CSE 140: Components and Design Techniques for Digital Systems

Lecture 7:
Sequential Networks

CK Cheng
Dept. of Computer Science and Engineering
University of California, San Diego
Part II: Sequential Networks

• Introduction
  – Sequential circuits
  – Memory hierarchy
  – Basic mechanism of memory

• Basic Building Blocks
  – Latches
  – Flip-Flops
  – Examples of Memory Modules

• Implementation
  – Finite state machine
What is a sequential circuit?

“A circuit whose output depends on current inputs and past outputs”

“A circuit with memory”

Memory: a key parameter is Time
**Sequential Networks: Key features**

Memory: Flip flops  
Specification: Finite State Machines  
Implementation: Excitation Tables  
Main Theme: Timing

**Present time** = \( t \) and **next time** = \( t+1 \)

Timing constraints to separate the present and next times.

\[
y_i = f_i(S^t, X) \\
s_i^{t+1} = g_i(S^t, X)
\]
Sequential Networks: Key features

Main Theme: Timing

**Present time = t and next time = t+1**

Timing constraints to separate the present and next times.

\[ y_i = f_i(S^t, X) \]
\[ s_i^{t+1} = g_i(S^t, X) \]
Memory Hierarchy

• What are registers made of?
  Flip-Flops, Latches
Fundamental Memory Mechanism
Memory Mechanism: Capacitive Load

• Fundamental building block of sequential circuits
• Two outputs: $\overline{Q}$, $Q$
• There is a feedback loop!
  • In a typical combinational logic, there is no feedback loop.
• No inputs
Capacitive Loads

• Consider the two possible cases:
  – \( Q = 0 \): then \( Q' = 1 \) and \( Q = 0 \) (consistent)
  – \( Q = 1 \): then \( Q' = 0 \) and \( Q = 1 \) (consistent)
  – Bistable circuit stores 1 bit of state in the state variable, \( Q \) (or \( Q' \))
  – Hold the value due to capacitive charges and feedback loop strengthening

• But there are no inputs to control the state
Q. Given a memory component made out of a loop of inverters, the number of inverters in the loop has to be
A. Even
B. Odd
C. No constraints
Basic Building Blocks

• Latches (Level Sensitive)
  – SR Latches, D Latches
• Flip-Flops (Edge Triggered)
  – D FFs, JK FFs, T FFs
• Examples of Memory Modules
  – Registers, Shift Registers, Pattern Recognizers, Counters, FIFOs
Flight attendant call button

• Flight attendant call button
  - Press call: light turns on
    • **Stays on** after button released
  - Press cancel: light turns off
  - Logic gate circuit to implement this?

• SR latch implementation
  - Call=1 : sets Q to 1 and keeps it at 1
  - Cancel=1 : resets Q to 0
SR (Set/Reset) Latch

• SR Latch

• Consider the four possible cases:
  – \( S = 1, R = 0 \)
  – \( S = 0, R = 1 \)
  – \( S = 0, R = 0 \)
  – \( S = 1, R = 1 \)
SR Latch Analysis

- $S = 1$, $R = 0$: then $Q = 1$ and $\overline{Q} = 0$

- $S = 0$, $R = 1$: then $Q = 0$ and $\overline{Q} = 1$
SR Latch Analysis

- $S = 0, R = 0$: then $Q = Q_{prev}$
  - $Q_{prev} = 0$
  - $Q_{prev} = 1$

- $S = 1, R = 1$: then $Q = 0$ and $\overline{Q} = 0$
SR Latch

Inputs: S, R            State: (Q, y)

\[ y = (S+Q)' \]

\[ Q = (R+y)' \]

Inputs: S, R            State: (Q, y)
<table>
<thead>
<tr>
<th>id</th>
<th>S</th>
<th>R</th>
<th>Qt</th>
<th>yt</th>
<th>Qt</th>
<th>yt</th>
<th>Qt</th>
<th>yt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
Q = (R+y)'
\]

\[
y = (S+Q)'
\]
### State Table

<table>
<thead>
<tr>
<th>id</th>
<th>S</th>
<th>R</th>
<th>Qt</th>
<th>yt</th>
<th>Qt</th>
<th>yt</th>
<th>Qt</th>
<th>yt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### “State Table”

<table>
<thead>
<tr>
<th>Qy\SR</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>11</td>
<td>01</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>01</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>01</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>00</td>
</tr>
</tbody>
</table>

18
CASES:
SR=01: (Q,y) = (0,1)
SR=10: (Q,y) = (1,0)
SR=11: (Q,y) = (0,0)
SR= 00: (Q,y) does not change if (Q,y) = (1,0) or (0,1)
  However, when (Q,y) = (0,0) or (1,1), the output keeps changing

Remark: To verify the design, we need to enumerate all combinations.
State diagram

<table>
<thead>
<tr>
<th>Qy \ SR</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>11</td>
<td>01</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>01</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>01</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>00</td>
</tr>
</tbody>
</table>
CASES:
SR=01: (Q,y) = (0,1)
SR=10: (Q,y) = (1,0)
SR=11: (Q,y) = (0,0)
SR= 00: (Q,y) does not change if (Q,y)=(1,0) or (0,1)
   However, when (Q,y) = (0,0) or (1,1), the output keeps changing.

Q. Suppose that we can set the initial state (Q,y) =(0,1). To avoid the SR latch output from toggling or behaving in an undefined way which input combinations should be avoided:
A. (S, R) = (0, 0)
B. (S, R) = (1, 1)
C. None of the above
CASES:
SR=01: (Q,y) = (0,1)
SR=10: (Q,y) = (1,0)
SR=11: (Q,y) = (0,0)
SR= 00: (Q,y) does not change if (Q,y)=(1,0) or (0,1)
   However, when (Q,y) = (0,0) or (1,1), the output keeps changing

We set the initial state (Q,y)=(0,1) or (1,0). To avoid the state (Q,y)= (0,0) or (1,1), we block the input SR=11. Thus, without input SR=11, the state can only be (Q,y)=(0,1) or (1,0).
The only way to reach state \((Q,y)=(0,0)\) or \((1,1)\) is via edge labeled \(SR=11\).
SR Latch Analysis

– \( S = 0, R = 0 \): then \( Q = Q_{prev} \) and \( \bar{Q} = \bar{Q}_{prev} \) (memory!)

\[
\begin{array}{c}
\text{Q}_{prev} = 0 \\
\begin{array}{c}
\text{N1} \\
R \quad 0 \\
\quad 0 \\
\quad 1 \\
\end{array}
\quad \begin{array}{c}
\text{N2} \\
S \quad 0 \\
\quad 0 \\
\quad 0 \\
\end{array}
\quad 0 \\
\quad 1 \\
\quad Q
\end{array}
\quad
\begin{array}{c}
\text{Q}_{prev} = 1 \\
\begin{array}{c}
\text{N1} \\
R \quad 0 \\
\quad 0 \\
\quad 0 \\
\quad 1 \\
\end{array}
\quad \begin{array}{c}
\text{N2} \\
S \quad 0 \\
\quad 0 \\
\quad 1 \\
\end{array}
\quad 1 \\
\quad 0 \\
\quad 0 \\
\quad 0 \\
\quad \bar{Q}
\end{array}
\]

– \( S = 1, R = 1 \): then \( Q = 0 \) and \( \bar{Q} = 0 \) (invalid state: \( Q \neq \text{NOT} \ \bar{Q} \))

\[
\begin{array}{c}
\begin{array}{c}
\text{N1} \\
R \quad 1 \\
\quad 0 \\
\quad 0 \\
\quad 0 \\
\end{array}
\quad \begin{array}{c}
\text{N2} \\
S \quad 1 \\
\quad 0 \\
\quad 0 \\
\quad 1 \\
\end{array}
\quad 0 \\
\quad 0 \\
\quad \bar{Q}
\end{array}
\]

\[24\]
C A S E S
SR=01: \((Q, y) = (0, 1)\)
SR=10: \((Q, y) = (1, 0)\)
SR=11: \((Q, y) = (0, 0)\)
SR = 00: if \((Q, y) = (0, 0)\) or \((1, 1)\), the output keeps changing
Solutions: Avoid the case that \(SR = (1, 1)\).

\[
\begin{array}{cccc}
\text{inputs} & \text{SR} & \text{Characteristic Expression} \\
\hline
\text{00} & 00 & Q(t+1) = S(t) + R' (t)Q(t) \\
\text{01} & 01 & \ \\
\text{10} & 10 & \ \\
\text{11} & 11 & \ \\
\end{array}
\]

\[
\begin{array}{ccccc}
\text{PS} & \text{Q(t)} & 00 & 01 & 10 & 11 \\
\hline
0 & 0 & 0 & 1 & - \\
1 & 1 & 0 & 1 & - \\
\end{array}
\]

\[
Q(t+1) \quad \text{NS (next state)}
\]
SR Latch Symbol

• SR stands for Set/Reset Latch
  – Stores one bit of state ($Q$)

• Control what value is being stored with $S$, $R$ inputs
  – Set: Make the output 1 ($S = 1$, $R = 0$, $Q = 1$)
  – Reset: Make the output 0 ($S = 0$, $R = 1$, $Q = 0$)

• Must do something to avoid invalid state (when $S = R = 1$)

SR Latch Symbol

\[ \text{SR} \quad \text{Latch} \quad \text{Symbol} \]
D Latch

- Two inputs: $CLK$, $D$
  - $CLK$: controls when the output changes
  - $D$ (the data input): controls what the output changes to

- Function
  - When $CLK = 1$, $D$ passes through to $Q$ (the latch is transparent)
  - When $CLK = 0$, $Q$ holds its previous value (the latch is opaque)

- Avoids invalid case when $Q \neq \text{NOT } \overline{Q}$
D Latch Internal Circuit

![D-Latch Symbol](image)
D Latch Internal Circuit

<table>
<thead>
<tr>
<th>CLk</th>
<th>D</th>
<th>$\overline{D}$</th>
<th>S</th>
<th>R</th>
<th>Q</th>
<th>$\overline{Q}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
D Latch Internal Circuit

CLK | D | \overline{D} | S | R | Q | \overline{Q} \\
---|---|---|---|---|---|---
0  | X | X  | 0  | 0  | Q \text{prev} | \overline{Q} \text{prev} \\
1  | 0 | 1  | 0  | 1  | 0  | 1 \\
1  | 1 | 0  | 1  | 0  | 1  | 0
D Flip-Flop

- Two inputs: \( CLK, D \)
- Function
  - The flip-flop “samples” \( D \) on the rising edge of \( CLK \)
    - 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
D Flip-Flop Internal Circuit

CLK

D Q
N1

CLK

L1 L2
Q Q

D Q
Q

CLK

D Q
Q Q
D Flip-Flop Internal Circuit

- Two back-to-back latches (L1 and L2) controlled by complementary clocks
- When $CLK = 0$
  - L1 is transparent, L2 is opaque
  - $D$ passes through to N1
- When $CLK = 1$
  - L2 is transparent, L1 is opaque
  - N1 passes through to $Q$
- Thus, on the edge of the clock (when $CLK$ rises from 0 → 1)
  - $D$ passes through to $Q$
D Flip-Flop vs. D Latch

CLK

D Q

Q

D Q

Q

CLK

D

Q (latch)

Q (flop)
D Flip-Flop vs. D Latch

CLK

D  Q

Q

D  Q

Q

CLK

D

Q (latch)

Q (flop)
Latch and Flip-flop (two latches)

A latch can be considered as a door

CLK = 0, door is shut
CLK = 1, door is unlocked

A flip-flop is a two door entrance

CLK = 1
CLK = 0
CLK = 1
D Flip-Flop (Delay)

Characteristics Expression: \( Q(t+1) = D(t) \)

State Table:

<table>
<thead>
<tr>
<th>Id</th>
<th>D</th>
<th>Q(t)</th>
<th>Q(t+1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\( NS = Q(t+1) \)
Can D flip-flop serve as a memory component?
A. Yes
B. No
JK F-F

State table

\[
\begin{array}{c|cccc}
\text{PS} & \text{JK} & 00 & 01 & 10 & 11 \\
0 & 0 & 0 & 1 & \? \\
1 & 1 & 0 & 1 & \? \\
\end{array}
\]

\(Q(t+1)\)
JK F-F

Characteristic Expression

\[ Q(t+1) = Q(t)K'(t) + Q'(t)J(t) \]
T Flip-Flop (Toggle)

Characteristic Expression

\[ Q(t+1) = Q'(t)T(t) + Q(t)T'(t) \]
Using a JK F-F to implement a D and T F-F

![Circuit Diagram]

iClicker
What is the function of the above circuit?
A. D F-F
B. T F-F
C. None of the above
Using a JK F-F to implement a D and T F-F

T flip flop
Rising vs. Falling Edge D Flip-Flop

The triangle means clock input, edge triggered.

Symbol for rising-edge triggered D flip-flop

Symbol for falling-edge triggered D flip-flop

Internal design: Just invert servant clock rather than master.

The triangle means clock input, edge triggered.

rising edges

falling edges

Clk
Enabled D-FFs

• **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
Resettable Flip-Flops

- **Inputs:** \( CLK, D, \text{Reset} \)
- **Function:**
  - \( \text{Reset} = 1 \): \( Q \) is forced to 0
  - \( \text{Reset} = 0 \): flip-flop behaves as ordinary D flip-flop
- **Two types:**
  - **Synchronous:** resets at the clock edge only
  - **Asynchronous:** resets immediately when \( \text{Reset} = 1 \)
- Asynchronously resettable flip-flop requires changing the internal circuitry of the flip-flop
- Synchronously resettable flip-flop circuit:

  ![Synchronously Resettable Flip-Flop Circuit](image)

- There are also synch/asynch settable FFs
Bit Storage Overview

SR latch

S (set)  Q
R (reset)

Level-sensitive SR latch

S  C  Q
R  R1  Q

D latch

D  C  Q

SR latch

S=1 sets Q to 1, S and R only have effect when C=1. We can design outside circuit so SR=11 never happens when C=1. Problem: avoiding SR=11 can be a burden.

SR can’t be 11 if D is stable before and while C=1, and will be 11 for only a brief glitch even if D changes while C=1. *Transition may cross many levels of latches.

Only loads D value present at rising clock edge, so values can’t propagate to other flip-flops during same clock cycle. *Transition happens between two level of flip-flops.

D flip-flop

Dm  Qm
Cs  Qs

Ds  Qs'

Q
Building blocks with FFs: Basic Register
### Shift register

- Holds & shifts samples of input

---

#### Diagram

- **IN** is connected to the first **D Q** flip-flop.
- **CLK** is connected to all flip-flops.
- The output of each flip-flop is connected to the input of the next flip-flop, forming a chain:
  - **OUT1** connects to **D Q**
  - **OUT2** connects to **D Q**
  - **OUT3** connects to **D Q**
  - **OUT4** connects to **D Q**

#### Table

<table>
<thead>
<tr>
<th>Time</th>
<th>Input</th>
<th>OUT1</th>
<th>OUT2</th>
<th>OUT3</th>
<th>OUT4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Shift register

- Holds & shifts samples of input

<table>
<thead>
<tr>
<th>Time</th>
<th>Input</th>
<th>OUT1</th>
<th>OUT2</th>
<th>OUT3</th>
<th>OUT4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</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>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Pattern Recognizer

- Combinational function of input samples
Counters

• Sequences through a fixed set of patterns
Describing Sequential Ckts

- State diagrams and next-state tables are not suitable for describing very large digital designs.
  - Large circuits must be described in a modular fashion -- as a collection of cooperating FSMs.
- BSV is a modern programming language to describe cooperating FSMs.
  - We will give various examples of FSMs in BSV.
Modulo-4 counter circuit

\[ q_0^{t+1} = \sim \text{inc} \cdot q_0^t + \text{inc} \cdot \sim q_0^t \]
\[ q_1^{t+1} = \sim \text{inc} \cdot q_1^t + \text{inc} \cdot \sim q_1^t \cdot q_0^t + \text{inc} \cdot q_1^t \cdot \sim q_0^t \]

“Optimized” logic
\[ q_0^{t+1} = \text{inc} \oplus q_0^t \]
\[ q_1^{t+1} = (\text{inc} == 1) \ ? q_0^t \oplus q_1^t : q_1^t \]
Modulo-4 counter circuit

$q_0^{t+1} = \neg inc \cdot q_0^t + inc \cdot \neg q_0^t$
$q_1^{t+1} = \neg inc \cdot q_1^t + inc \cdot \neg q_1^t \cdot q_0^t$
$+ inc \cdot q_1^t \cdot \neg q_0^t$

“Optimized” logic
$q_0^{t+1} = inc \oplus q_0^t$
$q_1^{t+1} = (inc == 1) ? q_0^t \oplus q_1^t : q_1^t$

<table>
<thead>
<tr>
<th>PS\input</th>
<th>inc=0</th>
<th>inc=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>00</td>
</tr>
</tbody>
</table>

PS: $q_1^t \ q_0^t$, NS: $q_1^{t+1} \ q_0^{t+1}$
module moduloCounter(Counter);
  Reg#(Bit#(2)) cnt <- mkReg(0);
  method Action inc;
    cnt <=(!cnt[1]&cnt[0] | cnt[1]&!cnt[0], !cnt[0]);
  endmethod
  method Bit#(2) read;
    return cnt;
  endmethod
endmodule

State specification

Initial value

An action to specify how the value of the cnt is to be set
Interface

• Modulo counter has the following interface, i.e., type

```verilog
interface Counter;
    method Action inc;
    method Bit#(2) read;
endinterface
```

• An interface can have many different implementations
  – For example, the numbers may be represented as Gray code
Modules

- A module in BSV is like a class definition in Java or C++
  - It has internal state
  - The internal state can only be read and manipulated by the (interface) methods
  - An action specifies which state elements are to be modified
  - Actions are atomic -- either all the specified state elements are modified or none of them are modified (no partially modified state is visible)
FIFO Interface

```haskell
interface Fifo#(numeric type size, type t);
    method Bool notFull;
    method Bool notEmpty;
    method Action enq(t x);
    method Action deq;
    method t first;
endinterface
```

- enq should be called only if notFull returns True;
- deq and first should be called only if notEmpty returns True
module mkCFFifo (Fifo#(1, t));
    Reg#(t)   d  <- mkRegU;
    Reg#(Bool) v  <- mkReg(False);
    method Bool notFull;
        return !v;
    endmethod
    method Bool notEmpty;
        return v;
    endmethod
    method Action enq(t x);
        v <= True; d <= x;
    endmethod
    method Action deq;
        v <= False;
    endmethod
    method t first;
        return d;
    endmethod
endmodule
FIFO Module: methods with guarded interfaces

- Every method has a *guard* (rdy wire); the value returned by a value method is meaningful only if its guard is true
- Every action method has an *enable signal* (en wire); an action method is invoked (en is set to true) only if the guard is true
- Guards make it possible to transfer the responsibility of the correct use of a method from the user to the compiler
- Guards are extraordinarily convenient for programming and also enhance modularity of the code

```
interface Fifo#{numeric type size,
    type t);
    method Action enq(t x);
    method Action deq;
    method t first;
endinterface
```
module mkCFFifo (Fifo#(1, t));
    Reg#(t) d <- mkRegU;
    Reg#(Bool) v <- mkReg(False);
    method Action enq(t x) if (!v); not full
        v <= True; d <= x;
    endmethod
    method Action deq if (v); not empty
        v <= False;
    endmethod
    method t first if (v); not empty
        return d;
    endmethod
endmodule