Logic Design (Part 3)
Sequential Logic
(Chapter 3)

So Far: Combinational Logic

Combinational Logic:
• Always gives the same output for a given set of inputs
• Aka “state-less” (i.e., no “state” or “memory”)

Sequential Logic:
• Its output depends on its inputs & its last output!
• Forms the basis for “state” or “memory” for a computer

Combinational vs. Sequential

Combinational Circuit
• always gives the same output for a given set of inputs
  ➢ ex: adder always generates sum and carry, regardless of previous inputs

Sequential Circuit
• stores information
• output depends on stored information (state) plus input
  ➢ so a given input might produce different outputs, depending on the stored information
• example: elevator
  ➢ Current floor increases when you go up
  ➢ output depends on previous state and ‘inputs’ (request for current floor)
• useful for building “memory” elements and “state machines”

Finite State Machines

The behavior of sequential circuits can be expressed using characteristic tables or finite state machines (FSMs).
• FSMs consist of a set of nodes that hold the states of the machine and a set of arcs that connect the states.
• Directed graph to represent a FSM

Moore and Mealy machines are two types of FSMs that are equivalent.
• They differ only in how they express the outputs of the machine.
• Moore machines place outputs on each node/state
  ➢ Associate an output with each state
• Mealy machines present their outputs on the transitions.
Example: A Vending Machine
• Accept user input (coins), when total is at least 75 cents dispense output (soda)
• Input valid coins:
  • Q (25 cents) D (10) or N (5)
• What should it keep track of?
  • current total
  • Is it 75 cents or more?
• When it reaches 75 or more:
  • Generate output

• States of the machine?

Design a Counter: counts from 0 to 7

Finite State Machine Representation of Counter

Bubbles represent all possible "states" for the machine (aka your flip-flop based circuit)
Arrows show movement from one state to the next
Transitions occur at pulse of the clock

State Machine
type of sequential circuit
• Combines combinational logic with storage
• "Remembers" state, and changes output (and state) based on inputs and current state

Inputs
State Machine
Combinational Logic Circuit
Storage Elements
Outputs
Sequential Logic
- Where do we start:
  - Build a device, using combinational logic devices, to store a value
    - RS Latch (also called SR Latch)
    - concept of memory
  - Build it using the devices we have thus far
    - How? Use “feedback” circuit
- What is the methodology behind design of sequential logic circuits
  - Finite State Machines
  - Example of Vending machine
- Combine sequential and combinational logic devices to “assemble” a simple processor!

Feedback Circuits
- What happens if we feed the output of a combinational logic circuit to an input in the circuit?
  - This is the key to circuits that can store values!
- Stable circuit
  - Output point of circuit retains value indefinitely
- Unstable circuit
  - State that remains constant only for a duration of a few gate delays

Feedback circuits
- download Sequential Circuits: sequential1.cdl
  - Open in Cedar Logic
- Page 1 of Sequential Circuits: Example 1
  - How do the two circuits behave?
Latches and Flip-Flops

Latch: basic circuit for storage
- Operate on changes in Level (i.e., 1 or 0)

Flip-flop:
- Sequential circuits take input from output of storage
- Latches that work on change of level can lead to unstable sequential circuits
  - As level changes the outputs change --- inputs change!
- Flip-Flop circuits designed to operate properly when they are part of a sequential circuit

Most Basic Sequential Logic Circuit: **R-S Latch**

Most fundamental unit for static memory
- Has the ability to “store” its last output

R-S Latch – Cross-Coupled NAND gates
- Output of each NAND gate serves as input to the other
- Two inputs: S (SET) & R (RESET)
- Two outputs: Q and NOT(Q) \(\text{Recall: } NOT(Q) = \overline{Q} = Q' = \overline{\overline{Q}}\)
- Called a “Latch” because it can “Latch” onto data coming in

R-S Latch
- The R-S latch is a bi-stable circuit which means that it can happily exist in either of two stable states. Just like a see-saw.
- You can push the latch from one state to another by setting or resetting it with the S-R signals
- The logic levels are maintained because of the feedback paths from outputs to inputs.

Another common way of drawing the same circuit
RS Latch

- Page 2 of Sequential Circuits: Example 1
  - Figure on top of page (with one circuit)
- First, set S=0, R=1
  - What is the output?
- Next, set S=1, R=1
  - What is the output?
- Next, set R=0, S=1
  - What is the output?
- Next, set R=1, S=1
  - What is the output?

Most Basic Sequential Logic Circuit: R-S Latch

First, recall truth table for a NAND gate:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

R-S Latch Operation:

- Best place to start is S=1, R=0

Next, look at top NAND gate:
- Its inputs are: 1 and 1
- Blue 1, comes from lower NAND
- Produces a 0 at its output

Therefore, when S=1, R=0
- The output of latch is: Q=0, ~Q=1

Truth Table for R-S Latch:

<table>
<thead>
<tr>
<th>ACTION</th>
<th>S</th>
<th>R</th>
<th>Q</th>
<th>~Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>RESET</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>HOLD</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Called the “RESET” action, as Q is set to 0
- Also, notice: Q and ~Q opposite

Most Basic Sequential Logic Circuit: R-S Latch

Truth table for a NAND gate:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Truth Table for R-S Latch:

<table>
<thead>
<tr>
<th>ACTION</th>
<th>S</th>
<th>R</th>
<th>Q</th>
<th>~Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>RESET</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>HOLD</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>HOLD</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

HOLD’s last value on its outputs!
- OUTPUT depends on input and last output
Most Basic Sequential Logic Circuit: R-S Latch

Truth table for a NAND gate:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

R-S Latch Operation:

- Next input case is called the "SET", when inputs are: S=0, R=1
  - 1st look at upper NAND gate
    - Its inputs are: 0 and X (anything)
    - Produces a 1 at its output
  - Lower NAND gate
    - Inputs are: 1 and 1
    - Produces a 0 at its output

Truth Table for R-S Latch:

<table>
<thead>
<tr>
<th>ACTION</th>
<th>S</th>
<th>R</th>
<th>Q</th>
<th>~Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>RESET</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>HOLD</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Most Basic Sequential Logic Circuit: R-S Latch

Truth table for a NAND gate:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

R-S Latch Operation:

- Last valid input case is the "HOLD" S=1, R=1
  - If we have just "SET" Latch, we will have Q=1, ~Q=0, already on outputs

Upper NAND gate

- Has S=1 & former value of ~Q=0
- Produces a 1 at its output
  - (same ~Q as when it started)

Lower NAND gate

- Inputs are: 1 and 1
- Produces a 0 at its output (same Q)
**Storage - Cross-Coupled NANDs (R-S Latch)**

What happens with S=0 and R=0?
- Short answer: confusion
- Real circuits depend on both Q and ~Q
- Strange things may happen if both are 1

<table>
<thead>
<tr>
<th>ACTION</th>
<th>S</th>
<th>R</th>
<th>Q</th>
<th>~Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILLEGAL</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SET</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>RESET</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>HOLD</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

D-Latches shows a way to prevent the RS Latch from ever getting S=R=0 as its input.

**Using multiple RS latches**
- Figure on Page 3: Multiple latches.
  - What is this circuit doing?
  - Store 3 bit number!

**Storage - Cross-Coupled NANDs (R-S Latch)**

How does this device “store” data?
- Each latch can store 1-bit of information, 3 SR Latches, holds 3-bits
- Let’s assume we wish to store the number 5₁₀ (10₁₂ in binary):

<table>
<thead>
<tr>
<th>ACTION</th>
<th>S</th>
<th>R</th>
<th>Q</th>
<th>~Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>RESET</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>HOLD</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

“Setting” the data we wish to store

“Holding” the data we set in the last phase

**Gated D-Latch: Preventing “Illegal State” of RS Latch**

Add logic to an R-S latch
- Create a more convenient interface, prevent S=0 && R=0

Two inputs: D (data) and WE (write enable)
- When WE = 1, latch is set to value of D
  - S = NOT(D), R = D

![Gated D-Latch Diagram]

D=1 && WE=1
So Q=1
What to do about the illegal inputs in RS latch...can we make the latch simpler?

- Page 3
- Place an input at D, set WE=1
  - What happens to output
- Set WE=0, set input at D
  - What happens to output
- We have a D latch – simplest way to store a bit
  - If Write Enable =1 then input is stored
  - If write enable =0, previous value remains in ‘storage’

Gated D-Latch: Preventing “Illegal State” of RS Latch

Add logic to an R-S latch
• Create a more convenient interface, prevent S=0 && R=0
Two inputs: D (data) and WE (write enable)
  • When WE = 1, latch is set to value of D
    ➢ S = NOT(D), R = D
  • When WE = 0, latch continues to hold previous value
    ➢ S = R = 1 (hold condition for SR latch)
  • Extra logic does not allow S=0, R=0 case to occur

D-Latch Timing Diagram
• The diagram below is called a “Timing” Diagram
  ➢ Our D-Latch is previous-state dependent
    – We can think of this as a time dependency
    – Moving to the right on diagram, represents forward moving time
  ➢ The inputs & outputs to our D-Latch are on left
    – Inputs/Outputs can be either “HIGH” (logic 1) or “LOW” (logic )
    ➢ Think of this as a time-dependent truth table

When latch is OPEN (WE=1): Notice, Q follows D

- When the WE signal is high the latch is said to be open and the output signal, Q, follows the input signal, D.
  ➢ As in any combinational circuit there will be a small delay between the time that the input changes and the time that the output follows suit.
**D-Latch Timing Diagram**

- When the WE signal is low the latch is closed and the output signal, Q retains its value.

When latch is CLOSED (WE=0): Notice, Q doesn’t follow D
(Instead, Q has previous value)

<table>
<thead>
<tr>
<th>WE</th>
<th>D</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>closed</td>
<td>open</td>
</tr>
</tbody>
</table>

**Setup / Hold Times**

- The input signal should (D) be stable a certain amount of time before the WE signal is set to CLOSED (WE=0)
  - This is referred to as the SETUP time
- In addition, the input signal (D) must be stable for a time after the WE is set to CLOSED (WE=0)
  - This is referred to as HOLD time
- Why? Time must be given for inputs to propagate through NAND gates! Gates are not instantaneous!

**Next... Storage Devices**

- Ok...we now have a device (D-Latch) that can store a bit
  - Use this to build ‘real’ storage devices....
  - Temporary storage in a computer?
    - Where are variables stored before being sent to the arithmetic unit for operations on them?
  - Register
    - Can we build an n-bit register using latches?
  - What about “main” memory
  - Disk
    - Later...

**Page 4**

- Use 4 D latches
  - Set input (from keypad) and set WE=1
    - What gets stored
  - Set WE=0, try entering inputs
  - 4-bit register
Register
A register stores a multi-bit value.
- We use a collection of D-latches, all controlled by a common WE.
- When WE=1, n-bit value D is written to register.

Multi–Bit D-Latch
- A collection of D-latches, controlled by a common WE
- When WE=1, n-bit value D is written to the outputs

Recall: A Basic Model of a Computer

Memory
We know how to store m-bit number in a register
How about many m-bit numbers?
- Bank of registers?
How to fetch a specific m-bit number?
- addressing

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-20</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>-7</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ADD 0, -20, 40</td>
</tr>
<tr>
<td>1</td>
<td>ADD 0, 10, 51</td>
</tr>
<tr>
<td>2</td>
<td>SUB 50, 41, 60</td>
</tr>
<tr>
<td>3</td>
<td>MUL 60, 94, 55</td>
</tr>
<tr>
<td>4</td>
<td>ADD 60, 65, 60</td>
</tr>
<tr>
<td>5</td>
<td>DIV 60, 5, 90</td>
</tr>
</tbody>
</table>

Essential Part of Computer!
Basic Components: Address: Looks up data
Note: both are in binary
Memory
Now that we know how to store bits, we can build a memory – a logical $k$ by $m$ array of stored bits

Address Space:
number of locations
(usually a power of 2)

Addressability:
number of bits per location
(e.g., byte-addressable)

Memory Interface
There are two basic operations on a memory
• Selecting one of the memory locations to read from
• Selecting one of the memory locations to write to

Interface signals
• A – $n$-bit address lines to select a location
• Dout – Contents of selected location during read ($m$ bits)
• Din – Value to be stored during write ($m$ bits)
• WE – If WE = 1 – write operation, WE = 0, read operation

Memory
Looking from the outside, what do we need?

Memory
Looking from the outside, what do we need?
Memory

Address Space

n bits allow the addressing of $2^n$ memory locations.
- Example: 24 bits can address $2^{24} = 16,777,216$ locations (i.e. 16M locations).
- If each location holds 1 byte (= 8 bits) then the memory is 16MB.
- If each location holds one word (32 bits = 4 bytes) then it is 64 MB.

Memory

Addressability

- Computers are either byte or word addressable - i.e. each memory location holds either 8 bits (1 byte), or a full standard word for that computer (16 bits for the LC-3, more typically 32 bits, though now many machines use 64 bit words).
- Normally, a whole word is written and read at a time:
  - If the computer is word addressable, this is simply a single address location.
  - If the computer is byte addressable, and uses a multi-byte word, then the word address is conventionally either that of its most significant byte (big endian machines) or of its least significant byte (little endian machines).

Page 6

- If we want output to come from one of many ‘locations’
  - Multiplexer
  - Control lines to Multiplexer = Address

- How about writing into one of many locations
  - Enable one out of many ‘locations’
  - Decoder
  - Control lines to Decoder = Address
2^2 by 3-bit memory

Read operation

Selects “address” to read

But how do we select/enable ONE of the D-latches?
Given 2 bit address, Select ONE latch

2^2 or 4 registers

2^2 by 3-bit memory

Write operation

**Limitation:**
You can only read or write at any given time

Recall: The Decoder

\( n \) inputs, \( 2^n \) outputs
- ONLY one AND gate outputs a 1 for each possible input pattern

2-bit decoder

Think of it like this: put a binary # on AB, turns on the corresponding output wire!

Ex: AB=10, 3rd wire turns high, all the other are low

2^2 by 3-bit memory - Multiple “Ports”

Independent Read/Write

You can read from one address and write to another with this arrangement

(notice 1 address line for R, 1 address line for W)
More Memory Details

This is still not the way actual memory is implemented
- Real memory: fewer transistors, denser, relies on analog properties

But the logical structure is similar
- Address decoder
- Word select line, word write enable
- Bit line

Two basic kinds of RAM (Random Access Memory)

Static RAM (SRAM) - 6 transistors per bit
- Fast, maintains data as long as power applied

Dynamic RAM (DRAM) - 1 transistor per bit
- Denser but slower, relies on “capacitance” to store data, needs constant “refreshing” of data to hold charge on capacitor

Also, non-volatile memories: ROM, PROM, flash, …

Dynamic RAM

- Information stored as charge on capacitors.
- Capacitors leak so values have to be ‘refreshed’ continually
- As memory chips get larger, access times tend to increase. The processor spends more time waiting for data.
  ➢ This is a major issue limiting computer systems performance
Example
Intel Core i5 – Processor
- Clock rates approx 2.5GHz, Clock period approx 0.4 ns
DDR2-667 PC2-5300 SO-DIMM – 2 GB Memory
- Can deliver at most 1 64-bit word every 1.5 ns
Mismatch between processor speed and memory speed

Memory Hierarchy
- Modern computers try to mitigate memory delays by exploiting locality of reference through caches.
- Smaller, faster memory stores are placed closer to the CPU and bulk transfers from slower memory are used

CPU
Cache Memories
Main Memory
Disks
Magnetic, Flash etc.

Storage in MegaBytes, access times single clock cycles
Storage in GigaBytes, access time 10s of clock cycles
Storage in TeraBytes, access time 1000s of clock cycles

Memory Hierarchy
Will return to this at the end of the course....!

Are we ready to design sequential circuits and finite state machines?
Is something missing?
When do states change in a machine?
Do we let states change at arbitrary times?
What do you think happens in a computer?
Clocked Flip-Flops/Circuits
• Subsystem in a computer consists of a large number of combinational and sequential devices
  • Each sequential device is like latch which is in one of two states
  • As machine executes its cycle, the states of all sequential devices change with time
• To control large collection of devices in an orderly (synchronized) fashion, machine maintains a clock
  • Requires all devices to change their states at the same time
  • Clock generates sequence of pulses
  • Much easier to design, debug, implement, and test
• How do we change latches so that they allow change in state synchronized with the clock?
• Sequential logic circuits require a means by which events can be sequenced.....clock!

Attaching Clock to D-Latch
• Let’s attach CLOCK to the WE on D-Latch

- We create “windows” of time that we can store data into latch
  ➢ When the CLOCK is “HIGH” – D-latch is open
  ➢ When the CLOCK is “LOW” – D-latch is closed
- We have to prepare what we wish to store, right before latch closes

Introducing - The Clock!
A clock controls when stored values are “updated”
• Electrical waveform – sends pulses through a circuit
• Oscillating global signal with fixed period

The clock will act as the ‘heartbeat’ of our system
• The number of cycles per second is the clock frequency measured in cycles per second or Hertz (Hz)
• The clock period refers to the duration of one clock cycle. The period and frequency are inversely related.
  ➢ Typical clock frequency: 2.5GHz = 2.5 x 10⁹ Hz
  ➢ So corresponding clock period = 1/(2.5 x 10⁹) = .4x10⁻⁹ sec
  ➢ That would be: 0.4 nanoseconds

Clock...Page 7
Input to D latch can be written only when clock is high

CLOCK

Input to D latch can be written only when clock is high
Let's Try to Build a Counter using the D-Latch

![D-Latch Diagram]

What is a counter?
Counter increments value by 1 at each cycle of clock
Example: time = 0, counter = 000
time = 1, counter = 001
time = 2, counter = 010
...
time = 7, counter = 111

Simple 2-bit Counter
- count from 0 to 3
  - 2 bit encoding A,B of states: 00, 01,10,11
  - Counts 00 01 10 11
- functions:
  - A,B current state
  - A*, B* next state
  - A* = (A'B) + (A B') = A XOR B
  - B* = (A'B') + (AB') = B'

Let's Try to Build a Counter using the D-Latch

![D-Latch Diagram]

We can't use a D-latch to build a counter 😞
Why not?
Let's say at time=0, D-latch has: 000
This is the input to incrementer, so output = 001
Now 001 is input to the D-latch
Problem:
We can't guarantee the clock will be low in time to store this new value into the D-latch

D Flip-Flop (or master-slave flip-flop)
D Flip-Flop is a pair of D latches
- Stupid name, but it stuck
- Isolates next state from current state

![D Flip-Flop Diagram]

Two phases:
- Clock = 1: WE₁ = 0; Latch #1 closed, WE₂ = 1; Latch #2 open
- Clock = 0: WE₁ = 1; Latch #1 open, WE₂ = 0; Latch #2 closed
D Flip-Flop timing Diagram

Latch #1

Latch #2

D Flip-Flop vs. D-Latch

We refer to the D flip flop as an edge-triggered device.
- D=Q ONLY when WE changes from 0 to 1

This differs from D latch, which is: level-triggered
- D=Q anytime WE equals 1

Timing Diagram for DFF:

WE __
D __
Q __

D Flip-Flop vs. D-Latch

We refer to the D flip flop as an edge-triggered device.
- D=Q ONLY when WE changes from 0 to 1

This differs from D latch, which is: level-triggered
- D=Q anytime WE equals 1

Timing Diagram for D-Latch:

WE __
D __
Q __

D Flip-Flop

- We can think of the D Flip-Flop as a 1 bit storage container with an input, D, and an output, Q.
- The D flip-flop takes a clock input (often denoted with a triangle)
- A set of D flip-flops can be grouped together with common Clock and WE inputs to form a register. A key component in our processor

Timing Diagram for D-Latch:

Flip-Flop

D → Q

Clock
### 3 “Storage” Devices

- **RS Latch** – Stores 1 Bit, Level-Triggered
  - 1 “forbidden” input: S=0, R=0
  - Holds Data when RS=11

- **D-Latch** – Stores 1 Bit, Level-Triggered
  - No “forbidden” inputs (fixes RS Latch)
  - D=Q when WE=1
  - Holds Data when WE=0

- **D-Flip-Flop** – Stores 1 Bit, Edge-Triggered
  - No “forbidden” inputs
  - D=Q when WE (CLK) transitions from 0 to 1
  - Holds Data for WE=1 or WE=0
  - Except when WE transitions from 0 to 1

### Working Counter

Use a clocked register (made of D flip-flops)

- Incrementer (+1) computes the next value of the state register

### Counter Timing Diagram

- **Clock**
- **D-ff**
- **Q**

### Finite State Machine Representation of Counter

- **Reset**
- **Bubbles represent all possible “states” for the machine (aka your flip-flop based circuit)**
- **Arrows show movement from one state to the next**
- **Transitions occur at pulse of the clock**
Truth Table Representation of Counter

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>D_2 (t)</td>
<td>D_1 (t)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Finite State Machine

- The counter we designed is an example of a finite state machine.
- In general a Finite State Machine consists of
  - An n-bit register which stores the state of the machine
  - A block of logic that computes the next state as a function of the current state and the inputs, if any
  - A block of logic which computes the output based on the current state.

One Last Thing...
D Flip-Flop with Additional Write Enable

- From previous slides, we attached clock to WE of the D-flip-flop
- Now, we add another WE line to the flip flop
  - Just holds onto data already stored in DFF
  - Give it the ability to “ignore” the clock!

Next....

- Procedure for designing Sequential circuits (to implement Finite State Machine)
  - Storage Device to store state: D flip flop
  - Logic to implement next state: combinational gates/devices
  - How to derive the logic: truth tables