Review: Designing with FSM

- FSMs are critical tool in your design toolbox
  - Adapters, Protocols, Datapath Controllers, ...
- They often interact with other FSMs
- Important to design each well and to make them work together well.
- Keep your Verilog FSMs clean
  - Separate combinational part from state update
- Good state machine design is an iterative process
  - State encoding
  - Reduction
  - Assignment

Outline

- Review
- Registers
- Simple, important FSMs
  - Ring counters
  - Binary Counters
- Universal Shift Register
- Using Counters to build controllers
  - Different approach to FSM design

Registers

- Collections of flip-flops with similar controls and logic
  - Stored values somehow related (e.g., form binary value)
  - Share clock, reset, and set lines
  - Similar logic at each stage
- Examples
  - Shift registers
  - Counters
Shift-registers

- **Parallel load shift register:**
  - “Parallel-to-serial converter”
  - Also, works as “Serial-to-parallel converter”, if Q values are connected out.
  - Also get used as controllers (ala “ring counters”)

**Shift Register Verilog**

```verilog
module shift_reg (out, in, clk);
  output [4:1] out;
  input  in, clk;
  reg    [4:1] out;

  always @(posedge clk)
  begin
    out <= {out[3:1], in};
  end
endmodule
```

**Shift Register Application**

- **Parallel-to-serial conversion for serial transmission**
Register with selective load

- We often use registers to hold values for multiple clocks
  - Wait until needed
  - Used multiple times
- How do we modify our D flipflop so that it holds the value till we are done with it?
- A very simple FSM

<table>
<thead>
<tr>
<th>En</th>
<th>State</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>Q</td>
</tr>
<tr>
<td>1</td>
<td>Q</td>
<td>D</td>
</tr>
</tbody>
</table>

IQ: Design Register with Set/Reset

- Set forces state to 1
- Reset forces state to 0
- What might be a useful fourth option?

Counters

- Special sequential circuits (FSMs) that repeatedly sequence through a set of outputs.
- Examples:
  - binary counter: 000, 001, 010, 011, 100, 101, 110, 111, 000, 001, ...
  - gray code counter: 000, 010, 110, 100, 101, 111, 011, 001, 000, 010, 110, ...
  - one-hot counter: 0001, 0010, 0100, 0101, 1000, 0001, ...
  - BCD counter: 0000, 0001, 0010, ..., 1000, 0000, 0001
  - pseudo-random sequence generators: 10, 01, 00, 11, 10, 01, 00, ...
- Moore machines with “ring” structure to STD:

What are they used?

- Examples:
  - Clock divider circuits
  - Delays, Timing
  - Protocols
  - Counters simplify controller design…
    » More on this later
How do we design counters?

• For binary counters (most common case) incrementer circuit would work:

• In Verilog, a counter is specified as: x = x+1;  
  – This does not imply an adder  
  – An incrementer is simpler than an adder  
  – And a counter is simpler yet.

• In general, the best way to understand counter design is to think of them as FSMs, and follow general procedure. Here’s a important examples…

Counters

• Sequences through a fixed set of patterns  
  – In this case, 1000, 0100, 0010, 0001  
  – If one of the patterns is its initial state (by loading or set/reset)

• Mobius (or Johnson) counter  
  – In this case, 1000, 1100, 1110, 1111, 0111, 0011, 0001, 0000

Ring Counters – getting started

• “one-hot” counters 0001, 0010, 0100, 1000, 0001, ...

• What are these good for?

Ring Counters (cont)

• “Self-starting” version:

• What are these good for?
Announcements

- Reading: K&B 7.1, app C.
- Midterm 9/27 (week from thurs)
  - Regular class time. In Lab 125 Cory
  - Covers all material thru 9/23
  - 9/23 lecture will be “putting it all together”
- Review session 9/27 8-10
  - See web page for additional specifics
- HW 3 (current) is a good exercise (and short)
- HW 4 (out thurs) will be light, then skip a week
- Thurs Evening Lab is a problem!
  - Too many people in that section
  - Too many people from other sections
- Lab ‘DO NOT DISTURB’ rules
  - You must receive checkoff in your own section (first 30 mins)
  - You are welcome to use the lab outside your section, but if it is during some other lab section, you must let the TA concentrate on their section.
  - TAs will leave promptly at 8 pm
  - It is your job to read lab and write Verilog before you arrive.

Synchronous Counters

All outputs change with clock edge.

- Binary Counter Design:  
  Start with 3-bit version and generalize:
  \[ \begin{array}{cccc}
  c & b & a & c' + b' + a' \\
  0 & 0 & 0 & 0 \\
  0 & 0 & 1 & 1 \\
  0 & 1 & 0 & 1 \\
  0 & 1 & 1 & 0 \\
  1 & 0 & 0 & 1 \\
  1 & 0 & 1 & 0 \\
  1 & 1 & 0 & 1 \\
  1 & 1 & 1 & 0 \\
  \end{array} \]

\[ a' = a' \]
\[ b' = a \oplus b \]
\[ c' = a'c + ab'c' + b'c = c(a' + b') + c'(ab) = c(ab) + c(ab) = c \oplus ab \]

Binary Counter

- Logic between registers (not just multiplexer)
  - XOR decides when bit should be toggled
  - Always for low-order bit, only when first bit is true for second bit, and so on

```
module counter (out4, out3, out2, out1, clk);
  output out4, out3, out2, out1;
  input  in, clk;
  reg    out4, out3, out2, out1;
  always @(posedge clk)
  begin
    out4 <= (out1 & out2 & out3) ^ out4;
    out3 <= (out1 & out2) ^ out3;
    out2 <= out1 ^ out2;
    out1 <= out1 ^ 1b'1;
  end
endmodule
```
Binary Counter Verilog

module counter (out4, out3, out2, out1, clk);
    output [4:1] out;
    input in, clk;
    reg [4:1] out;

    always @(posedge clk)
        out <= out + 1;

endmodule

Synchronous Counters

• How do we extend to n-bits?
• Extrapolate c+: d* = d ⊕ abc, e* = e ⊕ abcd

• Has difficulty scaling (AND gate inputs grow with n)

• CE is “count enable”, allows external control of counting.
• TC is “terminal count”, is asserted on highest value, allows cascading, external sensing of occurrence of max value.

Synchronous Counters

How does this one scale?
• Generation of TC signals very similar to generation of carry signals in adder.
• “Parallel Prefix” circuit reduces delay:

Four-bit Binary Synchronous Up-Counter

• Standard component with many applications
  – Positive edge-triggered FFs w/ sync load and clear inputs
  – Parallel load data from D, C, B, A
  – Enable inputs: must be asserted to enable counting
  – RCO: ripple-carry out used for cascading counters
    » high when counter is in its highest state 1111
    » implemented using an AND gate

• CE is “count enable”, allows external control of counting.
• TC is “terminal count”, is asserted on highest value, allows cascading, external sensing of occurrence of max value.
“Ripple” counters

- Each stage is +2 of previous.
- Look at output waveforms:

![Ripple counter circuit diagram](image)

- Often called “asynchronous” counters.
- A “T” flip-flop is a “toggle” flip-flop. Flips it state on cycles when T=1.

Discouraged
- Know it exists
- Don’t use it

Odd Counts

- Extra combinational logic can be added to terminate count before max value is reached:
  - Example: count to 12

![Odd count circuit diagram](image)

- Alternative:

![Alternative count circuit diagram](image)

Offset Counters

- Starting offset counters – use of synchronous load
  - e.g., 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1111, 0110, ...

- Ending offset counter – comparator for ending value
  - e.g., 0000, 0001, 0010, ..., 1100, 1101, 0000

- Combinations of the above (start and stop value)
Universal Shift Register

- Holds 4 values
  - Serial or parallel inputs
  - Serial or parallel outputs
  - Permits shift left or right
  - Shift in new values from left or right

<table>
<thead>
<tr>
<th>s0</th>
<th>s1</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>hold state</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>shift right</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>shift left</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>load new input</td>
</tr>
</tbody>
</table>

clear sets the register contents and output to 0
s1 and s0 determine the shift function

Design of Universal Shift Register

- Consider one of the four flip-flops
  - New value at next clock cycle:

```
module univ_shift (out, lo, ro, in, li, ri, s, clr, clk);
output [3:0] out;
output lo, ro;
input  [3:0] in;
input  [1:0] s;
input  li, ri, clr, clk;
reg   [3:0] out;
assign lo = out[3];
assign ro = out[0];
always @(posedge clk or clr)
begin
  if (clr) out <= 0;
  else
    case (s)
    3: out <= in;
    2: out <= {out[2:0], ri};
    1: out <= {li, out[3:1]};
    0: out <= out;
    endcase
end
endmodule
```

Pattern Recognizer

- Combinational function of input samples
  - In this case, recognizing the pattern 1001 on the single input signal
Counters for Control

- Big idea: to solve a big controller problem, build a very simple controller and then use it as a tool.

Recall: Byte-bit stream with Rate Matching

- How would you implement this FSM?

Counter for Sequencing States

CLR for “back to top”
Count_Enable for Self-loop

CE = rdy or (state == 0000)

Branch with LD (jump counter)

LD = (State == 0000 & ~rdy) or (state == 1111)
S = (state == 0000)

IQ: How would you simplify this further
### State Complexity vs Counter Usage

**Diagram**

- States and transitions between states for a 3-bit counter with control signals.

### Another Controller using Counters

#### Example, Bit-serial multiplier:

- **Control Algorithm:**
  
  ```
  repeat n cycles {  // outer (i) loop
      repeat n cycles{  // inner (j) loop
          shiftA, selectSum, shiftHI
      }
      shiftB, shiftHI, shiftLOW, reset
  }
  ```

#### Control Algorithm:

- The occurrence of a control signal x means x=1. The absence of x means x=0.

### Counter provides subsidiary state

- **State Transition Diagram:**
  - Assume presence of two binary counters. An “i” counter for the outer loop and “j” counter for inner loop.

- **Counter:**
  - TC is asserted when the counter reaches its maximum count value.
  - CE is “clock enable”. The counter increments its value on the rising edge of the clock if CE is asserted.

### Summary

- **Basic registers**
  - Common control, MUXes

- **Simple, important FSMs**
  - Simple internal feedback
  - Ring counters, Pattern detectors
  - Binary Counters

- **Universal Shift Register**

- **Using Counters to build controllers**
  - Simplify control by controlling simpler FSM