L5: Simple Sequential Circuits and Verilog

Acknowledgements: Nathan Ickes and Rex Min
Key Points from L4 (Sequential Blocks)

Classification:

- **Latch**: level sensitive (positive latch passes input to output on high phase, hold value on low phase)
- **Register**: edge-triggered (positive register samples input on rising edge)
- **Flip-Flop**: any element that has two stable states. Quite often Flip-flop also used denote an (edge-triggered) register

![Positive Latch and Positive Register Diagram]

- Latches are used to build Registers (using the Master-Slave Configuration), but are almost NEVER used by itself in a standard digital design flow.
- Quite often, latches are inserted in the design by mistake (e.g., an error in your Verilog code). Make sure you understand the difference between the two.
- Several types of memory elements (SR, JK, T, D). We will most commonly use the D-Register, though you should understand how the different types are built and their functionality.
System Timing Parameters

Register Timing Parameters

- $T_{cq}$ : worst case rising edge clock to $q$ delay
- $T_{cq, cd}$ : contamination or minimum delay from clock to $q$
- $T_{su}$ : setup time
- $T_{h}$ : hold time

Logic Timing Parameters

- $T_{logic}$ : worst case delay through the combinational logic network
- $T_{logic, cd}$ : contamination or minimum delay through logic network
System Timing (I): Minimum Period

\[ T > T_{cq} + T_{logic} + T_{su} \]
System Timing (II): Minimum Delay

\[ T_{eq,cd} + T_{logic,cd} > T_{hold} \]
Edge-triggered circuits are described using a sequential always block.

### Combinational

```verilog
module combinational(a, b, sel, out);
    input a, b;
    input sel;
    output out;
    reg out;
    always @ (a or b or sel)
        begin
            if (sel) out = a;
            else out = b;
        end
endmodule
```

### Sequential

```verilog
module sequential(a, b, sel, clk, out);
    input a, b;
    input sel, clk;
    output out;
    reg out;
    always @ (posedge clk)
        begin
            if (sel) out <= a;
            else out <= b;
        end
endmodule
```

![Combinational Circuit Diagram](image1)

![Sequential Circuit Diagram](image2)
The use of `posedge` and `negedge` makes an `always` block sequential (edge-triggered)

Unlike a combinational `always` block, the sensitivity list does determine behavior for synthesis!

D Flip-flop with synchronous clear

```verilog
module dff_sync_clear(d, clearb, clock, q);
  input d, clearb, clock;
  output q;
  reg q;
  always @(posedge clock)
  begin
    if (!clearb) q <= 1'b0;
    else q <= d;
  end
endmodule
```

D Flip-flop with asynchronous clear

```verilog
module dff_async_clear(d, clearb, clock, q);
  input d, clearb, clock;
  output q;
  reg q;
  always @(negedge clearb or posedge clock)
  begin
    if (!clearb) q <= 1'b0;
    else q <= d;
  end
endmodule
```

always block entered only at each positive clock edge

always block entered immediately when (active-low) `clearb` is asserted

Note: The following is incorrect syntax: `always @ (clear or negedge clock)`

If one signal in the sensitivity list uses `posedge`/`negedge`, then all signals must.

- Assign any signal or variable from only one `always` block, be wary of race conditions: always blocks execute in parallel
- **DFF with Synchronous Clear**

- **DFF with Asynchronous Clear**
## Blocking vs. Nonblocking Assignments

- **Verilog** supports two types of assignments within `always` blocks, with subtly different behaviors.

- **Blocking assignment**: evaluation and assignment are immediate

  ```verilog
  always @ (a or b or c)
  begin
    x = a | b;  // 1. Evaluate a | b, assign result to x
    y = a ^ b ^ c;  // 2. Evaluate a^b^c, assign result to y
    z = b & ~c;  // 3. Evaluate b&(~c), assign result to z
  end
  ```

- **Nonblocking assignment**: all assignments deferred until all right-hand sides have been evaluated (end of simulation timestep)

  ```verilog
  always @ (a or b or c)
  begin
    x <= a | b;  // 1. Evaluate a | b but defer assignment of x
    y <= a ^ b ^ c;  // 2. Evaluate a^b^c but defer assignment of y
    z <= b & ~c;  // 3. Evaluate b&(~c) but defer assignment of z
    x <= a | b;  // 4. Assign x, y, and z with their new values
    y <= a ^ b ^ c;
    z <= b & ~c;
  end
  ```

- Sometimes, as above, both produce the same result. Sometimes, not!
Will nonblocking and blocking assignments both produce the desired result?

module nonblocking(in, clk, out);
    input in, clk;
    output out;
    reg q1, q2, out;
    always @(posedge clk)
        begin
            q1 <= in;
            q2 <= q1;
            out <= q2;
        end
endmodule

module blocking(in, clk, out);
    input in, clk;
    output out;
    reg q1, q2, out;
    always @(posedge clk)
        begin
            q1 = in;
            q2 = q1;
            out = q2;
        end
endmodule
Use Nonblocking for Sequential Logic

```verilog
always @ (posedge clk)
begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
end
```

“At each rising clock edge, \( q1 \), \( q2 \), and \( out \) simultaneously receive the old values of \( in \), \( q1 \), and \( q2 \).”

- Blocking assignments do not reflect the intrinsic behavior of multi-stage sequential logic
- **Guideline:** use nonblocking assignments for sequential always blocks
Simulation

- Non-blocking Simulation

- Blocking Simulation
# Use Blocking for Combinational Logic

## Blocking Behavior

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(Given) Initial Condition</strong></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>a changes; always block triggered</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>x = a &amp; b;</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>y = x</td>
<td>c;</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

## Nonblocking Behavior

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(Given) Initial Condition</strong></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>a changes; always block triggered</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>x &lt;= a &amp; b;</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>y &lt;= x</td>
<td>c;</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### Deferred Assignment Completion

- x <= 0
- x <= 0, y <= 1

### Block

```verilog
define module blocking(a,b,c,x,y);
    input a,b,c;
    output x,y;
    reg x,y;
    always @ (a or b or c)
    begin
        x = a & b;
        y = x | c;
    end
endmodule
```

### Nonblocking

```verilog
define module nonblocking(a,b,c,x,y);
    input a,b,c;
    output x,y;
    reg x,y;
    always @ (a or b or c)
    begin
        x <= a & b;
        y <= x | c;
    end
endmodule
```

- Nonblocking and blocking assignments will synthesize correctly. Will both styles simulate correctly?
- Nonblocking assignments do not reflect the intrinsic behavior of multi-stage combinational logic
- While nonblocking assignments can be hacked to simulate correctly (expand the sensitivity list), it’s not elegant
- **Guideline:** use blocking assignments for combinational always blocks
The Asynchronous Ripple Counter

A simple counter architecture

- uses only registers (e.g., 74HC393 uses T-register and negative edge-clocking)
- Toggle rate fastest for the LSB

...but ripple architecture leads to large skew between outputs
The Ripple Counter in Verilog

Single D Register with Asynchronous Clear:

module dreg_async_reset (clk, clear, d, q, qbar);
input d, clk, clear;
output q, qbar;
reg q;

always @ (posedge clk or negedge clear)
begins
if (!clear)
  q <= 1'b0;
else q <= d;
end
assign qbar = ~q;
endmodule

Structural Description of Four-bit Ripple Counter:

module ripple_counter (clk, count, clear);
input clk, clear;
output [3:0] count;
wire [3:0] count, countbar;

dreg_async_reset bit0(.clk(clk), .clear(clear), .d(countbar[0]),
  .q(count[0]), .qbar(countbar[0]));
dreg_async_reset bit1(.clk(countbar[0]), .clear(clear), .d(countbar[1]),
  .q(count[1]), .qbar(countbar[1]));
dreg_async_reset bit2(.clk(countbar[1]), .clear(clear), .d(countbar[2]),
  .q(count[2]), .qbar(countbar[2]));
dreg_async_reset bit3(.clk(countbar[2]), .clear(clear), .d(countbar[3]),
  .q(count[3]), .qbar(countbar[3]));
endmodule
Simulation of Ripple Effect
Logic for a Synchronous Counter

- Count (C) will retained by a D Register
- Next value of counter (N) computed by combinational logic

<table>
<thead>
<tr>
<th>C3</th>
<th>C2</th>
<th>C1</th>
<th>N3</th>
<th>N2</th>
<th>N1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</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>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

N1 := \overline{C1}

N2 := C1 \overline{C2} + \overline{C1} C2
    := C1 \text{xor} C2

N3 := C1 C2 \overline{C3} + \overline{C1} C3 + \overline{C2} C3
    := C1 C2 C3 + (C1 + C2) C3
    := (C1 C2) \text{xor} C3

From [Katz05]
The 74163 Catalog Counter

- Synchronous Load and Clear Inputs
- Positive Edge Triggered FFs
- Parallel Load Data from D, C, B, A
- P, T Enable Inputs: both must be asserted to enable counting
- Ripple Carry Output (RCO): asserted when counter value is 1111 (conditioned by T); used for cascading counters

Synchronous CLR and LOAD
If CLRb = 0 then Q <= 0
Else if LOADb=0 then Q <= D
Else if P * T = 1 then Q <= Q + 1
Else Q <= Q
Inside the 74163 (Courtesy TI) - Operating Modes

CLR = 0, LOAD = 0:
Clear takes precedence

CLR = 1, LOAD = 0:
Parallel load from DATA
CLR = 1, LOAD = 1, P T = 0:
Counting inhibited

CLR = 1, LOAD = 1, P T = 1:
Count enabled
Behavioral description of the ‘163 counter:

```verilog
module counter(LDbar, CLRbar, P, T, CLK, D, count, RCO);
  input LDbar, CLRbar, P, T, CLK;
  input [3:0] D;
  output [3:0] count;
  output RCO;
  reg [3:0] Q;

  always @(posedge CLK) begin
    if (!CLRbar) Q <= 4'b0000;
    else if (!LDbar) Q <= D;
    else if (P && T) Q <= Q + 1;
  end

  assign count = Q;
endmodule
```

Priority logic for control signals

RCO gated by T input
Notice the glitch on RCO!
Output Transitions

- Any time multiple bits change, the counter output needs time to settle.
- Even though all flip-flops share the same clock, individual bits will change at different times.
  - Clock skew, propagation time variations
- Can cause glitches in combinational logic driven by the counter
- The RCO can also have a glitch.

Care is required of the Ripple Carry Output:
It can have glitches:
Any of these transition paths are possible!
Cascading the 74163: Will this Work?

- '163 is enabled only if P and T are high
- When first counter reaches $Q = 4'b1111$, its RCO goes high for one cycle
- When RCO goes high, next counter is enabled ($P T = 1$)

So far, so good...then what’s wrong?
Incorrect Cascade for 74163

Everything is fine up to 8’b11101111:

Problem at 8’b11110000: one of the RCOs is now stuck high for 16 cycles!
P input takes the master enable
T input takes the ripple carry

Summary

- Use blocking assignments for combinational always blocks
- Use non-blocking assignments for sequential always blocks
- Synchronous design methodology usually used in digital circuits
  - Single global clocks to all sequential elements
  - Sequential elements almost always of edge-triggered flavor (design with latches can be tricky)
- Today we saw simple examples of sequential circuits (counters)