Reset-able and Enable-able Registers

- Sometimes it is convenient or necessary to have flip-flops with special inputs like reset and enable.
- When designing flip-flops/registers, it is ok (possibly required) for there to be cases where the always block is entered, but the reg is not assigned.
- No fancy code, just make it work.
- Normally use synchronous reset instead of asynchronous reset (easier to test).
Reset-able and Enable-able Registers

- Example FF with reset and enable (reset has priority)

```verilog
always @(posedge clk) begin
    if (reset) // highest priority
        out <= #1 1'b0;
    else if (enable)
        out <= #1 c_out;
    // ok if no assignment (out holds value)
end
```
Reset-able and Enable-able Registers

- Example FF with reset and enable (enable has priority)

```verilog
counter0 <= #1 #1'b0;  
else 
  counter0 <= #1 c_out;
end 
// ok if no assignment (out holds value)
en
```
Reset-able and Enable-able Registers

- Use reset-able FFs only where needed
  - FFs are a little larger and higher power
  - Requires the global routing of the high-fanout reset signal

Diagram:
-reset only these two registers, but now reset must be enabled for at least 3 clock cycles

B. Baas, © 2011
Three types of “case” statements in verilog

1) case
   - Normal case statement

2) casez
   - Allows use of wildcard “?” character for don’t cares.
     ```verilog
     casez(in)
         4’b1???: out = a;
         4’b01??: out = b;
         4’b00??: out = c;
         default: out = d;
     endcase
     ```

3) casex
   - Don’t use it. Could use “z” or “x” logic.
   - default
     - Normally set output to an easily-recognizable value (such as x’s) in a default statement to make mistakes easier to spot
Complex or “arbitrary” functions are not uncommon

Examples

- sin/cos
- tangent$^{-1}$
- log

$\theta$

out_real
out_imag
Hardwired Function in Verilog using a Lookup Table

- `always @(input) begin
  case (input)
  4'b0000: begin  real=3'b100;  imag=3'b001; end
  4'b0001: begin  real=3'b000;  imag=3'b101; end
  4'b0010: begin  real=3'b110;  imag=3'b011; end
  ...
  default: begin  real=3'bxxx;  imag=3'bxxx; end
  endcase
end

- Often best to write a matlab program to write the verilog table as plain text
  - You will need several versions to get it right
  - Easy to adapt to other specifications
- Not efficient for very large tables
- Tables with data that is less random will have smaller synthesized area
D-Flipflop with Enable

- Storage controlled by a clock-enable
  - stores only when CE = 1 on a rising edge of the clock

- CE is a *synchronous control* input
Register with Enable

- One flipflop per bit
  - clk and CE wired in common

```verilog
wire [n:0] d;
wire ce;
reg [n:0] q;
...

always @(posedge clk)
  if (ce) q <= d;
```
Register with Synchronous Reset

- Reset input forces stored value to 0
- Reset input must be stable around rising edge of clk

always @(posedge clk)
  if (reset) q <= 0;
else if (ce) q <= d;
Register with Asynchronous Reset

- Reset input forces stored value to 0
  - reset can become 1 at any time, and effect is immediate
  - reset should return to 0 synchronously

```
<table>
<thead>
<tr>
<th>D</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>CE</td>
<td>reset</td>
</tr>
<tr>
<td>clk</td>
<td></td>
</tr>
</tbody>
</table>
```

![Waveform diagram showing clk, reset, CE, D, and Q signals over time]

Digital Design — Chapter 4 — Sequential Basics
Asynch Reset in Verilog

```verilog
always @(posedge clk or posedge reset)
  if (reset) q <= 0;
  else if (ce) q <= d;
```

- reset is an *asynchronous control* input here
  - include it in the event list so that the process responds to changes immediately
Example: Accumulator

- Sum a sequence of signed numbers
- A new number arrives when data_en = 1
- Clear sum to 0 on synch reset

```verilog
module accumulator  
  ( output reg signed [7:-12] data_out,  
    input  signed [3:-12] data_in,  
    input data_en, clk, reset );  
wire signed [7:-12] new_sum;  
assign new_sum = data_out + data_in;  
always @(posedge clk)  
  if ( reset ) data_out <= 20'b0;  
  else if ( data_en ) data_out <= new_sum;  
endmodule
```
module flip_flop_n (output reg Q,
   output Q_n,
   input pre_n, clr_n, D,
   input clk_n, CE);

always @( negedge clk_n or
               negedge pre_n or negedge clr_n ) begin
  if (!pre_n && !clr_n)
    $display("Illegal inputs: pre_n and clr_n both 0");
  if (!pre_n) Q <= 1'b1;
  else if (!clr_n) Q <= 1'b0;
  else if (CE)     Q <= D;
end
assign Q_n = ~Q;
endmodule
Shift Registers

- Performs shift operation on stored data
  - Arithmetic scaling
  - Serial transfer of data

![Shift Register Diagram]

Verilog

Digital Design — Chapter 4 — Sequential Basics 15
Example: Sequential Multiplier

- 16×16 multiply over 16 clock cycles, using one adder
  - Shift register for multiplier bits
  - Shift register for lsb’s of accumulated product
Latches

- Level-sensitive storage
  - Data transmitted while enable is '1'
    - *transparent* latch
  - Data stored while enable is '0'

![Latch Diagram](image)
Feedback Latches

- Feedback in gate circuits produces latching behavior
  - Example: reset/set (RS) latch

- Current RTL synthesis tools don’t accept Verilog models with unclocked feedback
Latches in Verilog

- Latching behavior is usually an error!

```verilog
always @(*)
  if (~sel) begin
    z1 <= a1; z2 <= b1;
  end
  else begin
    z1 <= a2; z3 <= b2;
  end
```

Oops! Should be `z2 <= ...`

- Values must be stored
  - for z2 while sel = 1
  - for z3 while sel = 0
Counters

- Stores an unsigned integer value
  - increments or decrements the value
- Used to count occurrences of
  - events
  - repetitions of a processing step
- Used as timers
  - count elapsed time intervals by incrementing periodically
Free-Running Counter

- Increments every rising edge of clk
  - up to $2^n - 1$, then wraps back to 0
  - i.e., counts modulo $2^n$
- This counter is *synchronous*
  - all outputs governed by clock edge
Example: Periodic Control Signal

- Count modulo 16 clock cycles
  - Control output = 1 every 8th and 12th cycle
  - decode count values 0111 and 1011
Example: Periodic Control Signal

```verilog
module decoded_counter ( output ctrl, 
                          input  clk );
    reg [3:0] count_value;
    always @(posedge clk)
        count_value <= count_value + 1;
    assign ctrl = count_value == 4'b0111 ||
                   count_value == 4'b1011;
endmodule
```
Count Enable and Reset

- Use a register with control inputs
  - Increments when CE = 1 on rising clock edge
  - Reset: synch or asynch
Terminal Count

- Status signal indicating final count value

- TC is 1 for one cycle in every $2^n$ cycles
  - frequency = clock frequency / $2^n$

- Called a *clock divider*
Divider Example

- Alarm clock beep: 500Hz from 1MHz clock
Divide by \( k \)

- Decode \( k-1 \) as terminal count and reset counter register
  - Counter increments modulo \( k \)
- Example: decade counter
  - Terminal count = 9
module decade_counter ( output reg [3:0] q, input clk );

always @(posedge clk)
    q <= q == 9 ? 0 : q + 1;
endmodule
Down Counter with Load

- Load a starting value, then decrement
  - Terminal count = 0
- Useful for interval timer
module interval_timer_rtl ( output tc, 
    input [9:0] data, 
    input load, clk );

    reg [9:0] count_value;
    always @(posedge clk) 
        if (load) count_value <= data;
        else count_value <= count_value - 1;
    assign tc = count_value == 0;
endmodule
module interval_timer_repetitive (output tc, 
               input [9:0] data, 
               input load, clk );

reg [9:0] load_value, count_value;

always @(posedge clk) 
  if (load) begin
    load_value <= data;
    count_value <= data;
  end
  else if (count_value == 0)
    count_value <= load_value;
  else
    count_value <= count_value - 1;

assign tc = count_value == 0;
endmodule
Ripple Counter

- Each bit toggles between 0 and 1
  - when previous bit changes from 1 to 0
Ripple or Synch Counter?

- Ripple counter is ok if
  - length is short
  - clock period long relative to flipflop delay
  - transient wrong values can be tolerated
  - area must be minimal
- E.g., alarm clock
- Otherwise use a synchronous counter
Datapaths and Control

- Digital systems perform sequences of operations on encoded data

**Datapath**
- Combinational circuits for operations
- Registers for storing intermediate results

**Control section**: control sequencing
- Generates *control signals*
  - Selecting operations to perform
  - Enabling registers at the right times
- Uses *status signals* from datapath
Example: Complex Multiplier

- Cartesian form, fixed-point
  - operands: 4 integer, 12 fraction bits
  - result: 8 pre-, 24 post-binary-point bits
- Subject to tight area constraints
  \[
  a = a_r + ja_i \quad b = b_r + jb_i
  \]
  \[
  p = ab = p_r + jp_i = (a_r b_r - a_i b_i) + j(a_r b_i + a_i b_r)
  \]
- 4 multiplies, 1 add, 1 subtract
  - Perform sequentially using 1 multiplier, 1 adder/subtractor
Complex Multiplier Datapath
module multiplier
 ( output reg signed [7:-24] p_r, p_i,
   input signed [3:-12] a_r, a_i, b_r, b_i,
   input clk, reset, input_rdy );

reg a_sel, b_sel, pp1_ce, pp2_ce, sub, p_r_ce, p_i_ce;
wire signed [3:-12] a_operand, b_operand;
wire signed [7:-24] pp, sum
reg signed [7:-24] pp1, pp2;
...
Complex Multiplier in Verilog

```verilog
assign a_operand = ~a_sel ? a_r : a_i;
assign b_operand = ~b_sel ? b_r : b_i;
assign pp = {{4{a_operand[3]}}, a_operand, 12'b0} * 
            {{4{b_operand[3]}}, b_operand, 12'b0};
always @(posedge clk) // Partial product 1 register
    if (pp1_ce) pp1 <= pp;
always @(posedge clk) // Partial product 2 register
    if (pp2_ce) pp2 <= pp;
assign sum = ~sub ? pp1 + pp2 : pp1 - pp2;
always @(posedge clk) // Product real-part register
    if (p_r_ce) p_r <= sum;
always @(posedge clk) // Product imaginary-part register
    if (p_i_ce) p_i <= sum;
...
endmodule
```
Multiplier Control Sequence

- Avoid resource conflict
- First attempt
  1. \(a_r \times b_r \rightarrow pp1\_reg\)
  2. \(a_i \times b_i \rightarrow pp2\_reg\)
  3. \(pp1 - pp2 \rightarrow p\_r\_reg\)
  4. \(a_r \times b_i \rightarrow pp1\_reg\)
  5. \(a_i \times b_r \rightarrow pp2\_reg\)
  6. \(pp1 + pp2 \rightarrow p\_i\_reg\)
- Takes 6 clock cycles
Multiplier Control Sequence

- Merge steps where no resource conflict

Revised attempt

1. \( a_r \times b_r \rightarrow pp1\_reg \)
2. \( a_i \times b_i \rightarrow pp2\_reg \)
3. \( pp1 - pp2 \rightarrow p_r\_reg \)  
   \[ a_r \times b_i \rightarrow pp1\_reg \]
4. \( a_i \times b_r \rightarrow pp2\_reg \)
5. \( pp1 + pp2 \rightarrow p_i\_reg \)

- Takes 5 clock cycles
### Multiplier Control Signals

<table>
<thead>
<tr>
<th>Step</th>
<th>a_sel</th>
<th>b_sel</th>
<th>pp1_ce</th>
<th>pp2_ce</th>
<th>sub</th>
<th>p_r_ce</th>
<th>p_i_ce</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>–</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>–</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>–</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>–</td>
<td>–</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Finite-State Machines

- Used the implement control sequencing
- A FSM is defined by
  - set of inputs
  - set of outputs
  - set of states
  - initial state
  - transition function
  - output function
- States are steps in a sequence of transitions
  - There are “Finite” number of states.
FSM in Hardware

- Mealy FSM: outputs depend on state and inputs
- Moore FSM: outputs depend on state only (no dash)
- Mealy and Moore FSM can convert to each other
FSM Example: Multiplier Control

- One state per step
- Separate idle state?
  - Wait for input_rdy = 1
  - Then proceed to steps 1, 2, ...
  - But this wastes a cycle!
- Use step 1 as idle state
  - Repeat step 1 if input_rdy ≠ 1
  - Proceed to step 2 otherwise
- Output function
  - Defined by table on slide 43
  - Moore or Mealy?

Transition function

<table>
<thead>
<tr>
<th>current_state</th>
<th>input_rdy</th>
<th>next_state</th>
</tr>
</thead>
<tbody>
<tr>
<td>step1</td>
<td>0</td>
<td>step1</td>
</tr>
<tr>
<td>step1</td>
<td>1</td>
<td>step2</td>
</tr>
<tr>
<td>step2</td>
<td>–</td>
<td>step3</td>
</tr>
<tr>
<td>step3</td>
<td>–</td>
<td>step4</td>
</tr>
<tr>
<td>step4</td>
<td>–</td>
<td>step5</td>
</tr>
<tr>
<td>step5</td>
<td>–</td>
<td>step1</td>
</tr>
</tbody>
</table>
State Encoding

- Encoded in binary
  - $N$ states: use at least $\lceil \log_2 N \rceil$ bits
- Encoded value used in circuits for transition and output function
  - encoding affects circuit complexity
- Optimal encoding is hard to find
  - CAD tools can do this well
- One-hot works well in FPGAs
- Often use 000...0 for idle state
  - reset state register to idle
FSMs in Verilog

- Use parameters for state values
  - Synthesis tool can choose an alternative encoding

```verilog
parameter [2:0] step1 = 3'b000, step2 = 3'b001,
    step3 = 3'b010, step4 = 3'b011,
    step5 = 3'b100;
reg [2:0] current_state, next_state;
...
```
Multiplier Control in Verilog

```verilog
always @(posedge clk or posedge reset) // State register
  if (reset) current_state <= step1;
  else current_state <= next_state;
always @* // Next-state logic
  case (current_state)
    step1: if (!input_rdy) next_state = step1;
           else next_state = step2;
    step2: next_state = step3;
    step3: next_state = step4;
    step4: next_state = step5;
    step5: next_state = step1;
  endcase
```
Multiplier Control in Verilog

```verilog
always @* begin // Output_logic
    a_sel = 1'b0; b_sel = 1'b0; pp1_ce = 1'b0; pp2_ce = 1'b0;
    sub = 1'b0; p_r_ce = 1'b0; p_i_ce = 1'b0;
    case (current_state)
        step1: begin
            pp1_ce = 1'b1;
        end
        step2: begin
            a_sel = 1'b1; b_sel = 1'b1; pp2_ce = 1'b1;
        end
        step3: begin
            b_sel = 1'b1; pp1_ce = 1'b1;
            sub = 1'b1; p_r_ce = 1'b1;
        end
        step4: begin
            a_sel = 1'b1; pp2_ce = 1'b1;
        end
        step5: begin
            p_i_ce = 1'b1;
        end
    endcase
end
```
State Transition Diagrams

- Bubbles to represent states
- Arcs to represent transitions

Example

- \( S = \{s1, s2, s3\} \)
- Inputs \((a1, a2)\):
  \( \Sigma = \{(0,0), (0,1), (1,0), (1,1)\} \)
- \( \delta \) defined by diagram
State Transition Diagrams

- Annotate diagram to define output function
  - Annotate states for Moore-style outputs
  - Annotate arcs for Mealy-style outputs

- Example
  - $x_1, x_2$: Moore-style
  - $y_1, y_2, y_3$: Mealy-style