Unit 13

Sequential Logic Constructs
Learning Outcomes

• I understand the difference between level-sensitive and edge-sensitive

• I understand how to create an edge-triggered FF from 2 latches
How sequential building blocks work

LATCHES AND FLIP-FLOPS
Suppose we have a sequence of input numbers on X[3:0] that are entered over time that we want to sum up.

Possible solution: Route the outputs back to the inputs so we can add the current sum to the input X.
Sequential Logic

- Suppose we have a sequence of input numbers on X[3:0] that are entered over time that we want to sum up.
- Possible solution: Route the outputs back to the inputs so we can add the current sum to the input X.
- Problem 1: No way to initialize sum.
- Problem 2: Outputs can race around to inputs and be added more than once per input number.

Possible Solution
- Outputs can feedback to inputs and update them sum more than once per input.
Sequential Logic

• Add logic at outputs to just capture and remember the new sum until we’re ready to input the next number in the sequence.

This logic should *remember* (i.e. sequential logic) the sum and only update it when the next number arrives.

The data can still loop around and add up again (2+2=4) but if we just hold our output = 2 then the feedback loop will be broken.

We remember initial sum of 2 until input 3 arrives at which point we’d capture & remember the sum 5.
Sequence Adder

- If $X$ changes once per cycle then $Z$ should also change once per cycle.
- That is why we will use a register (flip-flops) to ensure the outputs can only update once per cycle.
Sequence Adder

- The 0 on Clear will cause Z to be initialized to 0, but then Z can’t change until the next positive edge.
- That means we will just keep adding $0 + 2 = 2$.
Sequence Adder

- At the edge the flip-flops will sample the D inputs and then remember 2 until the next positive edge.
- That means we will just keep adding $3 + 2 = 5$.
Sequence Adder

• Finally, at the positive edge the flip-flops will sample the D inputs and then remember 14
Sequential Logic

• But how do flip-flops work?
• Our first goal will be to design a circuit that can remember one bit of information
• Easiest approach...

• But how do you change the input? 
  – A signal should only have one driver
D-Latches

- The primary building block of sequential logic is a D-Latch
- D-Latches (Data latches) store/remember/hold data when the clock is low (CLK=0) and pass data when the clock is high (CLK=1)

These "switches" which can be closed or open are really transistors that can be on or off.
Transparent & Hold Mode of D-Latches

• The D-Latch operates in either transparent or hold mode based on the clock value.

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th>Q</th>
<th>Q'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>Q₀</td>
<td>Q₀'</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Function Table
Description of D-Latch

Transparent Mode
(Q=D when CLK=1)

Hold Mode
(Q=Q₀ when CLK=0)
D-Latches

**Hold Mode**

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th>Q</th>
<th>Q'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>Q₀</td>
<td>Q₀'</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Triggering Rule:** The Q output follows the D input (i.e., Q = D) when the clock or gate input is high (i.e., the latch is enabled). When the latch is disabled (Clock = LOW) the output remains put.

**D-LATCH 7475**

As clock is LOW, don’t look at the D input

Complete waveform for Q

0 1 2 3 6 7
D-Latches

When C=0, outputs don’t change no matter what the inputs do.

When C=1, outputs change based on inputs.

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th>Q</th>
<th>Q'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>Q₀</td>
<td>Q₀'</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Notation

• To show that Q remembers its value we can put it in the past tense:
  – \( Q = Q_0 \) (Current Value of Q = Old Value of Q)

• OR put it in the future tense
  – \( Q^* = Q \) (Next Value of Q = Current Value of Q)

Indicates “next-value” of Q

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th>Q</th>
<th>Q’</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>( Q_0 )</td>
<td>( Q_0' )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Current Value = Old Value

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th>Q*</th>
<th>Q’*</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>( Q )</td>
<td>( Q' )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Next Value = Current Value
Adding a Sequence of Numbers

• What if we put D-Latches at the outputs
Adding a Sequence of Numbers

- We’ll change X on every clock period

When $C=0 \Rightarrow Q^* = Q$
When $C=1 \Rightarrow Q^* = D$
Adding a Sequence of Numbers

- Since the clock starts off low, the outputs of the latches can’t change and just hold at 0

When C=0 => Q* = Q
When C=1 => Q* = D
Adding a Sequence of Numbers

- When the clock goes high the D goes through to Q and is free to loop back around

When C=0 => Q* = Q
When C=1 => Q* = D
Adding a Sequence of Numbers

- Once it loops back around it will be added again, change the Y value and go through to Z and loop back around again.

When $C=0 \Rightarrow Q^* = Q$
When $C=1 \Rightarrow Q^* = D$
Adding a Sequence of Numbers

- This feedback loop continues until the clock goes low again

When C=0 => Q^* = Q
When C=1 => Q^* = D
Adding a Sequence of Numbers

- When the clock goes low again, the outputs will hold at their current value 8 until the clock goes high

When C=0 => Q* = Q
When C=1 => Q* = D
Adding a Sequence of Numbers

- When the clock goes high, the outputs will be free to change and we will get the feedback problem

\[
\begin{align*}
\text{When } C=0 & \Rightarrow Q^* = Q \\
\text{When } C=1 & \Rightarrow Q^* = D
\end{align*}
\]
Adding a Sequence of Numbers

- Latches clearly don’t work
- The goal should be to get \textit{one change of the outputs per clock period}

When \( C = 0 \) => \( Q^* = Q \)
When \( C = 1 \) => \( Q^* = D \)
FLIP-FLOPS
Flip-Flops vs. Latches

**Latches**
- Asynchronous
- Clock/Enable input
- Level Sensitive
  - Outputs can change anytime Clock = 1

**Flip-Flops**
- Synchronous
- Clock Input
- Edge-Sensitive
  - Outputs change only on the positive (negative) edges

\[ D \rightarrow Q \]

**D-Latch**

\[ D \rightarrow Q \]

**D-FF**

\[ D \rightarrow Q \]
Flip-Flops

• Change D Latches to D Flip-Flops

Diagram showing a comparison between a D-latch and a D-flip-flop, with a note indicating that a triangle at the clock input indicates edge-sensitive FF.
Flip-Flops

• To indicate negative-edge triggered use a bubble in front of the clock input

Positive-Edge Triggered D-FF

No bubble indicates positive-edge triggered

Negative-Edge Triggered D-FF

Bubble indicates negative-edge triggered
Positive-Edge Triggered D-FF

- Q looks at D only at the positive-edge

<table>
<thead>
<tr>
<th>CLK</th>
<th>D</th>
<th>Q*</th>
<th>Q'*</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>Q</td>
<td>Q'</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>Q</td>
<td>Q'</td>
</tr>
<tr>
<td>↑</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>↑</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Q only samples D at the positive edges and then holds that value until the next edge.
Negative-Edge Triggered D-FF

- Q looks at D only at the negative-edge

<table>
<thead>
<tr>
<th>CLK</th>
<th>D</th>
<th>Q*</th>
<th>Q’*</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>Q</td>
<td>Q’</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>Q</td>
<td>Q’</td>
</tr>
<tr>
<td>↓</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>↓</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Q only samples D at the negative edges and then holds that value until the next edge.
D FF Example

- Assume positive edge-triggered FF
Shift Register

- A shift register is a device that acts as a ‘queue’ or ‘FIFO’ (First-in, First-Out).
- It can store n bits and each bit moves one step forward each clock cycle
  - One bit comes in the overall input per clock
  - One bit ‘falls out’ the output per clock

```
Data In = 1

S7  S6  S5  S4  S3  S2  S1  S0
0    1   1   0   1   0   1   1
1    0   1   1   0   1   0   1
```

Data during clock i

Last Data

Data during clock i+1
Shift Register

Shift Register w/ Latches

CLK
D_IN
Q0
Q1
Q2
Q3

D_IN
Clock
D
Q
C
Q0
D
Q
C
Q1
D
Q
C
Q2
D
Q
C
Q3
When we want to ensure an output updates only ONCE per clock, we need to use **flip-flops** (not latches or bistables)!
BUILDING A FLIP FLOP
Building an Edge-Triggered Device

- We generally build FFs from latches
- To build a device that can only change at 1 instant (clock edge) we can:
  - Try to only enable 1 latch for a small instant in time
  - Use two latches running on opposite clock phases

Input Queue

Door opening and closing almost instantly

Two door system (when 1 is open the other is closed)
Master-Slave D-FF

• To build an **edge-triggered** D-FF we can use two D-Latches

![Diagram of Master-Slave D-FF]

These 2 latches form a flip-flop
Complete the Waveform

Diagram of a D flip-flop circuit with master and slave sections connected by a clock signal.

Waveforms for D, CLK, Q_{Master}, and Q_{Slave} are shown with eight time steps (1 to 8) labeled along the bottom.
Master-Slave D-FF

- To implement a positive edge-triggered D-FF change the clock inversion

Negative-Edge Triggered

Positive-Edge Triggered
INITIALIZING OUTPUTS
Initializing Outputs

- Need to be able to initialize Q to a known value (0 or 1)
- FF inputs are often connected to logic that will produce values after initialization
- Two extra inputs are often included: (PRE)SET and CLEAR

\[ \text{When CLEAR = active} \]
\[ Q^* = 0 \]
\[ \text{When SET = active} \]
\[ Q^* = 1 \]
\[ \text{When NEITHER = active} \]
Normal FF operation

Note: CLR and SET have priority over normal FF inputs
Initializing Outputs

- To help us initialize our FF’s use a RESET signal
  - Generally produced for us and given along with CLK
- It starts at *Active (1)* when power turns on and then goes to *Inactive (0)* for the rest of time
- When it’s active use it to initialize the FF’s and then it will go inactive for the rest of time and the FF’s will work based on their inputs

![Diagram](image-url)
Initializing Outputs

- Suppose we want our FF to initialize to 0 when the power turns on
  - Connect RESET to the CLR input
  - Connect 0 (off) to the SET input

When RESET = 0,
- CLR is inactive and Q looks at D at each clock edge

When RESET = 1,
- CLR is active and Q is forced to 0 regardless of D

Q* = 0

Q* = D
Implementing an Initial State

When RESET is activated Q’s initialize and then when it goes back to 1 the Q’s look at the D inputs.

- Forces Q’s to 0 because it’s connected to the CLR inputs.
- Once RESET goes to 0, the FF’s look at the D inputs.
Synchronous vs. Asynchronous

- The new preset and clear inputs can be built to be **synchronous** or **asynchronous**.
- These terms refer to when the initialization takes place:
  - Asynchronous...initialize when signal is activated
  - Synchronous...initialize at clock edge

**Synchronous**

- Clock
- CLR
- Q’s

Synchronous SET or CLR means the signal must be active at a clock edge before Q will initialize.

**Asynchronous**

- Clock
- CLR
- Q’s

Asynchronous SET or CLR means Q will initialize as soon as the SET or CLR signal is activated.
Set / Clear Example

- Complete the waveform for a D-FF with *asynchronous* SET and CLR
Exercise
Using muxes to control when register save data

REGISTER WITH ENABLES
Register Resets/Clears

- When the power turns on the bit stored in a flip-flop will initialize to a random value
- Better to initialize it to a known value (usually 0's)
- Can use an asynchronous or synchronous "reset" to force the flip-flops to 0's

<table>
<thead>
<tr>
<th>CLK</th>
<th>RST</th>
<th>D_i</th>
<th>Q_i*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,0</td>
<td>X</td>
<td>X</td>
<td>Q_i</td>
</tr>
<tr>
<td>↑</td>
<td>1</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>↑</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>↑</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

4-bit Register
Register Problem

- Whatever the D value is at the clock edge is sampled and passed to the Q output until the next clock edge.
- Problem: Register will save data on EVERY edge.
  - Often we want the ability to save on one edge and then keep that value for many more cycles.

4-bit Register – On clock edge, D is passed to Q.
Solution

• Registers (D-FF’s) will sample the D bit every clock edge and pass it to Q
• Sometimes we may want to hold the value of Q and ignore D even at a clock edge
• We can add an enable input and some logic in front of the D-FF to accomplish this

<table>
<thead>
<tr>
<th>CLK</th>
<th>RST</th>
<th>EN</th>
<th>D_i</th>
<th>Q_i*</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Q_i</td>
</tr>
<tr>
<td>↑↑</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>↑↑</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>Q_i</td>
</tr>
<tr>
<td>↑↑</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>↑↑</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Registers w/ Enables

- When EN=0, Q value is passed back to the input and thus Q will maintain its value at the next clock edge.
- When EN=1, D value is passed to the input and thus Q will change at the edge based on D.

When EN=0, Q is recycled back to the input.

When EN=1, D input is passed to FF input.
4-bit Register w/ Data (Load) Enable

- Registers (D-FF’s) will sample the D bit every clock edge and pass it to Q
- Sometimes we may want to hold the value of Q and ignore D even at a clock edge
- We can add an enable input and some logic in front of the D-FF to accomplish this

<table>
<thead>
<tr>
<th>CLK</th>
<th>RST</th>
<th>EN</th>
<th>$D_i$</th>
<th>$Q_i^*$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>$Q_i$</td>
</tr>
<tr>
<td>↑↑</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>↑↑</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>$Q_i$</td>
</tr>
<tr>
<td>↑↑</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>↑↑</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

4-bit register with 4-bit wide 2-to-1 mux in front of the D inputs
Registers w/ Enables

- The D value is sampled at the clock edge only if the enable is active.
- Otherwise the current Q value is maintained.
COUNTERS
Counters

- Count (Add 1 to Q) at each clock edge
  - Up Counter: $Q^* = Q + 1$
  - Can also build a down counter as well ($Q^* = Q - 1$)

- Standard counter components include other features
  - Resets: Reset count to 0
  - Enables: Will not count at edge if EN=0
  - Parallel Load Inputs: Can initialize count to a value P (i.e. $Q^* = P$ rather than $Q+1$)
Sample 4-bit Counter

- **4-bit Up Counter**
  - RST: synchronous reset input
  - PE and $P_i$ inputs: loads $Q$ with $P$ when PE is active
  - CE: Count Enable
    - Must be active for the counter to count up
  - TC (Terminal Count) output
    - Active when $Q=1111$ AND counter is enabled
    - $TC = EN \cdot Q_3 \cdot Q_2 \cdot Q_1 \cdot Q_0$
      - Mealy output
    - Indicates that on the next edge it will roll over to 0000

<table>
<thead>
<tr>
<th>CLK</th>
<th>RST</th>
<th>PE</th>
<th>CE</th>
<th>Q*</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\uparrow$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\uparrow$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\uparrow$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\uparrow$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Counters

SR=active at clock edge, thus Q=0
Q*=Q+1
Enable = off, thus Q holds
Q*=Q+1
Q*=Q+1
PE = active, thus Q=P
Q*=Q+1
Q*=Q+1

Mealy TC output:
EN•Q3•Q2•Q1•Q0
Counter Exercise

CLK
RST
PE
CE
P[3:0]
Q[3:0]
0011 1101 1001
Counter Design

• Sketch the design of the 4-bit counter presented on the previous slides