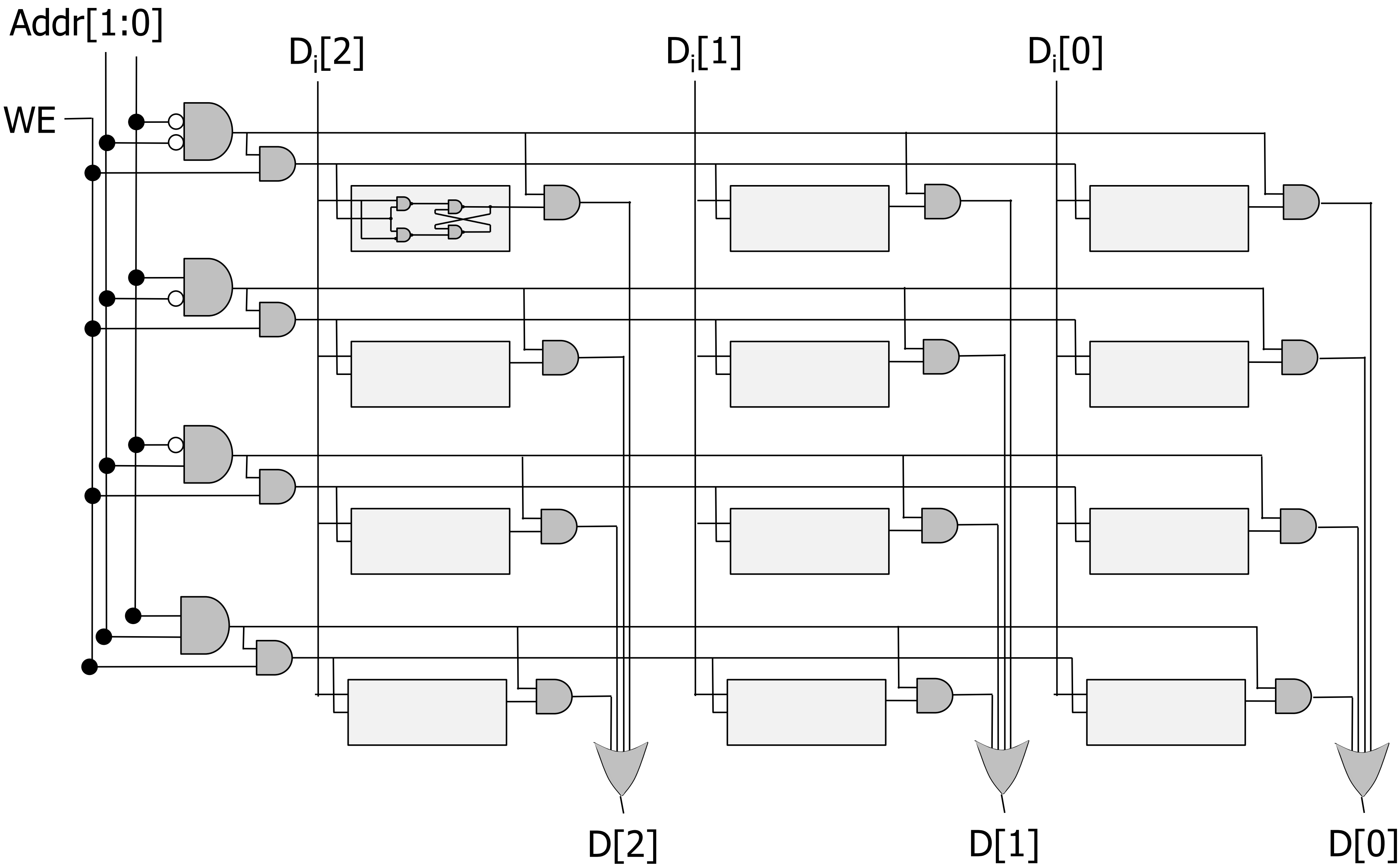


Putting it all Together

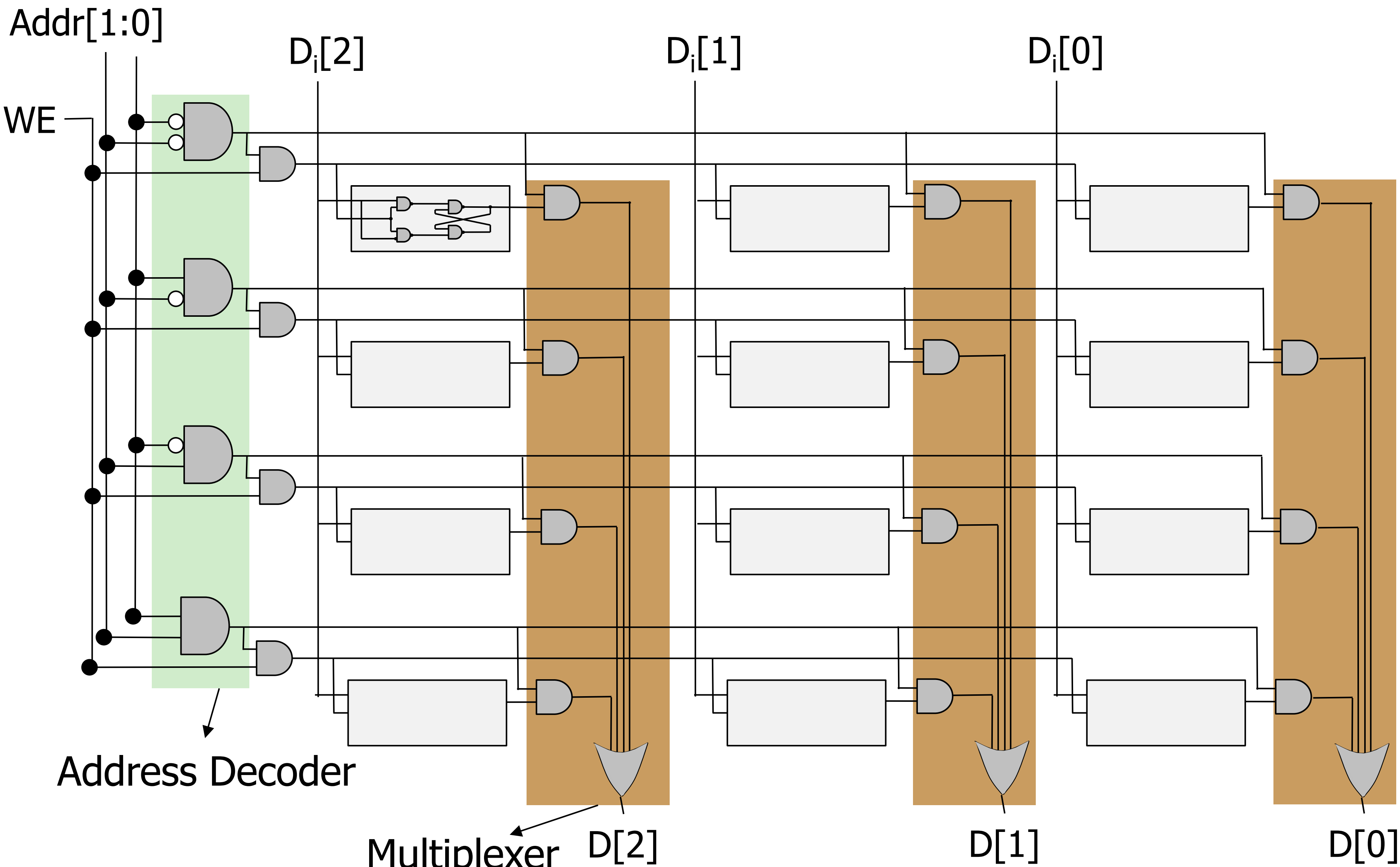
Let's enable reading from and writing to a memory array



A Bigger Memory Array (4 locations X 3 bits)



A Bigger Memory Array (4 locations X 3 bits)



(together w/ decoder, which provides the select input)

Example: Reading Location 3

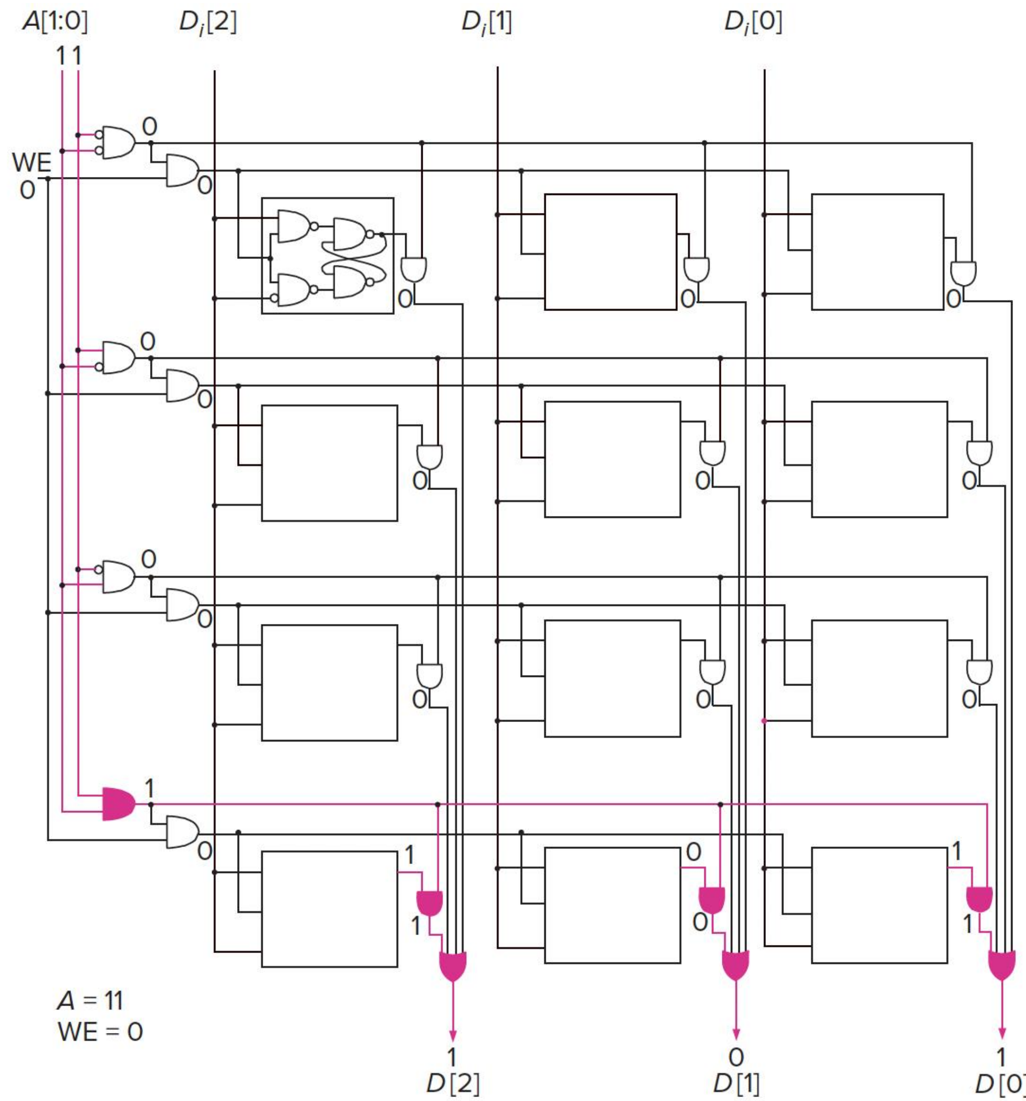
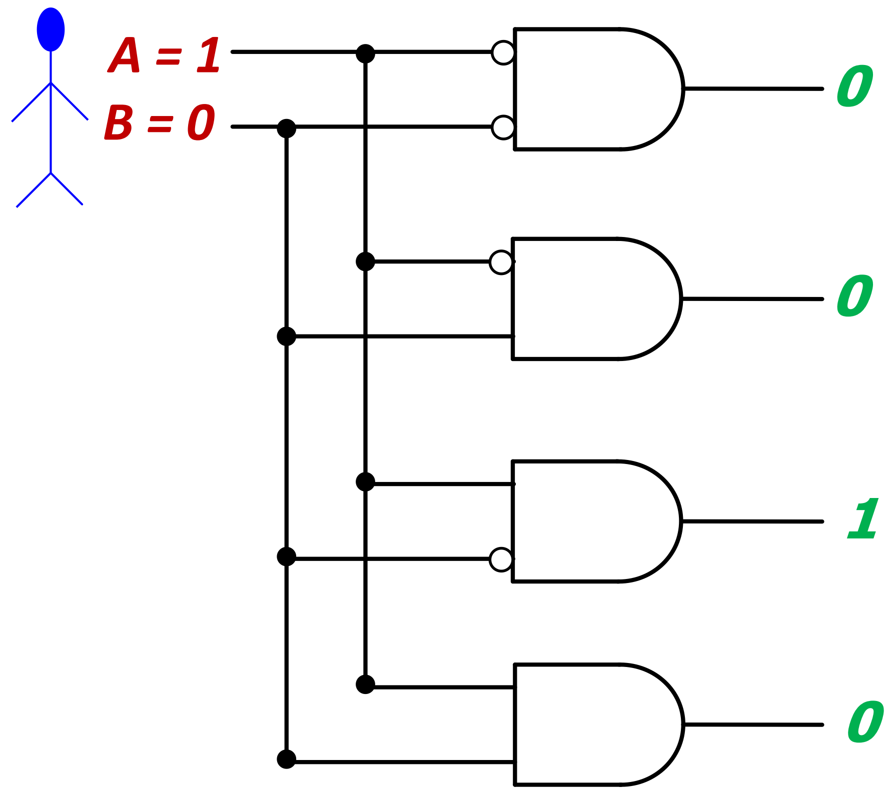


Figure 3.21 Reading location 3 in our 2^2-by-3-bit memory.

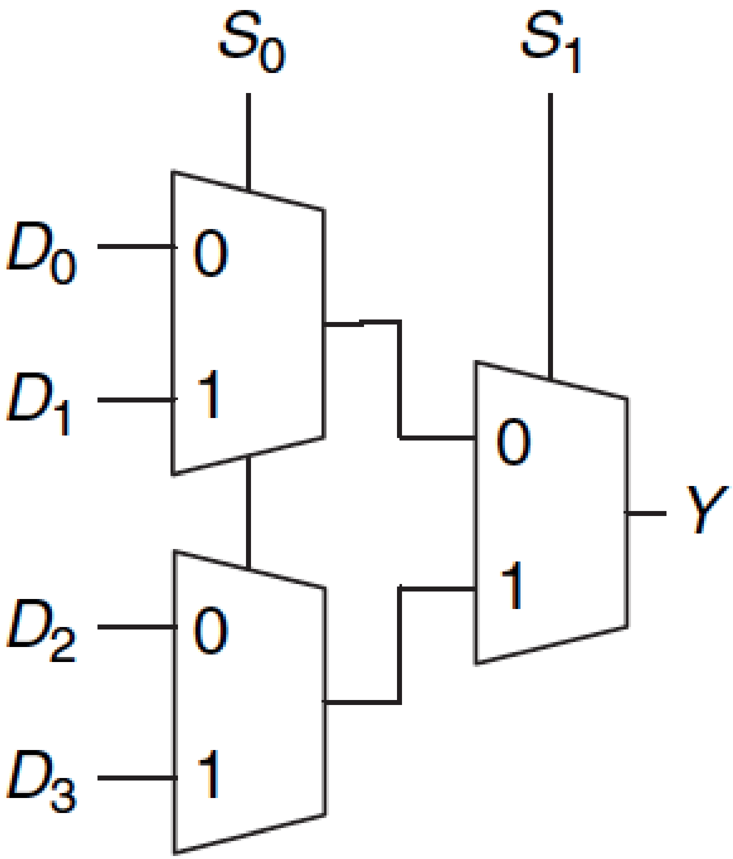
Image source: Patt and Patel, "Introduction to Computing Systems", 3rd ed., page 78.

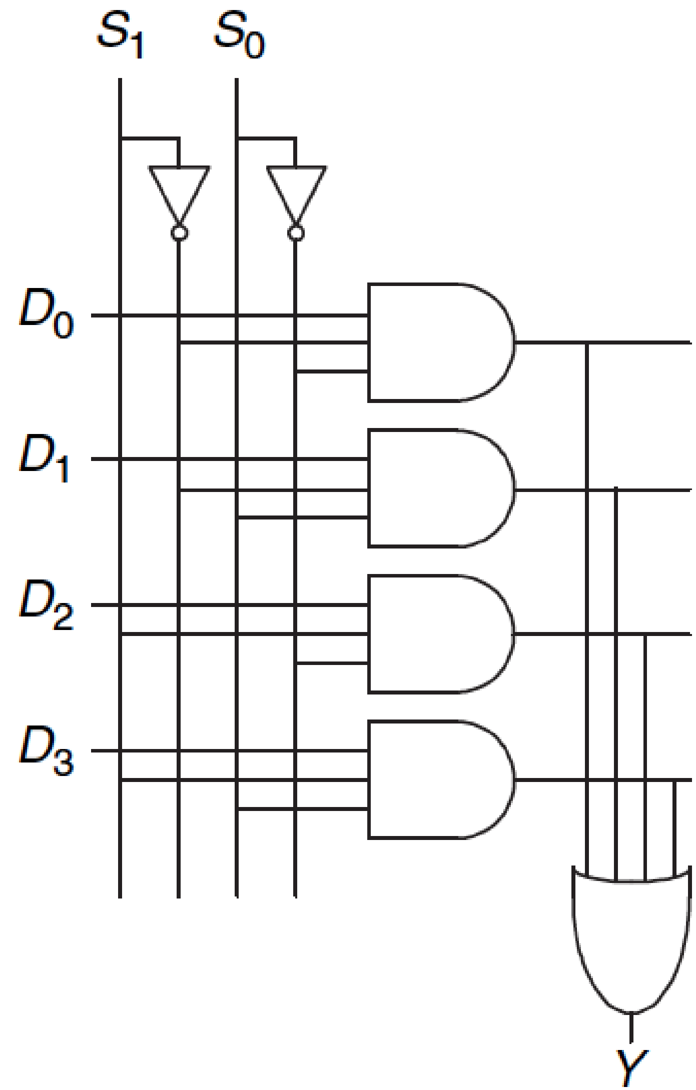
Recall: Decoder (II)

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



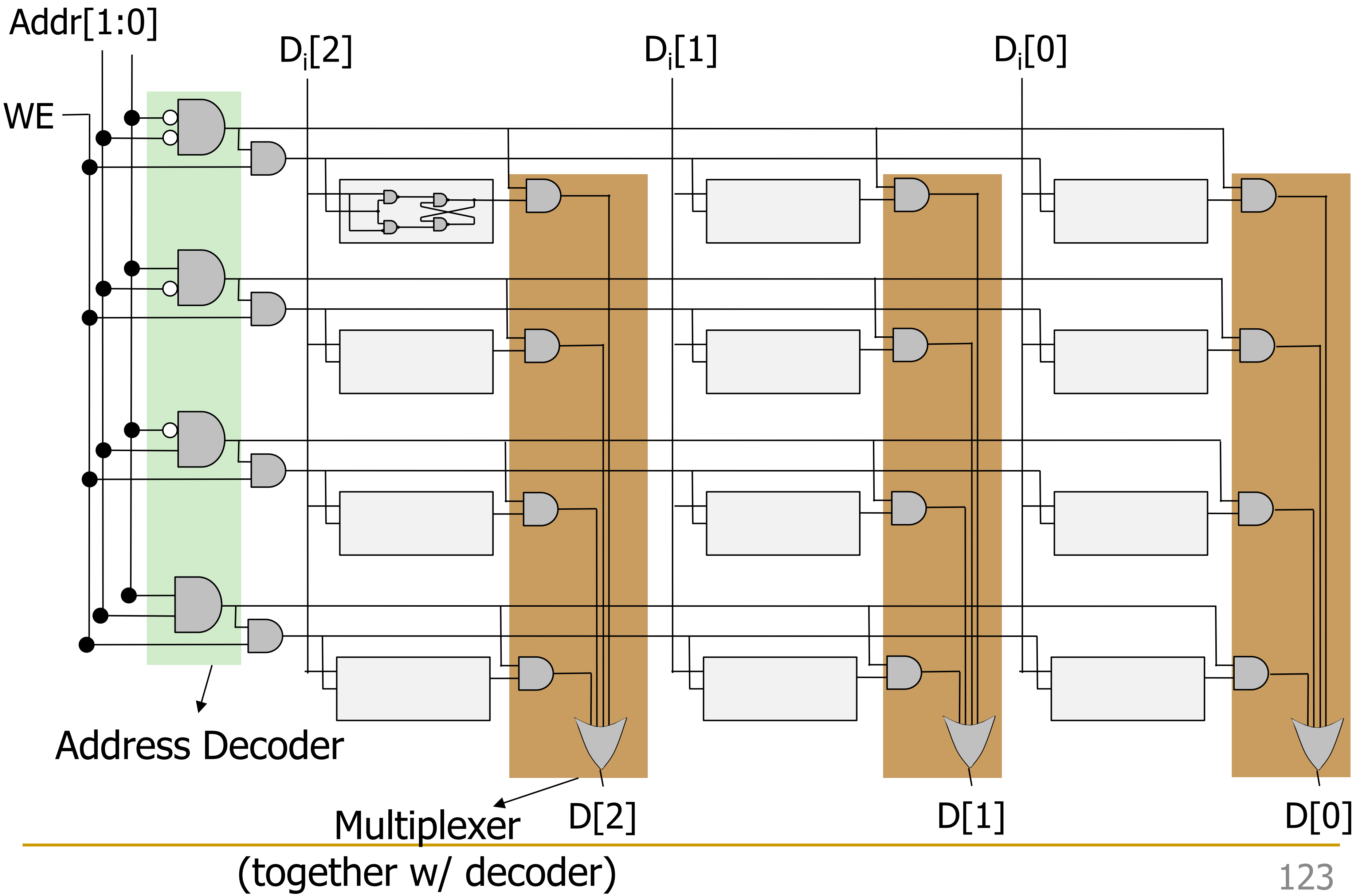
Recall: A 4-to-1 Multiplexer





Aside: Implementing Logic Functions Using Memory

Recall: A Bigger Memory Array (4 locations X 3 bits)



Memory-Based Lookup Table Example

- Memory arrays can also perform Boolean Logic functions
  - 2^N-location M-bit memory can perform any N-input, M-output function
  - Lookup Table (LUT): Memory array used to perform logic functions
  - Each address: row in truth table; each data bit: corresponding output value

Truth Table

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

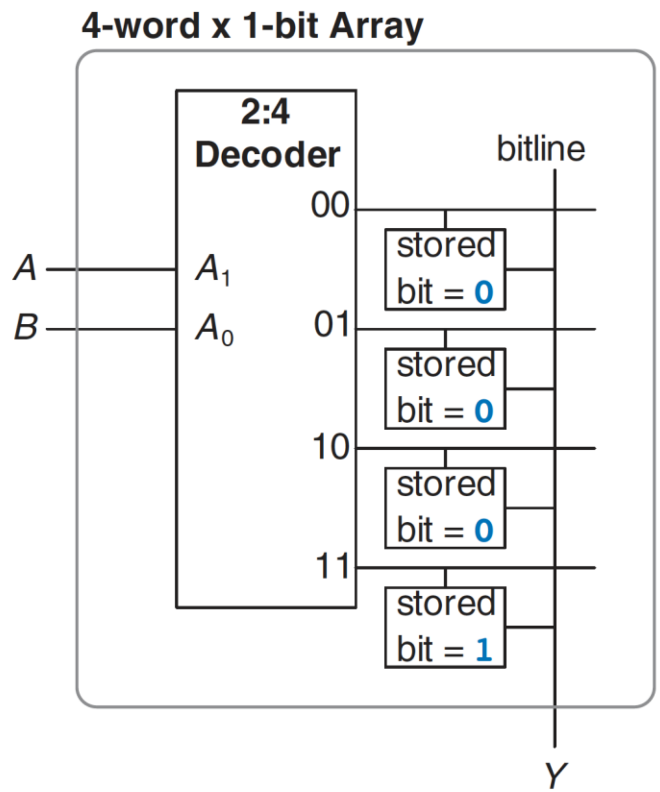


Figure 5.52 4-word × 1-bit memory array used as a lookup table

Aside: Lookup Tables (LUTs) in FPGAs

- LUTs are commonly used in FPGAs
  - To enable programmable/reconfigurable logic functions
  - To enable easy integration of combinational and sequential logic

(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

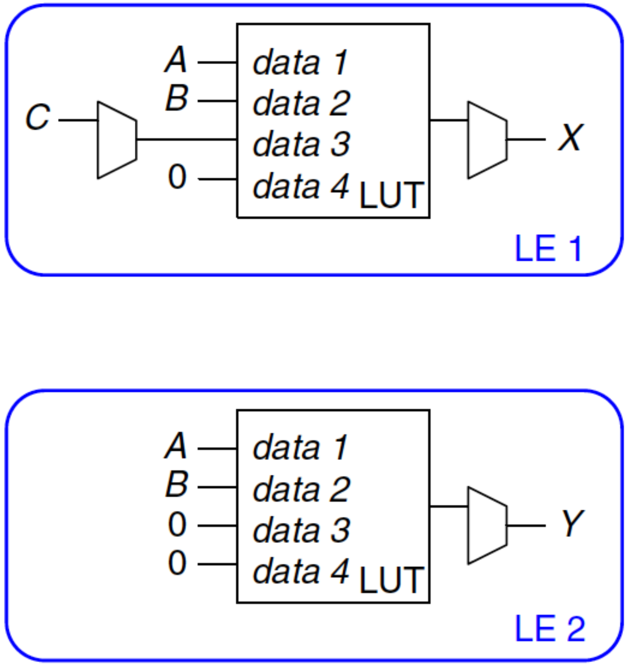


Figure 5.59 LE configuration for two functions of up to four inputs each

Read H&H Chapter 5.6.2

Recall: A Multiplexer-Based LUT

- Let's implement a function that outputs '1' when there are at least two '1's in a 3-bit input

In C:

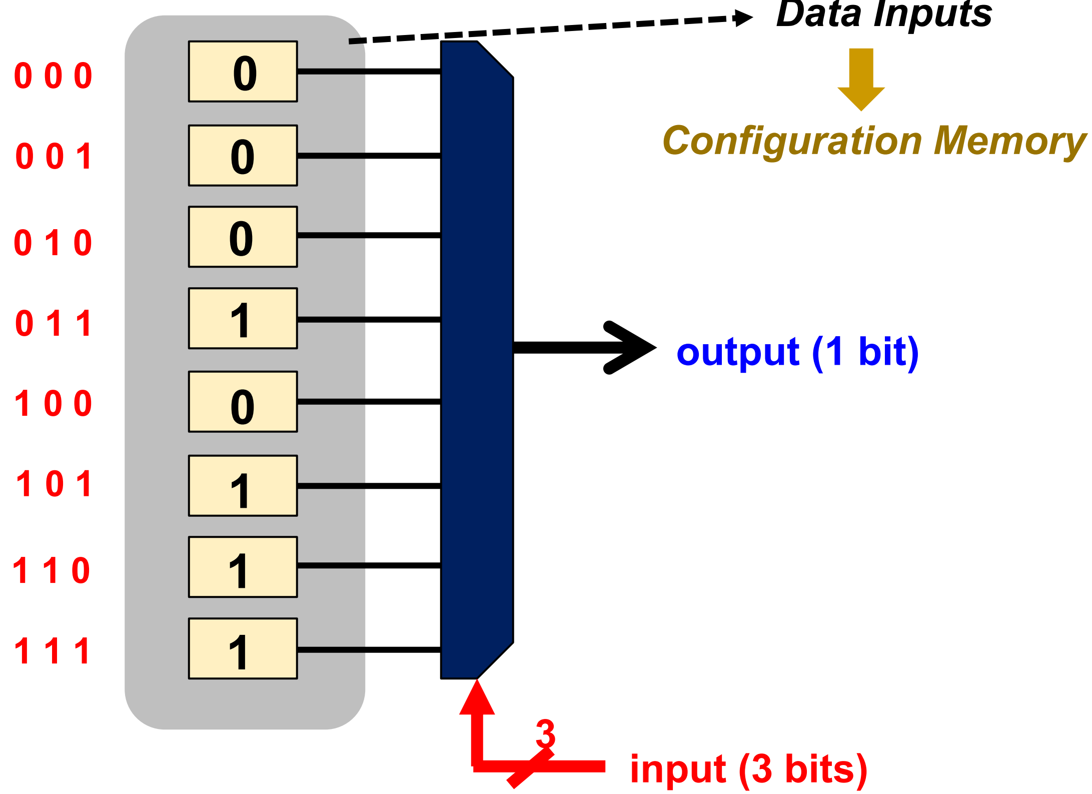
```
int count = 0;
for(int i = 0; i < 3; i++) {
    count += input & 1;
    input = input >> 1;
}

if(count > 1) return 1;

return 0;
```

```
switch(input){
   case 0:
   case 1:
   case 2:
   case 4:
      return 0;
   default:
      return 1;}
```

In Hardware (e.g., FPGA):



Sequential Logic Circuits

Sequential Logic Circuits

- We have examined designs of circuit elements that can **store information**
- Now, we will use these elements to build circuits that **remember** past inputs

Combinational
Only depends on current inputs

Sequential
Opens depending on past inputs

https://www.easykeys.com/228_ESP_Combination_Lock.aspx
https://www.fosmon.com/product/tsa-approved-lock-4-dial-combo

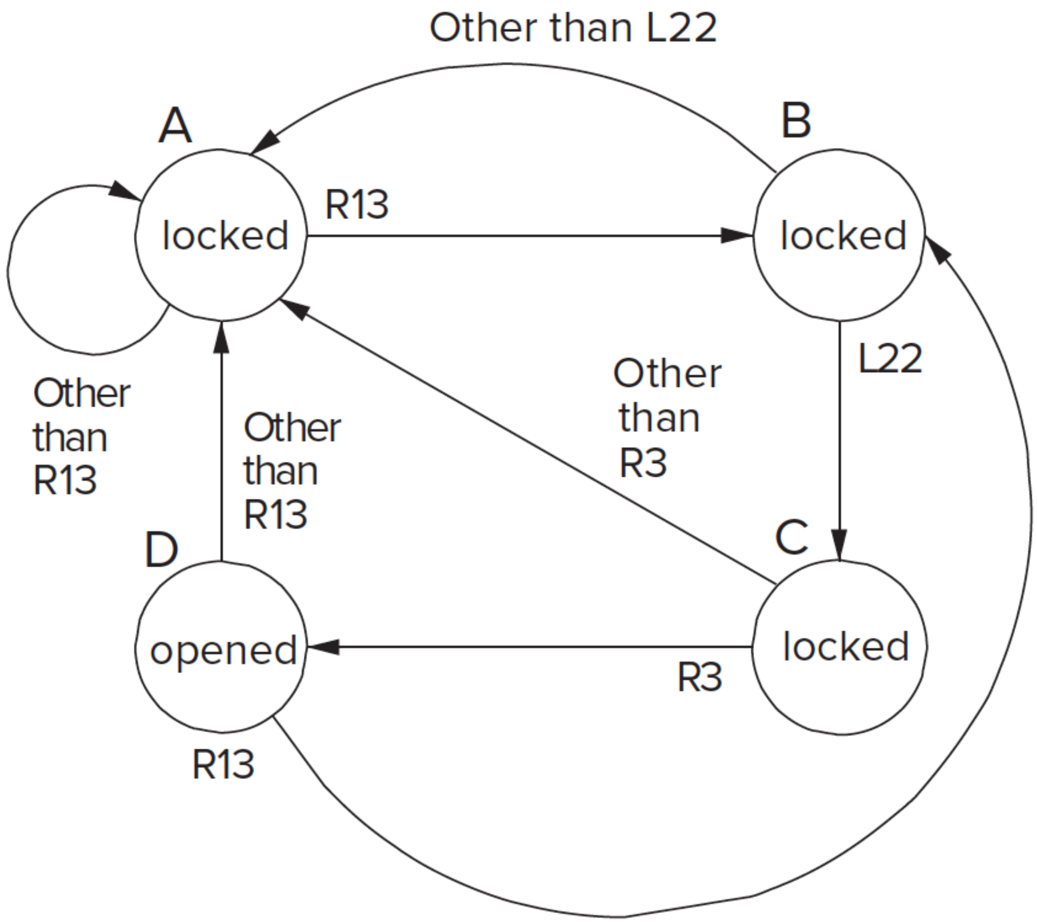
State

- In order for this lock to work, it has to keep track (**remember**) of the past events!
- If passcode is **R13-L22-R3**, sequence of **states** to unlock:
  A. The lock is not open (locked), and no relevant operations have been performed
  B. Locked but user has completed R13
  C. Locked but user has completed R13-L22
  D. Unlocked: user has completed R13-L22-R3

- The **state** of a system is a snapshot of all relevant elements of the system at the moment of the snapshot
  - To open the lock, **states A-D must be completed in order**
  - If anything else happens (e.g., L5), lock **returns** to state A

State Diagram of Our Sequential Lock

- Completely describes the operation of the sequential lock



- We will understand "state diagrams" fully later today

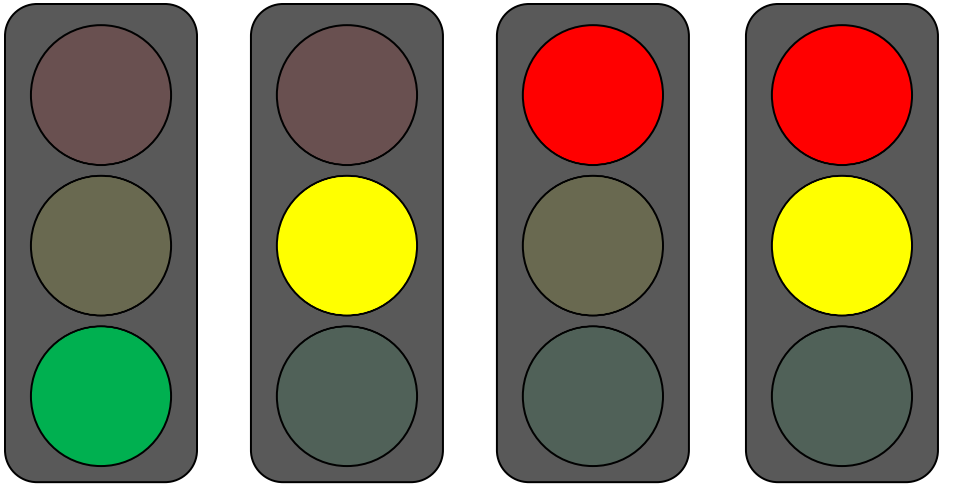
Image source: Patt and Patel, "Introduction to Computing Systems", 2nd ed., page 76.

Asynchronous vs. Synchronous State Changes

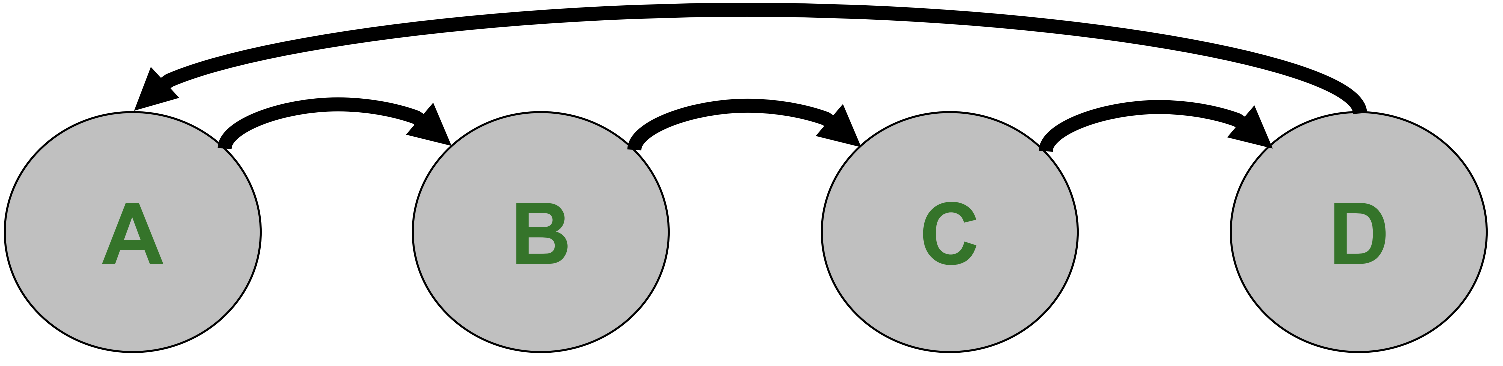
- Sequential lock we saw is an asynchronous "machine"
  - State transitions can take place immediately in response to input
  - There is nothing that synchronizes when each state transition must occur
- Most modern computers are synchronous "machines"
  - State transitions take place at fixed units of time (i.e., potentially delayed response to input, synchronized to an external signal)
  - Controlled in part by a clock, as we will see soon
- These are two different design paradigms, with tradeoffs

Another Simple Example of State

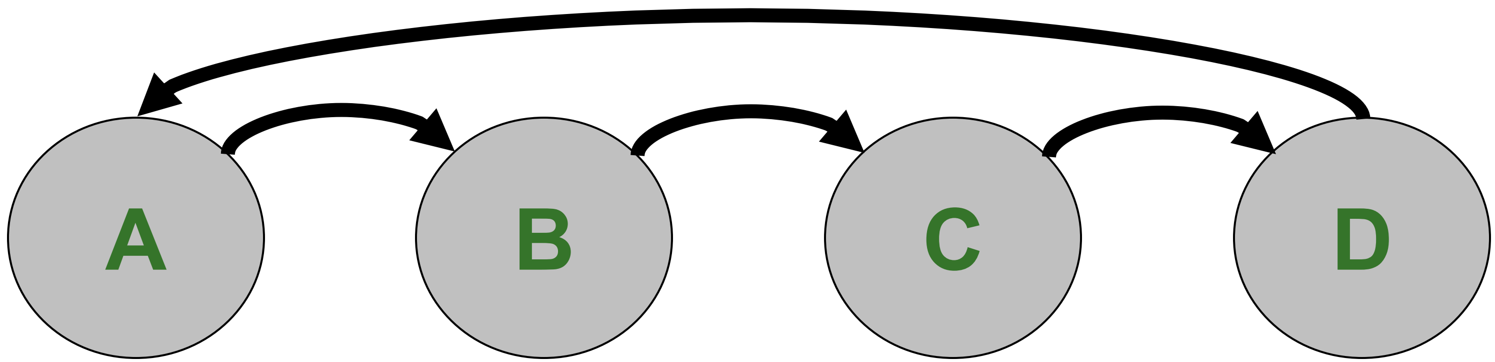
- A standard Swiss traffic light has **4 states**
  A. Green
  B. Yellow
  C. Red
  D. Red and Yellow



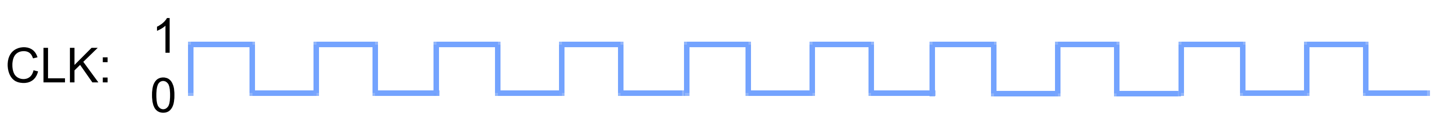
- The sequence of these states are always as follows



Changing State: The Notion of Clock (I)



- When should the light change from one state to another?
- We need a **clock** to dictate when to change state
  - Clock signal alternates between 0 & 1



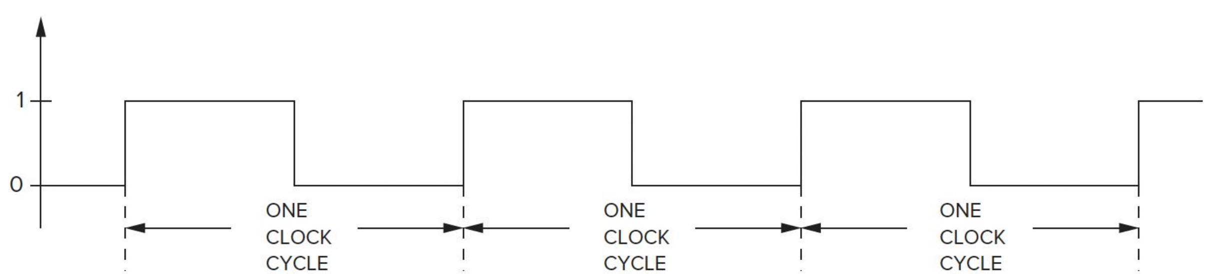
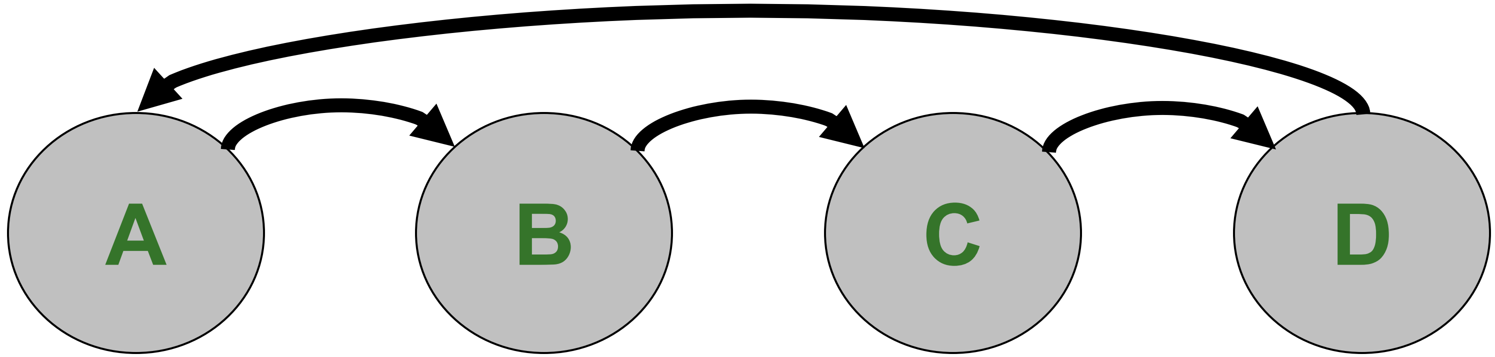
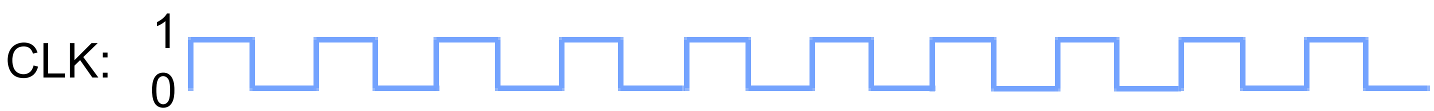


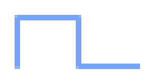
Figure 3.28 A clock signal.

Changing State: The Notion of Clock (I)



- When should the light change from one state to another?
- We need a **clock** to dictate when to change state
  - Clock signal alternates between 0 & 1



- At the start of a clock cycle (), system state changes
  - During a clock cycle, the state stays constant
  - In this traffic light example, we are assuming the traffic light stays in each state an equal amount of time

Changing State: The Notion of Clock (II)

- **Clock** is a general mechanism that triggers transition from one state to another in a (synchronous) sequential circuit
- Clock **synchronizes state changes** across many sequential circuit elements
- Combinational logic evaluates for the **length of the clock cycle**
- Clock cycle should be chosen to accommodate maximum combinational circuit delay
  - More on this later, when we discuss timing

Asynchronous vs. Synchronous State Changes

- Sequential lock we saw is an asynchronous "machine"
  - State transitions can take place immediately in response to input
  - There is nothing that synchronizes when each state transition must occur
- Most modern computers are synchronous "machines"
  - State transitions take place at fixed units of time (i.e., synch.)
  - Controlled in part by a clock, as we will see soon
- These are two different design paradigms, with tradeoffs
  - Synchronous control can be easier to get correct when the system consists of many components and many states
  - Asynchronous control can be more efficient (no clock overheads)

We will assume synchronous systems in most of this course

Finite State Machines

Finite State Machines

- What is a **Finite State Machine** (FSM)?
  - **A discrete-time model** of a stateful system
  - Each state represents a snapshot of the system at a given time

- An FSM pictorially shows
  1. the set of all possible **states** that a system can be in
  2. how the system transitions from one state to another

- An FSM can model
  - A traffic light, an elevator, fan speed, a microprocessor, etc.

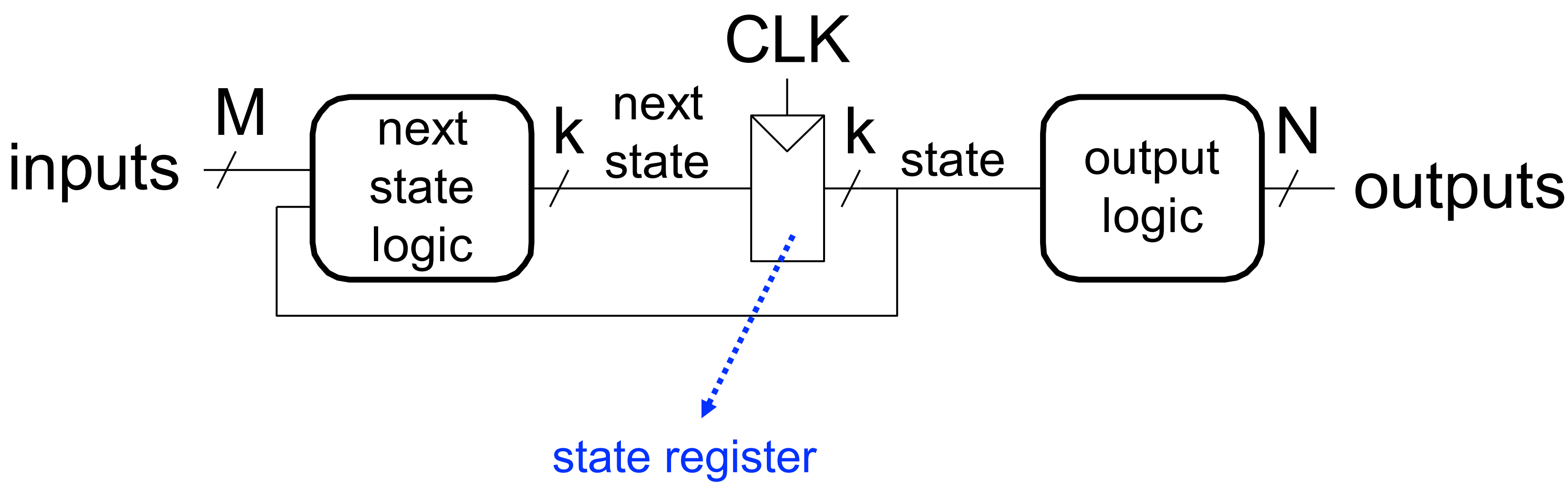
- **An FSM enables us to pictorially think of a stateful system using simple diagrams**

Finite State Machines (FSMs) Consist of:

- **Five elements:**
  1. A **finite** number of states
     - ***State***: snapshot of all relevant elements of the system at the time of the snapshot
  2. A **finite** number of external inputs
  3. A **finite** number of external outputs
  4. An explicit specification of all state transitions
     - How to get from one state to another
  5. An explicit specification of what determines each external output value

Finite State Machines (FSMs)

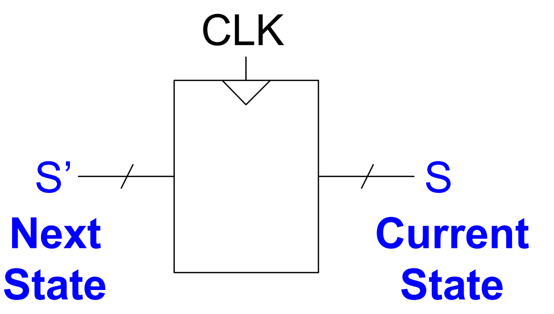
- Each FSM consists of three separate parts:
  - next state logic
  - state register
  - output logic



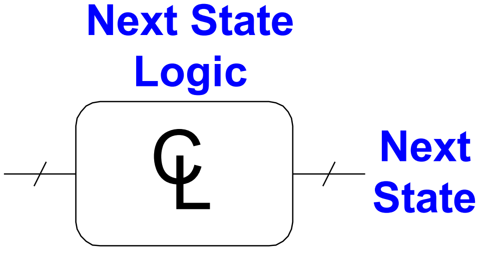
At the beginning of the clock cycle, next state is latched into the state register

Finite State Machines (FSMs) Consist of:

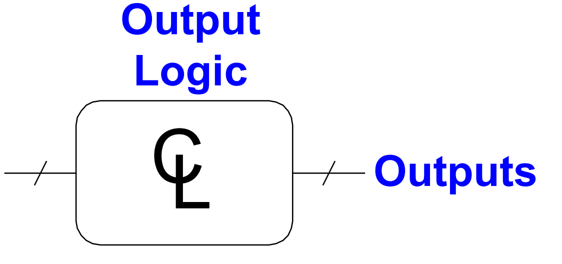
- **Sequential Circuits**
  - State register(s)
    - Store the current state and
    - Load the next state at the clock edge



- **Combinational Circuits**
  - Next state logic
    - Determines what the next state will be

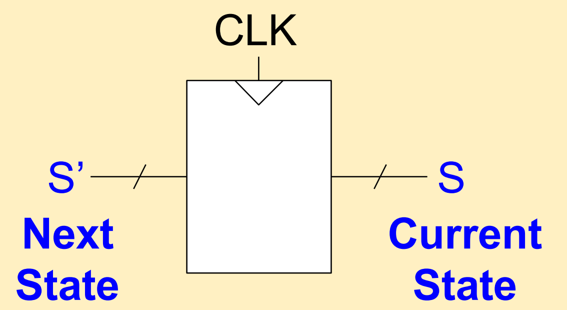


  - Output logic
    - Generates the outputs

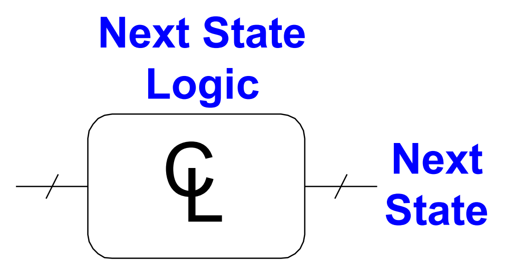


Finite State Machines (FSMs) Consist of:

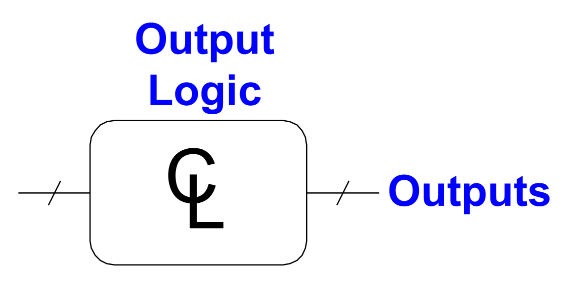
- **Sequential Circuits**
  - State register(s)
    - Store the current state and
    - Provide the next state at the clock edge



- **Combinational Circuits**
  - Next state logic
    - Determines what the next state will be



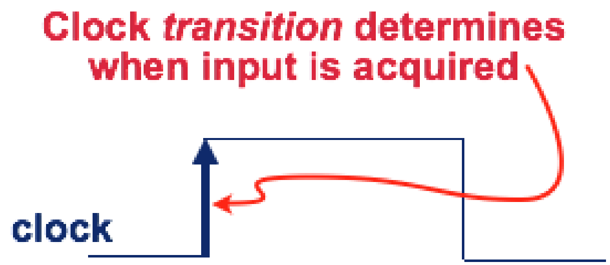
  - Output logic
    - Generates the outputs



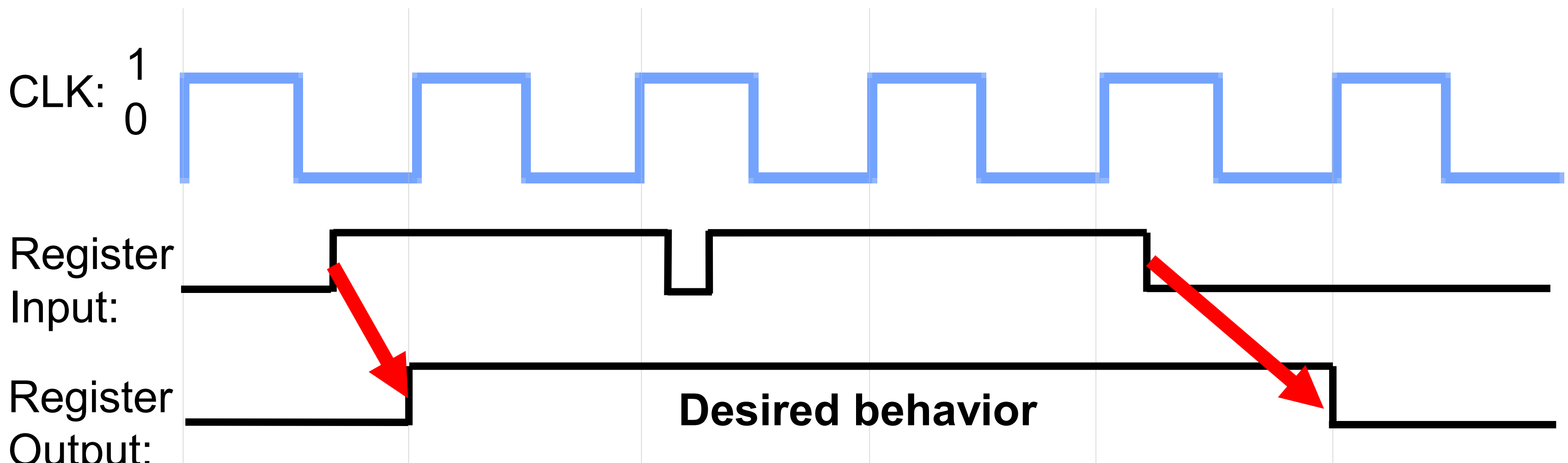
State Register Implementation

- How can we implement a **state register**? Two properties:

1. We need to store data at the **beginning** of every clock cycle

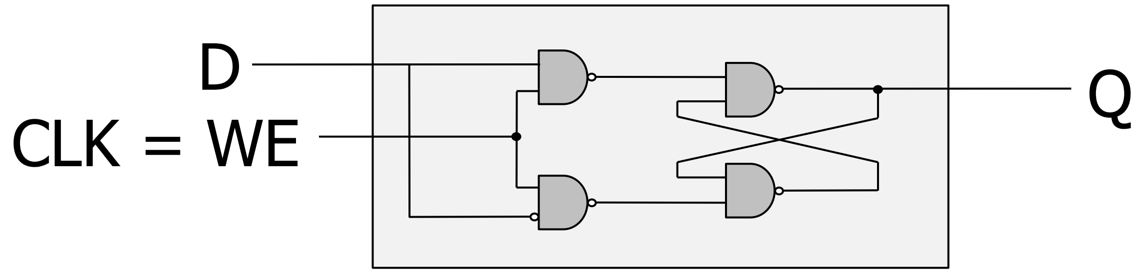


2. The data must be **available** during the entire **clock cycle**

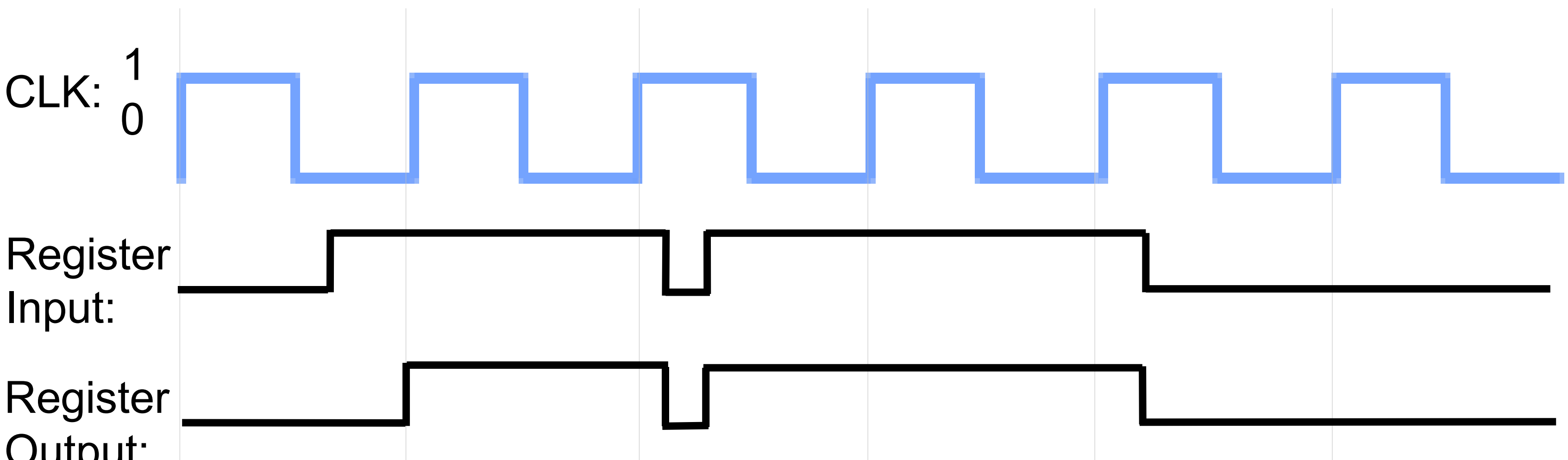


The Problem with Latches: Transparency

Recall the Gated D Latch

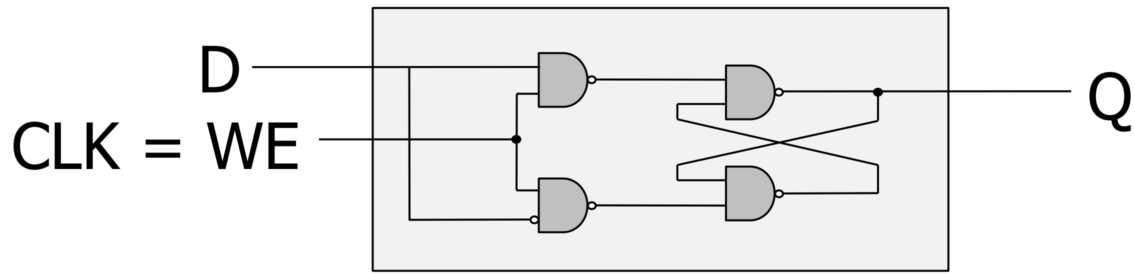


- Currently, we **cannot** simply wire a clock to WE of a latch
  - **Whenever the clock is high,** the latch propagates **D** to **Q**
  - **The latch is "transparent"**

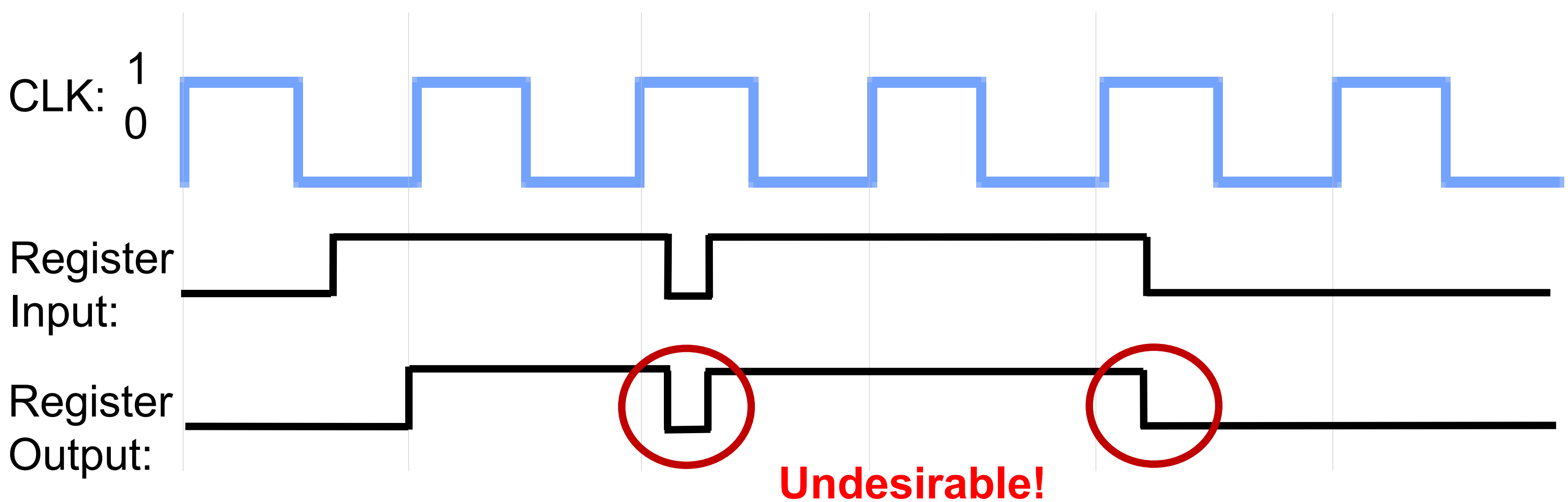


The Problem with Latches: Transparency

Recall the
Gated D Latch

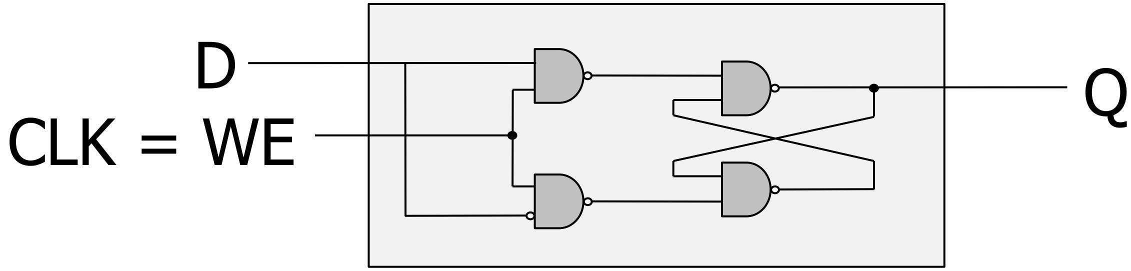


- Currently, we **cannot** simply wire a clock to WE of a latch
  - **Whenever the clock is high,** the latch propagates **D** to **Q**
  - **The latch is "transparent"**



The Problem with Latches: Transparency

Recall the
Gated D Latch



How can we change the latch, so that

1) D (input) is **observable** at **Q** (output) **only** at the **beginning of next** clock cycle?

2) Q is **available for the full clock cycle**

The Need for a New Storage Element

- To design viable FSMs

- We need storage elements that allow us to:

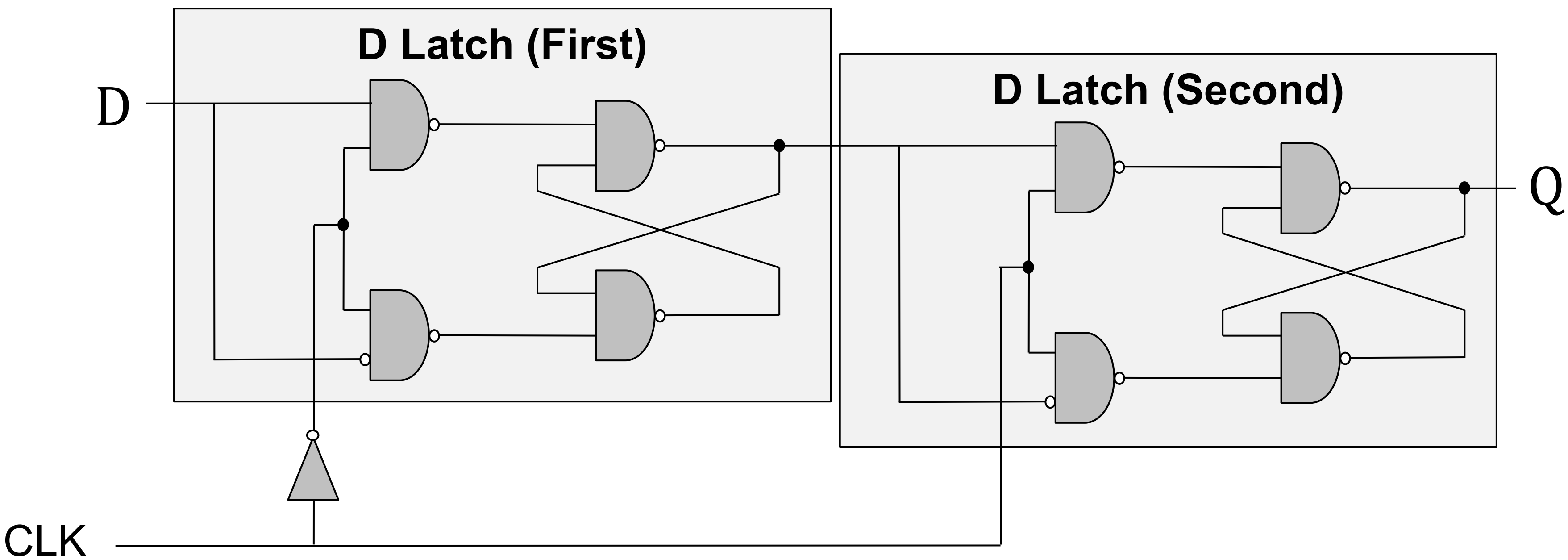
  - **read** the **current state** throughout the **entire current clock cycle**

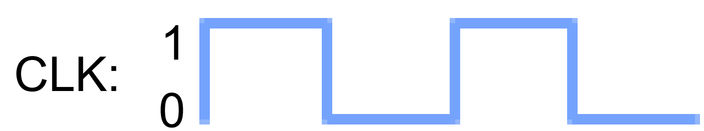
  AND

  - **not write** the **next state** values into the storage elements **until** the beginning of the **next clock cycle**

The D Flip-Flop

- 1) state change on clock edge, 2) data available for full cycle

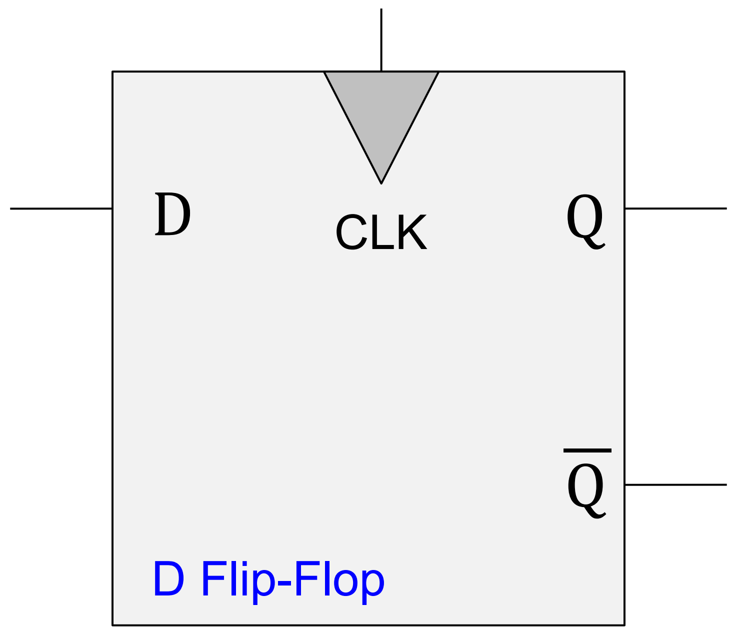


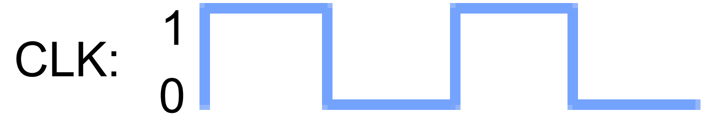


- When the clock is low, 1st latch propagates **D** to the input of the 2nd (Q unchanged)
- Only when the clock is high, 2nd latch latches **D (Q stores D)**
  - At the rising edge of clock (clock going from 0→1), Q gets assigned D

The D Flip-Flop

- 1) state change on clock edge, 2) data available for full cycle

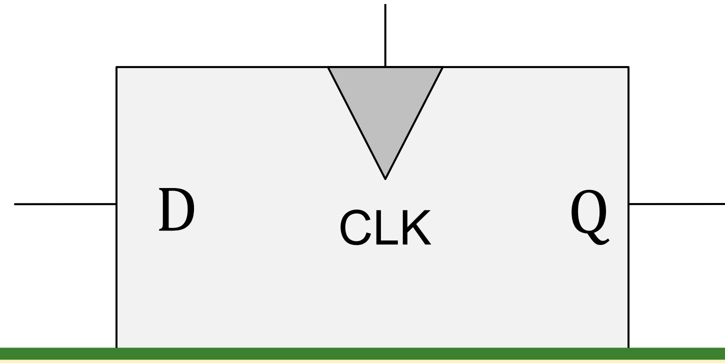




- At the rising edge of clock (clock going from 0→1), **Q** gets assigned **D**
- At all other times, Q is unchanged

The D Flip-Flop

- 1) state change on clock edge, 2) data available for full cycle

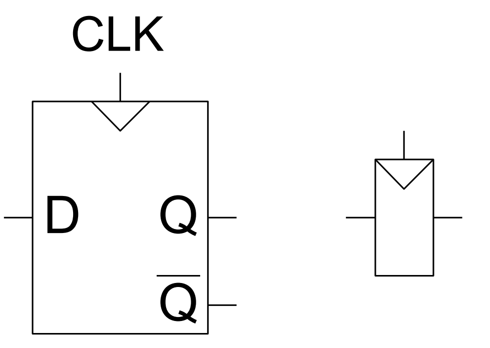


We can use **D Flip-Flops**
to implement the state register

- At the rising edge of clock (clock going from 0→1), **Q** gets assigned **D**
- At all other times, Q is unchanged

Rising-Clock-Edge Triggered Flip-Flop

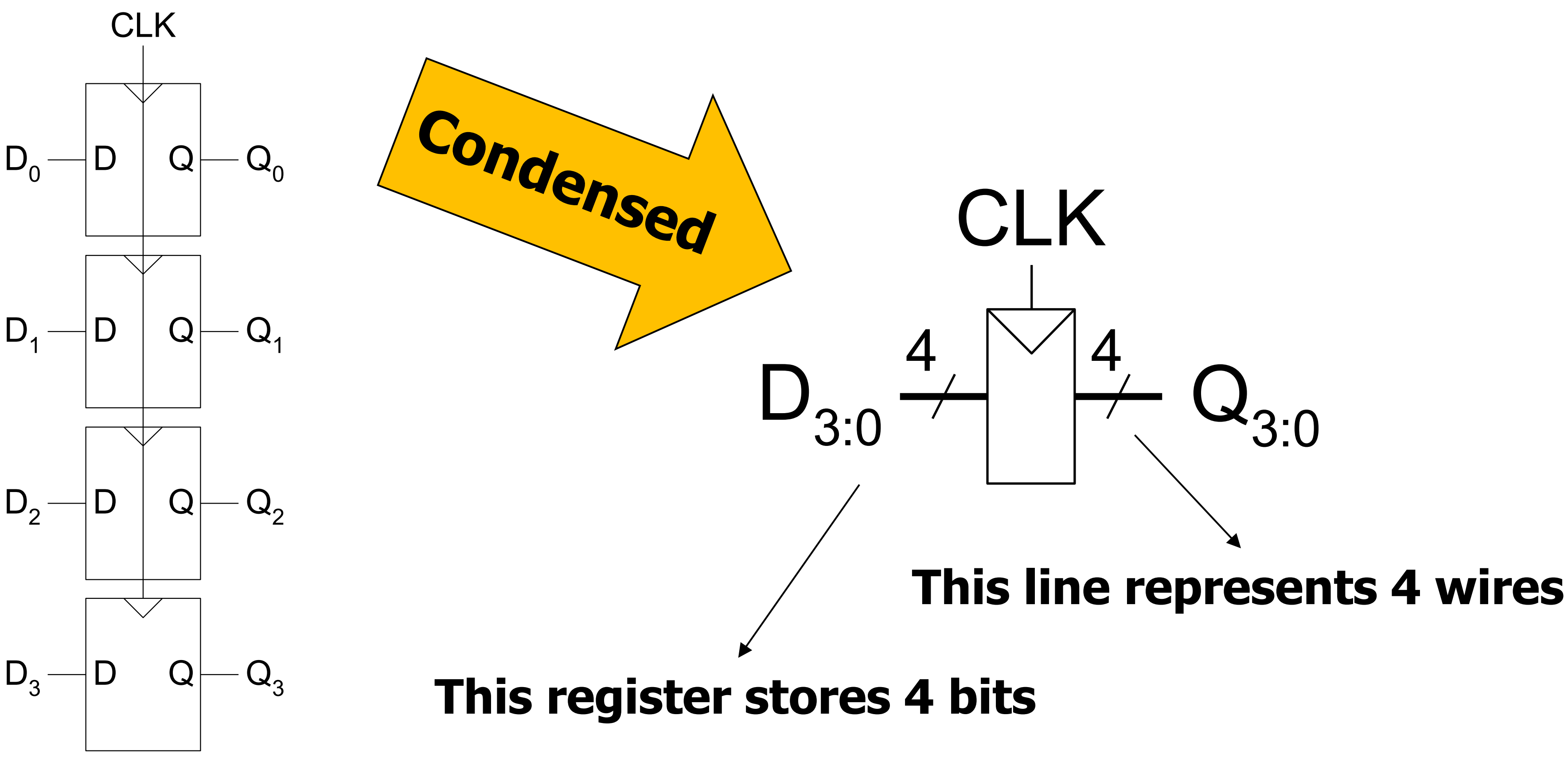
- **Two inputs**: CLK, D
- **Function**
  - The flip-flop "samples" **D** on the rising edge of CLK (**positive edge**)
  - When CLK rises from 0 to 1, **D** passes through to **Q**
  - Otherwise, **Q** holds its previous value
  - **Q** changes **only** on the rising edge of CLK



- A flip-flop is called an **edge-triggered state element** because it captures data on the clock edge
  - A latch is a **level-triggered** state element

D Flip-Flop Based Register

- Multiple parallel D flip-flops, each of which storing 1 bit



A 4-Bit D-Flip-Flop-Based Register (Internally)

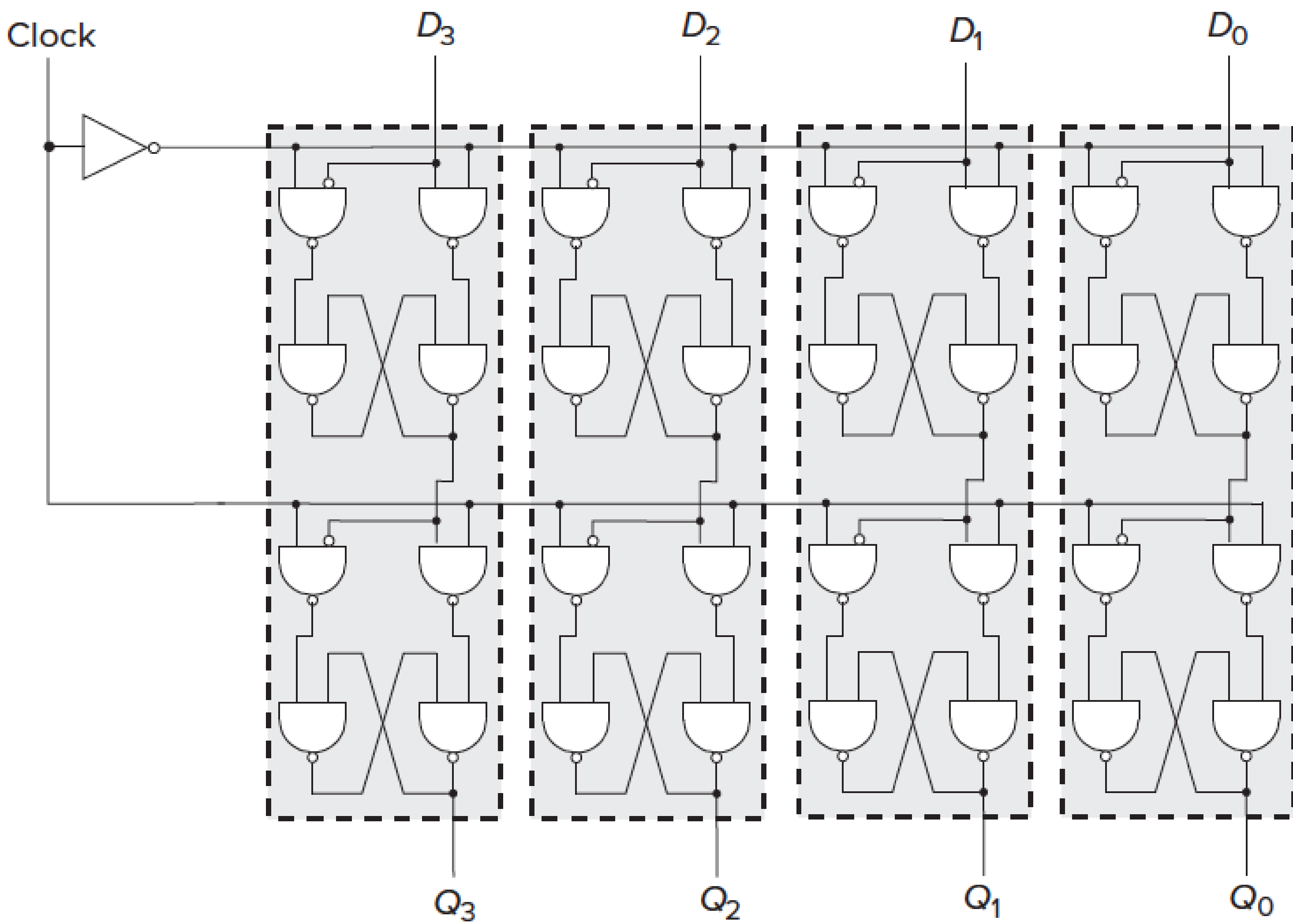
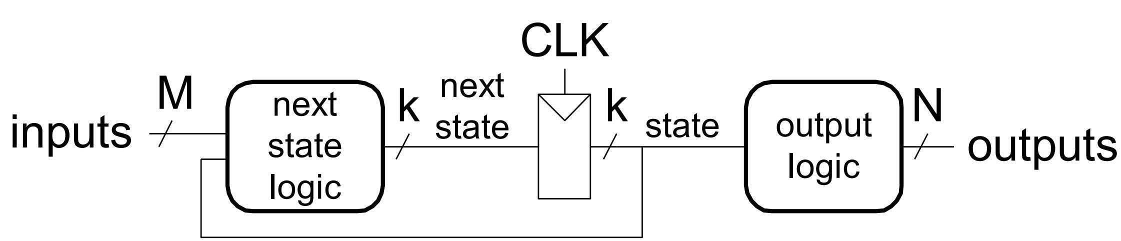


Image source: Patt and Patel, "Introduction to Computing Systems", 3rd ed., tentative page 95.

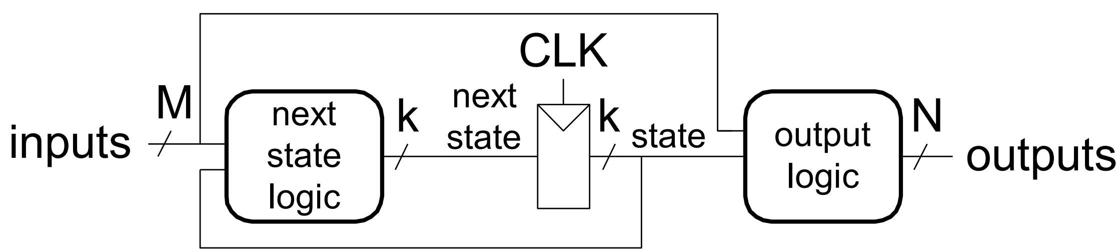
Finite State Machines (FSMs)

- Next state is determined by the current state and the inputs
- Two types of finite state machines differ in the **output logic**:
  - **Moore FSM**: outputs depend only on the current state

Moore FSM



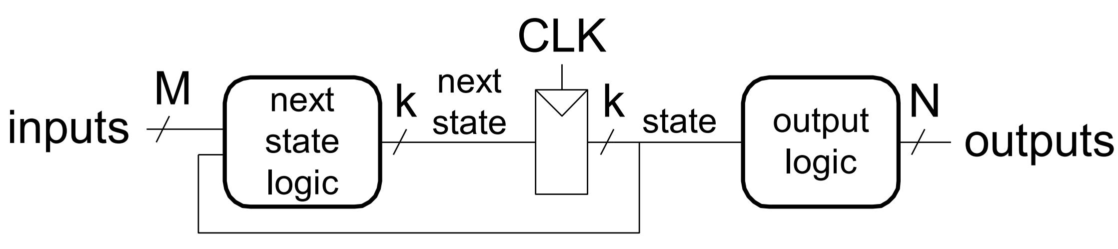
Mealy FSM



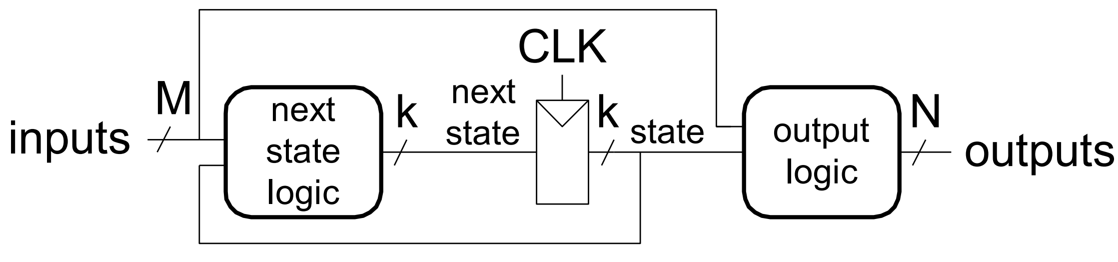
Finite State Machines (FSMs)

- Next state is determined by the current state and the inputs
- Two types of finite state machines differ in the **output logic**:
  - **Moore FSM**: outputs depend only on the current state
  - **Mealy FSM**: outputs depend on the current state and the inputs

Moore FSM

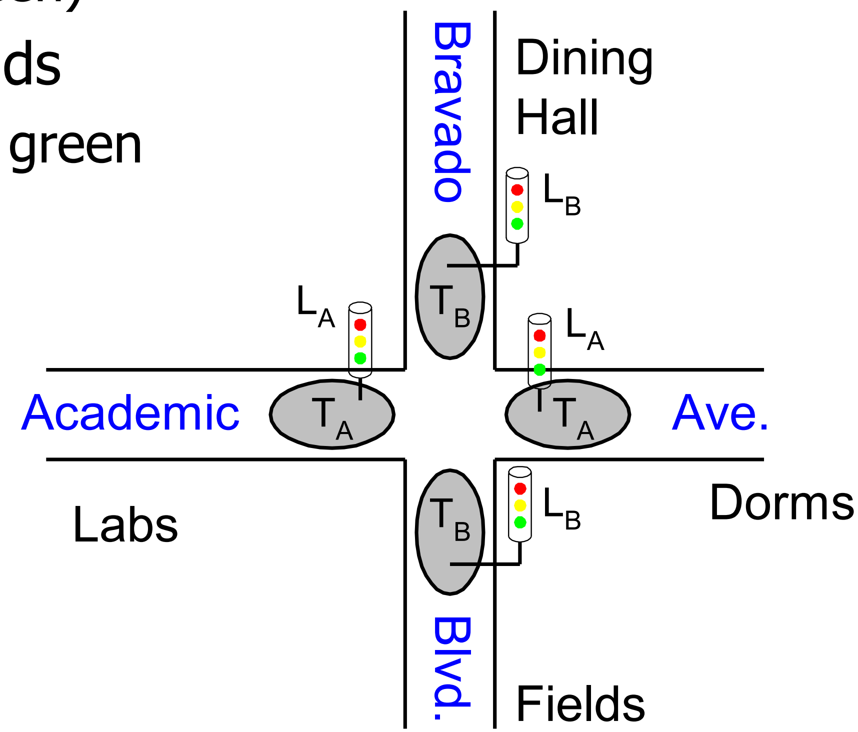


Mealy FSM



Finite State Machine Example

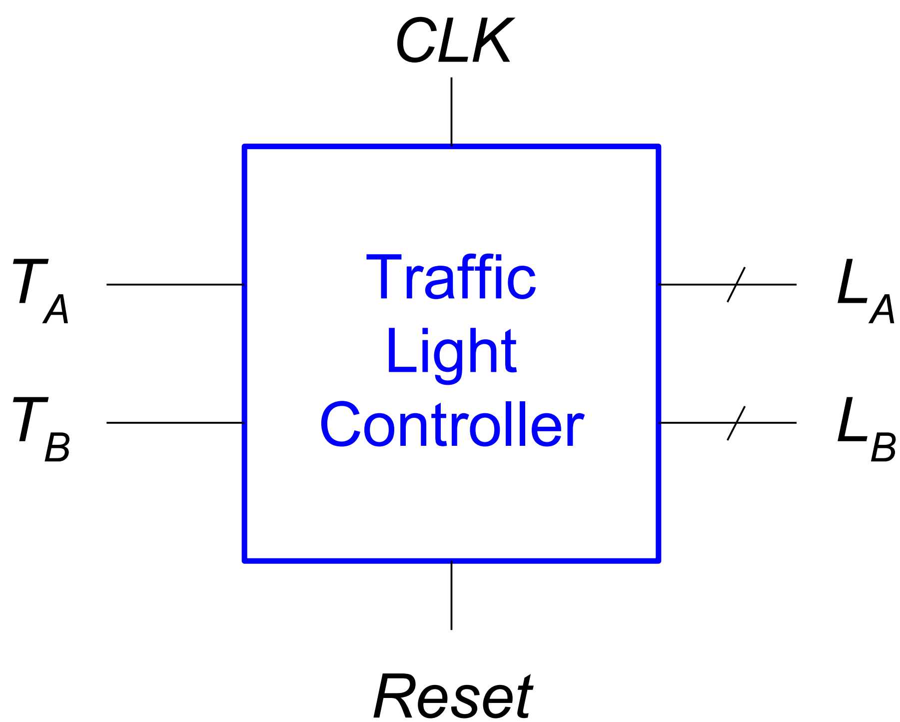
- "Smart" traffic light controller
  - **2 inputs**:
    - Traffic sensors: T_A , T_B (TRUE when there's traffic)
  - **2 outputs**:
    - Lights: L_A , L_B (Red, Yellow, Green)
  - State can change every 5 seconds
    - Except if green and traffic, stay green



From H&H Section 3.4.1

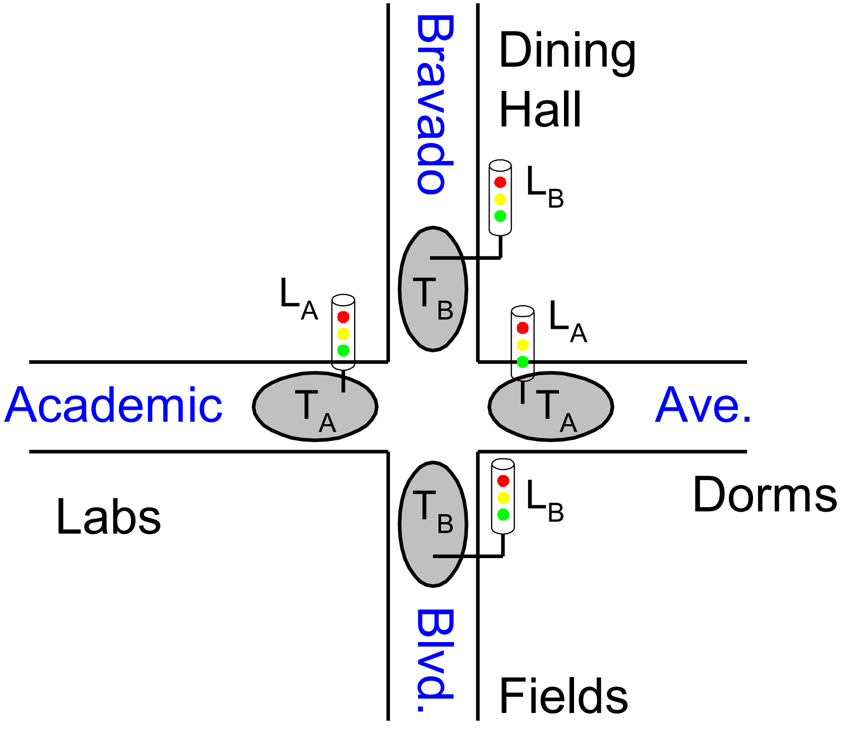
Finite State Machine Black Box

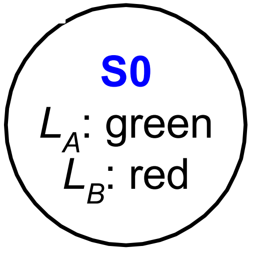
- **Inputs:** CLK, Reset, T_A , T_B
- **Outputs:** L_A , L_B



Finite State Machine Transition Diagram

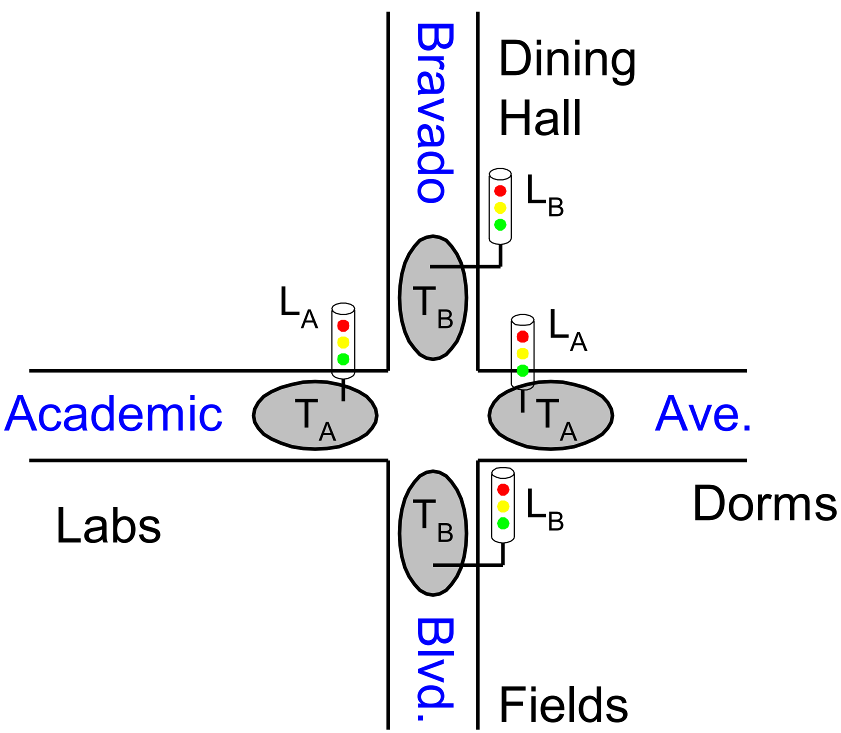
- **Moore FSM:** outputs labeled in each state
  - **States:** Circles
  - **Transitions:** Arcs

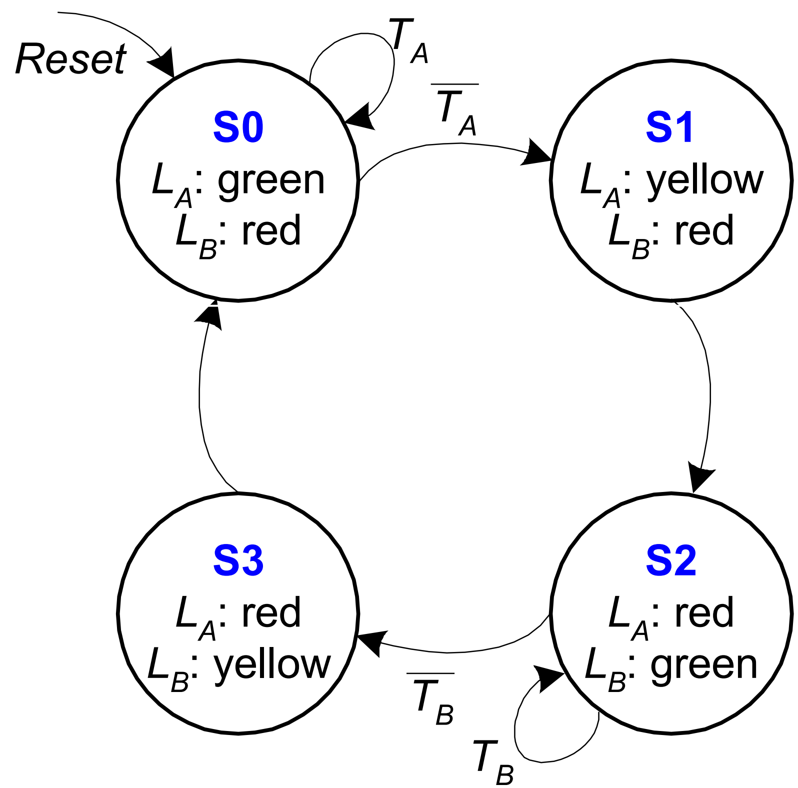




Finite State Machine Transition Diagram

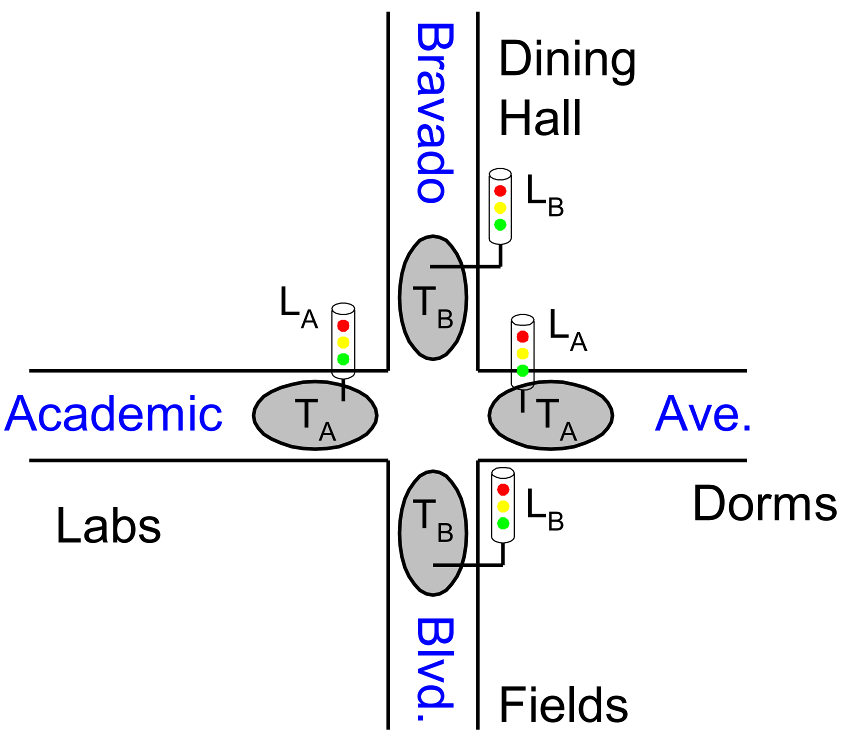
- **Moore FSM:** outputs labeled in each state
  - **States:** Circles
  - **Transitions:** Arcs

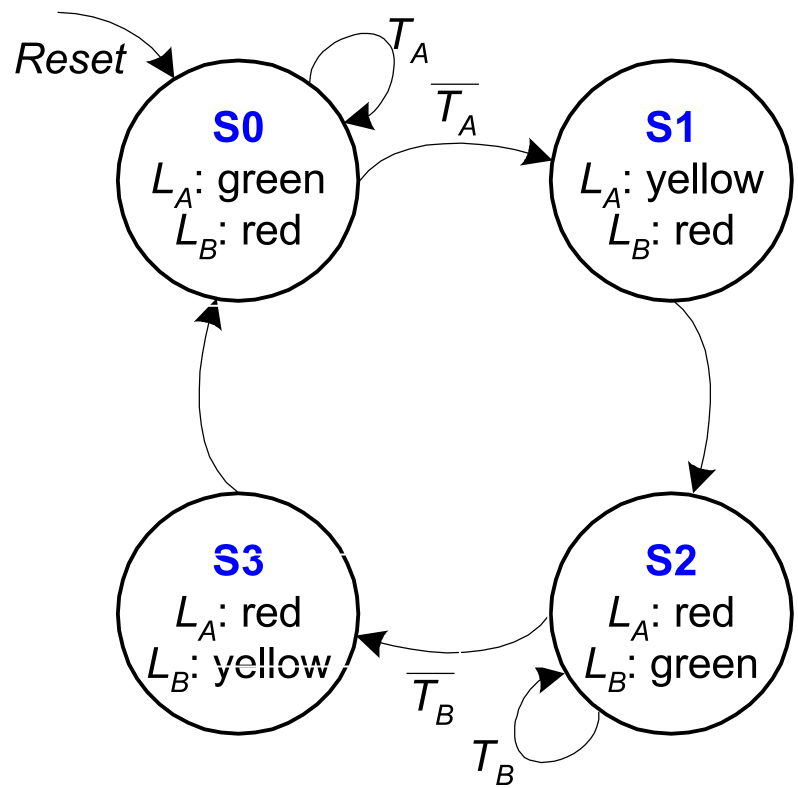




Finite State Machine Transition Diagram

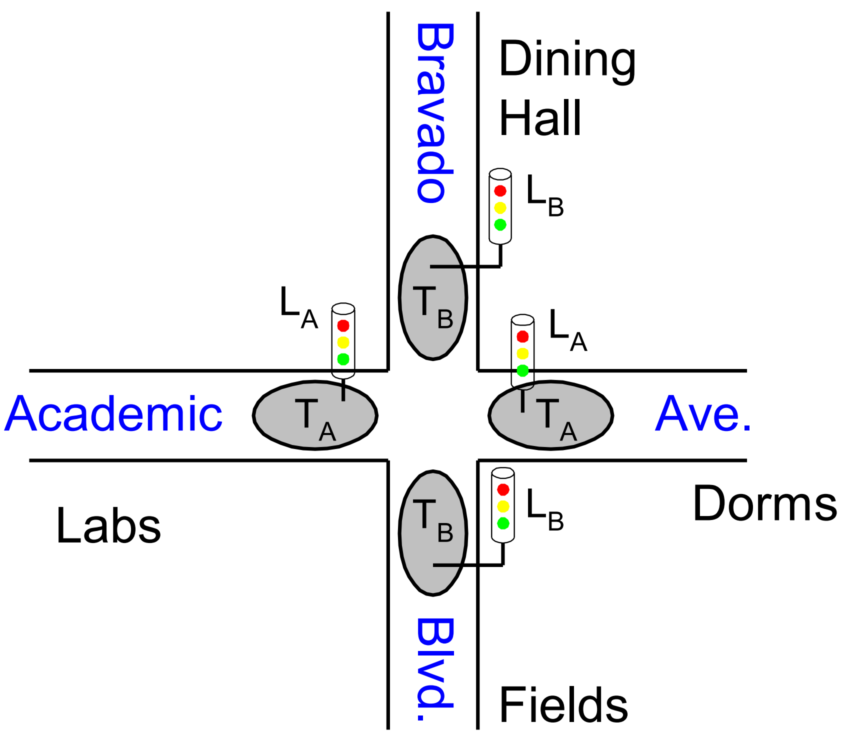
- **Moore FSM:** outputs labeled in each state
  - **States:** Circles
  - **Transitions:** Arcs

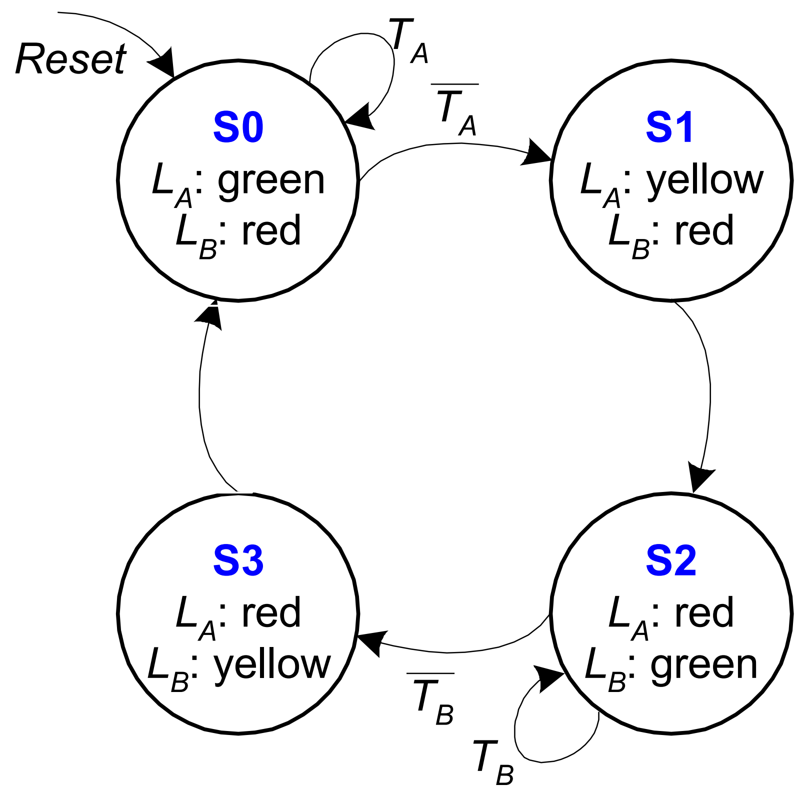




Finite State Machine Transition Diagram

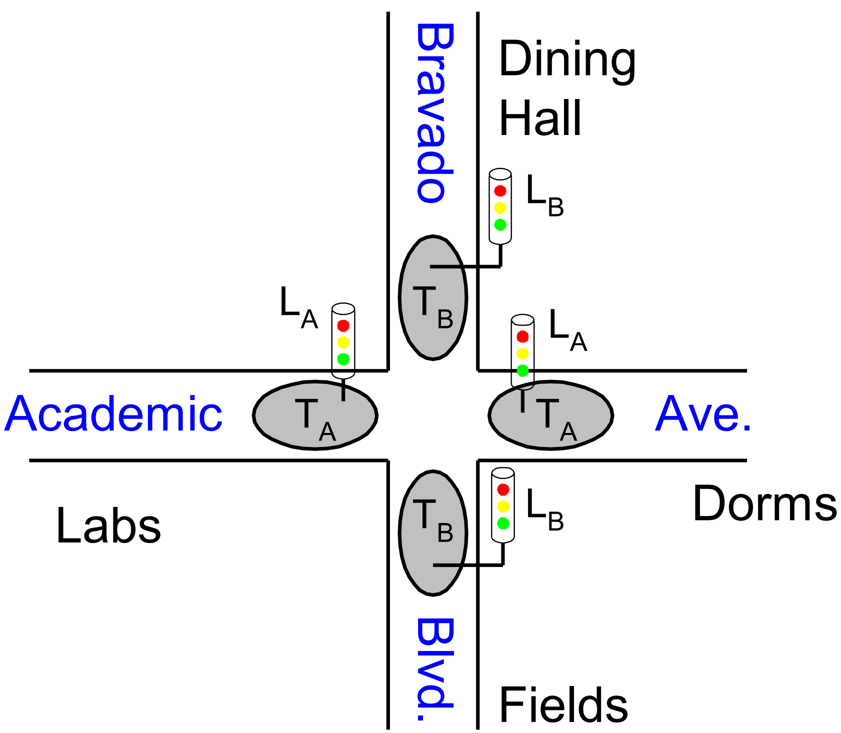
- **Moore FSM:** outputs labeled in each state
  - **States:** Circles
  - **Transitions:** Arcs

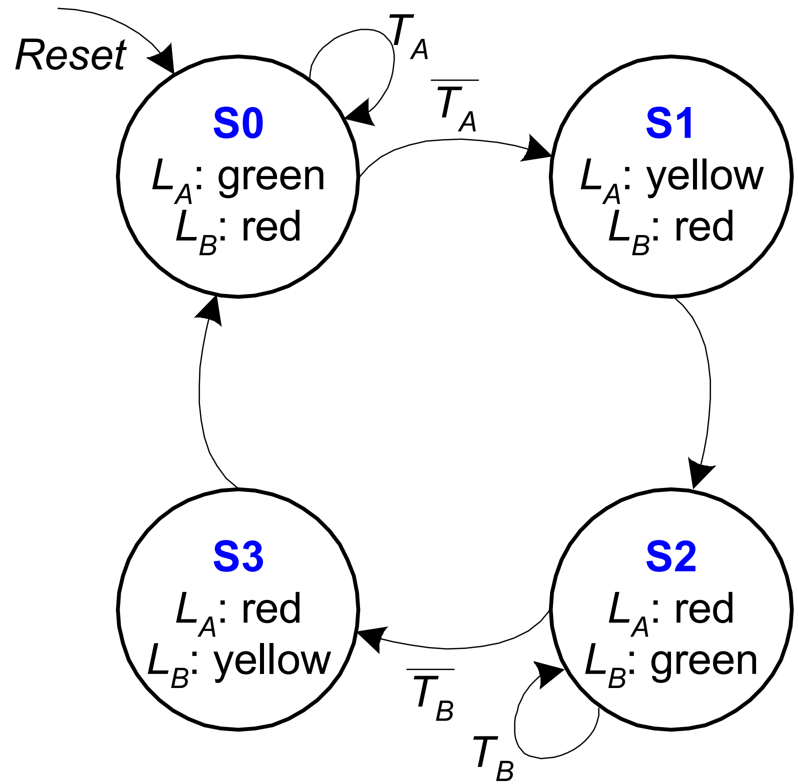




Finite State Machine Transition Diagram

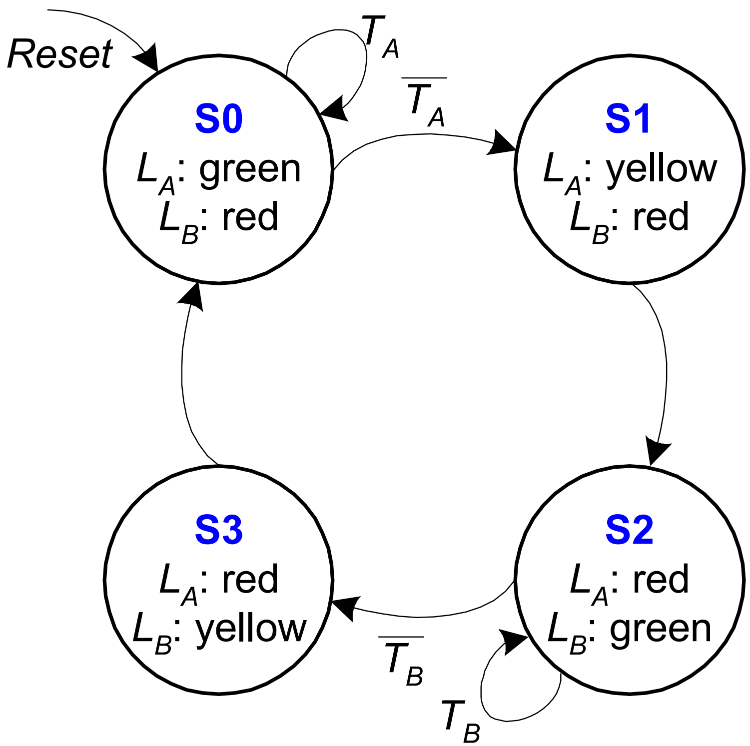
- **Moore FSM:** outputs labeled in each state
  - **States:** Circles
  - **Transitions:** Arcs





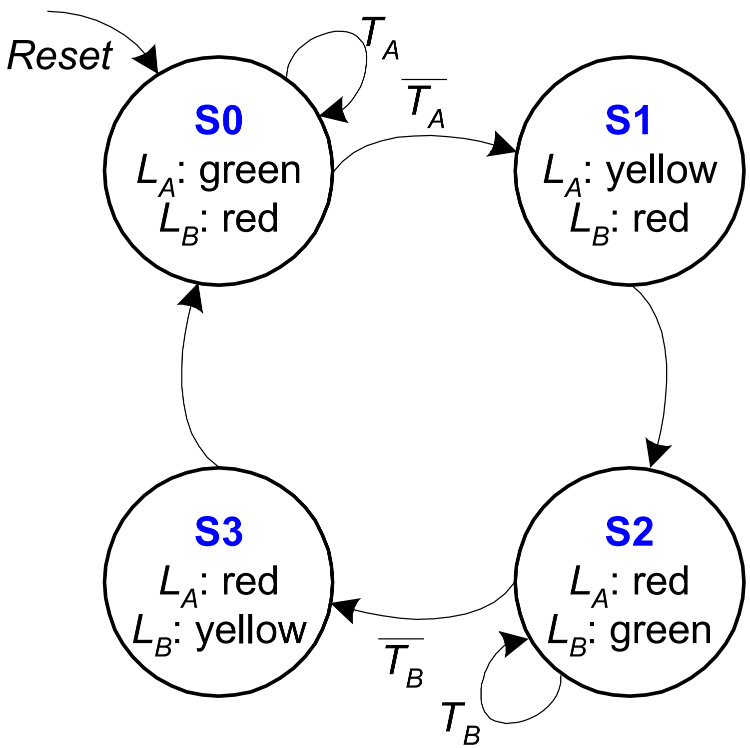
Finite State Machine: State Transition Table

FSM State Transition Table



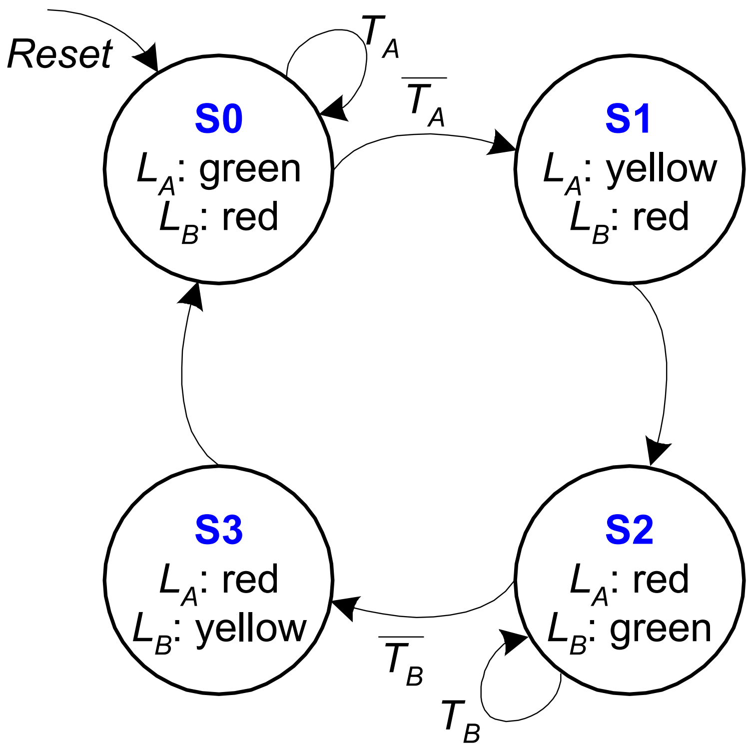
Current State	Inputs		Next State
S	T_A	T_B	S'
S0	0	X	
S0	1	X	
S1	X	X	
S2	X	0	
S2	X	1	
S3	X	X	

FSM State Transition Table



Current State	Inputs		Next State
S	T_A	T_B	S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

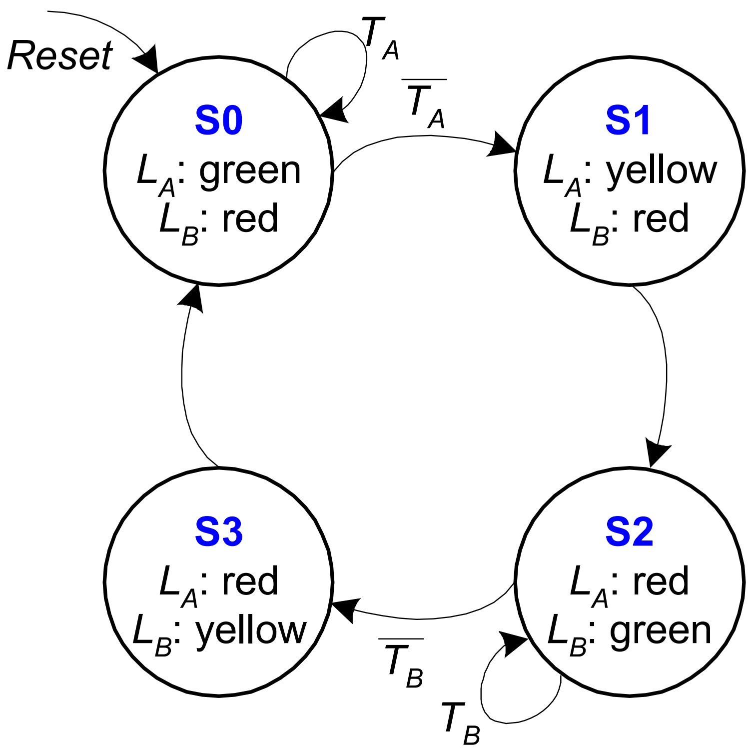
FSM State Transition Table



Current State	Inputs		Next State
S	T_A	T_B	S'
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0

State	Encoding
S0	00
S1	01
S2	10
S3	11

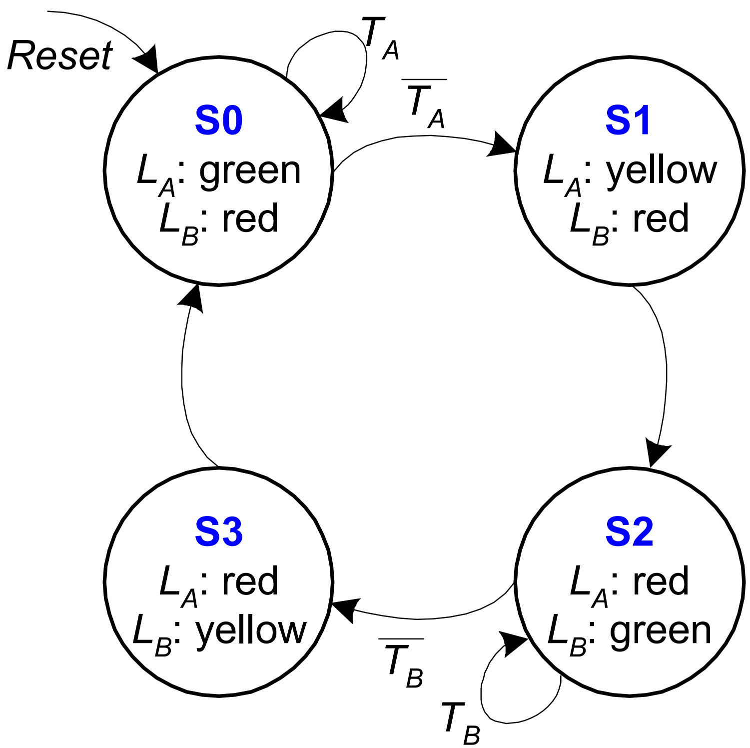
FSM State Transition Table



Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

FSM State Transition Table

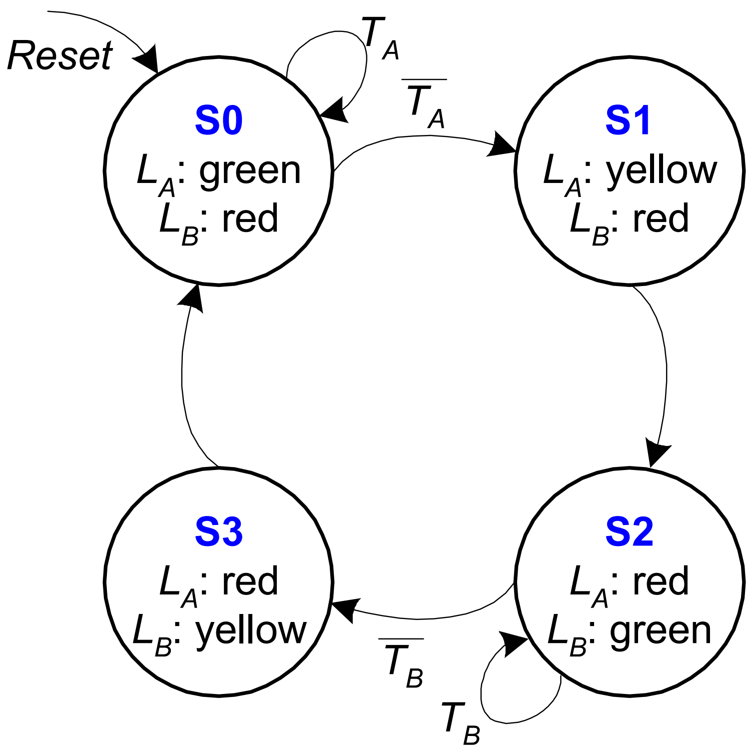


Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

$S'_1 = ?$

FSM State Transition Table

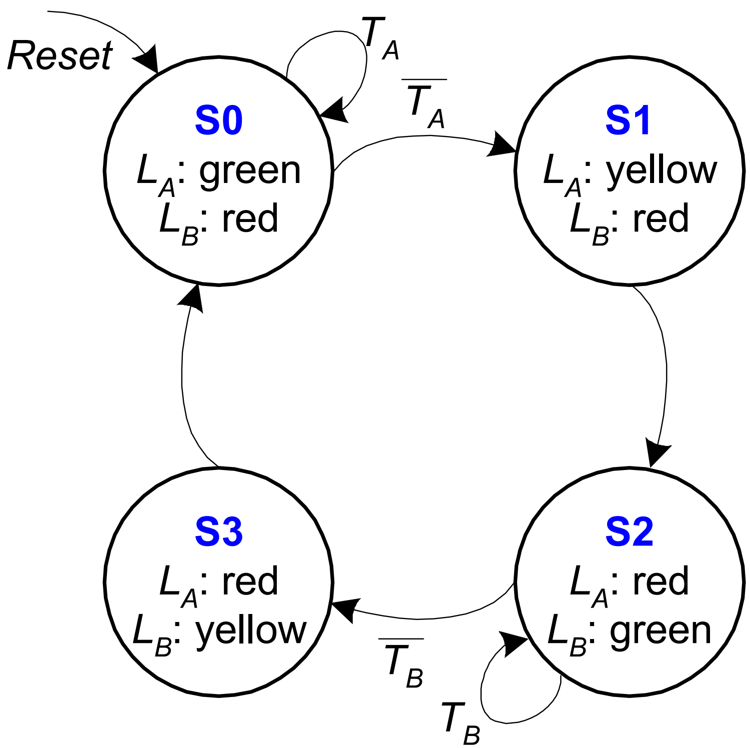


Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

$$S'_1 = (\overline{S_1} \cdot S_0) + (S_1 \cdot \overline{S_0} \cdot \overline{T_B}) + (S_1 \cdot \overline{S_0} \cdot T_B)$$

FSM State Transition Table



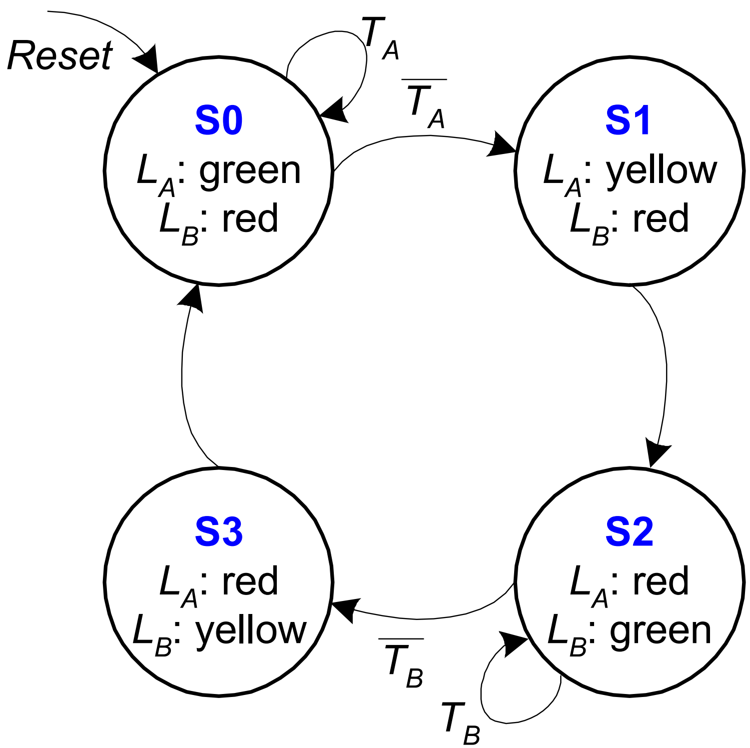
Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

$$S'_1 = (\overline{S_1} \cdot S_0) + (S_1 \cdot \overline{S_0} \cdot \overline{T_B}) + (S_1 \cdot \overline{S_0} \cdot T_B)$$

$$S'_0 = ?$$

FSM State Transition Table



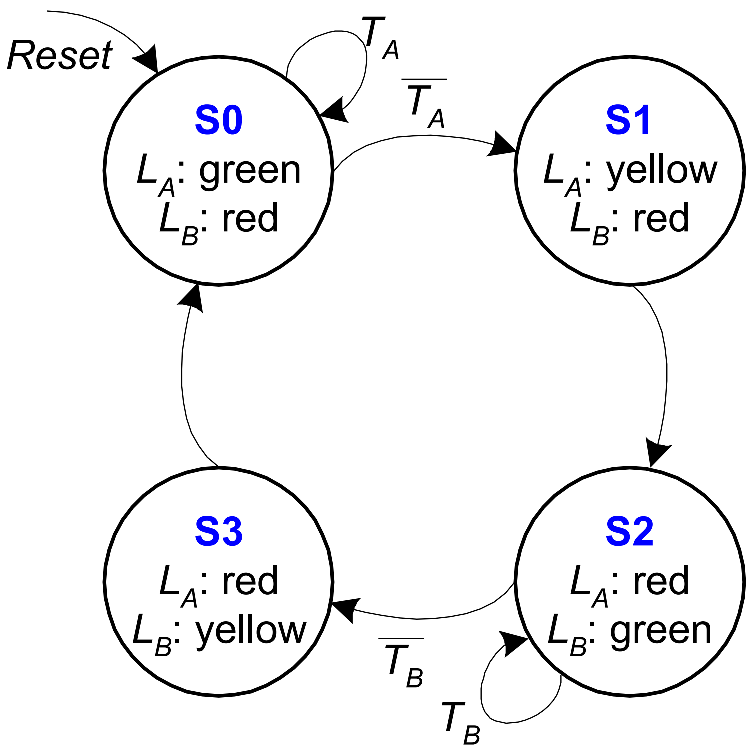
Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

State	Encoding
S0	00
S1	01
S2	10
S3	11

$$S'_1 = (\overline{S_1} \cdot S_0) + (S_1 \cdot \overline{S_0} \cdot \overline{T_B}) + (S_1 \cdot \overline{S_0} \cdot T_B)$$

$$S'_0 = (\overline{S_1} \cdot \overline{S_0} \cdot \overline{T_A}) + (S_1 \cdot \overline{S_0} \cdot \overline{T_B})$$

FSM State Transition Table



Current State		Inputs		Next State	
S_1	S_0	T_A	T_B	S'_1	S'_0
0	0	0	X	0	1
0	0	1	X	0	0
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

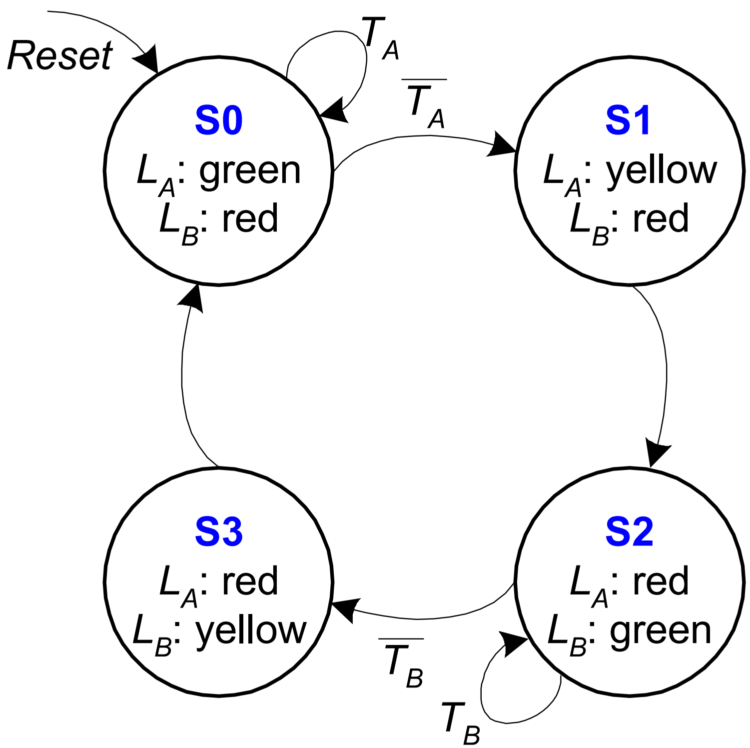
State	Encoding
S0	00
S1	01
S2	10
S3	11

$S'_1 = S_1 \text{ xor } S_0$ **(Simplified)**

$S'_0 = (\overline{S_1} \cdot \overline{S_0} \cdot \overline{T_A}) + (S_1 \cdot \overline{S_0} \cdot \overline{T_B})$

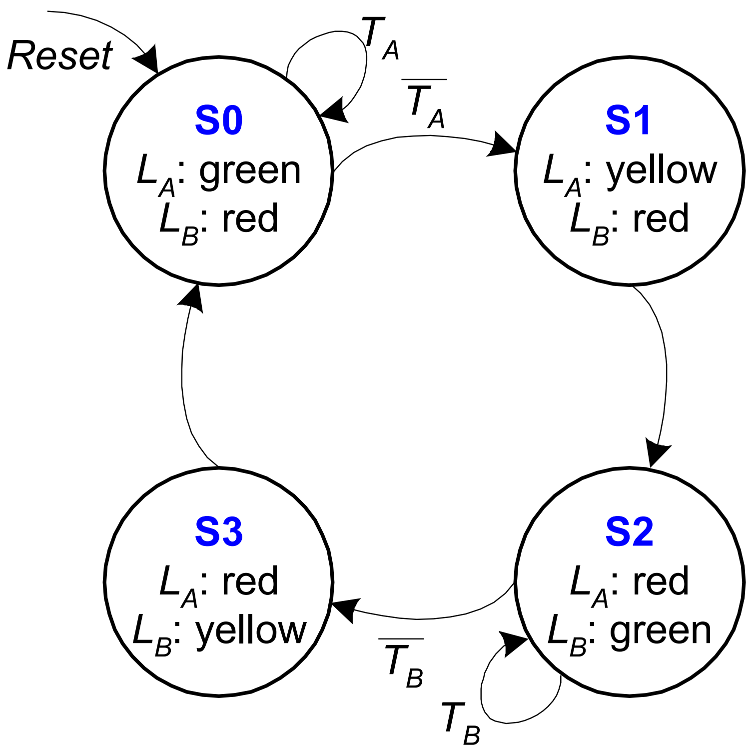
Finite State Machine: Output Table

FSM Output Table



Current State		Outputs	
S_1	S_0	L_A	L_B
0	0	green	red
0	1	yellow	red
1	0	red	green
1	1	red	yellow

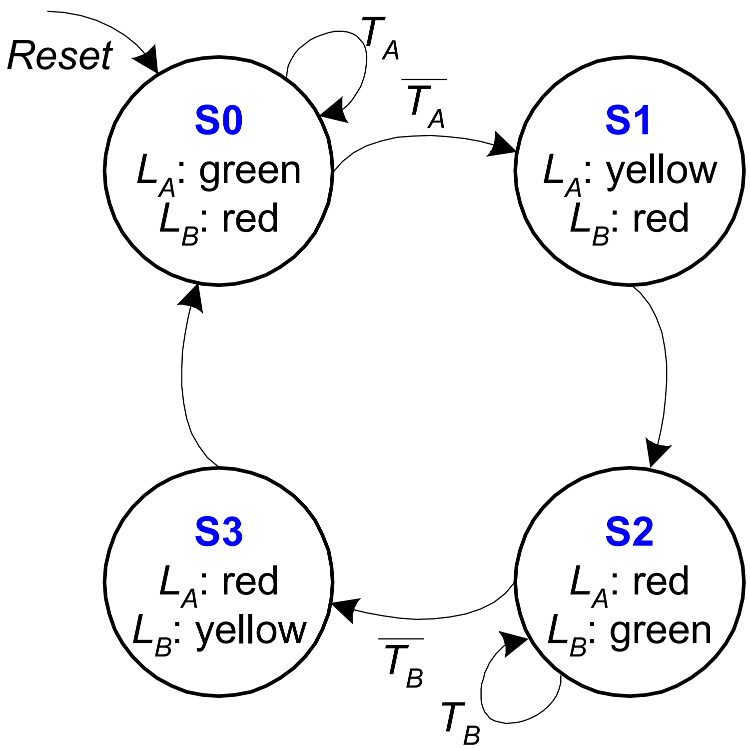
FSM Output Table



Current State		Outputs	
S_1	S_0	L_A	L_B
0	0	green	red
0	1	yellow	red
1	0	red	green
1	1	red	yellow

Output	Encoding
green	00
yellow	01
red	10

FSM Output Table

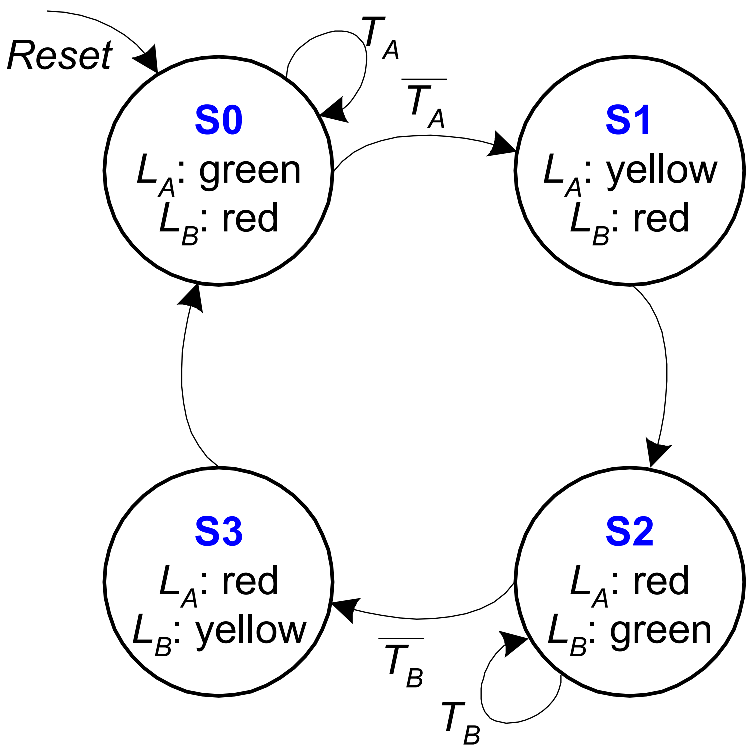


Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

$$L_{A1} = S_1$$

Output	Encoding
green	00
yellow	01
red	10

FSM Output Table

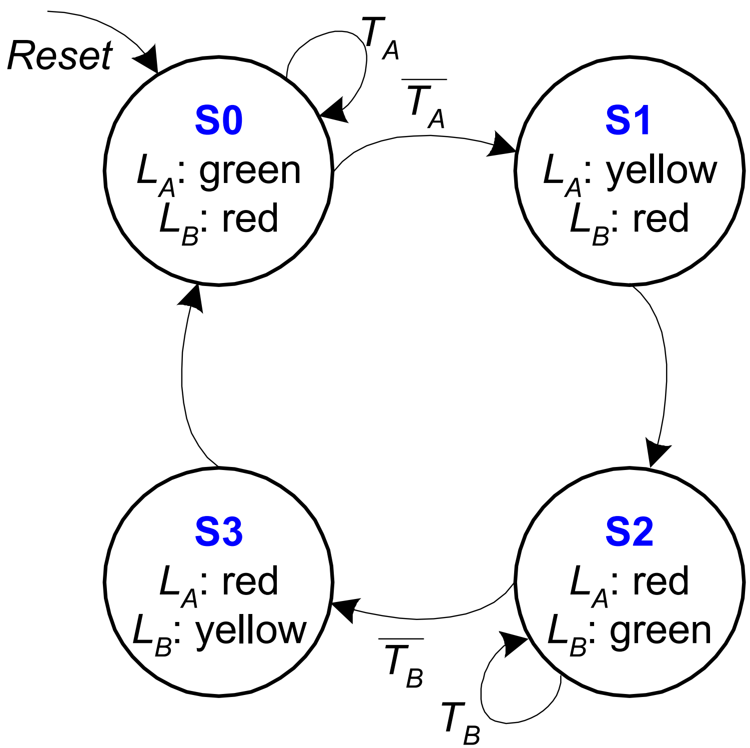


Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

Output	Encoding
green	00
yellow	01
red	10

$$L_{A1} = S_1$$
$$L_{A0} = \overline{S_1} \cdot S_0$$

FSM Output Table

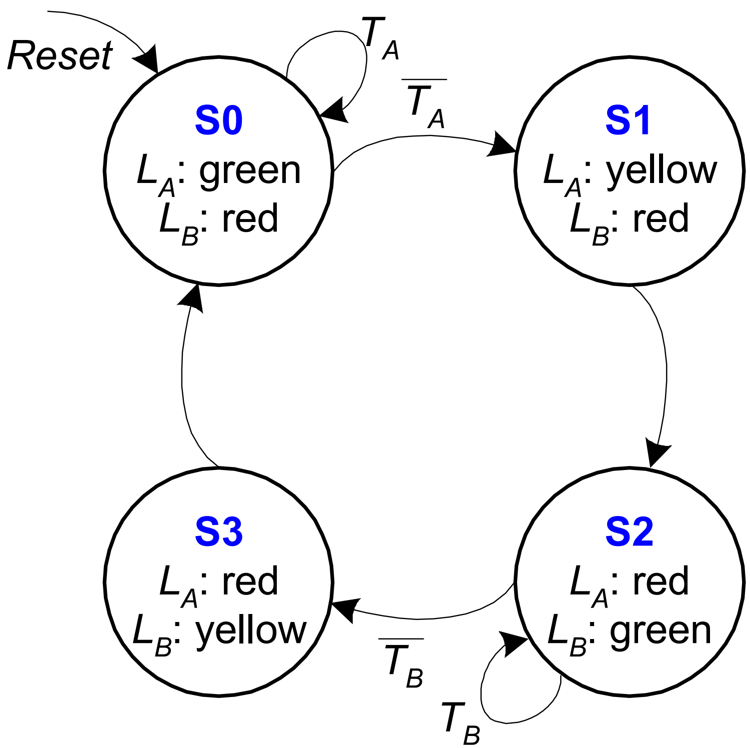


Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

$$L_{A1} = S_1$$
$$L_{A0} = \overline{S_1} \cdot S_0$$
$$L_{B1} = \overline{S_1}$$

Output	Encoding
green	00
yellow	01
red	10

FSM Output Table



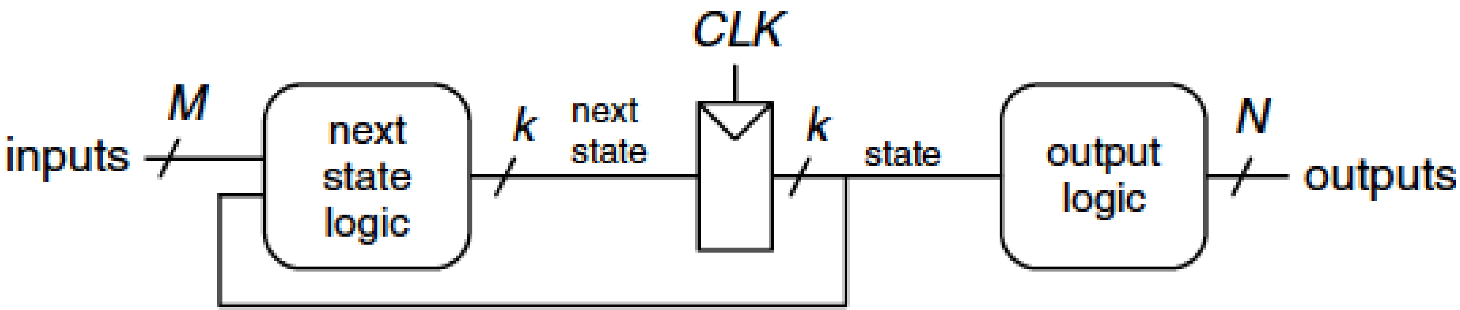
Current State		Outputs			
S_1	S_0	L_{A1}	L_{A0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

Output	Encoding
green	00
yellow	01
red	10

$$L_{A1} = S_1$$
$$L_{A0} = \overline{S_1} \cdot S_0$$
$$L_{B1} = \overline{S_1}$$
$$L_{B0} = S_1 \cdot S_0$$

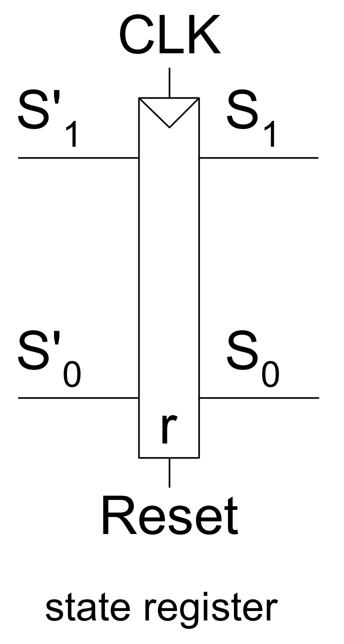
Finite State Machine: Schematic

FSM Schematic: State Register

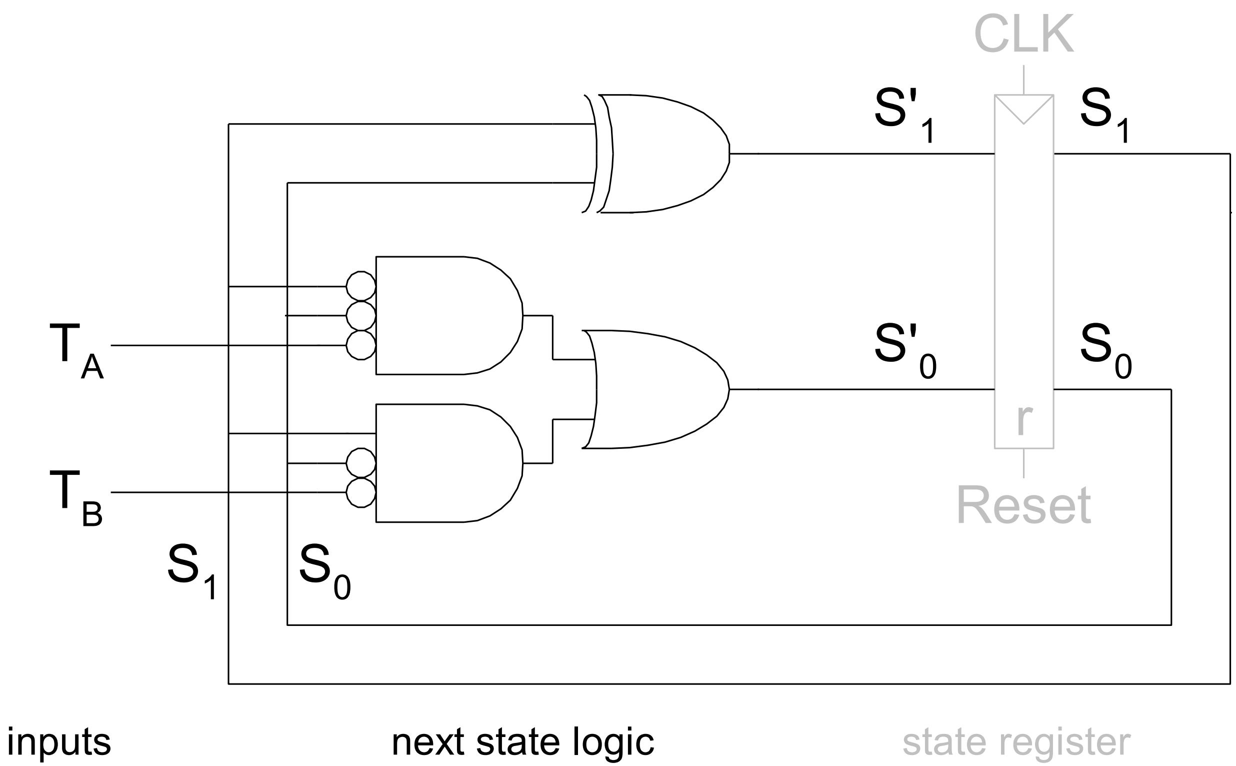


(a)

FSM Schematic: State Register



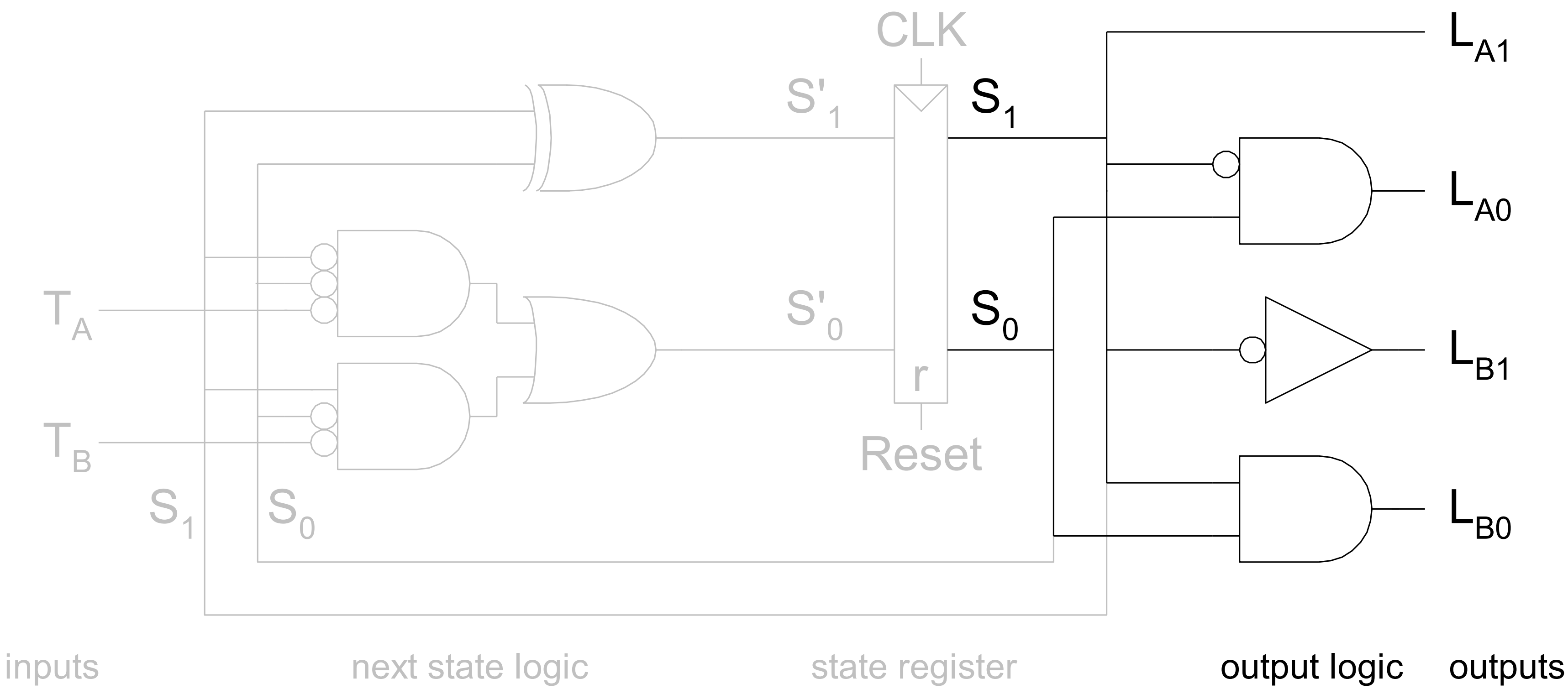
FSM Schematic: Next State Logic



$$S'_1 = S_1 \text{ xor } S_0$$

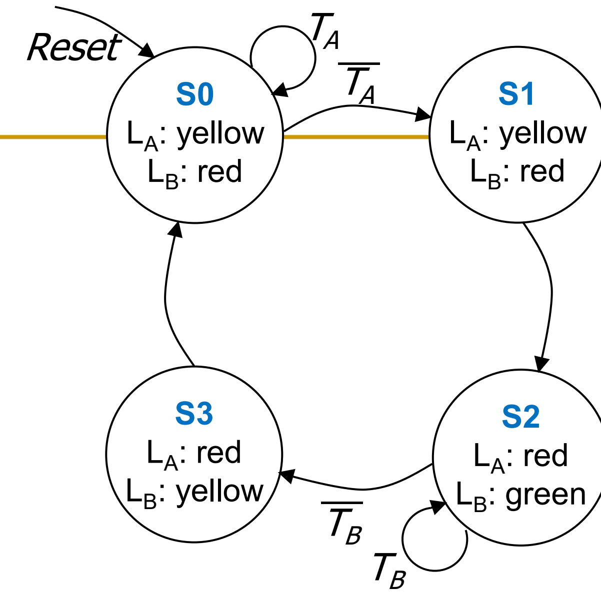
$$S'_0 = (\overline{S_1} \cdot \overline{S_0} \cdot \overline{T_A}) + (S_1 \cdot \overline{S_0} \cdot \overline{T_B})$$

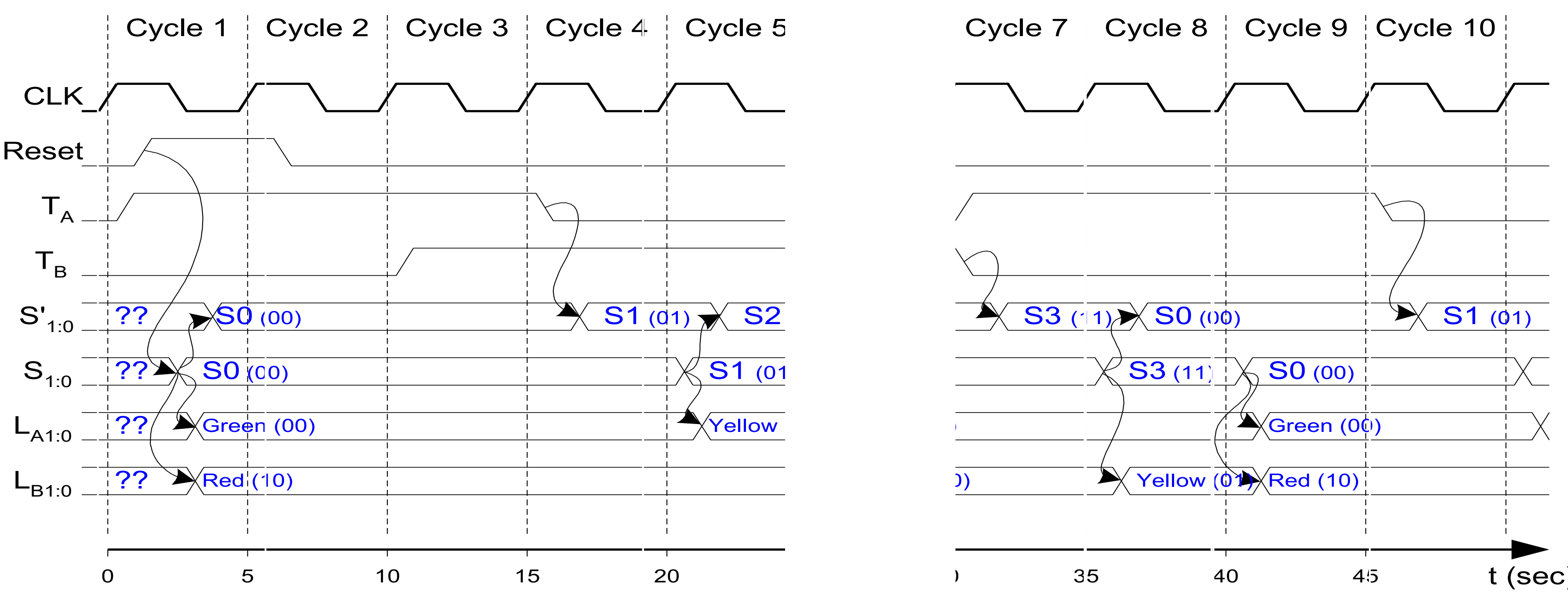
FSM Schematic: Output Logic



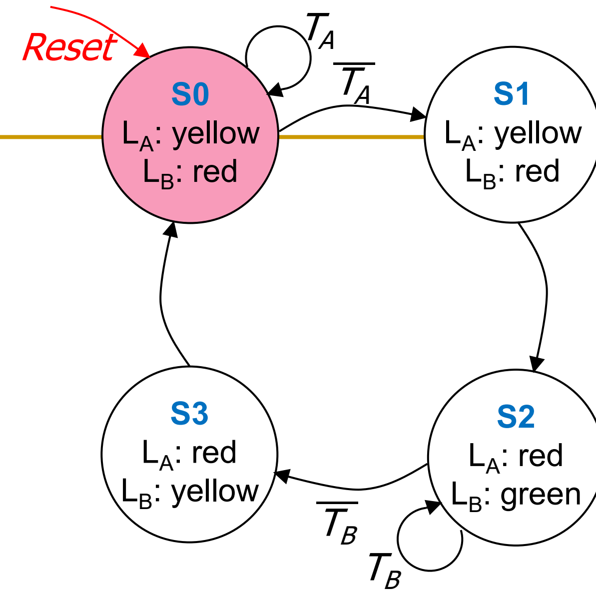
$$L_{A1} = S_1$$
$$L_{A0} = \overline{S_1} \cdot S_0$$
$$L_{B1} = \overline{S_1}$$
$$L_{B0} = S_1 \cdot S_0$$

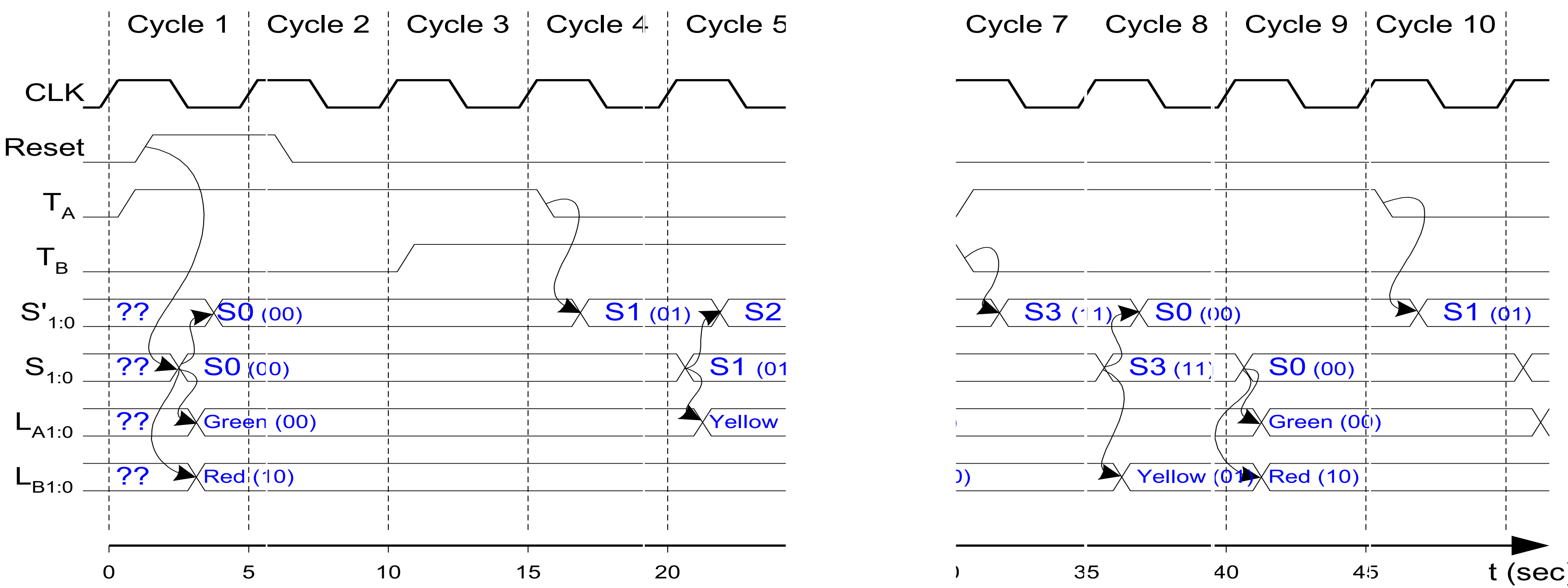
FSM Timing Diagram



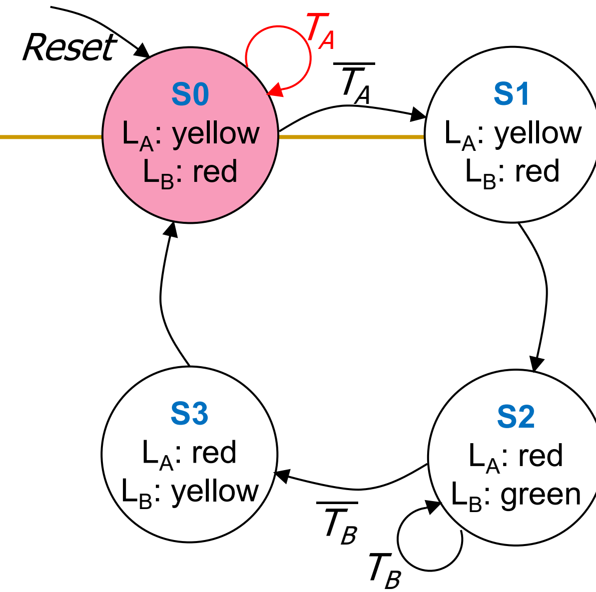


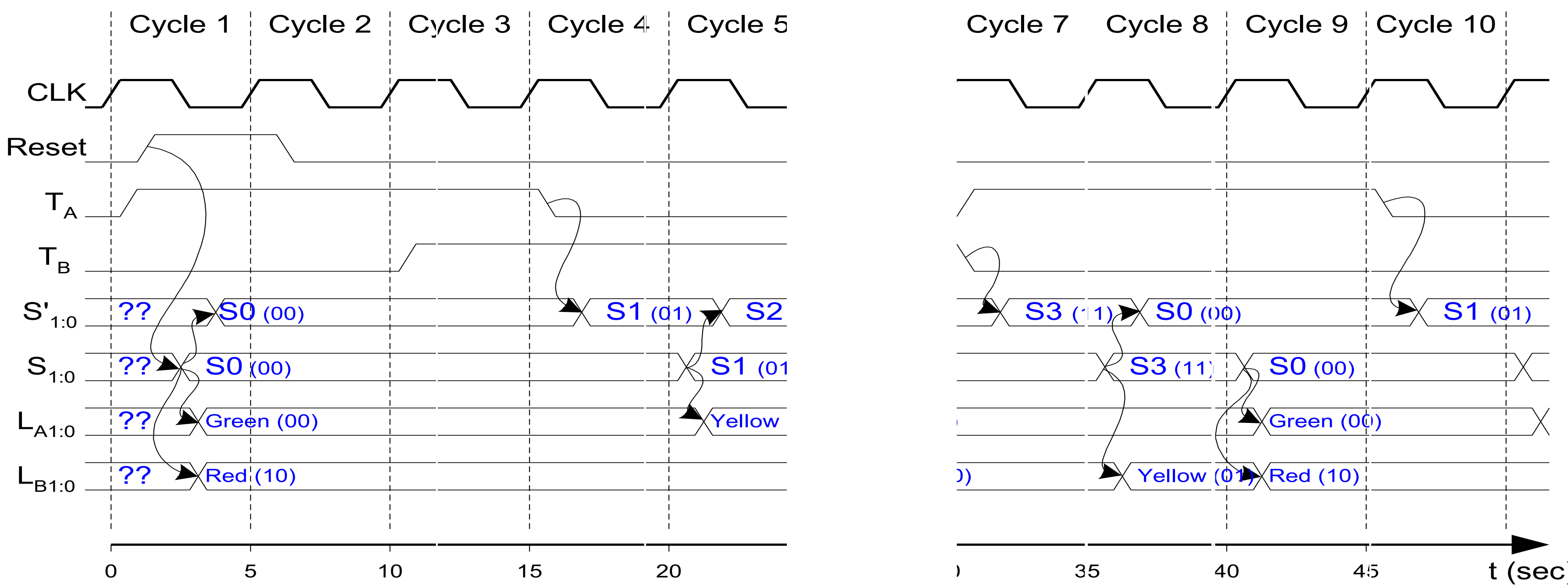
FSM Timing Diagram



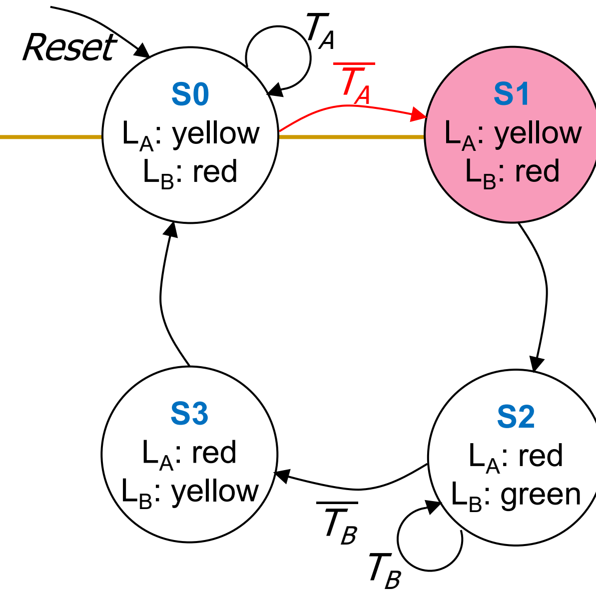


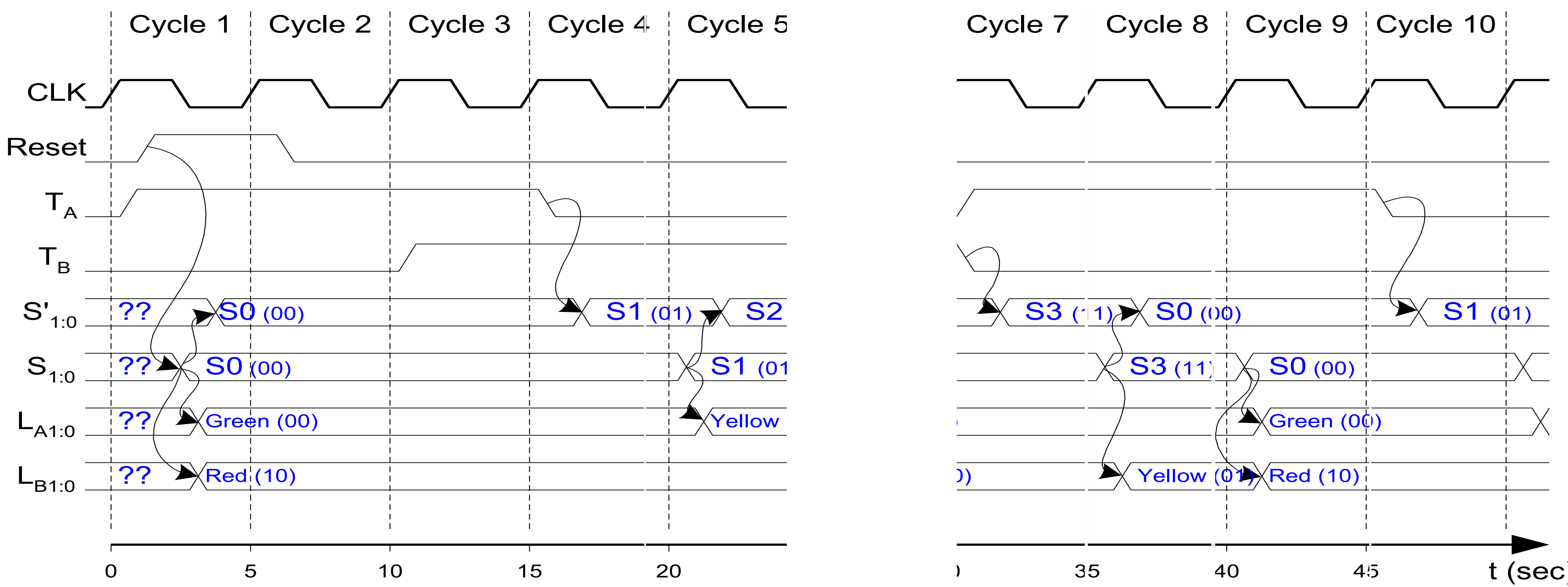
FSM Timing Diagram



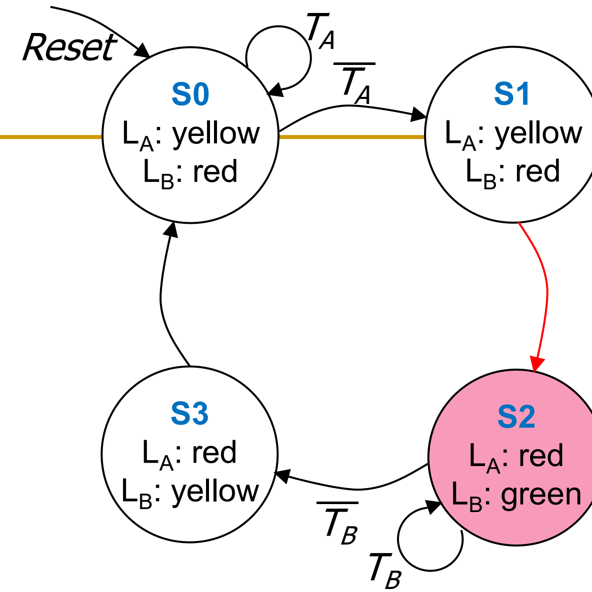


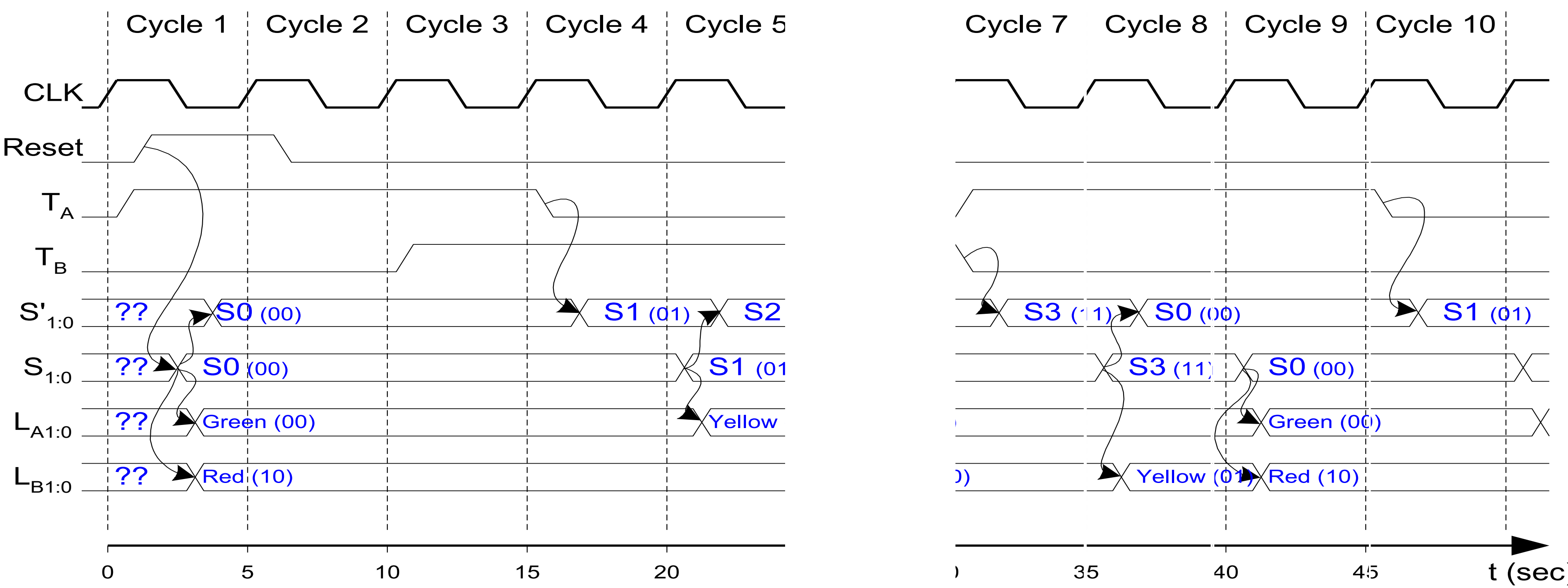
FSM Timing Diagram



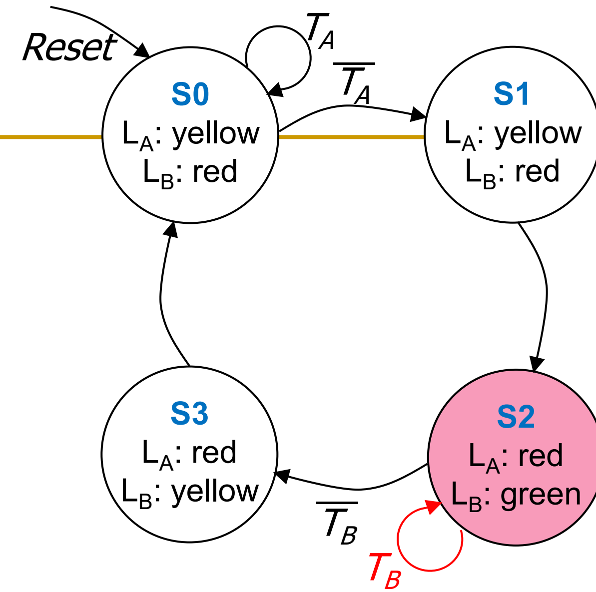


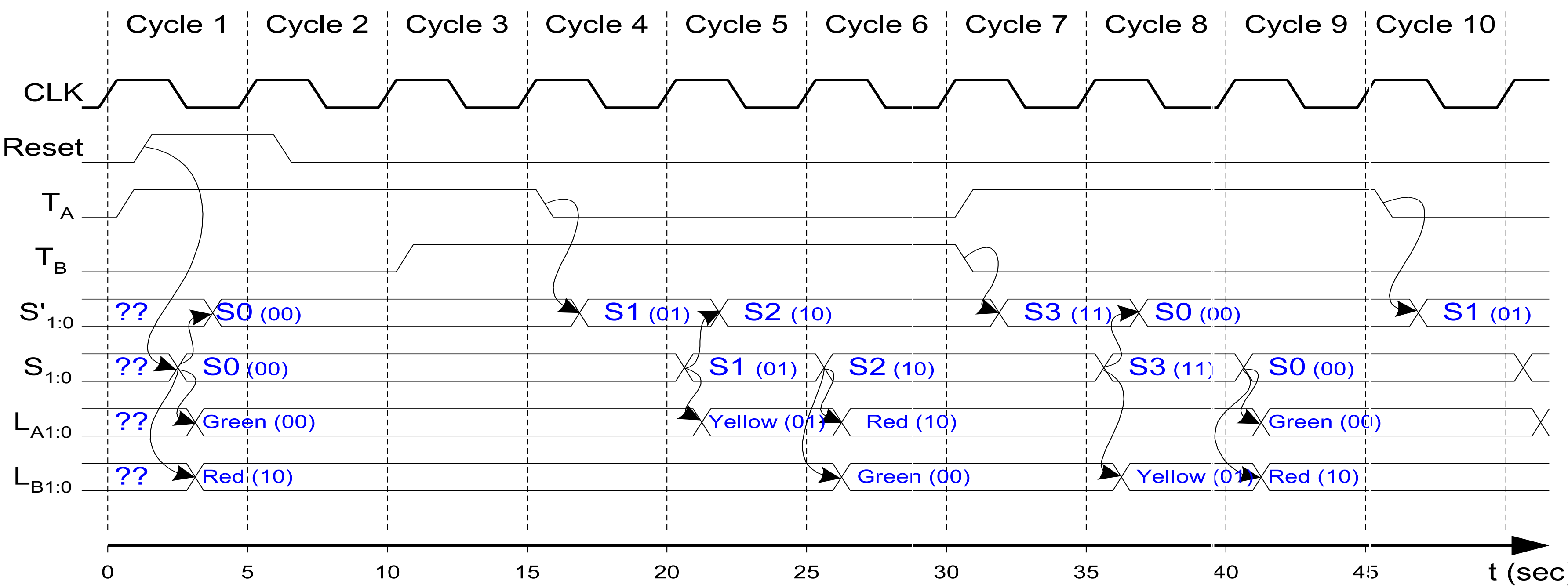
FSM Timing Diagram



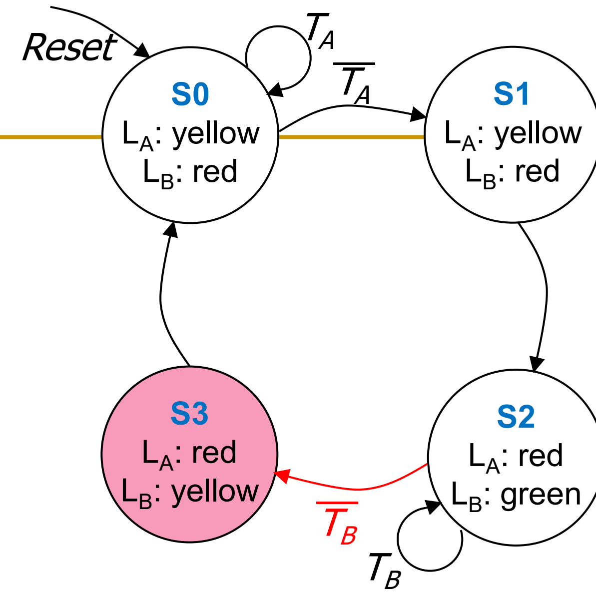


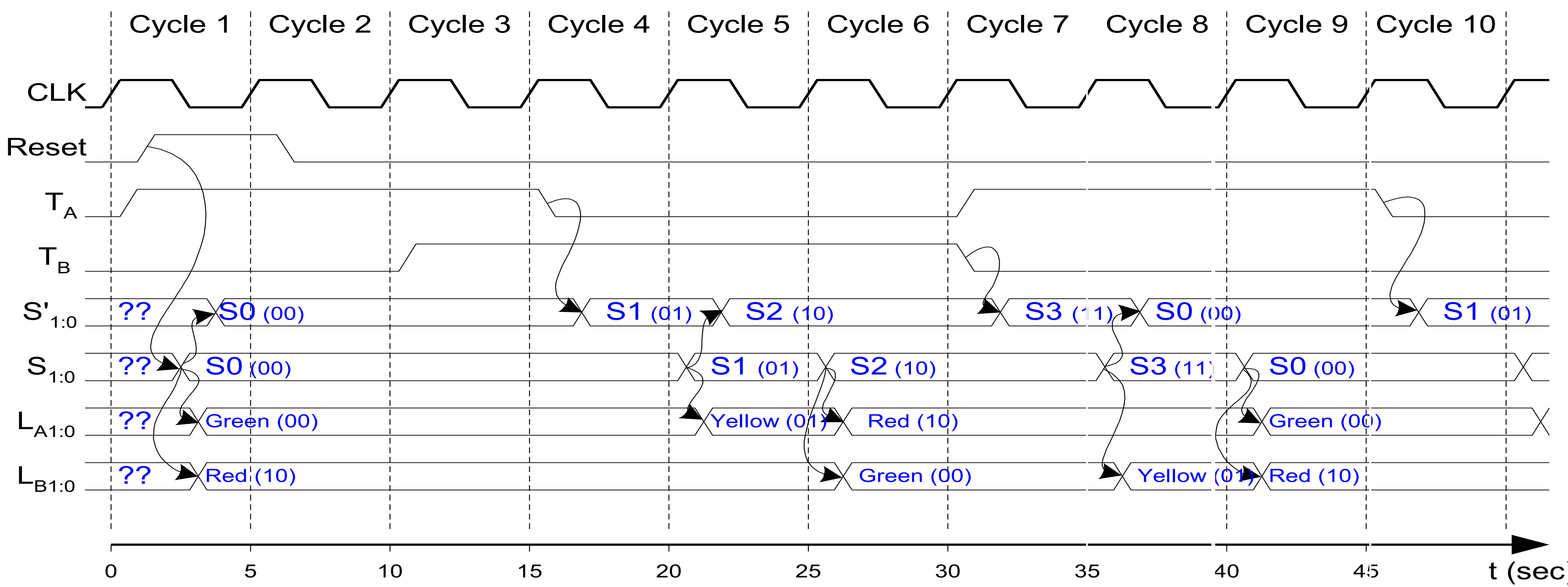
FSM Timing Diagram



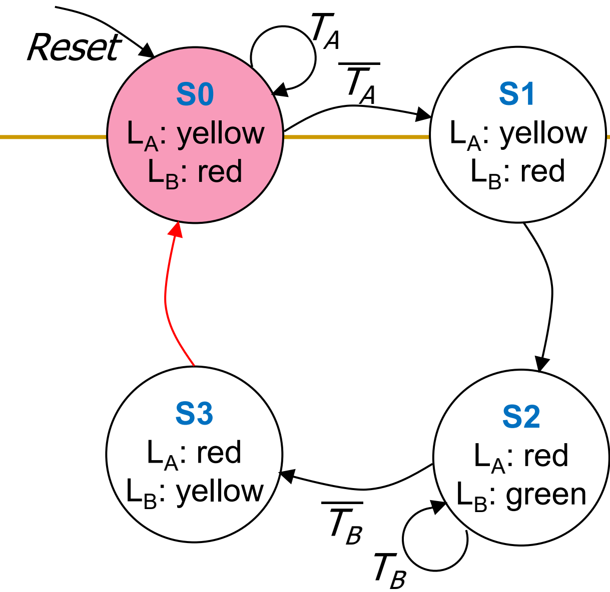


FSM Timing Diagram

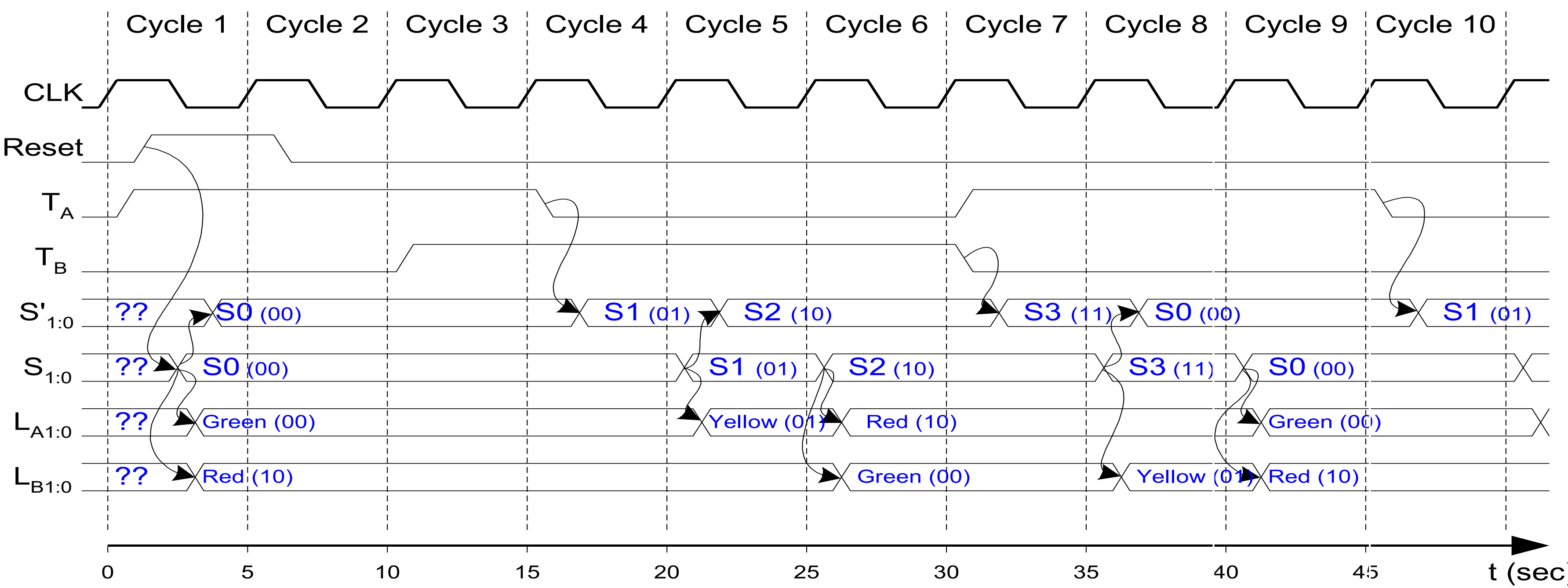




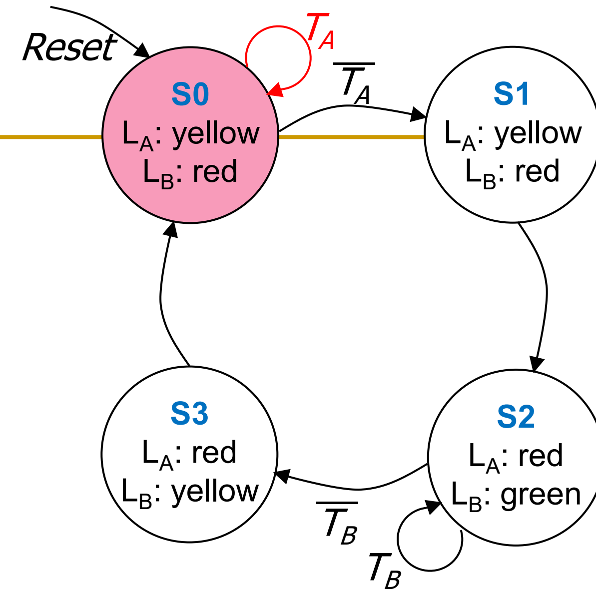
FSM Timing Diagram

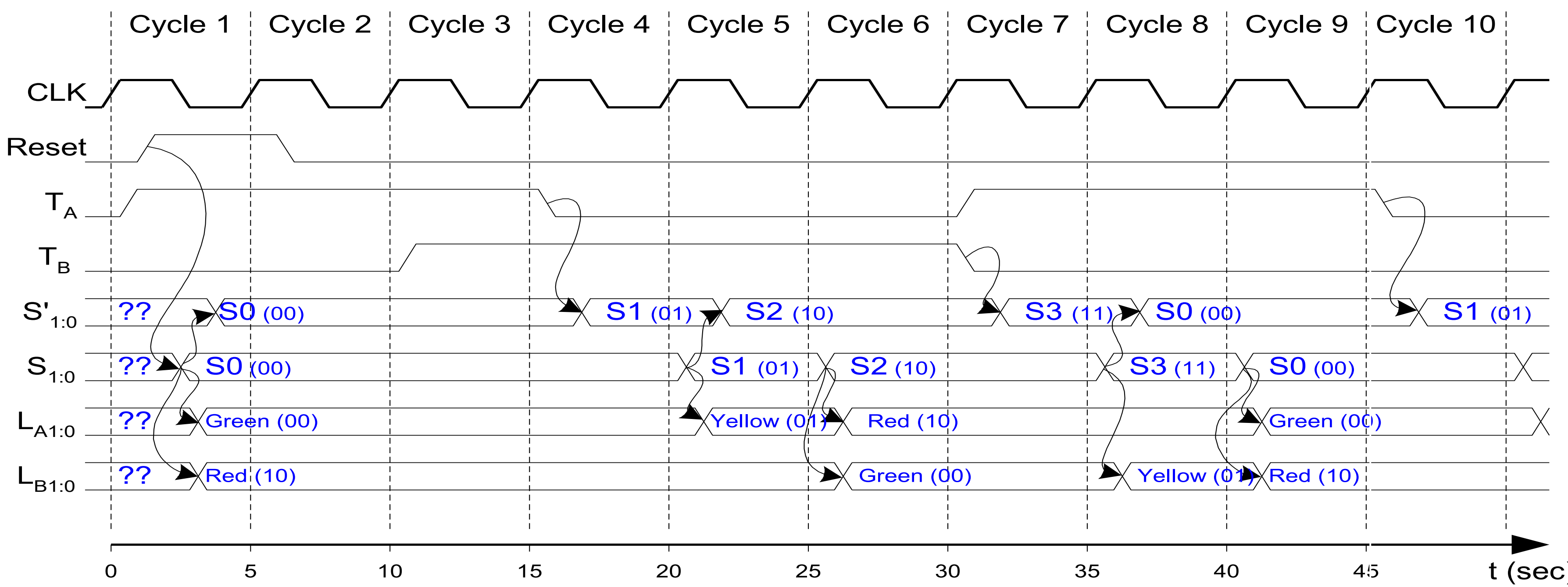


This is from H&H Section 3.4.1

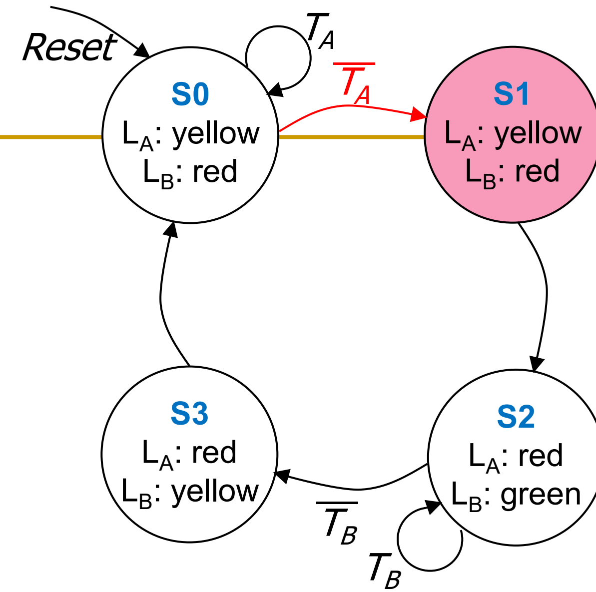


FSM Timing Diagram

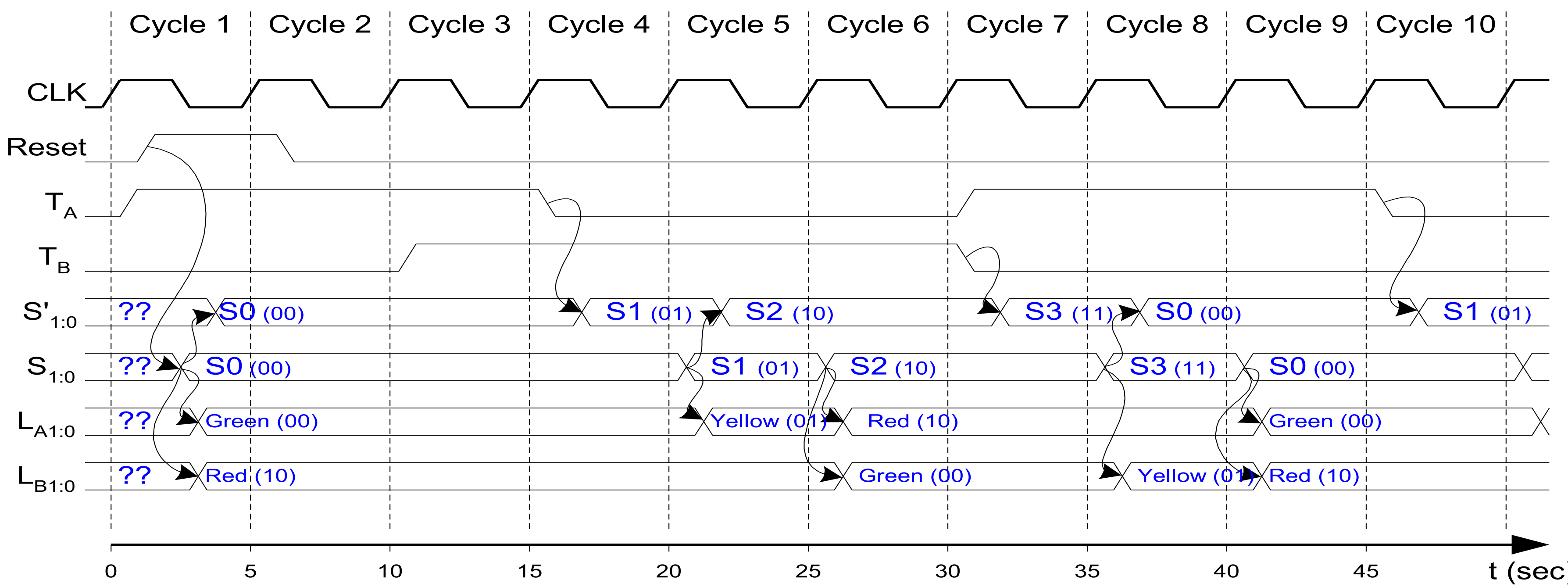




FSM Timing Diagram



See H&H Chapter 3.4

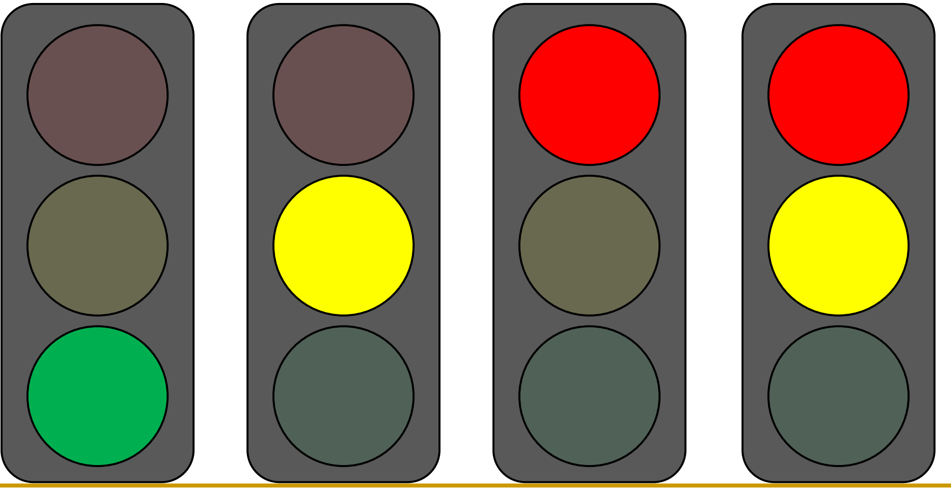


Finite State Machine: State Encoding

FSM State Encoding

- How do we encode the state bits?
  - Three common state binary encodings with different tradeoffs
    1. **Fully Encoded**
    2. **1-Hot Encoded**
    3. **Output Encoded**

- Let's see an example **Swiss** traffic light with 4 states
  - Green, Yellow, Red, Yellow+Red



FSM State Encoding (II)

1. **Binary Encoding (Full Encoding):**
   - Use the minimum possible number of bits
     - Use $log_2(num\_states)$ bits to represent the states
   - *Example state encodings:* 00, 01, 10, 11
   - **Minimizes** # flip-flops, but not necessarily output logic or next state logic

2. **One-Hot Encoding:**
   - Each bit encodes a different state
     - Uses *num_states* bits to represent the states
     - Exactly 1 bit is "hot" for a given state
   - *Example state encodings:* 0001, 0010, 0100, 1000
   - **Simplest design process** – very automatable
   - **Maximizes** # flip-flops, **minimizes** next state logic

FSM State Encoding (III)

3. **Output Encoding:**
   - Outputs are **directly accessible** in the state encoding
   - For example, since we have **3 outputs** (light color), encode state with **3 bits**, where each bit represents a color
   - *Example states:* 001, 010, 100, 110
     - Bit_0 encodes **green** light output,
     - Bit_1 encodes **yellow** light output
     - Bit_2 encodes **red** light output
   - **Minimizes** output logic
   - Only works for Moore Machines (output function of state)

FSM State Encoding (III)

3. Output Encoding:

- Outputs are **directly accessible** in the state encoding

The **designer** must **carefully** choose an encoding scheme to **optimize** the design under given constraints

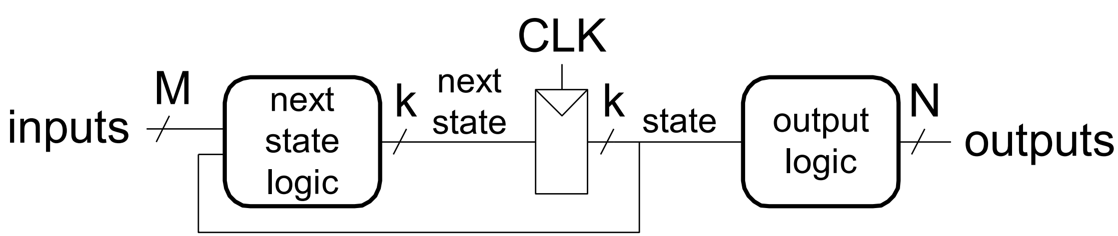
- **Minimizes** output logic
- Only works for Moore Machines (output depends only on state)

Moore vs. Mealy Machines

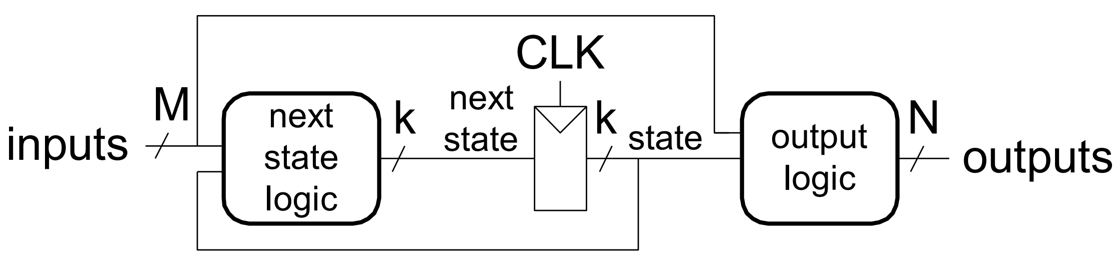
Recall: Moore vs. Mealy FSMs

- Next state is determined by the current state and the inputs
- Two types of FSMs differ in the **output logic**:
  - **Moore FSM**: outputs depend only on the current state
  - **Mealy FSM**: outputs depend on the current state and the inputs

Moore FSM

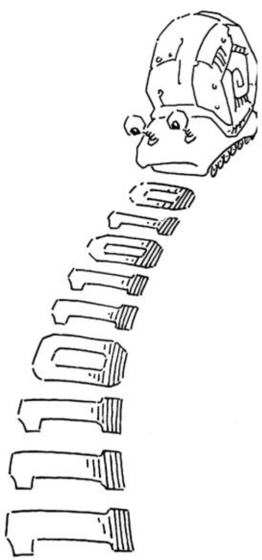


Mealy FSM

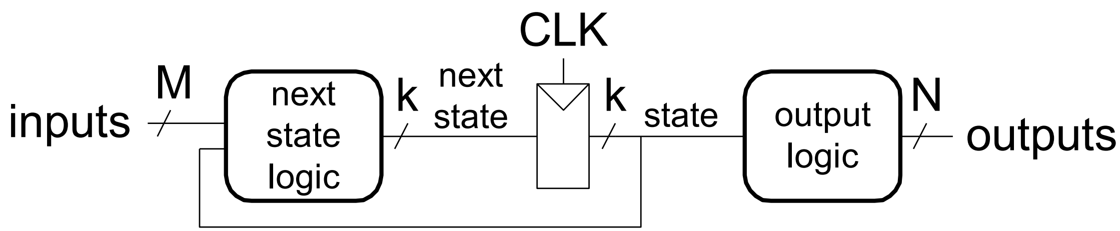


Moore vs. Mealy FSM Examples

- Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it.
- The snail smiles whenever the last four digits it has crawled over are 1101.
- Design Moore and Mealy FSMs of the snail's brain.

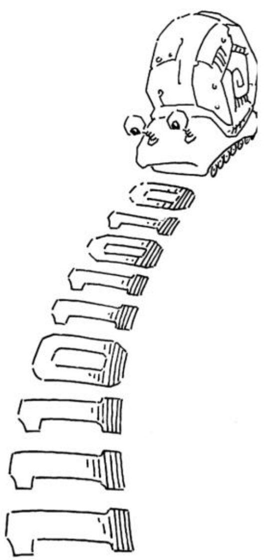


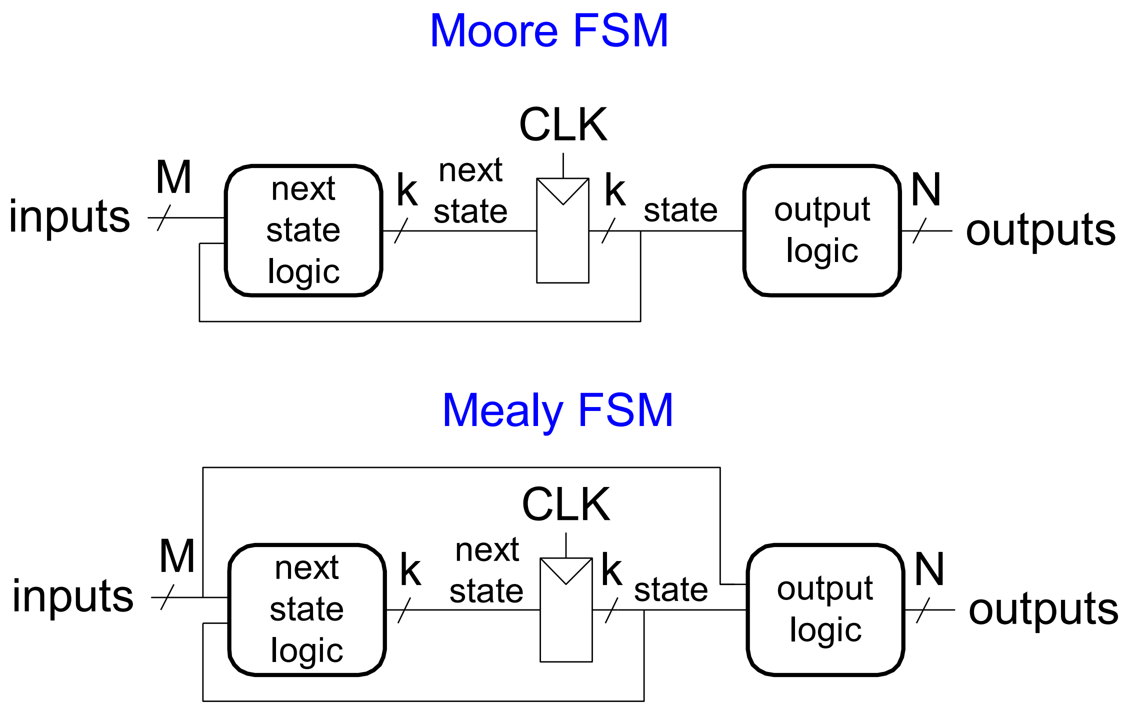
Moore FSM



Moore vs. Mealy FSM Examples

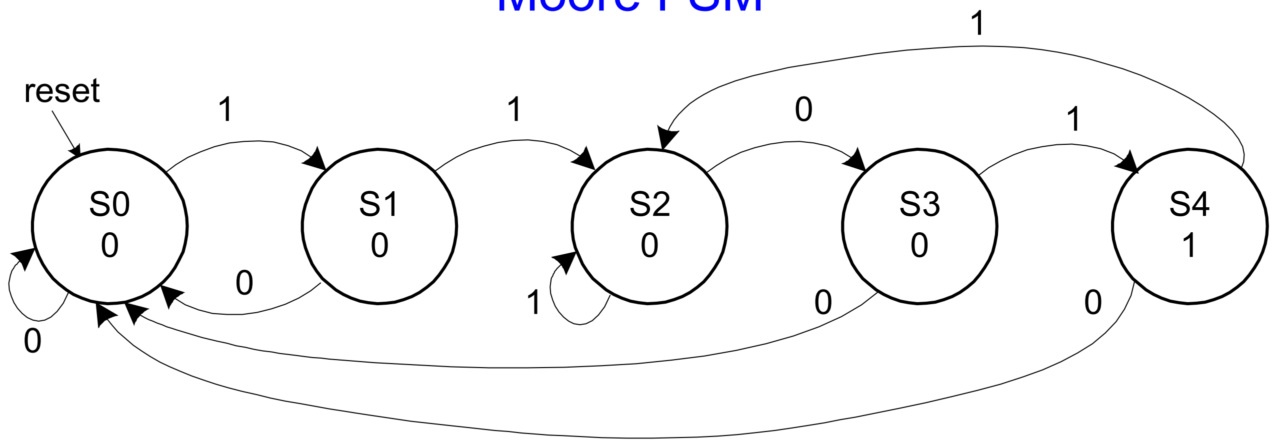
- Alyssa P. Hacker has a snail that crawls down a paper tape with 1's and 0's on it.
- The snail smiles whenever the last four digits it has crawled over are 1101.
- Design Moore and Mealy FSMs of the snail's brain.





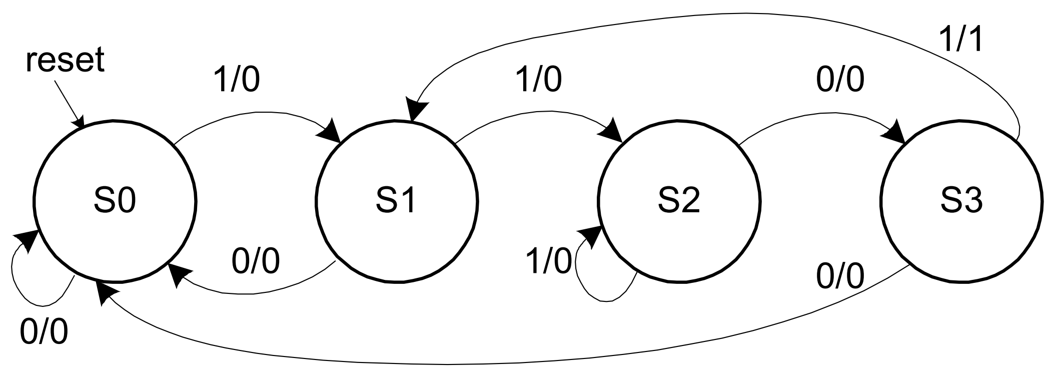
State Transition Diagrams

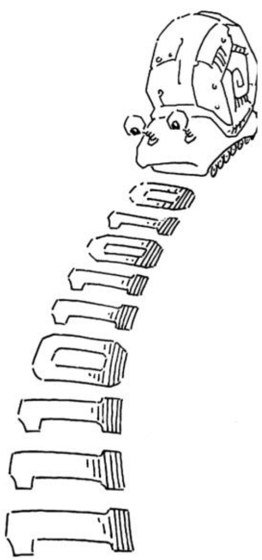
Moore FSM



What are the tradeoffs?

Mealy FSM





FSM Design Procedure

- **Determine** all possible states of your machine

- **Develop** a **state transition diagram**
  - Generally this is done from a textual description
  - You need to 1) determine the **inputs** and **outputs** for each **state** and 2) figure out how to get from one state to another

- **Approach**
  - Start by defining the **reset state** and what happens from it – this is typically an easy point to start from
  - Then continue to add **transitions** and **states**
  - Picking **good state names** is very important
  - Building an FSM is **like** programming (but it *is not* programming!)
    - An FSM has a sequential "control-flow" like a program with conditionals and goto's
    - The if-then-else construct is controlled by one or more inputs
    - The outputs are controlled by the state or the inputs
  - In hardware, we typically have many concurrent FSMs

What is to Come: LC-3 Processor

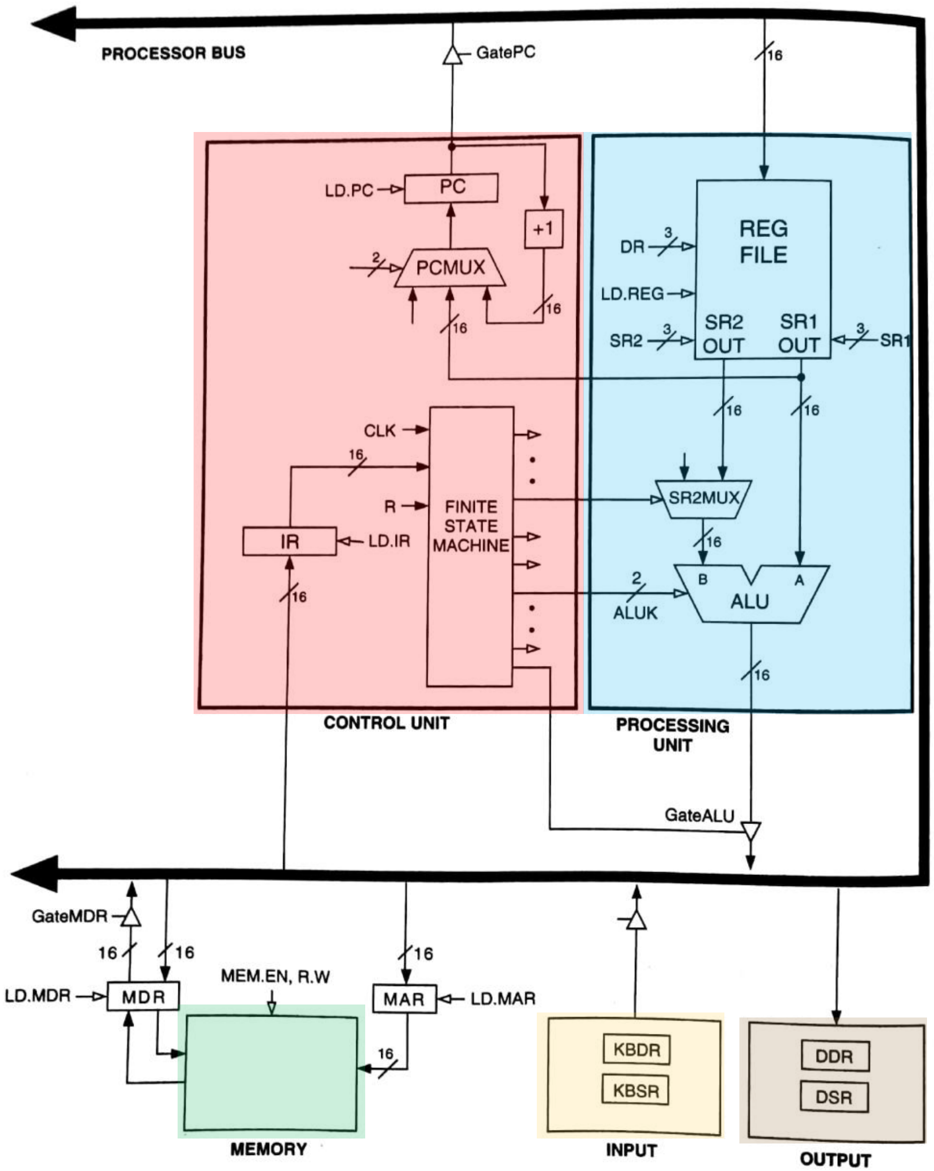
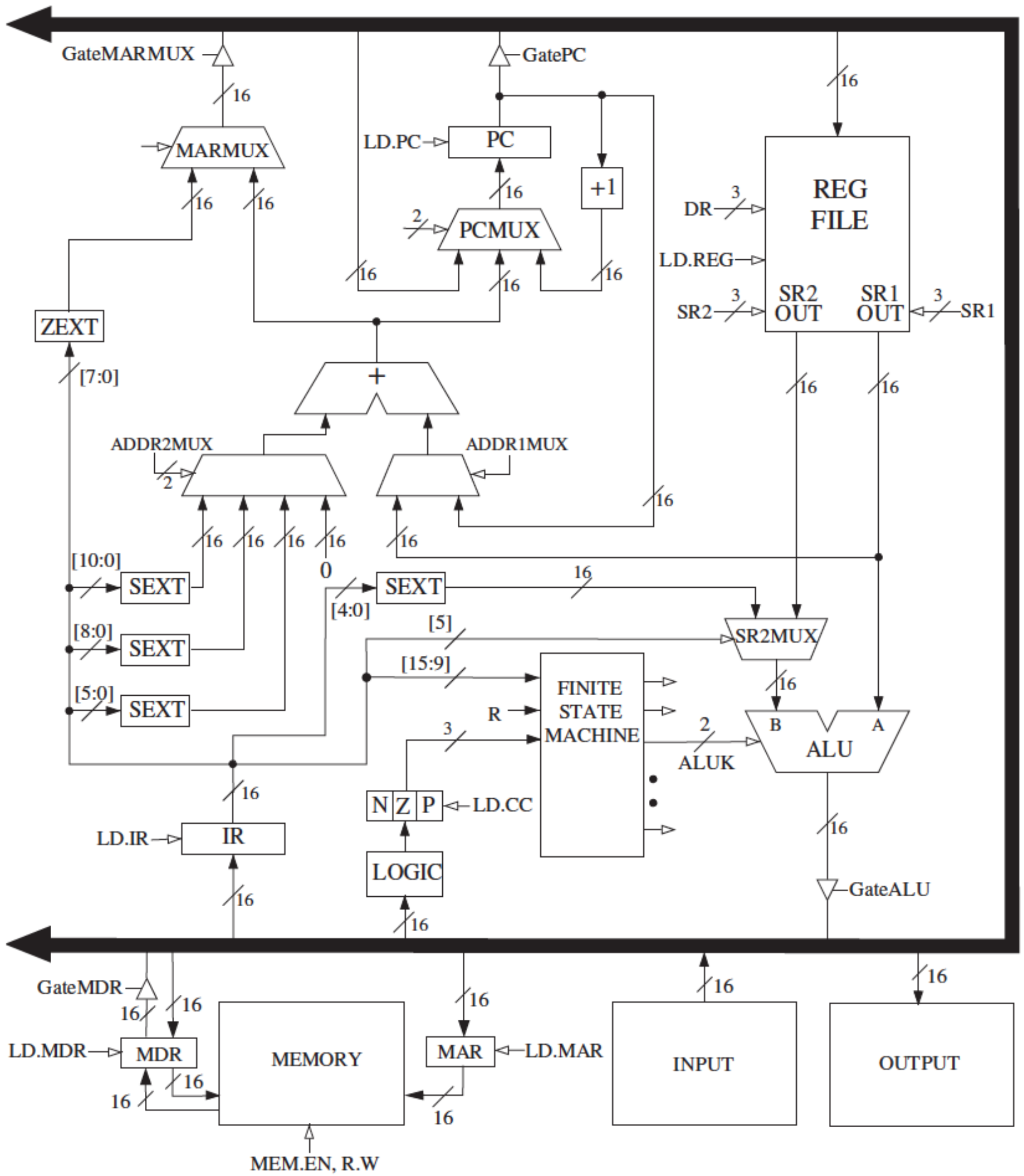


Figure 4.3 The LC-3 as an example of the von Neumann model

What is to Come: LC-3 Datapath



Digital Design & Computer Arch.

Lecture 3: Combinational Logic II and Sequential Logic

Prof. Onur Mutlu

ETH Zürich

Spring 2026

26 February 2026