Readings

- Please study Slides 102-120 from Lecture 6 on your own

- This week
  - 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 + Chapter 5

- Next week
  - Von Neumann Model, LC3, and MIPS
    - P&P Chapter 4-5 + H&H Chapter 6
  - Digital Building Blocks
    - H&H Chapter 5
What We Will Learn Today

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

- Finite State Machines (FSM)
  - Moore Machine
  - Mealy Machine

- Verilog implementations of sequential circuits
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**?
Basic Storage Element: The R-S Latch
The R-S (Reset-Set) Latch

- The simplest implementation of the R-S Latch
  - Two **NAND gates** with outputs feeding into each other’s input
  - Data is stored at \( Q \) (inverse at \( Q' \))
  - \( S \) and \( R \) inputs are held at 1 in **quiescent (idle) state**
    - **\( S \) (set):** drive \( S \) to 0 (keeping \( R \) at 1) to change \( Q \) to 1
    - **\( R \) (reset):** drive \( R \) to 0 (keeping \( S \) at 1) to change \( Q \) to 0
  - \( S \) and \( R \) should never **both** be 0 at the same time

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R )</td>
<td>( S )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Why not \( R=S=0? \)

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 **metastability** to come in Lecture 8)
The Gated D Latch
The Gated D Latch

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

![Gated D Latch Diagram]
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
The 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

Register x (Rx)

This register holds 4 bits, and its data is referenced as Q[3:0]
Memory
- Memory is comprised of locations that can be written to or read from. An example memory array with 4 locations:

<table>
<thead>
<tr>
<th>Addr(00):</th>
<th>Addr(01):</th>
<th>Addr(10):</th>
<th>Addr(11):</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100 1001</td>
<td>0100 1011</td>
<td>0010 0010</td>
<td>1100 1001</td>
</tr>
</tbody>
</table>

- 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 full set of unique locations in memory is referred to as the address space.

- Typical memory is MUCH larger (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

<table>
<thead>
<tr>
<th>Addr(0)</th>
<th>Bit₂</th>
<th>Bit₁</th>
<th>Bit₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr(1)</td>
<td>Bit₂</td>
<td>Bit₁</td>
<td>Bit₀</td>
</tr>
</tbody>
</table>
Reading from Memory

How can we select the address to read?

- Because there are 2 addresses, address size is $\log(2) = 1$ bit
**Reading from Memory**

**How can we select the address to read?**
- Because there are 2 addresses, address size is \( \log(2) = 1 \) bit

![Diagram showing selection of address and data output](image)
How can we select the address to read?

- Because there are 2 addresses, address size is $\log(2) = 1$ bit
How can we select the address to read?
• Because there are 2 addresses, address size is \( \log(2) = 1 \) bit
How can we select the address and write to it?
**Writing to Memory**

**How can we select the address and write to it?**
- Input is indicated with $D_i$

![Diagram](image)
Putting it all Together

Enable reading and writing to a memory array
A Bigger Memory Array
A Bigger Memory Array
Sequential Logic Circuits
Sequential Logic Circuits

- We have looked at 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_Combo_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 L22
  D. Locked but user has completed R3
  E. The lock can now be opened

- 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-E must be completed in order**
  - If anything else happens (e.g., L5), lock **returns** to state A
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
  - Clock: \[ \begin{array}{c}
  1 \\
  0 
  \end{array} \]
- At the start of a clock cycle (\[ \square \]), 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 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.
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
  - the set of all possible states that a system can be in
  - 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
    - 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
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

Clock transition determines when input is acquired

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

CLK: 1 0

Input:

Output:
The Problem with Latches

Currently, we **cannot** simply wire a clock to WE of a latch

- **When the clock is high,** $Q$ will not take on $D$’s value AND
- **When the clock is low,** the latch will propagate $D$ to $Q$

Recall the Gated D Latch

CLK: 1 0

Input:  

Output:  

![Gated D Latch Diagram]
The Problem with Latches

Recall the Gated D Latch

- **Current**, we **cannot** simply wire a clock to **WE** of a latch
  - **When the clock is high**, **Q** will not take on **D**’s value **AND**
  - **When the clock is low**, the latch will propagate **D** to **Q**
The Problem with Latches

Recall the Gated D Latch

CLK = WE

D

Q

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 D Flip-Flop

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

- When the clock is low, master propagates D to the input of slave (Q unchanged)
- Only when the clock is high, slave 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

- How do we implement this?

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

We can use these **Flip-Flops** to implement the state register!
Rising-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 device because it is activated on the clock edge
Register

- Multiple parallel flip-flops, each of which store 1 bit

This register stores 4 bits

This line represents 4 wires

Condensed
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
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
“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)
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 Diagram]

- **States**:
  - S0
  - $\text{L}_A$: green
  - $\text{L}_B$: red
Finite State Machine Transition Diagram

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

![Finite State Machine Transition Diagram](image)
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
**Moore FSM:** outputs labeled in each state

- **States:** Circles
- **Transitions:** Arcs

---

### Finite State Machine Transition Diagram

- **S0**
  - $L_A$: green
  - $L_B$: red

- **S1**
  - $L_A$: yellow
  - $L_B$: red

- **S2**
  - $L_A$: red
  - $L_B$: green

- **S3**
  - $L_A$: red
  - $L_B$: yellow

---

**Academic Ave.**

**Bravado Blvd.**

**Dining Hall**

**Fields**

**Dorms**

**Reset**
Finite State Machines
State Transition Table
## FSM State Transition Table

<table>
<thead>
<tr>
<th>Current State</th>
<th>Inputs</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>S0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>S1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>S2</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>S2</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>S3</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
**FSM State Transition Table**

<table>
<thead>
<tr>
<th>Current State</th>
<th>Inputs</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>S0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>S1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>S2</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>S2</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>S3</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Legend:**
- **S0**: $L_A$: green, $L_B$: red
- **S1**: $L_A$: yellow, $L_B$: red
- **S2**: $L_A$: red, $L_B$: green
- **S3**: $L_A$: red, $L_B$: yellow

**Inputs:**
- $T_A$: green
- $T_B$: red
- $\overline{T_A}$: yellow
- $\overline{T_B}$: red

**Reset:**
- S0 to S0
- S1 to S1
- S2 to S2
- S3 to S3
FSM State Transition Table

<table>
<thead>
<tr>
<th>Current State</th>
<th>Inputs</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>S0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>S1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>S2</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>S2</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>S3</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

State Encoding

<table>
<thead>
<tr>
<th>State</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>00</td>
</tr>
<tr>
<td>S1</td>
<td>01</td>
</tr>
<tr>
<td>S2</td>
<td>10</td>
</tr>
<tr>
<td>S3</td>
<td>11</td>
</tr>
</tbody>
</table>
### FSM State Transition Table

<table>
<thead>
<tr>
<th>Current State</th>
<th>Inputs</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>$S_0$</td>
<td>$T_A$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>00</td>
</tr>
<tr>
<td>S1</td>
<td>01</td>
</tr>
<tr>
<td>S2</td>
<td>10</td>
</tr>
<tr>
<td>S3</td>
<td>11</td>
</tr>
</tbody>
</table>
FSM State Transition Table

Current State | Inputs | Next State
---|---|---
S₁ | S₀ | Tₐ | Tₜₜ | 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

\[ S'_1 = (S_1 \cdot \overline{S_0}) + (S_1 \cdot \overline{S_0} \cdot \overline{T_B}) + (S_1 \cdot \overline{S_0} \cdot T_B) \]
FSM State Transition Table

![FSM Diagram]

<table>
<thead>
<tr>
<th>Current State</th>
<th>Inputs</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_1 )</td>
<td>( S_0 )</td>
<td>( T_A )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>( X )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>( X )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>( X )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>( X )</td>
</tr>
</tbody>
</table>

\[
S'_1 = (\neg S_1 \cdot S_0) + (S_1 \cdot \neg S_0 \cdot T_B) + (S_1 \cdot S_0 \cdot T_B)
\]

\[
S'_0 = ?
\]
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

\[
S'_1 = (\overline{S}_1 \cdot S_0) + (S_1 \cdot \overline{S}_0 \cdot T_B) + (S_1 \cdot \overline{S}_0 \cdot T_B)
\]

\[
S'_0 = (\overline{S}_1 \cdot \overline{S}_0 \cdot \overline{T_A}) + (S_1 \cdot \overline{S}_0 \cdot \overline{T_B})
\]

State | Encoding
---|---
S0 | 00
S1 | 01
S2 | 10
S3 | 11
**FSM State Transition Table**

<table>
<thead>
<tr>
<th>Current State</th>
<th>Inputs</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>$S_0$</td>
<td>$T_A$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

$S'_1 = S_1 \text{ xor } S_0$  \hspace{1cm} \textbf{(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 Machines

Output Table
### FSM Output Table

#### Current State

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>$S_0$</td>
<td>$L_A$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>green</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>yellow</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>red</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>red</td>
</tr>
</tbody>
</table>

#### Diagram

- **Reset**
- **States:**
  - $S_0$: $L_A$: green, $L_B$: red
  - $S_1$: $L_A$: yellow, $L_B$: red
  - $S_3$: $L_A$: red, $L_B$: yellow
  - $S_2$: $L_A$: red, $L_B$: green
- **Transitions:**
  - $T_A$: from $S_0$ to $S_1$
  - $T_A$: from $S_1$ to $S_0$
  - $T_B$: from $S_0$ to $S_3$
  - $T_B$: from $S_3$ to $S_0$
  - $T_B$: from $S_1$ to $S_2$
  - $T_B$: from $S_2$ to $S_1$

---

69
FSM Output Table

<table>
<thead>
<tr>
<th>Current State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>$S_1$</td>
</tr>
<tr>
<td>$L_A$: green</td>
<td>$L_A$: yellow</td>
</tr>
<tr>
<td>$L_B$: red</td>
<td>$L_B$: red</td>
</tr>
<tr>
<td></td>
<td>green</td>
</tr>
<tr>
<td></td>
<td>red</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Current State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>$S_0$</td>
</tr>
<tr>
<td>$L_A$: yellow</td>
<td>$L_A$: red</td>
</tr>
<tr>
<td>$L_B$: red</td>
<td>$L_B$: green</td>
</tr>
<tr>
<td></td>
<td>green</td>
</tr>
<tr>
<td></td>
<td>red</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Current State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_2$</td>
<td>$S_0$</td>
</tr>
<tr>
<td>$L_A$: red</td>
<td>$L_A$: red</td>
</tr>
<tr>
<td>$L_B$: yellow</td>
<td>$L_B$: green</td>
</tr>
<tr>
<td></td>
<td>red</td>
</tr>
<tr>
<td></td>
<td>yellow</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Current State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_3$</td>
<td>$S_0$</td>
</tr>
<tr>
<td>$L_A$: red</td>
<td>$L_A$: red</td>
</tr>
<tr>
<td>$L_B$: yellow</td>
<td>$L_B$: green</td>
</tr>
<tr>
<td></td>
<td>red</td>
</tr>
<tr>
<td></td>
<td>yellow</td>
</tr>
</tbody>
</table>

Output Encoding

<table>
<thead>
<tr>
<th>Output</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>green</td>
<td>00</td>
</tr>
<tr>
<td>yellow</td>
<td>01</td>
</tr>
<tr>
<td>red</td>
<td>10</td>
</tr>
</tbody>
</table>
 FSM Output Table

\[ L_{A1} = S_1 \]
**FSM Output Table**

- **S0**
  - $L_A$: green
  - $L_B$: red

- **S1**
  - $L_A$: yellow
  - $L_B$: red

- **S2**
  - $L_A$: red
  - $L_B$: green

- **S3**
  - $L_A$: red
  - $L_B$: yellow

**Current State**

<table>
<thead>
<tr>
<th>$S_1$</th>
<th>$S_0$</th>
<th>$L_{A1}$</th>
<th>$L_{A0}$</th>
<th>$L_{B1}$</th>
<th>$L_{B0}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Outputs**

<table>
<thead>
<tr>
<th>Output</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>green</td>
<td>00</td>
</tr>
<tr>
<td>yellow</td>
<td>01</td>
</tr>
<tr>
<td>red</td>
<td>10</td>
</tr>
</tbody>
</table>

**Output Encoding**

- $L_{A1} = S_1$
- $L_{A0} = S_1 \cdot S_0$
FSM Output Table

Current State | Outputs
---|---
| | L\textsubscript{A1} | L\textsubscript{A0} | L\textsubscript{B1} | L\textsubscript{B0}
\textit{S}\textsubscript{1} | \textit{S}\textsubscript{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
FSM Output Table

### FSM Transition Diagram

- **S0**
  - $L_A$: green
  - $L_B$: red

- **S1**
  - $L_A$: yellow
  - $L_B$: red

- **S2**
  - $L_A$: red
  - $L_B$: green

- **S3**
  - $L_A$: red
  - $L_B$: yellow

### Current State and Outputs Table

<table>
<thead>
<tr>
<th>Current State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>$S_0$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Output Encoding

<table>
<thead>
<tr>
<th>Output</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>green</td>
<td>00</td>
</tr>
<tr>
<td>yellow</td>
<td>01</td>
</tr>
<tr>
<td>red</td>
<td>10</td>
</tr>
</tbody>
</table>
Finite State Machines
Schematic
FSM Schematic: State Register
FSM Schematic: State Register

CLK

S'₁  S₁

S'₀  S₀

Reset

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

next state logic

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

inputs

next state logic

state register

output logic

outputs
FSM Timing Diagram
FSM Timing Diagram

Cycle 1 | Cycle 2 | Cycle 3
---|---|---
CLK
Reset
T_A
T_B
S'_1:0
S_1:0
L_A1:0
L_B1:0

0 | 5 | 10

S0: (00)
S1: (10)
S2: (11)
S3: (01)

L_A: yellow
L_B: red

L_A: red
L_B: green

L_A: red
L_B: yellow

Reset
T_A
T_B
T_B

Green (00)
Red (10)
FSM Timing Diagram

- **Cycle 1:**
  - **S0:** L_A: yellow, L_B: red
  - **S1:** L_A: yellow, L_B: red
  - **S2:** L_A: red, L_B: green
  - **S3:** L_A: red, L_B: yellow

- **Cycle 2:**
  - **S0:** L_A: yellow (00)
  - **S1:** L_B: red (10)
  - **S2:** L_B: green (10)
  - **S3:** L_B: yellow (10)

- **Cycle 3:**
  - **S0:** L_B: red (10)
  - **S1:** L_B: green (10)
  - **S2:** L_B: yellow (10)
  - **S3:** L_B: red (10)

- **Cycle 4:**
  - **S0:** L_B: red (10)
  - **S1:** L_B: green (10)
  - **S2:** L_B: yellow (10)
  - **S3:** L_B: red (10)

- **Cycle 5:**
  - **S0:** L_B: red (10)
  - **S1:** L_B: green (10)
  - **S2:** L_B: yellow (10)
  - **S3:** L_B: red (10)

- **Cycle 6:**
  - **S0:** L_B: red (10)
  - **S1:** L_B: green (10)
  - **S2:** L_B: yellow (10)
  - **S3:** L_B: red (10)

- **Cycle 7:**
  - **S0:** L_B: red (10)
  - **S1:** L_B: green (10)
  - **S2:** L_B: yellow (10)
  - **S3:** L_B: red (10)

- **Cycle 8:**
  - **S0:** L_B: red (10)
  - **S1:** L_B: green (10)
  - **S2:** L_B: yellow (10)
  - **S3:** L_B: red (10)

- **Cycle 9:**
  - **S0:** L_B: red (10)
  - **S1:** L_B: green (10)
  - **S2:** L_B: yellow (10)
  - **S3:** L_B: red (10)

- **Cycle 10:**
  - **S0:** L_B: red (10)
  - **S1:** L_B: green (10)
  - **S2:** L_B: yellow (10)
  - **S3:** L_B: red (10)
FSM Timing Diagram

![FSM Timing Diagram](image-url)
FSM Timing Diagram

- **S0**: $L_A$: yellow, $L_B$: red
- **S1**: $L_A$: yellow, $L_B$: red
- **S2**: $L_A$: red, $L_B$: green
- **S3**: $L_A$: red, $L_B$: yellow

CLK

- **T_A**: Transition from S0 to S1
- **T_B**: Transition from S3 to S2

- **Reset**
- **S'_{1:0}**: ??? → S0 (00) → S1 (01) → S2 (10) → S3 (11)
- **S_{1:0}**: ??? → S0 (00) → S1 (01) → S2 (10)
- **L_{A1:0}**: ??? → Green (00) → Yellow (01) → Red (10)
- **L_{B1:0}**: ??? → Red (10) → Green (00)

Time in cycles: 0, 5, 10, 15, 20, 25, 30, 35
FSM Timing Diagram

- **S0**: L_A: yellow, L_B: red
- **S1**: L_A: yellow, L_B: red
- **S2**: L_A: red, L_B: green
- **S3**: L_A: red, L_B: yellow

- **CLK**
- **Reset**
- **T_A**
- **T_B**

- **S1**: (00) Start
- **S2**: (01) Green, Red
- **S3**: (10) Yellow, Red
- **S0**: (11) Green, Yellow

- **L_A1:0**: Green (00), Yellow (01), Red (10)
- **L_B1:0**: Red (10), Green (00), Yellow (01)

- **Cycle 1**: S0 (00)
- **Cycle 2**: S0 (00)
- **Cycle 3**: S1 (01)
- **Cycle 4**: S0 (00)
- **Cycle 5**: S1 (01)
- **Cycle 6**: S2 (10)
- **Cycle 7**: S3 (11)
- **Cycle 8**: S0 (00)
- **Cycle 9**: S0 (00)
FSM Timing Diagram

- **States:**
  - S0: $L_A$: yellow, $L_B$: red
  - S1: $L_A$: yellow, $L_B$: red
  - S2: $L_A$: red, $L_B$: green
  - S3: $L_A$: red, $L_B$: yellow

- **Transitions:**
  - $T_A$: Transition from $S0$ to $S1$
  - $T_B$: Transition from $S1$ to $S2$

- **Clock Phases:**
  - **CLK**
  - **Reset**
  - **$T_A$**
  - **$T_B$**

- **Outputs:**
  - $S'_{1:0}$
  - $S_{1:0}$
  - $L_{A1:0}$
  - $L_{B1:0}$

- **Time Course:**
  - 0-5 seconds: $S'_{1:0}$ transitions from $??$ to $S0 (00)$
  - 5-10 seconds: $S_{1:0}$ transitions from $??$ to $S0 (00)$
  - 10-15 seconds: $L_{A1:0}$ transitions from $??$ to Green (00)
  - 15-20 seconds: $L_{B1:0}$ transitions from $??$ to Red (10)
  - 20-25 seconds: $S'_{1:0}$ transitions from $S0 (00)$ to $S1 (01)$
  - 25-30 seconds: $S_{1:0}$ transitions from $S0 (00)$ to $S1 (01)$
  - 30-35 seconds: $L_{A1:0}$ transitions from Green (00) to Yellow (01)
  - 35-40 seconds: $L_{B1:0}$ transitions from Red (10) to Green (00)
  - 40-45 seconds: $S'_{1:0}$ transitions from $S1 (01)$ to $S0 (00)$
  - 45-50 seconds: $S_{1:0}$ transitions from $S1 (01)$ to $S0 (00)$
  - 50-55 seconds: $L_{A1:0}$ transitions from Yellow (01) to Red (10)
  - 55-60 seconds: $L_{B1:0}$ transitions from Green (00) to Yellow (01)
FSM State Encoding

- 3 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
1. **Fully Encoded:**
   - **Minimizes** # flip-flops, but not necessarily output logic or next state logic
   - Use $\log_2(num\_states)$ bits to represent the states
   - *Example states:* 00, 01, 10, 11

2. **1-Hot Encoded:**
   - **Maximizes** # flip-flops, **minimizes** next state logic
   - Simplest design process – very automatable
   - Use $num\_states$ bits to represent the states
   - *Example states:* 0001, 0010, 0100, 1000
3. **Output Encoded:**

- **Minimizes** output logic
- Only works for Moore Machines
- Each state has to be encoded uniquely, but the outputs must be 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
3. Output Encoded:

- **Minimizes** output logic
- Only works for Moore Machines
- Each state has to be encoded **uniquely**, but the outputs

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

- $\text{Bit}_1$ encodes **yellow** light output
- $\text{Bit}_2$ encodes **red** light output
Moore vs. Mealy Machines
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 vs. Mealy FSM

- 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

Mealy FSM

What are the tradeoffs?
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
Implementing Sequential Logic Using Verilog
We have seen an overview of Verilog

Discussed *structural* and *behavioral* modeling

Showed *combinational logic* constructs

**This week**

- Sequential logic constructs in Verilog
- Developing *testbenches* for simulation
Combinational + Memory = Sequential Circuit

Sequential Circuit

Combinational Circuit

Storage Element

inputs

outputs
Sequential Logic in Verilog

- Define blocks that have memory
  - *Flip-Flops, Latches, Finite State Machines*

- **Sequential Logic** state transition is triggered by a ‘CLOCK’ event
  - Latches are sensitive to level of the signal
  - Flip-flops are sensitive to the transitioning of clock

- Combinational constructs are **not** sufficient
  - We need **new constructs**:
    - *always*
    - *posedge/negedge*
The “always” Block

```verbatim
always @(sensitivity list)
    statement;
```

Whenever the event in the sensitivity list occurs, the statement is executed.
Example: D Flip-Flop

module flop(input clk,
            input [3:0] d,
            output reg [3:0] q);

always @(posedge clk)
q <= d; // pronounced “q gets d”

endmodule

- **posedge** defines a rising edge (transition from 0 to 1).
- Statement executed when the **clk signal rises (posedge of clk)**
- Once the **clk signal rises**: the value of **d** is copied to **q**
Example: D Flip-Flop

```
module flop(input clk,
            input [3:0] d,
            output reg [3:0] q);

  always @(posedge clk)
  begin
    q <= d;  // pronounced “q gets d”
  end
endmodule
```

- `assign` statement is **not** used within an always block
- `<=` describes a **non-blocking** assignment
  - We will see the difference between **blocking assignment** and **non-blocking assignment** soon
Example: D Flip-Flop

module flop(input clk,  
            input [3:0] d,  
            output reg [3:0] q);

always @(posedge clk)
    q <= d;                // pronounced “q gets d”

endmodule

- Assigned variables need to be declared as reg
- The name reg does not necessarily mean that the value is a register (It could be, but it does not have to be)
- We will see examples later
Asynchronous and Synchronous Reset

- **Reset** signals are used to initialize the hardware to a known state
  - Usually activated at system start (on power up)

- **Asynchronous Reset**
  - The reset signal is sampled independent of the clock
  - Reset gets the highest priority
  - Sensitive to glitches, may have metastability issues
    - Will be discussed in Lecture 8

- **Synchronous Reset**
  - The reset signal is sampled with respect to the clock
  - The reset should be active long enough to get sampled at the clock edge
  - Results in completely synchronous circuit
In this example: two events can trigger the process:

- A **rising edge** on clk
- A **falling edge** on reset
D Flip-Flop with Asynchronous Reset

```verbatim
module flop_ar (input clk,
              input reset,
              input [3:0] d,
              output reg [3:0] q);

always @ (posedge clk, negedge reset)
begin
  if (reset == 0) q <= 0;   // when reset
  else               q <= d;   // when clk
end
endmodule
```

- For longer statements, a `begin-end` pair can be used
  - To improve readability
  - In this example, it was not necessary, but it is a good idea
D Flip-Flop with Asynchronous Reset

- First reset is checked: if reset is 0, q is set to 0.
  - This is an asynchronous reset as the reset can happen independently of the clock (on the negative edge of reset signal)
- If there is no reset, then regular assignment takes effect
The process is only sensitive to clock

- Reset **happens only** when the **clock rises**. This is a **synchronous** reset
A flip-flop with **enable** and **reset**
- Note that the **en** signal is *not* in the **sensitivity list**
- **q** gets **d** only when **clk** is rising and **en** is **1**
Example: D Latch

module latch (input clk,
    input [3:0] d,
    output reg [3:0] q);

always @ (clk, d)
    if (clk) q <= d; // latch is transparent when
                     // clock is 1
endmodule
Summary: Sequential Statements So Far

- Sequential statements are within an `always` block

- The sequential block is triggered with a change in the sensitivity list

- Signals assigned within an `always` must be declared as `reg`

- We use `<=` for (non-blocking) assignments and do not use `assign` within the always block.
Basics of **always** Blocks

```verilog
module example (input clk,
        input [3:0] d,
        output reg [3:0] q);

wire [3:0] normal;       // standard wire
reg [3:0] special;       // assigned in always

always @ (posedge clk)
    special <= d;       // first FF array

assign normal = ~ special; // simple assignment

always @ (posedge clk)
    q <= normal;       // second FF array
endmodule
```

You can have as many **always** blocks as needed!

Assignment to the same signal in different **always** blocks is not allowed!
Why Does an **always** Block Memorize?

```
module flop (input clk,
            input [3:0] d,
            output reg [3:0] q);

    always @(posedge clk)
      begin
        q <= d;   // when clk rises copy d to q
      end

endmodule
```

- This statement describes what happens to signal `q`
- ... but what happens when the clock is not rising?
- The value of `q` is preserved (memorized)
An always Block Does NOT Always Memorize

module comb (input inv,
             input [3:0] data,
             output reg [3:0] result);

    always @ (inv, data)   // trigger with inv, data
        if (inv) result <= ~data;  // result is inverted data
        else result <= data;    // result is data

endmodule

- This statement describes what happens to signal result
  - When inv is 1, result is ~data
  - When inv is not 1, result is data

- The circuit is combinational (no memory)
  - result is assigned a value in all cases of the if .. else block, always
always Blocks for Combinational Circuits

- An **always** block defines **combinational logic** if:
  - All outputs are always (**continuously**) updated
  1. All right-hand side signals are in the sensitivity list
     - You can use **always @*** for short
  2. All left-hand side signals get assigned in every possible condition of **if .. else** and **case** blocks

- It is easy to make mistakes and **unintentionally describe memorizing elements** (latches)
  - **Vivado** will most likely warn you. Make sure you check the warning messages

- **Always** blocks allow powerful combinational logic statements
  - **if .. else**
  - **case**
Sequential or Combinational?

Sequential

```verilog
wire enable, data;
reg out_a, out_b;

always @ (*) begin
  out_a = 1'b0;
  if (enable) begin
    out_a = data;
    out_b = data;
  end
end
```

No assignment for ~enable

Sequential

```verilog
wire enable, data;
reg out_a, out_b;

always @ (data) begin
  out_a = 1'b0;
  out_b = 1'b0;
  if (enable) begin
    out_a = data;
    out_b = data;
  end
end
```

Not in the sensitivity list
The always Block is NOT Always Practical/Nice

Both statements describe the same multiplexer

In this case, the always block is more work
always Block for Case Statements (Handy!)

module sevensegment (input [3:0] data,
    output reg [6:0] segments);

always @ ( * ) // * is short for all signals
    case (data) // case statement
        4'd0: segments = 7'b111_1110; // when data is 0
        4'd1: segments = 7'b011_0000; // when data is 1
        4'd2: segments = 7'b110_1101;
        4'd3: segments = 7'b111_1001;
        4'd4: segments = 7'b011_0011;
        4'd5: segments = 7'b101_1011;
        // etc etc
        default: segments = 7'b000_0000; // required
    endcase

endmodule
Summary: always Block

- **If .. else** can only be used in **always** blocks

- The **always** block is **combinational** only if all **regs** within the block are always assigned to a signal
  - Use the **default** case to make sure you do not forget an unimplemented case, which may otherwise result in a latch

- Use **casex** statement to be able to check for don’t cares
Non-Blocking and Blocking Assignments

**Non-blocking (<=)**

```verilog
always @(a)
begin
    a <= 2'b01;
b <= a;
// all assignments are made here
// b is not (yet) 2'b01
end
```

- All assignments are made at the end of the block
- All assignments are made in parallel, process flow is not-blocked

**Blocking (=)**

```verilog
always @(a)
begin
    a = 2'b01;
// a is 2'b01
    b = a;
// b is now 2'b01 as well
end
```

- Each assignment is made immediately
- Process waits until the first assignment is complete, it blocks progress

---
Why use (Non)-Blocking Statements

- There are technical reasons why both are required
  - It is out of the scope of this course to discuss these

- Blocking statements allow sequential descriptions
  - More like a programming language

- If the sensitivity list is correct, blocks with non-blocking statements will always evaluate to the same result
  - This may require some additional iterations
Example: Blocking Assignment

- Assume all inputs are initially ‘0’

```verbatim
always @ ( * )
begin
    p = a ^ b ;  // p = 0 1
    g = a & b ;  // g = 0 0
    s = p ^ cin ;  // s = 0 1
    cout = g | (p & cin) ;  // cout = 0 0
end
```

- If `a` changes to ‘1’
  - All values are updated in order
The Same Example: Non-Blocking Assignment

- Assume all inputs are initially ‘0’

```verbatim
always @ ( * )
begin
    p    <= a ^ b ;       // p    = 0 1
    g    <= a & b ;       // g    = 0 0
    s    <= p ^ cin ;     // s    = 0 0
    cout <= g | (p & cin) ; // cout = 0 0
end
```

- If `a` changes to ‘1’
  - All assignments are concurrent
  - When `s` is being assigned, `p` is still 0
After the first iteration, \( p \) has changed to ‘1’ as well

```vhdl
always @ ( * )
begin
    p    <= a ^ b ;       // p    = 1 1
    g    <= a & b ;       // g    = 0 0
    s    <= p ^ cin ;     // s    = 0 1
    cout <= g | (p & cin) ; // cout = 0 0
end
```

Since there is a change in \( p \), the process triggers again
- This time \( s \) is calculated with \( p=1 \)
Rules for Signal Assignment

- Use `always @(posedge clk)` and non-blocking assignments (<=) to model synchronous sequential logic.

```verilog
always @(posedge clk)
  q <= d; // non-blocking
```

- Use continuous assignments (`assign`) to model simple combinational logic.

```verilog
assign y = a & b;
```
Rules for Signal Assignment (Cont.)

- Use **always @ (*)** and **blocking assignments (=)** to model more complicated combinational logic.

- You **cannot** make assignments to the same signal in more than one always block or in a **continuous assignment**

```
always @ (*)
a = b;
always @ (*)
a = c;
```

```
always @ (*)
a = b;
assign a = c;
```
Recall: Finite State Machines (FSMs)

- Each FSM consists of three separate parts:
  - next state logic
  - state register
  - output logic
Recall: Finite State Machines (FSMs) Comprise

- **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
FSM Example 1: Divide the Clock Frequency by 3

The output $Y$ is HIGH for one clock cycle out of every 3. In other words, the output divides the frequency of the clock by 3.
module divideby3FSM (input clk, input reset, output q);

reg [1:0] state, nextstate;

parameter S0 = 2'b00;
parameter S1 = 2'b01;
parameter S2 = 2'b10;

- We define **state** and **nextstate** as 2-bit **reg**
- The parameter descriptions are **optional**, it makes reading easier
Implementing FSM Example 1: State Register

```vhdl
// state register
always @ (posedge clk, posedge reset)
  if (reset) state <= S0;
  else state <= nextstate;
```

- This part defines the state register (memorizing process)
- Sensitive to only clk, reset
- In this example, reset is active when it is ‘1’ (active-high)
Implementing FSM Example 1: Next State Logic

// next state logic

always @ (*)

    case (state)
        S0:   nextstate = S1;
        S1:   nextstate = S2;
        S2:   nextstate = S0;
        default: nextstate = S0;
    endcase
In this example, output depends only on state

- **Moore type FSM**
Implementation of FSM Example 1

```verilog
module divideby3FSM (input clk, input reset, output q);
    reg [1:0] state, nextstate;

    parameter S0 = 2'b00; parameter S1 = 2'b01; parameter S2 = 2'b10;

    always @(posedge clk, posedge reset) // state register
        if (reset) state <= S0;
        else state <= nextstate;

    always @(*) // next state logic
        case (state)
            S0:    nextstate = S1;
            S1:    nextstate = S2;
            S2:    nextstate = S0;
            default: nextstate = S0;
        endcase
    assign q = (state == S0); // output logic
endmodule
```
FSM Example 2: Smiling Snail

- 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.
We did not cover the following.

They are for your preparation.
module SmilingSnail (input clk,
    input reset,
    input number,
    output smile);

reg [1:0] state, nextstate;

parameter S0 = 2'b00;
parameter S1 = 2'b01;
parameter S2 = 2'b10;
parameter S3 = 2'b11;
This part defines the state register (memorizing process)

Sensitive to only clk, reset

In this example reset is active when ‘1’ (active-high)
// next state logic
always @ (*)
case (state)
    S0: if (number) nextstate = S1;
        else nextstate = S0;
    S1: if (number) nextstate = S2;
        else nextstate = S0;
    S2: if (number) nextstate = S2;
        else nextstate = S3;
    S3: if (number) nextstate = S1;
        else nextstate = S0;
    default: nextstate = S0;
endcase
Implementing FSM Example 2: Output Logic

// output logic
assign smile = (number & state == S3);

- In this example, output depends on state and input
  - Mealy type FSM

- We used a simple combinational assignment
module SmilingSnail (input clk, 
    input reset, 
    input number, 
    output smile);

    reg [1:0] state, nextstate;

    parameter S0 = 2'b00;
    parameter S1 = 2'b01;
    parameter S2 = 2'b10;
    parameter S3 = 2'b11;

    // state register
    always @ (posedge clk, posedge reset)
        if (reset) state <= S0;
        else state <= nextstate;

    // next state logic
    always @ (*) // next state logic
        case (state)
            S0: if (number)
                nextstate = S1;
            else
                nextstate = S0;
            S1: if (number)
                nextstate = S2;
            else
                nextstate = S0;
            S2: if (number)
                nextstate = S2;
            else
                nextstate = S3;
            S3: if (number)
                nextstate = S1;
            else
                nextstate = S0;
            default: nextstate = S0;
        endcase

    // output logic
    assign smile = (number & state==S3);

endmodule
What Did We Learn?

- Basics of defining **sequential circuits** in Verilog

- The **always** statement
  - Needed for defining memorizing elements (flip-flops, latches)
  - Can also be used to define **combinational circuits**

- **Blocking** vs **Non-blocking** statements
  - `=` assigns the value **immediately**
  - `<=` assigns the value **at the end of the block**

- Writing FSMs
  - Next state logic
  - State assignment
  - Output logic
Next Lecture: Timing and Verification
Design of Digital Circuits

Lecture 7: Sequential Logic Design

Prof. Onur Mutlu
ETH Zurich
Spring 2018
15 March 2018
Backup Slides

- Different types of flip flops
The D Flip-Flop
Enabled Flip-Flops

- **Inputs:** CLK, D, EN
  - The enable input (EN) controls when new data (D) is stored

- **Function:**
  - EN = 1: D passes through to Q on the clock edge
  - EN = 0: the flip-flop retains its previous state

![Internal Circuit Diagram]
Resetable Flip-Flop

- **Inputs:** CLK, D, Reset
  - The Reset is used to set the output to 0.
- **Function:**
  - \textbf{Reset} = 1: Q is forced to 0
  - \textbf{Reset} = 0: the flip-flop behaves like an ordinary D flip-flop

**Symbols**
Resettable Flip-Flops

- Two types:
  - **Synchronous**: resets at the clock edge only
  - **Asynchronous**: resets immediately when Reset = 1

- Asynchronously resettable flip-flop requires changing the internal circuitry of the flip-flop (see Exercise 3.10)

- Synchronously resettable flip-flop?
Settable Flip-Flop

- **Inputs**: CLK, D, Set
- **Function**:
  - **Set = 1**: Q is set to 1
  - **Set = 0**: the flip-flop behaves like an ordinary D flip-flop