Week 6 - DFFs and Testbenches

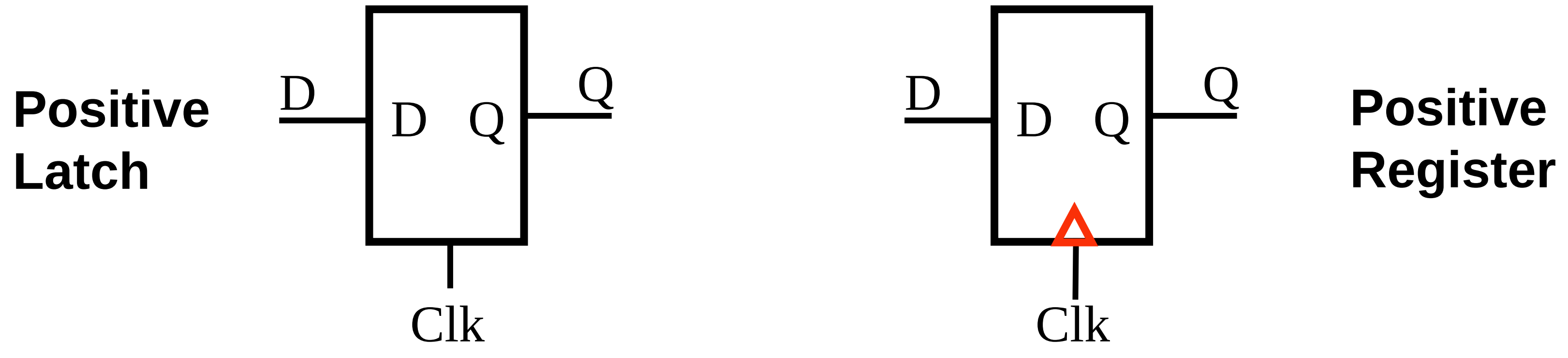
Agenda

- DFF Modeling
  - RTL Review
  - How else to model DFF
  - DFF Questions?
- Testbenches
  - TB template
  - RTL Review
  - TB Questions?
- BREAK
- SR and LFSR examples

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



- 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.

RTL Review - DFF

```
// FALLING EDGE D FLIP FLIP MODULE:
//=====================================
module d_ff_gates (d, clk, rst, q, q_bar);
input d, clk, rst;
output q, q_bar;
wire n1,n2,n3,q_bar_n, cn,dn,n4,n5,n6;
// First Latch not (n1,d);
nand (n2,d,clk); nand (n3,n1,clk); nand (dn,q_bar_n,n2); nand
(q_bar_n,dn,n3, !rst);
// Second Latch not (cn,clk); not (n4,dn); nand (n5,dn,cn); nand
(n6,n4,cn); nand (q,q_bar,n5); nand (q_bar,q,n6, !rst);
endmodule
```

RTL Review - DFF

- Master slave latch configuration works
- Questions
    - How does this get synthesized?
    - How do I use this?

RTL Review - DFF

- Master slave latch configuration works
- Questions
  - How does this get synthesized?
    - ASIC Tool - large amount of gates
    - FPGA Tool - Xilinx Synthesis output:
      - Warnings of Combinatorial loops
      - Each DFF occupies all the lookup tables for **TWO Spartan3 slices**! Wastes DFF's! :'-[
      - Cannot guarantee timing, since its clock and rst are routed through FPGA fabric and not the clock trees!
  - How do I use this?
    - Have to instance it each time we want to use it! Cannot imply the DFF!

The Sequential `always` Block

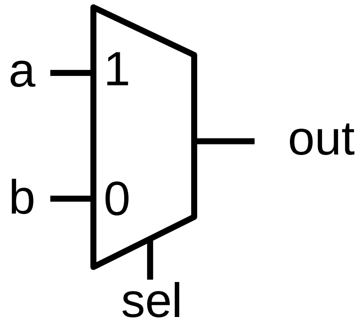
- **Edge-triggered circuits are described using a sequential `always` block**

Combinational

```
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
```

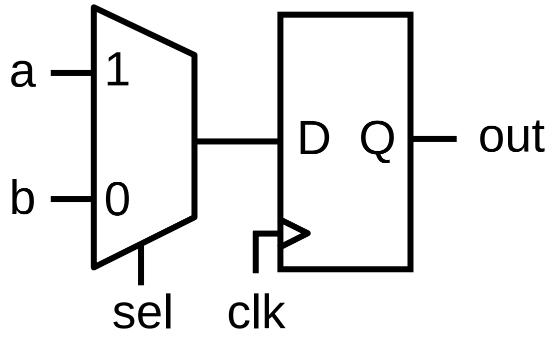


Sequential

```
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
```



Importance of the Sensitivity List

- **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

```
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
```

`always` block entered only at each positive clock edge

D Flip-flop with asynchronous clear

```
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 immediately when (active-low) clearb is asserted

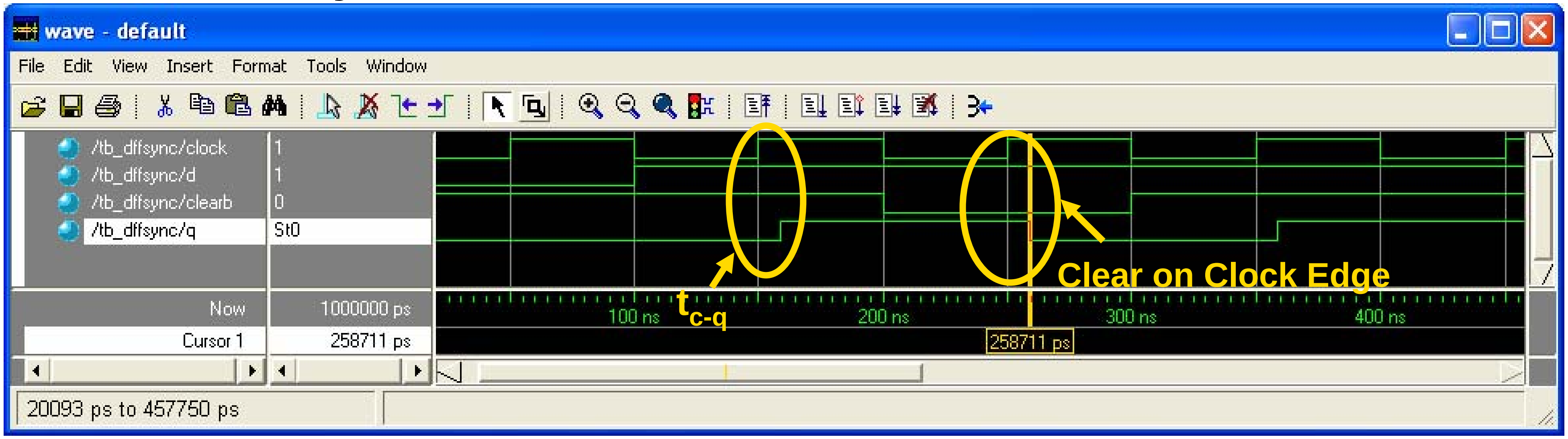
Note: The following is **incorrect** syntax: `always @ (clear or negedge clock)`

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

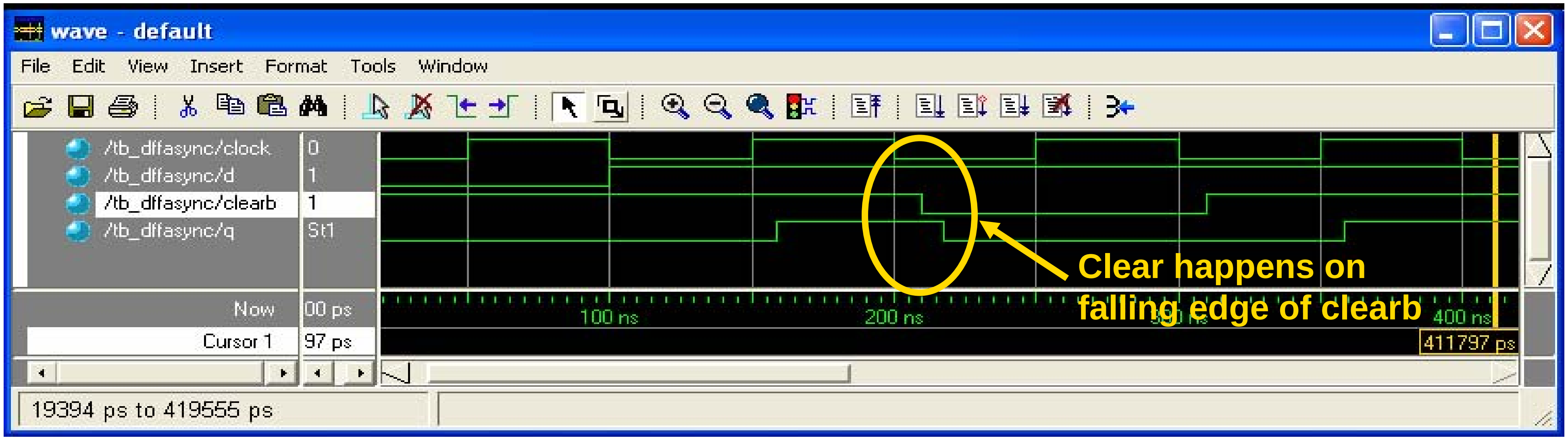
- **Assign any signal or variable from <u>only one</u> always block, Be wary of race conditions: always blocks execute in parallel**

Simulation (after Place and Route in Xilinx)

- **DFF with Synchronous Clear**



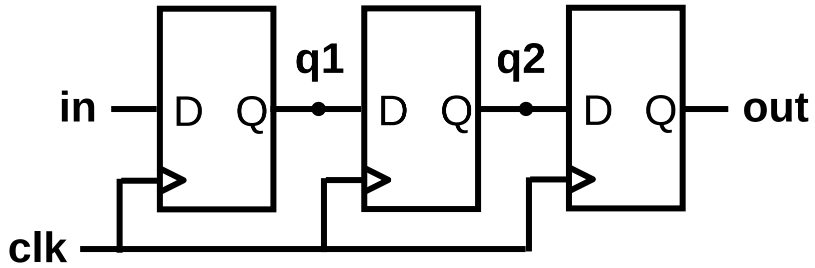
- **DFF with Asynchronous Clear**



Use Nonblocking for Sequential Logic

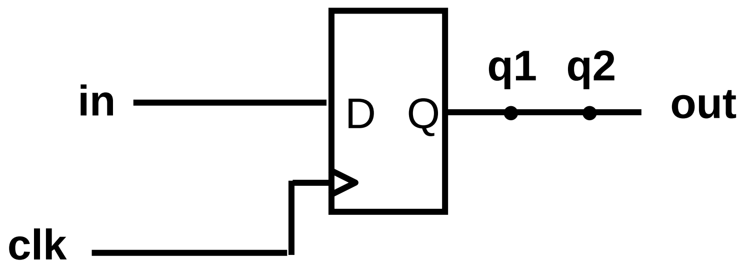
```
always @ (posedge clk)
begin
  q1 <= in;
  q2 <= q1;
  out <= q2;
end
```

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



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

“At each rising clock edge, *q1* = *in*.
After that, *q2* = *q1* = *in*.
After that, *out* = *q2* = *q1* = *in*.
Therefore *out* = *in*.”



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

How **TO** model / infer a DFF

Assume output reg Q, input wire D, etc..

```
---- DFF ----
always@(posedge clk)
        Q <= D;

---- DFF w/ Clock Enable ----
always@(posedge clk)
    if(en)
        Q <= D;
    else
        Q <= Q;
```

How **TO** model / infer a DFF

Assume output reg Q, input wire D, etc..

```
---- DFF with synchronous reset ----
always@(posedge clk)
   if(rst)
      Q <= 0;
   else
      Q <= D;

---- DFF w/ Clock Enable & async reset ----
always@(posedge clk or posedge rst)
   if(rst)
      Q <= 0;
   else if(en)
      Q <= D;
   else
      Q <= Q;
```

How **NOT TO** model / infer DFF

Assume output reg Q, input wire D, etc..

```
---- Which one is clock? ----
always@(posedge rst or posedge clk or posedge A)
    if (rst) Q<= 1;
    else if (clk) Q<= 0;
    else Q<= D;

---- What happens here? ----
always (posedge A)
    if (rst) Q<= 1;
    else if (clk) Q<= 0;
    else Q <= D;
```

How **NOT TO** model / infer DFF

Assume output reg Q, input wire D, etc..

```
---- Which one is clock? ----
always@(posedge rst or posedge clk or posedge A)
    if (rst) Q<= 1; //set
    else if (clk) Q<= 0; //reset
    else Q<= D;
 // A becomes the clock
---- What happens here? ----
always (posedge A)
    if (rst) Q<= 1;
    else if (clk) Q<= 0;
    else Q <= D;
// DFF, with the input being equal to rst + !clk&D
```

RTL Coding Timing Question

```
reg [2:0] my_var;

initial
begin
for (my_var=0; my_var<8; my_var=my_var+1)
#5 a=a+b;
#5 $finish;
end
```

Question - When will the simulation end?

RTL Coding Timing Question

```
reg [2:0] my_var;

initial
begin
for (my_var=0; my_var<8; my_var=my_var+1)
#5 a=a+b;
#5 $finish;
end
```

Question - When will the simulation end?
Answer - 45 units. Not seconds or nanoseconds, 45 time units. T
is no timescale definition here!
ex: `timescale 1ns / 10ps

General test bench structure

In general, your test bench may include all of the following items:

- Include statements
- Parameter definitions
- DUT Input (reg) data
- DUT Output (wires) data
- Any interconnects, additional registers, events
- DUT Instantiation
- Any other test structures (DUT vs. Model)
- Initial Conditions/Test prep (read in a file, load array, ...)
- Stimulus
- Event Definitions
- Monitor - Either data capture, self checking TB, et cetera...

General test bench structure - cont

- Include statements
  - Timescale.v
    - Place the timescale definition in a separate file. Not required.
  - module_defines.v
    - Place any global definitions in a separate file, include it in all required parts of the design.
- Parameter definitions
  - Clock period, finish time, control words, ... all may need to be defined as parameters.
  - Put these near the top of your testbench

General test bench structure - cont

- DUT Input regs
  - Need to place data on the input of your DUT, so you'll want to use REGs to store the data your using. Instance all your input regs together.
- DUT Output wires
  - Your output from your DUT needs to be connected to something, for this you'll want to use wires. Instance your output wires together.
    - Not restricted from storing output into reg datatypes, just not _required_
- Interconnects, other data...
  - If you have multiple modules to connect together with your DUT, instance their interconnects together.
  - Integers, other regs, memory arrays, all may be needed, so instance them after interconnects.

General test bench structure - cont

- DUT Instantiation
    - You'll have to instance your device under test.
    - Get in the practice of using name ports
        - Ditch this style
            - upcounter dut (rst, clk, enable, q, qb);
        - Use this style
            - upcounter dut (.rst(rst), .clk(clk), .enable(enable), q(q), .qb(qb));
- If you use other modules in your TB, be sure to instance them as well.

General test bench structure - cont

- Initial Conditions
    - Your first initial block, separate from data
    - Make sure you set all of your registers to an initial value
    - Reset may be set to a normal, non-reset state.
- May want to have a block for ending the simulation
    - Either a #ENDTIME $finish directive in your initial block
    - Or a separate initial block, just for causing the simulaiton to end
    - This is to prevent your TB from running forever...

General test bench structure - cont

- Stimulus
  - This is your sequence of input to your DUT, kept in its own initial block
  - This may be a series of assignment statements with delays dictating input, or a series of event triggers, or a mixture of the two.
- Event definitions
  - always @(event_name) begin....end blocks
  - These blocks enter when the event event_name is triggered.   -> event_name;
  - Can lead to very robust test benches.
- Monitor
  - Waveform Capture
  - Self checking
  - Status/Error messages

RTL Review - Testbench

```
module upcounter_tb;
reg rst, clk, enable;
wire [3:0] q, qb;
upcounter dut(rst, clk, enable, q, qb);
initial begin
rst = 1; enable = 0; clk=0;
#1; rst = 0;
clk=1; #1; clk=0; #1; clk=1; #1; clk=0; #1; clk=1; #1; clk=0; #1;
clk=1; #1; clk=0; #1; clk=1; #1; clk=0; #1; clk=1; #1; clk=0; #1;
clk=1; #1; clk=0; #1; clk=1; #1;
end
initial begin
$dumpfile ("waves.lxt"); $dumpvars (0, upcounter_tb);
end
endmodule
```

RTL Review - Testbench

- Items of Concern
  - Implicit ports when instantiating DUT
  - Same Initial block used for stimulus and initial conditions
  - Clock explicitly defined

Clock Generator

```
	//parameters
parameter CLKPERIOD = 20; //50Mhz @ 1ns timescale

//
	// lots of things going on in between here...
	//

	//clk
always
#(CLKPERIOD/2) clk = ~clk;

	// carry on, my friend, with your free running clock!
```

Lets code - Shift Register

```
module shift(D, Q, Q_bus, clk, rst);
parameter N=4;
parmeter TP=1;
input D, clk, rst;
output Q;
output [N-1:0] Q_regs;
reg [N-1:0] Q_regs;
assign Q = Q_regs[n-1];
always@(posedge clk)
    if(rst)
        Q_regs <= #Tp N'b0;
    else
        Q_regs <= #Tp {Q_regs[n-2:0],D);
end module
```

Lets code - Linear Feedback Shift Register

```
module lfsr_example(D, Q, Q_regs, clk, rst);
parameter N=3;
parameter Tp=1;
input D, clk, rst;
output Q;
output [N-1:0]
Q_regs; reg [N-1:0] Q_regs;
assign Q = Q_regs[0];
always@(posedge clk)
    if(rst)
        Q_regs<= #Tp N'b1;
    else
        Q_regs<= #Tp {Q_regs[1],Q_regs[0],Q_regs[1]^Q_regs
[2]};
end module
```

Lets code - Testbench

- Build a testbench for each module, SR and LFSR,
    - Observe their operation is correct through waveforms or with a self checking testbench
- Alternatively, add a clock enable to each module, and write a single testbench and test both modules together, one at a time.