Registers

- **Readings: 5.8-5.9.3**
- Storage unit. Can hold an n-bit value
- Composed of a group of n flip-flops
  - Each flip-flop stores 1 bit of information

\[
\begin{array}{c|c|c}
\text{Reset} & \text{Load} & \text{Action} \\
0 & 0 & Q = \text{old } Q \\
1 & 0 & Q = 0 \\
0 & 1 & Q = D \\
\end{array}
\]
**Shift Register**

- Register that shifts the binary values in one or both directions

```
In  D ff  Q  D ff  Q  D ff  Q  D ff  Q  Out
  D ff  clk  D ff  clk  D ff  clk  D ff  clk
  Clock
```

**Transfer of Data**

- 2 modes of communication: Parallel vs. Serial
  - Parallel: all bits transferred at the same time
  - Serial: one bit transferred at a time
  - Shift register can be used for serial transfer
### Shift Register w/Parallel Load

<table>
<thead>
<tr>
<th>Shift</th>
<th>Load</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Q = old Q</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Shift</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>Parallel Load</td>
</tr>
</tbody>
</table>

### Conversion between Parallel & Serial

![Diagram of conversion between parallel and serial data]
Bidirectional Shift Register w/Parallel Load

<table>
<thead>
<tr>
<th>Shift</th>
<th>Load</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Q = old Q</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Parallel Load</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Shift Up (V*2)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Shift Down (V/2)</td>
</tr>
</tbody>
</table>

Counters

- A reg. that goes through a specific state sequence
- \textit{n-bit Binary Counter}: counts from 0 to 2^{N}-1 in binary
- \textit{Up Counter}: Binary value increases by 1
- \textit{Down Counter}: Binary value decreases by 1
- 3-bit binary up counter state diagram:
Binary Up-Counter Imp.

Complex Binary Counter

<table>
<thead>
<tr>
<th>Load</th>
<th>Count</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Q = old Q</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Up Count</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Reset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Load Parallel</td>
</tr>
</tbody>
</table>
Arbitrary Sequence Counters

- Design a 3-bit count that goes through the sequence
  000->010->100->101->111->110->001->011->000->...

Counters in Verilog

module upcounter #(parameter WIDTH=8)
  (out, incr, reset, clk);

  output reg [WIDTH-1:0] out;
  input           incr, reset, clk;

endmodule
Memory

- Need method for storing large amounts of data
- Computer programs, data, pictures, etc.

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>00111110</td>
</tr>
<tr>
<td>000001</td>
<td>01101011</td>
</tr>
<tr>
<td>000010</td>
<td>01011101</td>
</tr>
<tr>
<td>000011</td>
<td>01100011</td>
</tr>
<tr>
<td>000100</td>
<td>00111110</td>
</tr>
<tr>
<td>000101</td>
<td>00000000</td>
</tr>
<tr>
<td>000110</td>
<td>11111111</td>
</tr>
<tr>
<td>000111</td>
<td>01010101</td>
</tr>
<tr>
<td>001000</td>
<td>10101010</td>
</tr>
<tr>
<td>001001</td>
<td>00100001</td>
</tr>
<tr>
<td>001010</td>
<td>11011101</td>
</tr>
</tbody>
</table>

- RAM: Random Access Memory, Read/Write
- ROM: Read-only Memory

8x4 RAM

<table>
<thead>
<tr>
<th>Write</th>
<th>In3</th>
<th>In2</th>
<th>In1</th>
<th>In0</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A2</th>
<th>A1</th>
<th>A0</th>
</tr>
</thead>
</table>
|    |    | Out3

<table>
<thead>
<tr>
<th>Out3</th>
<th>Out2</th>
<th>Out1</th>
<th>Out0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RAM Cell

- Requirements:
  - Store one bit of data
  - Change data based on input when row is selected

<table>
<thead>
<tr>
<th>Input</th>
<th>D</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row Select</td>
<td>en</td>
<td></td>
</tr>
</tbody>
</table>

RAM example

- Use a memory to do a programmable 32-picture animation on a 7-segment display
Verilog Memories

module memory16x6 (data_out, data_in, addr, we, clk);
output reg [5:0] data_out;
input [5:0] data_in;
input [3:0] addr;
input we, clk;
reg [5:0] mem [15:0];
always @(*)
data_out = mem[addr];
always @(posedge clk)
if (we)
    mem[addr] <= data_in;
endmodule

Field Programmable Gate Arrays (FPGAs)

- Readings: B.6-B.6.5
- Logic cells imbedded in a general routing structure
- Logic cells usually contain:
  - 6-input Boolean function calculator
  - Flip-flop (1-bit memory)
- All features electronically (re)programmable
Using an FPGA

Verilog

FPGA CAD Tools

Bitstream

Simulation

FPGA Programming

Bitstream

= 1 memory cell (stores 1 bit of info)
FPGA Combinational Logic

- How can we use Muxes and Programming bits to compute combinational binary function $F(A, B, C)$?

  ![8:1 MUX Diagram]

- Creates a “LUT” or lookup table.

FPGA Sequential Logic

- How do we put DFF’s onto LUT outputs only when we need them?

  ![3-LUT and D Flipflop Diagram]

- Creates a “LE” or logic block
How do we combine LE’s to build larger functions?

This is an Altera “LAB”.

Can’t do all-to-all/crossbar routing, so what?
FPGA CAD

- CAD = “Computer-Aided Design”
- Tech Mapping: Convert Verilog to LUTs
- Placement: Assign LUTs to specific locations
- Routing: Wire inputs to outputs
- Bitstream Generation: Convert mapping to bits

Verilog ➔ FPGA CAD Tools ➔ Bitstream

Verilog code for 2-input multiplexer:

```verilog
module AOI (F, A, B, C, D);
    output F;
    input A, B, C, D;
    assign F = ~((A & B) | (C & D));
endmodule
```

```verilog
module MUX2 (V, SEL, I, J);
    output V;
    input SEL, I, J;
    wire SELB, VB;
    not G1 (SELB, SEL);
    AOI G2 (VB, I, SEL, SELB, J);
    not G3 (V, VB);
endmodule
```