Sequential Logics / Flip-Flops

Some Circuits Need Storage

- So far, all of the logic circuits we have talked about have been **combinational**
  - Circuit outputs **only** depend on **current** input values
- But many digital logic circuits we use need to keep track of past information

Sequential Circuits

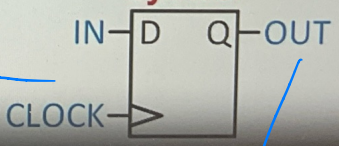
- Some systems need to store information about past behavior to know how to react to inputs
  - Generally not storing ***everything*** that happened, just some key information
  - A vending machine tracks **how much** money has been entered so it knows when it is "enough" to dispense an item
    - Not which coins or in what order

- Sometimes just need to store data for later access
  - **Examples:** audio files, digital photos, emails, student grades, bank balances…

Storing a Single Bit

- There are a variety of structures used in digital circuits to store a single bit of information
  - These vary in size, speed, complexity, and behavior
- We focus on one structure: the D flip-flop
  - It has one bit of data input, one bit of data output
  - It also has a "clock" input that controls **when** the value carried by the input signal is stored into the flip-flop
    - The value held by a flip-flop cannot change until the clock allows it (or power is lost to the system…)
  - The flip-flop's output is always **equal** to the value it is storing
    - The output updates when a new value is stored into the flip-flop

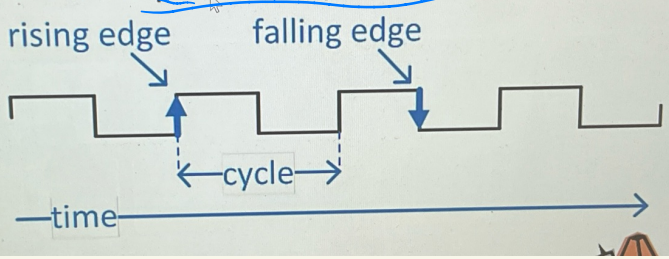
Flip-Flop Symbol



★ current value "stored"

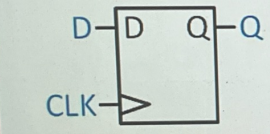
Clock Signal

- A clock is a special signal that oscillates between 1 and 0 at a specific *frequency*

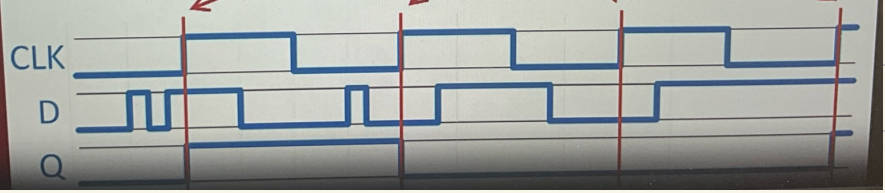


Flip-Flop Behavior

- This D flip-flop updates when the clock transitions from 0 to 1 (rising edge)

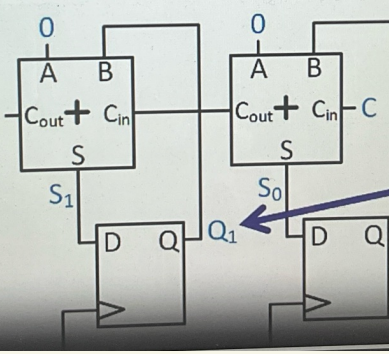


The flip-flop input is **only** stored at the rising clock edge ("triggering" edge)



A Simple Sequential Circuit

- The circuit has a 2-bit adder and 2 flip-flops
  - The flip-flops hold a 2-bit binary number Q_1Q_0
  - The adder implements the arithmetic operation
    $S_1S_0 = Q_1Q_0 + 00_2 + C$
  - At each triggering clock edge, the flip-flops store S_1S_0



At the clock edge, **Q** is always overwritten with the value of **D**.

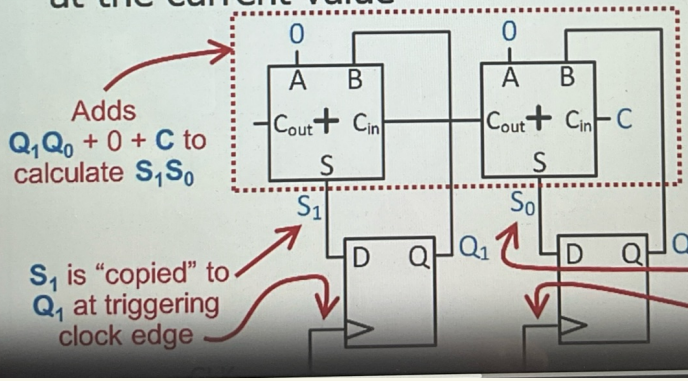
A flip-flop can only store a single bit of information!

Counter Circuit

- When **C**=1, the circuit counts 00→01→10→11→00→…
  - The current count value is Q_1Q_0
  - The next count value is S_1S_0
- When **C**=0, the count will hold at the current value

Current State: value ***currently*** stored in flip-flops

Next State: value that will be stored in flip-flops at ***next*** triggering clock edge

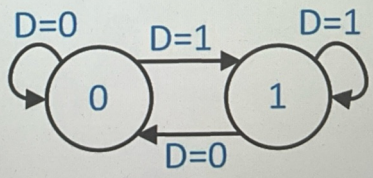


Finite State Machine

- Any practical sequential circuit is a Finite State Machine (FSM)
  - State: current "status" → the contents of the flip-flops
    - **Examples:** Current traffic light color, current score of a game, current floor of an elevator, …
  - Since there can only be a finite number of flip-flops in any machine, the machine only has a finite number of possible states it can be in, hence the name
- Any circuit built from flip-flops is an FSM
  - The counter circuit we just saw is an FSM
  - Even a single flip-flop is an FSM!

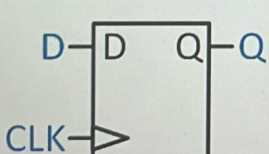
State Diagram

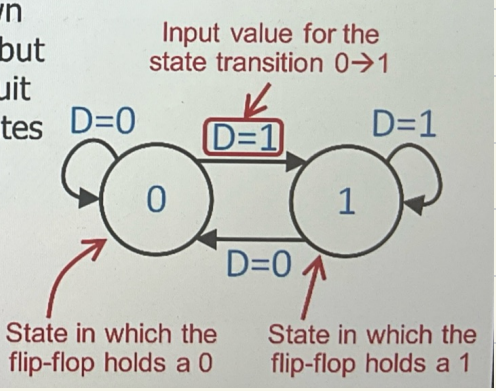
- A graphical representation of FSM behavior
  - State: current "status" → the contents of the flip-flops
    - Each possible state is represented by a circle
      - **N** flip-flops can represent 2^N different states
    - State circles also indicate the output(s) in that state
      - Sometimes the output is equal to the state (i.e. the counter)
  - State transitions
    - Represented by an arrow from one state circle to another
    - Labeled with the input condition that causes the FSM to make that transition



State Diagram of a Flip-Flop

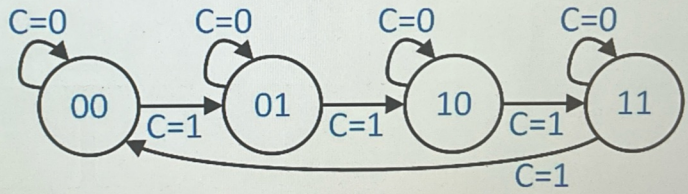
- A single flip-flop has 2 states (0 and 1)
- The flip-flop has a single input **D**
- The flip-flop transitions between states at the triggering clock edge based on the value of **D**
  - The clock is NOT shown on the state diagram, but governs when the circuit can move between states

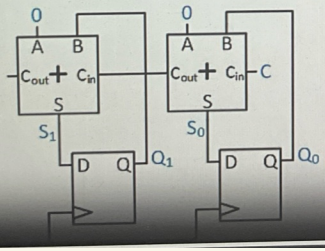


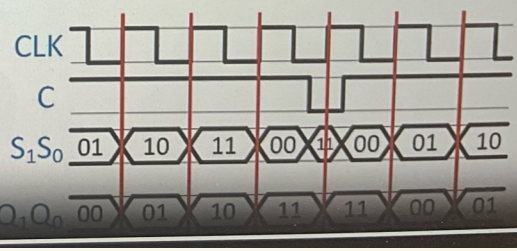


Counter State Diagram

- Four states (one per counter value)
  - Transition back to the same state if **C**=0
  - Transition to the state for the next count value if **C**=1





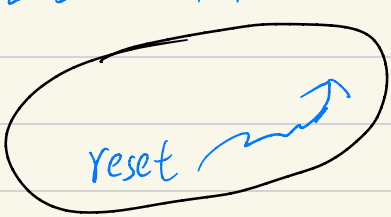


Finite State Machines

- The FSM's behavior depends both upon the current state and the input(s)
  - The next state is a function of the current state and current input values
  - In this class, the output is a function only of the current state

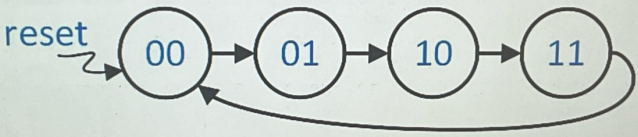
FSM contains combinational circuits to compute these

(☆ Normally, the state has a beginning to start — Labeled with "reset")



Common Shorthand Notation

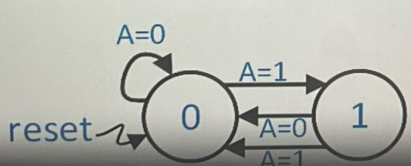
- Some state machines do not have an input, or the input does not affect a particular state transition
  - **Example:** a 2-bit counter that always counts



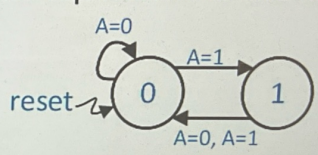
  - There is no input that controls what happens here
  - At each triggering clock edge, the circuit transitions to the next state (which represents the next count value)
- This type of transition—one not influenced by the input value(s)—is an **unconditional** transition

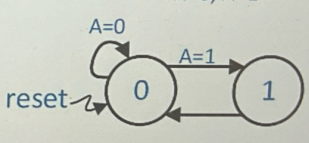
Example: Shorthand Notation

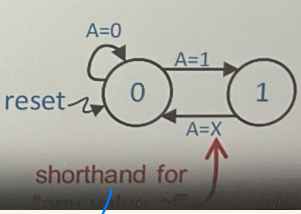
- A two-state FSM
- It starts in State 0
- If **A**=1 in State 0, it goes to State 1…
- …for one cycle before returning to State 0 (regardless of **A**)
- [if **A**=0 in State 0, it stays in State 0]



- All of the below diagrams are equivalent to the first





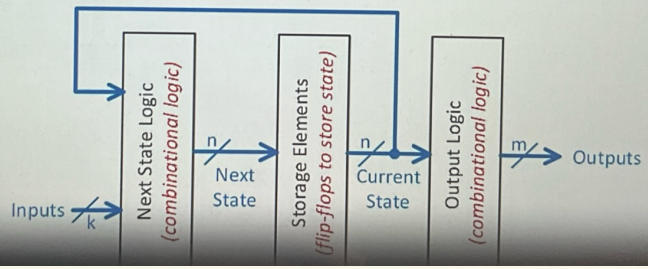


"Any value of"

How Do We Implement FSMs?

- Implementing an FSM involves several steps
  - Include flip-flops needed to store the state
  - Design the logic that computes the next state based on the current state and the inputs of the circuit
  - Design the logic that computes the output based on the current state

These are combinational circuits!



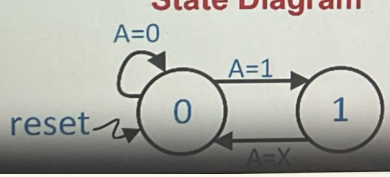
State Table

- Just like we use a **truth table** to characterize a **combinational** circuit, we use a **state table** to characterize a **sequential** circuit
  - **Example:**

The output (we will name it **Y**) is equal to the state

We will name the current state signal **S**

State Diagram

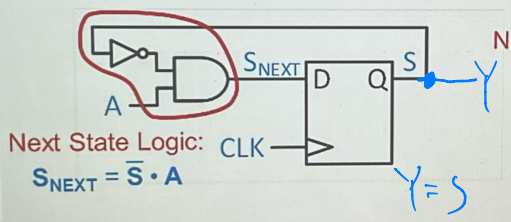


State Table

Current State S	Input A	Next State S_{NEXT}	Output Y
0	0	0	0
0	1	1	0
1	0	0	1
1	1	0	1

FSM Hardware

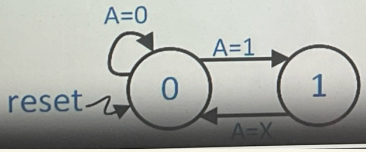
- What does the circuit look like?
- How do we use the state table to design it?



Inputs of the Next State Logic

Output of the Next State Logic

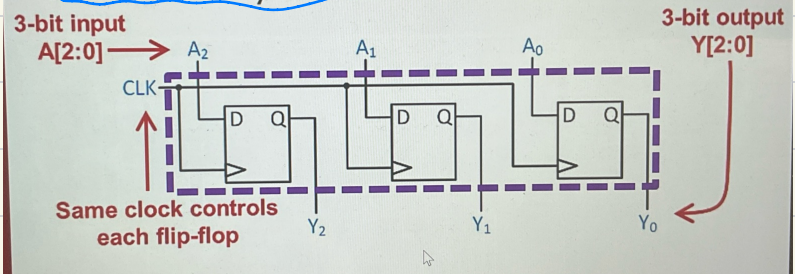
Current State S	Input A	Next State S_{NEXT}	Output Y
0	0	0	0
0	1	1	0
1	0	0	1
1	1	0	1



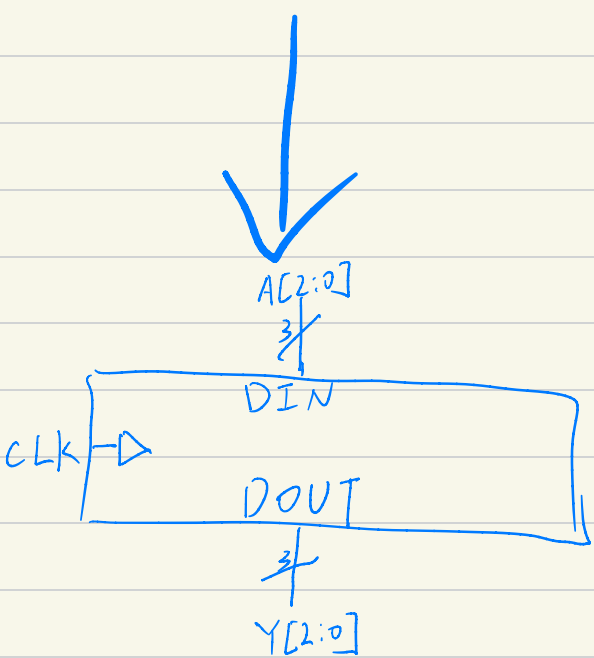
Registers and Memory

Registers

- A register is a group of flip-flops used to store multi-bit binary values

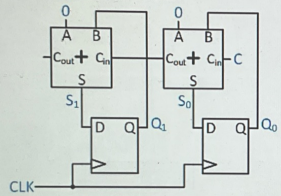


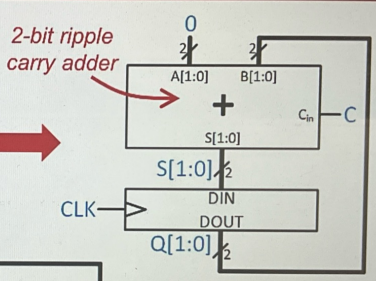
- We usually do not show each flip-flop in a register...



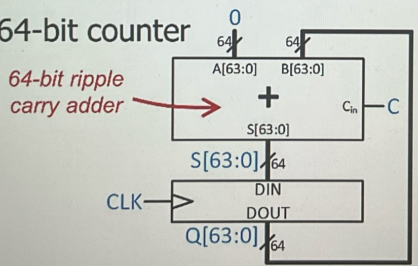
Example: Multi-Bit Counters

- 2-bit counter





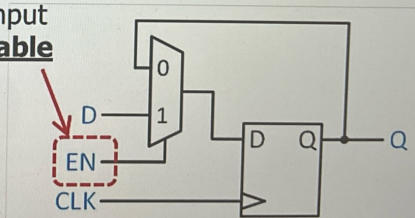
- 64-bit counter

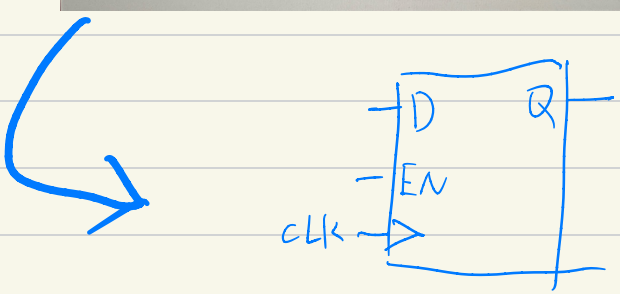


Enable

- Sometimes we want a flip-flop to "hold" its value for multiple clock cycles, regardless of its input
  - But a flip-flop **always** stores the input value at the triggering edge of the clock…
- Solution: add logic so that we can choose to store a new value or the value already in the flip-flop
  - Selection based on the value of a special input signal called an **enable**

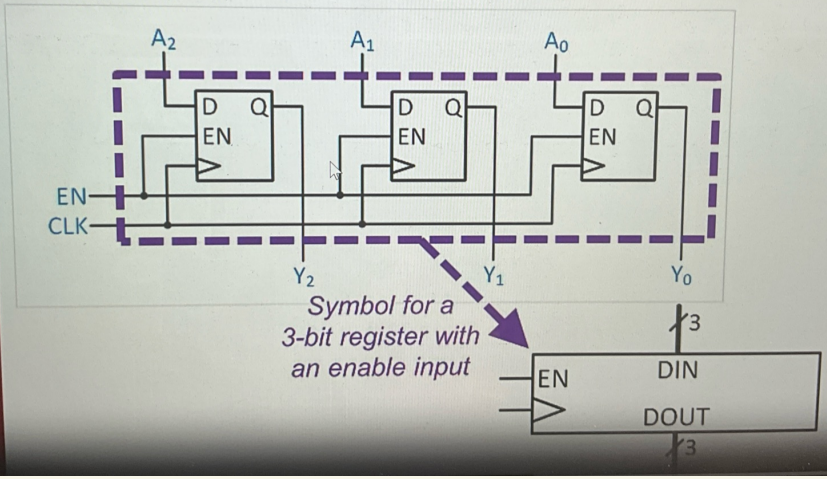
EN=1 → choose **D**
EN=0 → choose **Q**





Register With Enable

- We can also add "enable" capability to registers
  - Every flip-flop in the register shares a common enable



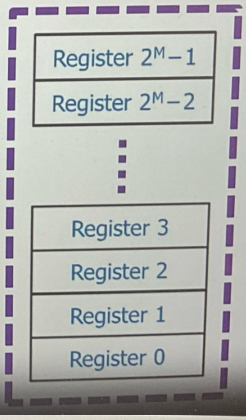
Memory Concepts

- A memory has locations where data is stored
- Each location has a unique address
  - An M-bit number that specifies which of the 2^M locations to access

- Each location stores a single N-bit word
  - N is the data size of the memory (# bits per location)
- The capacity of a memory is the total number of bits that it stores: (# locations) × (data size) = $2^M \times N$
- Only one location can be accessed at a time, either to write (store) a value or read a value
  - A write enable signal (**WE**) controls read vs. write

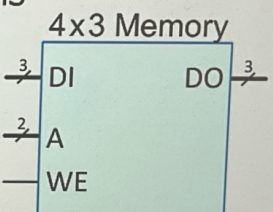
A Logical View of Memory

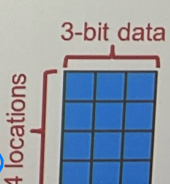
- A memory is conceptually like an array of registers
- To access a register, we index into the array using the address for that register
- To write a value, we supply the data to be stored and the address of the location
  - Also, set **WE** to 1 for a write...
- To read a value, we supply the address of the location
  - Also, set **WE** to 0 for a read... (i.e., do not write)



Example Memory

- Design first as a "black box"
  - Just the symbol—no inner details are shown
- Build a memory with four locations and a 3-bit data word size
  - Four locations:
    - 2-bit address → **A[1:0]**
  - 3-bit data:
    - Data in must be 3 bits → **DI[2:0]**
    - Data out must be 3 bits → **DO[2:0]**
  - Need to control read/write:
    - Write enable signal → **WE**
- The capacity of this memory is:
  (# locations) × (data size) = 4 × 3 = 12 bits

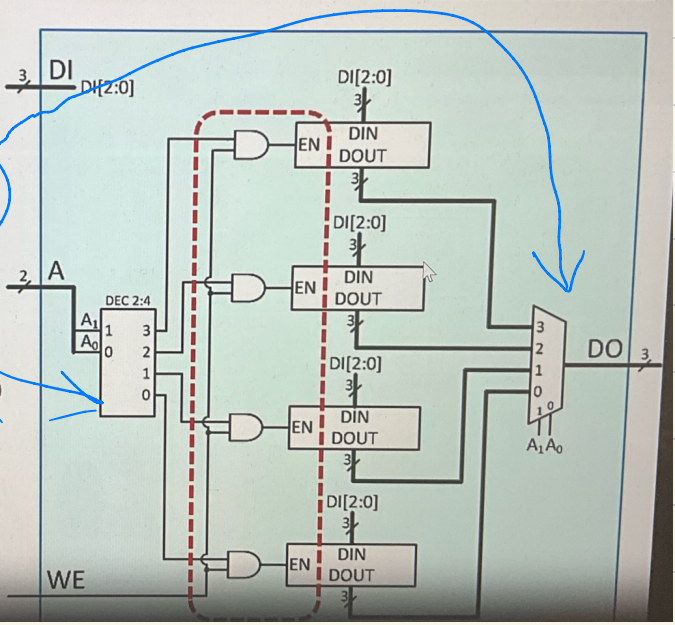




$2^2 \times 3 = 12$

Example Memory

- Use four 3-bit registers
  - Clock not shown
- Address selects which register supplies data out
- Address selects which register may be written to
- Selected register will **only** be written if WE=1



Example: Memory Read

INPUTS

XXX

data input does not matter

10

address (location) to read

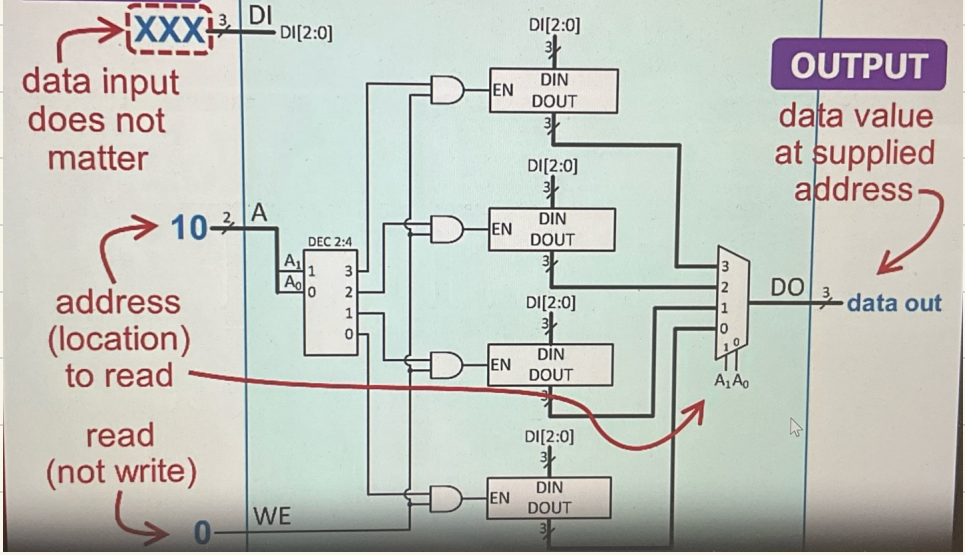
read (not write)

0

OUTPUT

data value at supplied address

data out



Example: Memory Write