Lecture 7: Sequential Networks

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

1. Introduction
   1. Sequential circuits
   2. Memory hierarchy
   3. Basic mechanism of memory

2. Basic Building Blocks
   1. Latches
   2. Flip-Flops
   3. Examples of Memory Modules

3. Implementation
   1. Finite state machine
1. Introduction: 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

![Diagram of a sequential circuit with inputs $x_i$, memory $S_i$, and output $y_i$.]
1. Introduction: Sequential Network 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.

\[ y_i = f_i(S^t, X) \]
\[ s_{i}^{t+1} = g_i(S^t, X) \]
1. Introduction: Sequential Network Key features

Main Theme: Timing

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

Timing constraints to separate the present and next times.

\[ y_i = f_i(S^t, X) \]
\[ s_{i+1} = g_i(S^t, X) \]
1. Introduction: Different Types of Memory/Storage

- Typical Computer Memory Hierarchy
- Tradeoff between speed (latency) and size
- Size relates to
  - Storage density (area/bit)
  - Power (power/bit)

- Registers/Latches
- Register File (Static Memory - SRAM)
- Cache Memory (Static Memory - SRAM)
- Main Memory (Dynamic Memory – DRAM)
- NVM Main Memory (Non-Volatile Memory– e.g. Flash)
- Disk
1. Introduction: Fundamental Memory Mechanism
1. Introduction: 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
1. Introduction: 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**
Example

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
1. Introduction: Memory Storage Mechanism

- Word line (WL) to access cell
- Read – measure voltage difference between B+ and B-
- Write – force values of B+ and B- which may flip the cell
- Static Random Access Memory - SRAM
2. 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
2. Basic Building Blocks: 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
2. Basic Building Blocks: SR (Set/Reset) Latch

- SR Latch

- Consider the four possible cases:
  - $S = 1$, $R = 0$
  - $S = 0$, $R = 1$
  - $S = 0$, $R = 0$
  - $S = 1$, $R = 1$
2. Basic Building Blocks: SR Latch Analysis

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

- \( S = 0, R = 1 \): then \( Q = 0 \) and \( \overline{Q} = 1 \)
2. Basic Building Blocks: 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 $\bar{Q} = 0$
2. Basic Building Blocks: SR Latch

Inputs: S, R

State: (Q, y)

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

\[ Q = (R+y)' \]
Truth table of SR latch with incremental steps in time

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

\[ Q = (R + y)' \]
### “State Table” of SR latch

<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>0</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>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</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>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</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>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>0</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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \text{SR} )</th>
<th>(00)</th>
<th>(01)</th>
<th>(10)</th>
<th>(11)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Qy} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>

19
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 Table and State Diagram

### State Table

<table>
<thead>
<tr>
<th>( \text{SR} ) ( \text{Qy} )</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>

### State Diagram

[Diagram showing state transitions with labels 00, 01, 10, 11 representing state changes.]
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 \(SR=11\).
The only way to reach state \((Q,y)=(0,0)\) or \((1,1)\) is via edge labeled \(SR=11\).
2. Basic Building Blocks: SR Latch Analysis

- $S = 0, R = 0$: then $Q = Q_{prev}$ and $\overline{Q} = \overline{Q}_{prev}$ (memory!)

  \[ Q_{prev} = 0 \]

  \[ Q_{prev} = 1 \]

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

  \[ Q_{prev} = 0 \]

  \[ Q_{prev} = 1 \]
2. Basic Building Blocks: SR Latch

CASES
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).

State table

<table>
<thead>
<tr>
<th>PS</th>
<th>SR</th>
<th>Characteristic Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00</td>
<td>Q(t+1) = S(t)+R’(t)Q(t)</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Q(t+1) NS (next state)
2. Basic Building Blocks: 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]
2. Basic Building Blocks: 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 \( Q \neq \text{NOT} \overline{Q} \)
2. Basic Building Blocks: D Latch Internal Circuit
2. Basic Building Blocks: D Latch Internal Circuit

**Truth Table:**

<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>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2. Basic Building Blocks: D Latch Internal Circuit

```
<table>
<thead>
<tr>
<th>CLK</th>
<th>D</th>
<th>̅D</th>
<th>S</th>
<th>R</th>
<th>Q prev</th>
<th>̅Q prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>̅X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

2. Basic Building Blocks: 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 \textit{edge-triggered} device because it is activated on the clock edge

D Flip-Flop
\[ D \quad Q \]
\[ Q \]
2. Basic Building Blocks: D FF Internal Circuit

![D FF Internal Circuit Diagram]
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 to 1)
  - $D$ passes through to $Q$
D Flip-Flop vs. D Latch

CLK

D Q

Q (latch)

Q (flop)

D Q

Q
D Flip-Flop vs. D Latch
2. Basic Building Blocks: Latch and FF (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
2. Basic Building Blocks: D Flip-Flop (Delay)

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

<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>

State table

\[
\begin{array}{c|cc}
PS & 0 & 1 \\
\hline
D   & 0 & 1 \\
0   & 0 & 1 \\
1   & 0 & 1 \\
\end{array}
\]

NS = Q(t+1)
2. Basic Building Blocks: Example

Can D flip-flip serve as a memory component?
A. Yes
B. No
3. Implementation: Rising vs. Falling Edge D FF

Symbol for rising-edge triggered D flip-flop

Symbol for falling-edge triggered D flip-flop

The triangle means clock input, edge triggered

Internal design: Just invert servant clock rather than master

rising edges

falling edges
3. Implementation: 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:** $CLK$, $D$, $Reset$
- **Function:**
  - $Reset = 1$: $Q$ is forced to 0
  - $Reset = 0$: flip-flop behaves as ordinary D flip-flop
- **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
- Synchronously resettable flip-flop circuit:
- There are also synch/asynch settable FFs

Sources: TSR, Katz, Boriello & Vahid
3. Implementation: Bit Storage Overview

S=1 sets Q to 1, R=1 resets Q to 0. Problem: SR=11 yield undefined Q.

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.
Building blocks with FFs: Basic Register
3. Implementation: 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></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>
3. Implementation: Shift register

- Holds & shifts samples of input

```
IN   D   Q   OUT1   D   Q   OUT2   D   Q   OUT3   D   Q   OUT4
CLK

<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>
```
3. Implementation: Pattern Recognizer

- Combinational function of input samples
3. Implementation: Counters

• Sequences through a fixed set of patterns
Appendix: slides on BSV

Describing Sequential Ckts

- State diagrams and next-state tables nor are 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

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

“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

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

“Optimized” logic

\[ q_0^{t+1} = \text{inc} \oplus q_0^t \]
\[ q_1^{t+1} = (\text{inc} == 1) \ ? \ q_0^t \oplus q_1^t : q_1^t \]

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

```plaintext
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)
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
```
One-Element FIFO Implementation with guards

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);
    return  d;
endmethod
endmodule