Lecture 4

Sequential Logic

Pset 3 due Thursday

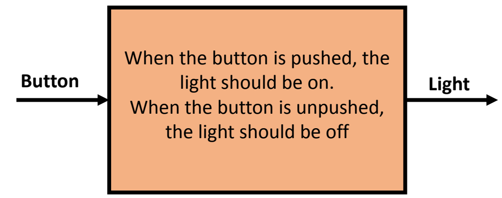
Lab 2 part 1 due Thursday

Lab 2 part 2 due Next Tuesday

Lab 3 out ~~this Thursday~~ Now!

Something We Can Build

What if you were given the following design specification:

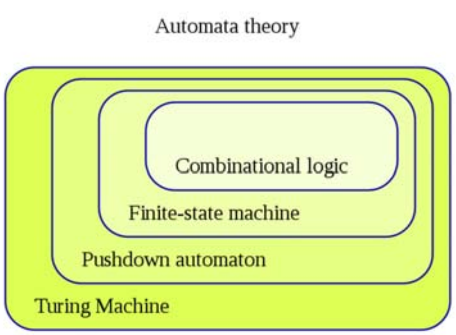


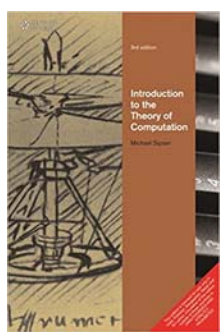

```
assign light = button;
```

*Done…This is all we need in life.

No it isn't

Levels of Complexity in Computation



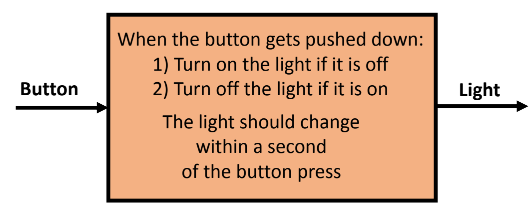


Sipser's Book

https://en.wikipedia.org/wiki/Automata_theory

Something We Cannot Build (Yet)

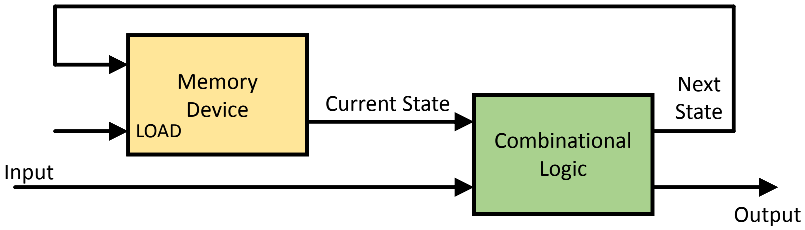
What if you were given the following design specification:



What makes this circuit so different from those we've discussed before?

1. "**State**" – i.e. the circuit has memory (become "state-ful")
2. The output was changed by an input "**event**" (pushing a button) rather than an input "value"

Digital State



Plan: Build a Sequential Circuit with stored digital STATE

- Memory stores CURRENT state
- Combinational Logic computes:
  - NEXT state (from input, current state)
  - OUTPUT values (from input, current state)
- State changes on LOAD control input

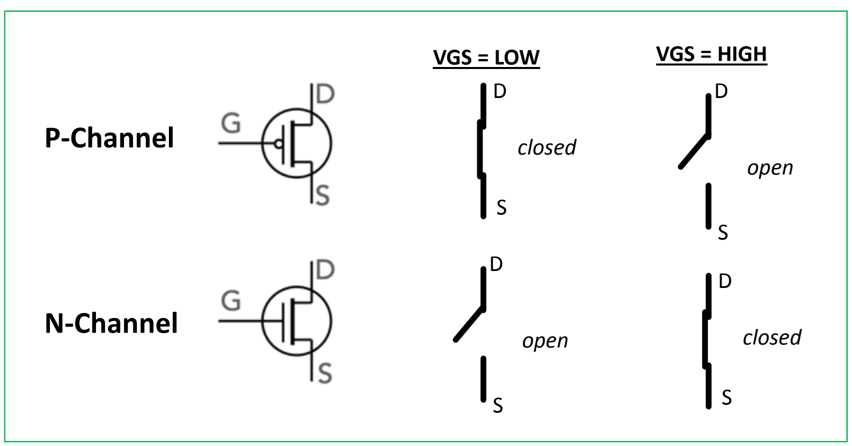
*When Output depends on input and current state, circuit is called a **Mealy** machine. If Output depends only on the current state, circuit is called a **Moore** machine.*

How is Digital State Created?

…Or How to Make Electronics Remember Previous Values

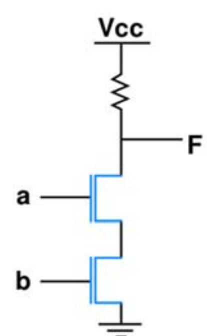
From the Ground Up:

- MOSFETs are electrically controlled switches:
  - Electricity can “gate” other electricity

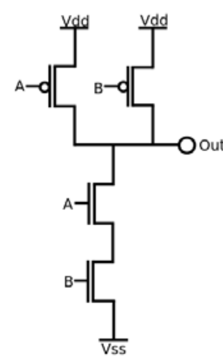


NAND Gate

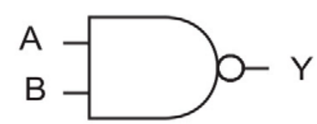
- Assemble higher functionality from the transistors



NMOS NAND gate



CMOS NAND gate

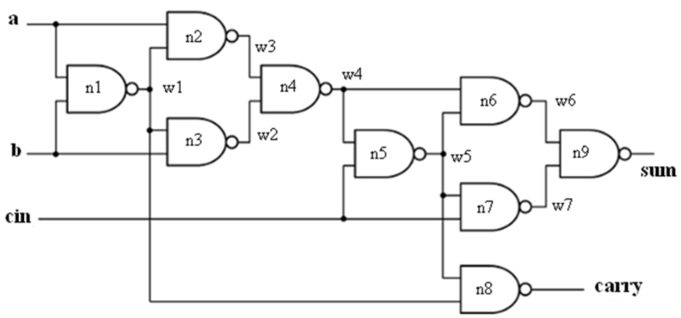


A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

(4 transistors)

Full Adder out of NANDs

Made completely of NAND gates and lets you add two one bit numbers plus a carry

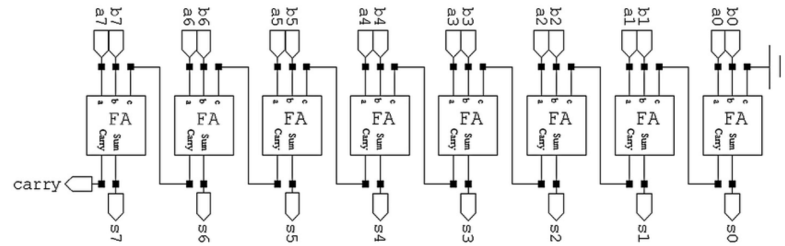


http://vlsiinnovator.blogspot.com/2015/02/full-adder-using-nand-gate-structural.html

(36 transistors)

8 bit Adder from Full Adders

Link together eight Full Adders to get an 8 bit adder… and so on (now you can add two numbers each up to 255)



(288 transistors)

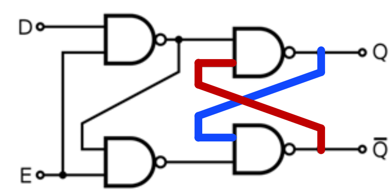
https://www.researchgate.net/figure/Eight-bit-Ripple-Carry-adder_fig2_283037309

And So On…

- We can keep wiring up larger and more complex digital systems which:
  - Given a set of inputs, provide a given set of outputs
  - Base outputs completely on layers of combinational logic
  - Will always respond the same way in time for a given input
- But there are other ways…

The D Latch

- Made of gates (which are made of transistors, which are made of sand(currently))
- Something different though…what is it?



"latch" means it holds whatever value was already present…basically: "Previous Q"

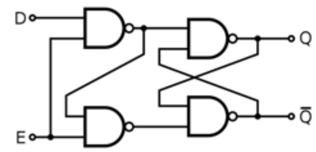
E	D	Q	$\overline{Q}$
0	0	latch	latch
0	1	latch	latch
1	0	0	1
1	1	1	0

E = "Enable" D = "Data" Q = not sure, but it is the output

https://www.allaboutcircuits.com/textbook/digital/chpt-10/d-latch/

The D Latch Provides Memory!

1. Set E=1
2. Set your D value
3. Set E=0
4. Whatever D was is stored at Q forever until E is 1 again!
5. **Can we do better/different?**

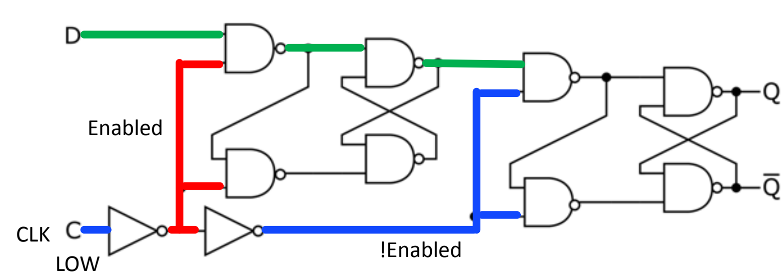


E	D	Q	$\overline{Q}$
0	0	latch	latch
0	1	latch	latch
1	0	0	1
1	1	1	0

E = "Enable" D = "Data" Q = not sure, but it is the output

The D Flip-Flop (Reg)

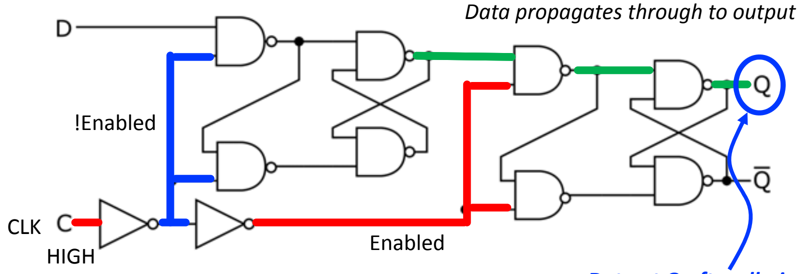
Two D-Latches in Series driven with opposite enable signals



CLK LINE is LOW

The D Flip-Flop (Reg)

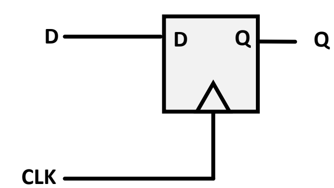
Two D-Latches in Series driven with opposite enable signals

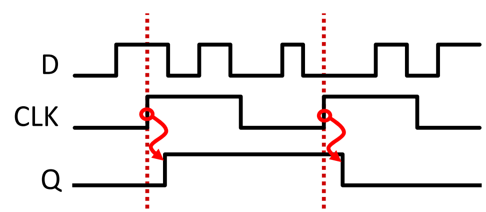


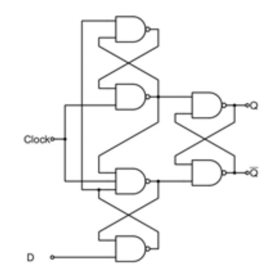
CLK LINE rises to HIGH

A New Building Block: the D Flip-Flop

- The edge-triggered D register: *on the rising edge of CLK*, the value of D is saved in the register and then appears shortly afterward on Q.







Example: 74LS74 internals

When you simplify some common/redundant logic between the two stages, you get to about ~25 transistors

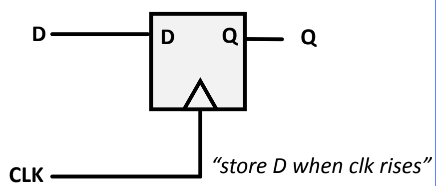
Registers, Latches, and Flip-Flops

- The terminology is a mess for historical reasons and just people in general, including myself. Here's one interpretation:
- A "register" is something that holds a value. Flip-flops and Latches *are* registers
- Further confusing the situation, people, including myself, often use "register" or "reg" to just refer to flip-flops

Use a lot!

D Flip-Flop

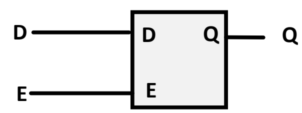
Edge-Triggered Sample-and-Hold Device



"store D when clk rises"

D Latch

Level-Triggered Sample-and-Hold Device



"store D when E is high"

Usage

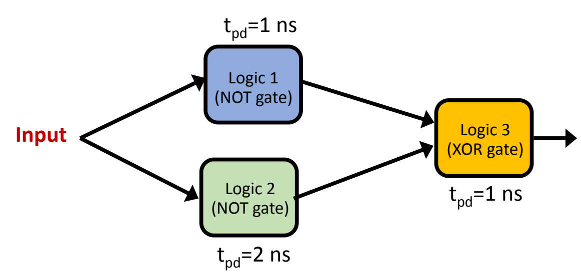
...Or what does this let us do now?

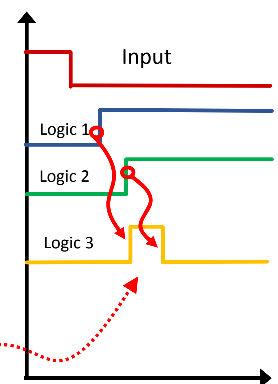
D Flip-Flop Registers Give Us A Few Critical Capabilities

- We can store values for later use
- We can sample values at precise times
  - *A rising edge is as close to a delta-function like event as we can get*
- We can design in stages:
  - Allow us to non-destructively limit signal propagation

Remember about Delays in Logic

- Every combinational circuit has delays regarding how slowly (or quickly) its outputs change in response to inputs, and this varies based on design/complexity
  - t_{cd} minimum time input takes to start to change output
  - t_{pd} maximum time input takes to finish changing output



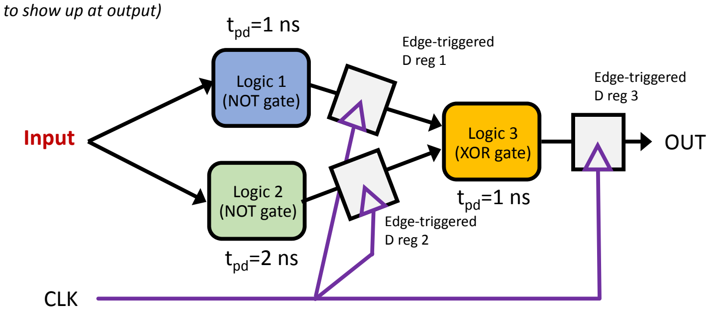


Glitch
Temporary solution based on differential delays

Remember about Delays in Logic

- Registers let us isolate/limit signal propagation and synchronize stages

t_{pd} is propagation delay (how long input takes to show up at output)

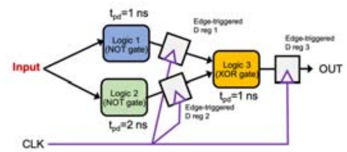


CLK is a synchronization signal

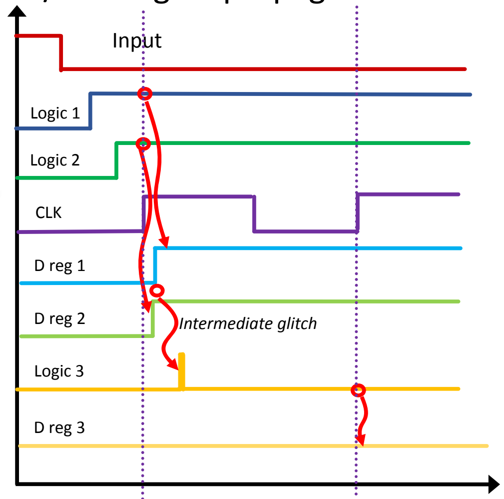
Remember about Delays in Logic

- Registers let us isolate/limit signal propagation and synchronize stages

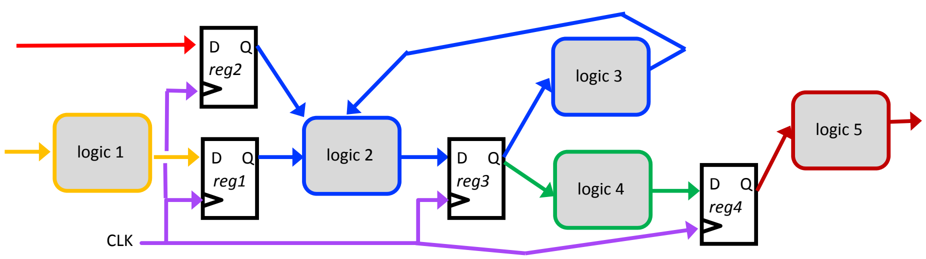
Previous page:



Intermediate glitches are minimized and suppressed in output

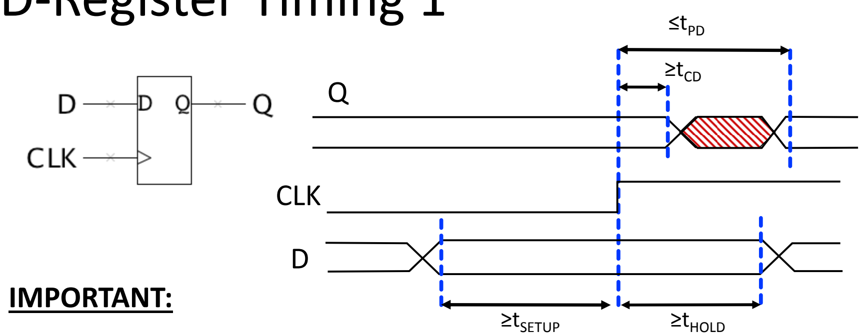


Design Complex Logic In Stages!



- D flip-flops regulate signal propagation!
- Design complex logic systems in stages
- Worry only about affects of delays (t_{pd} and t_{cd}) and glitches within a given stage, rather than how they all interplay!

D-Register Timing 1



IMPORTANT:

t_{PD}: maximum propagation delay, @posedge CLK D →Q
__Maximum__ time it takes for Q to change after rising edge of CLK

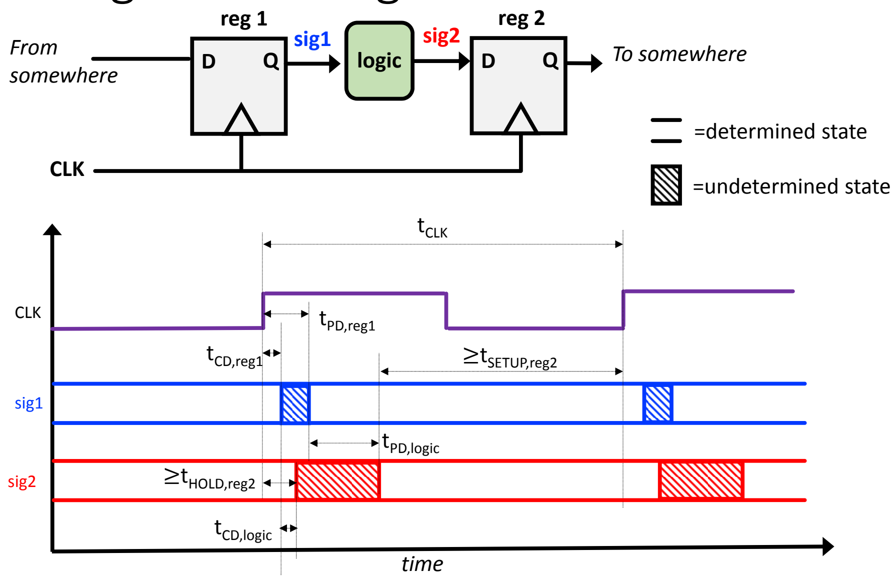
t_{CD}: minimum contamination delay, @posedge CLK D →Q
__Minimum__ time it takes for Q to start to change after rising edge of CLK

t_{SETUP}: setup time
How long D must be stable __before__ the rising edge of CLK

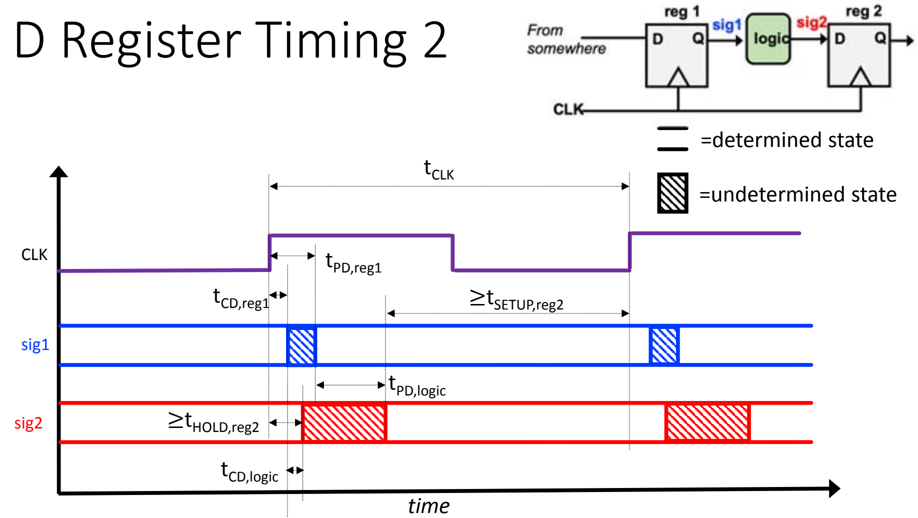
t_{HOLD}: hold time
How long D must be stable __after__ the rising edge of CLK

New timing attributes for registers

D Register Timing 2



D Register Timing 2



Two Requirements/ Conclusions:

$$t_{PD,reg1} + t_{PD,logic} + t_{SETUP,reg2} \leq t_{CLK}$$

$$t_{CD,reg1} + t_{CD,logic} \geq t_{HOLD,reg2}$$

D Register Timing 2

Two Requirements/ Conclusions:

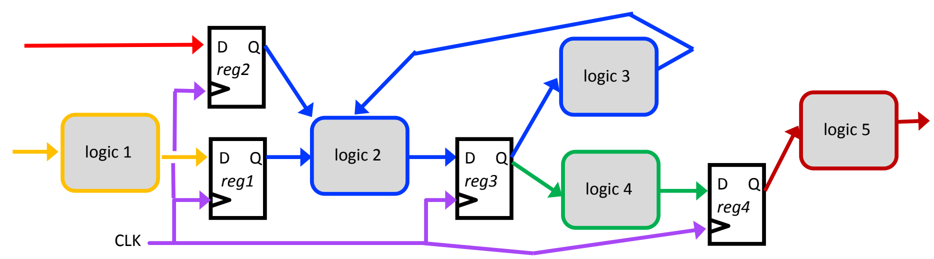
$$t_{PD,reg1} + t_{PD,logic} + t_{SETUP,reg2} \leq t_{CLK}$$

*We may encounter this in 6.111! If we try to make our combinational logic **tooooo complex** and we won't satisfying timing. How do we fix? Two options:*

$$t_{CD,reg1} + t_{CD,logic} \geq t_{HOLD,reg2}$$

*If you violate this, you have to change your design. This is more an issue for the device engineers…on our FPGAs the contamination delays (min change times) are usually longer than HOLD times, so it is hard for **us** to run into this problem in 6.111 (though it is a very real problem for people laying out circuits)*

Design Complex Logic In Stages!

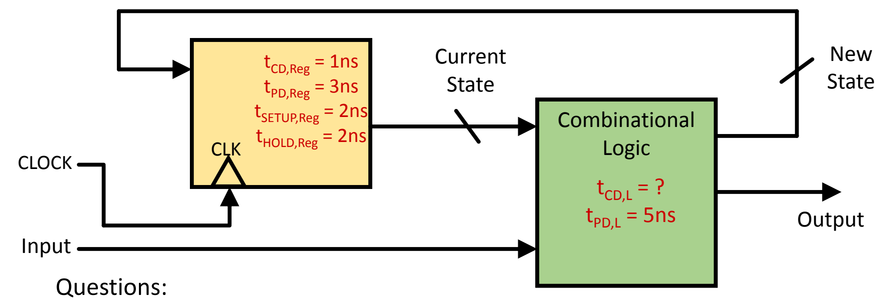


- Design complex logic systems in stages
- Worry only about affects of delays (t_{pd} and t_{cd}) within a given stage, rather than how they all interplay!

Single Clock Synchronous Discipline

- Single Clock signal shared among all clocked devices (one clock domain)
- Only care about the value of combinational circuits just before rising edge of clock
- Clock period greater than every combinational delay
- Change saved state after noise-inducing logic changes have stopped!

Sequential Circuit Timing



Questions:

- Constraints on t_{CD} for the logic?
- Minimum clock period?
- Setup, Hold times for Inputs?

This is a simple *Finite State Machine* … more in future classes!

Writing Sequential Logic

…Or how do we create sequential logic using SystemVerilog, overcome the problems of Verilog, and move forward as a Society

Lab 2 Starter Code

- Has a little bit of everything:

assign statements
Synthesize to combinational logic

always_comb blocks
Synthesize to combinational logic
Allow you to be more expressive than simple assign statements

always_ff blocks
Synthesize to sequential logic

```
module seven_seg_controller(input               clk_in,
                            input               rst_in,
                            input [31:0]        val_in,
                            output logic[7:0]   cat_out,
                            output logic[7:0]   an_out
    );

    logic[7:0]      segment_state;
    logic[31:0]     segment_counter;
    logic [3:0]     routed_vals;
    logic [6:0]     led_out;

    binary_to_seven_seg my_converter ( .val_in(routed_vals), .led_out(led_out));
    assign cat_out = ~led_out;
    assign an_out = ~segment_state;


    always_comb begin
        case(segment_state)
            8'b0000_0001:   routed_vals = val_in[3:0];
            8'b0000_0010:   routed_vals = val_in[7:4];
            8'b0000_0100:   routed_vals = val_in[11:8];
            8'b0000_1000:   routed_vals = val_in[15:12];
            8'b0001_0000:   routed_vals = val_in[19:16];
            8'b0010_0000:   routed_vals = val_in[23:20];
            8'b0100_0000:   routed_vals = val_in[27:24];
            8'b1000_0000:   routed_vals = val_in[31:28];
            default:        routed_vals = val_in[3:0];
        endcase
    end

    always_ff @(posedge clk_in)begin
        if (rst_in)begin
            segment_state <= 8'b0000_0001;
            segment_counter <= 32'b0;
        end else begin
            if (segment_counter == 32'd100_000)begin
                segment_counter <= 32'd0;
                segment_state <= {segment_state[6:0],segment_state[7]};
            end else begin
                segment_counter <= segment_counter +1;
            end
        end
    end

endmodule //seven_seg_controller
```

always

- In Verilog the always keyword is a way to specify logic (sequential, combinational) that is caused by an event (clock edge, change of state, etc)
- Very similar to an asynchronous callback in Javascript etc:
  - "When an event happens, do a certain thing:"
- Historically there was one always word and you would then specify a sensitivity list:
  - always @(x) = "when x changes"
  - always @(*) = "when anything changes (combinational)"
  - always @(posedge clk) = "when clk edge rises"
  - Etc…

Regs, Wires, Logics, and Life

- Original Verilog had two main datatypes
  - wire: Used for continuous assignment (combinational)
  - reg: Used to "store" values
- Despite its name being short for "register" a reg might not actually mean the design will synthesize to an actual *register*…It depended on usage in the Verilog.
- In particular it mostly depended on your sensitivity list in your always block and if you used block or non-blocking assignments (= or <=):
  - *posedge? Make a flip flop*
  - *values? Make it a combinational or possibly a latch*

SystemVerilog

- Drop the wire and reg terminology, just have logic and let compiler figure out if it becomes an actual register (flip-flop) or wire from use
- Use is specified more clearly now by replacing ambiguousness of generic always with specific use cases:
  - always_comb: build using combinational logic
  - always_ff: build using D-flip-flops (edge-trig sequential)
- What is synthesized is NOT "inferred" and more clearly based on user specification! ☺

Blocking vs. Nonblocking Assignment

- Within any type of always block you can assign things in two different ways:
- In both ways, you don't need the keyword assign
- *Blocking assignment (=):* evaluation and assignment are immediate; subsequent statements affected. (ORDER MATTERS)
- *Nonblocking assignment (<=):* all assignments deferred to end of simulation time step after all right-hand sides have been evaluated (*even those in other active* always *blocks*) (ORDER DOESN'T MATTER)

Blocking vs. Non-Blocking Assignments

- Verilog supports two types of assignments within always-type blocks, with subtly different behaviors.
- ***Blocking assignment (=):*** evaluation and assignment are immediate

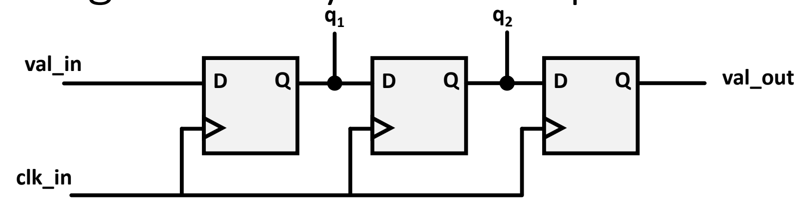
```
always_comb begin
  x = a | b;       // 1. evaluate a|b, assign result to x
  y = a ^ b ^ c;   // 2. evaluate a^b^c, assign result to y
  z = b & ~c;      // 3. evaluate b&(~c), assign result to z
end
```

- ***Nonblocking assignment (<=):*** all assignments deferred to end of simulation time step after all right-hand sides have been evaluated (*even those in other active* always *blocks*)

```
always_comb begin
  x <= a | b;     // 1. evaluate a|b, but defer assignment to x
  y <= a ^ b ^ c; // 2. evaluate a^b^c, but defer assignment to y
  z <= b & ~c;    // 3. evaluate b&(~c), but defer assignment to z
  // 4. end of time step: assign new values to x, y and z
end
```

Sometimes, as above, both produce the same result. **Sometimes, not!**

Assignment Styles for Sequential Logic



- Suppose we want to build the circuit above:
- Will nonblocking and blocking assignments both produce the desired result? ("old" means value **before** clock edge, "new" means the value **after** most recent assignment)

```
module nonblocking(
        input val_in, clk_in,
        output logic val_out
);
  logic q1, q2;
  always_ff @(posedge clk_in) begin
    q1 <= val_in;
    q2 <= q1;    // uses old q1
    val_out <= q2;  // uses old q2
  end
endmodule
```

```
module blocking(
  input val_in, clk_in,
  output logic val_out
);
  logic q1, q2;
  always_ff @(posedge clk_in) begin
    q1 = val_in;
    q2 = q1;    // uses new q1
    val_out = q2;  // uses new q2
  end
endmodule
```

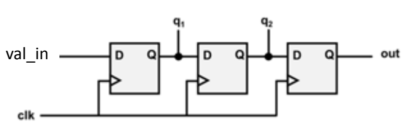

Use <u>Nonblocking</u> for Sequential Logic

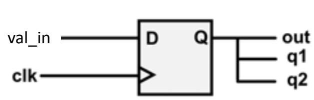
```
module nonblocking(
        input val_in, clk_in,
        output logic val_out
);
  logic q1, q2;
  always_ff @(posedge clk_in) begin
    q1 <= val_in;
    q2 <= q1;    // uses old q1
    val_out <= q2;  // uses old q2
  end
endmodule
```

"At each rising clock edge, q1, q2, and out simultaneously receive the old values of vin, q1, and q2."

```
module blocking(
  input val_in, clk_in,
  output logic val_out
);
  logic q1, q2;
  always_ff @(posedge clk_in) begin
    q1 = val_in;
    q2 = q1;    // uses new q1
    val_out = q2;  // uses new q2
  end
endmodule
```

"At each rising clock edge, q1 = vin.
After that, q2 = q1.
After that, out = q2.
Therefore out = vin.





- Blocking assignments ***do not*** reflect the intrinsic behavior of multi-stage sequential logic
- Guideline: use ***nonblocking*** assignments for always_ff blocks (Sequential always blocks)!!

General Strong Guidelines

- Blocking assignments (=) more closely align with how combinational works (use in always_comb)
- Non-blocking assignments (<=) more closely align with how we want to think about how sequential logic works (use in always_ff)
- Avoid mixing blocking and non-block assignments within one block!
  - Something will synthesize, but sometimes simulation will differ from what gets synthesized (built)
  - Really hard to comprehend for our limited human minds…so debugging is a nightmare

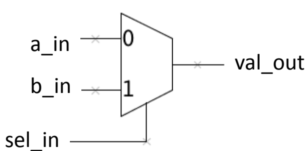
Example Uses with Assignments:

Combinatorial

```
module blob(input    a_in,
                     b_in,
                     sel_in,
                     output logic val_out);
  always_comb begin
    if (sel_in) val_out = b_in;
    else val_out = a_in;
  end

endmodule
```

Makes:

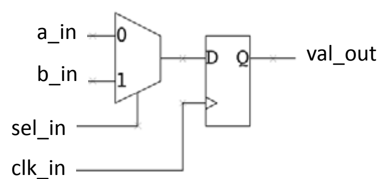


Sequential

```
module blob(input    a_in,
                     b_in,
                     sel_in,
                     clk_in,
                     output logic val_out);
  always_ff @(posedge clk_in) begin
    if (sel_in) val_out <= b_in;
    else val_out <= a_in;
  end

endmodule
```

Makes:



Coding Guidelines

- The following helpful guidelines are from this paper. If followed, they ensure your simulation results will match what they synthesized hardware will do:
  http://www.sunburst-design.com/papers/CummingsSNUG2000SJ_NBA.pdf

1. <u>When modeling sequential logic, use and always_ff with nonblocking assignments.</u>
2. <u>When modeling combinational logic with an always block, use always_comb with blocking assignments</u>.
3. When modeling both sequential and "combinational" logic within the same always block, use nonblocking assignments.
4. Do not mix blocking and nonblocking assignments in the same always block.
5. Do not make assignments to the same variable from more than one always block (this should throw errors, but might not if using blocking assignments)

- #1 thing we will be checking in your Verilog submissions!

The Sensitivity List in always_ff

- The use of posedge and negedge specifies edge you care about
- Can have combinational sensitivity lists, but must all be edge-based

D-Register with synchronous clear

```
module dff_sync_clear(
  input d_in, clearb_in, clk_in,
  output logic q_out
);
  always @(posedge clk_in)
    begin
      if (!clearb_in) q_out <= 1'b0;
      else q_out <= d_in;
    end
endmodule
```

always block entered only at each positive clock edge

D-Register with asynchronous clear

```
module dff_sync_clear(
  input d_in, clearb_in, clk_in,
  output logic q_out
);
  always @(negedge clearb_in or posedge clk_in)
    begin
      if (!clearb_in) q_out <= 1'b0;
      else q_out <= d_in;
    end
endmodule
```

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

Example: Simple Counter

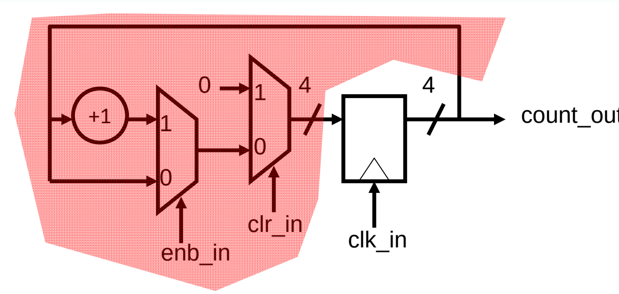
- Can still specify combinational logic when making sequential logic in always_ff blocks!

```
// 4-bit counter with enable and synchronous clear
module counter(input clk_in, enb_in, clr_in, output reg [3:0] count_out);

    always_ff @(posedge clk_in) begin
          count_out <= clr_in ? 4'b0 : (enb_in ? count_out+1 : count_out);
    end

endmodule
```

Combinational



Quite a bit similarity to Lab 2

Example: Simple Counter

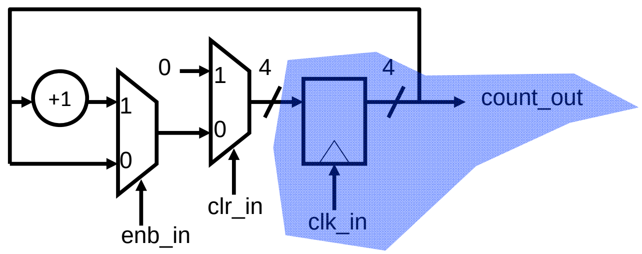
- Can still specify combinational logic when making sequential logic in always_ff blocks!

```
// 4-bit counter with enable and synchronous clear
module counter(input clk_in, enb_in, clr_in, output reg [3:0] count_out);

    always_ff @(posedge clk_in) begin
        count_out <= clr_in ? 4'b0 : (enb_in ? count_out+1 : count_out);
    end

endmodule
```

Sequential



Quite a bit similarity to Lab 2

Summary

- If a logic is assigned values with an assign statement OR inside a always_comb block, it will synthesize to the result of combinational logic
- If a logic is assigned values within a always_ff block, it will synthesize to a value on a D-flip-flop
- While you can mix = and <=, it is really, really discouraged:
  - Use = inside always_comb
  - Use <= inside always_ff

Lab 2 Starter Code

…With this all in mind, let's revisit Lab 2's starter code

Lab 2 Starter Code

- Has a little bit of everything:

assign statements
Synthesize to combinational logic

always_comb blocks
Synthesize to combinational logic
Allow you to be more expressive

always_ff blocks
Synthesize to sequential logic

```
module seven_seg_controller(input               clk_in,
                            input               rst_in,
                            input [31:0]        val_in,
                            output logic[7:0]   cat_out,
                            output logic[7:0]   an_out
    );

    logic[7:0]      segment_state;
    logic[31:0]     segment_counter;
    logic [3:0]     routed_vals;
    logic [6:0]     led_out;

    binary_to_seven_seg my_converter ( .val_in(routed_vals), .led_out(led_out));
    assign cat_out = ~led_out;
    assign an_out = ~segment_state;


    always_comb begin
        case(segment_state)
            8'b0000_0001:   routed_vals = val_in[3:0];
            8'b0000_0010:   routed_vals = val_in[7:4];
            8'b0000_0100:   routed_vals = val_in[11:8];
            8'b0000_1000:   routed_vals = val_in[15:12];
            8'b0001_0000:   routed_vals = val_in[19:16];
            8'b0010_0000:   routed_vals = val_in[23:20];
            8'b0100_0000:   routed_vals = val_in[27:24];
            8'b1000_0000:   routed_vals = val_in[31:28];
            default:        routed_vals = val_in[3:0];
        endcase
    end

    always_ff @(posedge clk_in)begin
        if (rst_in)begin
            segment_state <= 8'b0000_0001;
            segment_counter <= 32'b0;
        end else begin
            if (segment_counter == 32'd100_000)begin
                segment_counter <= 32'd0;
                segment_state <= {segment_state[6:0],segment_state[7]};
            end else begin
                segment_counter <= segment_counter +1;
            end
        end
    end

endmodule //seven_seg_controller
```

Implicit wire data types

- wire data types can only be given values through an assign statement (can't on left side of = or <= in an always block)
- Outputs on modules default to wires unless specified otherwise

```
module seven_seg_controller(input              clk_in,
                            input              rst_in,
                            input [31:0]       val_in,
                            output logic[7:0]  cat_out,
                            output logic[7:0]  an_out
    );
```

Could also say just `output [7:0] cat_out`

but then you can only do:

```
assign cat_out = blah blah blah;
```

If it is declared as a logic, then you can do either:

```
assign cat_out = blah blah blah;
```

OR

```
always_ff @(posedge clk_in)begin
    cat_out <= blah blah blah;
end
```

OR

```
always_comb begin
    cat_out = blah blah blah;
end
```

Depending on need…

Can write your combinational logic in whatever way you most prefer!

- Both types of assignments turn into combinational logic
- This one could be done with a nested ternary operator, but it would be gross

```
assign cat_out = ~led_out;
assign an_out = ~segment_state;

always_comb begin
    case(segment_state)
        8'b0000_0001:   routed_vals = val_in[3:0];
        8'b0000_0010:   routed_vals = val_in[7:4];
        8'b0000_0100:   routed_vals = val_in[11:8];
        8'b0000_1000:   routed_vals = val_in[15:12];
        8'b0001_0000:   routed_vals = val_in[19:16];
        8'b0010_0000:   routed_vals = val_in[23:20];
        8'b0100_0000:   routed_vals = val_in[27:24];
        8'b1000_0000:   routed_vals = val_in[31:28];
        default:        routed_vals = val_in[3:0];
    endcase
end
```


Same Thing

```
always_comb begin
    case(segment_state)
        8'b0000_0001:   routed_vals = val_in[3:0];
        8'b0000_0010:   routed_vals = val_in[7:4];
        8'b0000_0100:   routed_vals = val_in[11:8];
        8'b0000_1000:   routed_vals = val_in[15:12];
        8'b0001_0000:   routed_vals = val_in[19:16];
        8'b0010_0000:   routed_vals = val_in[23:20];
        8'b0100_0000:   routed_vals = val_in[27:24];
        8'b1000_0000:   routed_vals = val_in[31:28];
        default:        routed_vals = val_in[3:0];
    endcase
end
```

These do the same thing…create combinational logic

```
assign routed_vals = segment_state==8'b0000_0001? val_in[3:0]:
                     segment_state==8'b0000_0010? val_in[7:4]:
                     segment_state==8'b0000_0100? val_in[11:8]:
                     segment_state==8'b0000_1000? val_in[15:12]:
                     segment_state==8'b0001_0000? val_in[19:16]:
                     segment_state==8'b0010_0000? val_in[23:20]:
                     segment_state==8'b0100_0000? val_in[27:24]:
                     segment_state==8'b1000_0000? val_in[31:28]: val_in[3:0];
```


Sequential (Synchronous) Logic

"Every rising **clk** edge, if **rst_in** asserted, reset things to 0. Otherwise if **segment_counter** is 100,000, reset **segment_counter** and rotate **segment_state**. Otherwise, increment **segment_counter** by 1…"

```
always_ff @(posedge clk_in)begin
    if (rst_in)begin
        segment_state <= 8'b0000_0001;
        segment_counter <= 32'b0;
    end else begin
        if (segment_counter == 32'd100_000)begin
            segment_counter <= 32'd0;
            segment_state <= {segment_state[6:0],segment_state[7]};
        end else begin
            segment_counter <= segment_counter +1;
        end
    end
end
```

Resetting to a Known State!

- Usually one can't rely on registers powering-on to a particular initial state*. So most designs have a RESET signal that when asserted initializes all the state to known, mutually consistent initial values.

reset

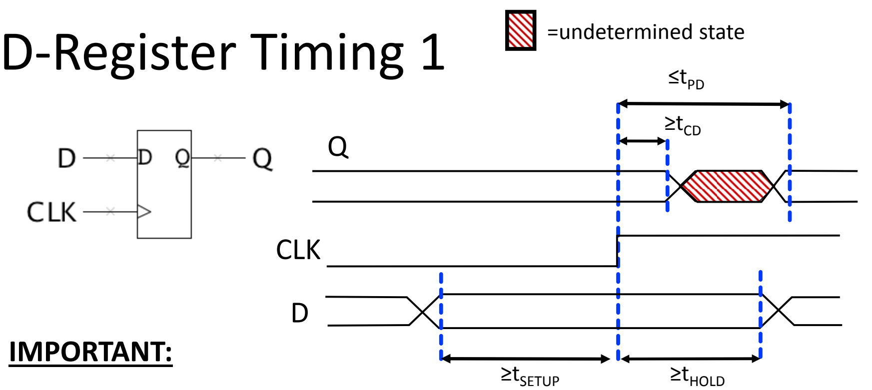
```
always_ff @(posedge clk_in)begin
    if (rst_in)begin
        segment_state <= 8'b0000_0001;
        segment_counter <= 32'b0;
    end else begin
        if (segment_counter == 32'd100_000)begin
            segment_counter <= 32'd0;
            segment_state <= {segment_state[6:0],segment_state[7]};
        end else begin
            segment_counter <= segment_counter +1;
        end
    end
end
```

** Actually, our FPGAs will reset all registers to 0 when the device is programmed, unless otherwise specified. But it's nice to be able to press a reset button to return to a known state rather than starting from scratch by reprogramming the device.*

Interfacing to Sequential Logic

…Or what are the problems with working with Sequential Logic?…. ***Optional for today depending on timing***

D-Register Timing 1



IMPORTANT:

t_{PD}: maximum propagation delay, @posedge CLK D →Q
__Maximum__ time it takes for Q to change after rising edge of CLK

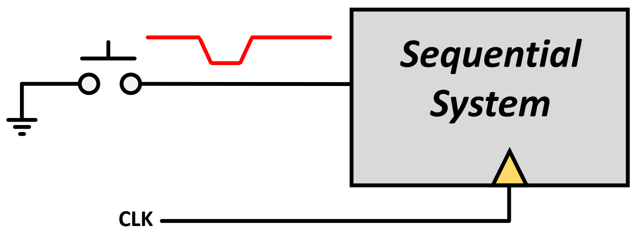
t_{CD}: minimum contamination delay, @posedge CLK D →Q
__Minimum__ time it takes for Q to start to change after rising edge of CLK

t_{SETUP}: setup time
How long D must be stable __before__ the rising edge of CLK

t_{HOLD}: hold time
How long D must be stable __after__ the rising edge of CLK

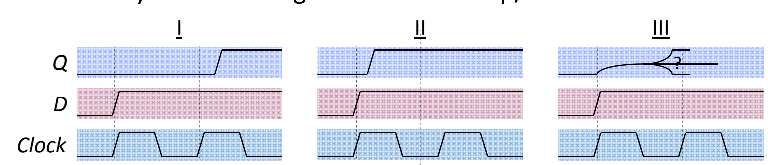
New timing attributes for registers

Asynchronous Inputs in Sequential Systems



Can't guarantee setup and hold times will be met!

When an asynchronous signal causes a setup/hold violation...



Transition is missed on first clock cycle, but caught on next clock cycle.

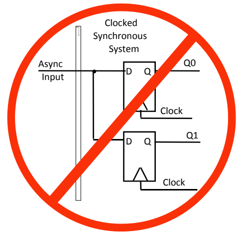
Transition is caught on first clock cycle.

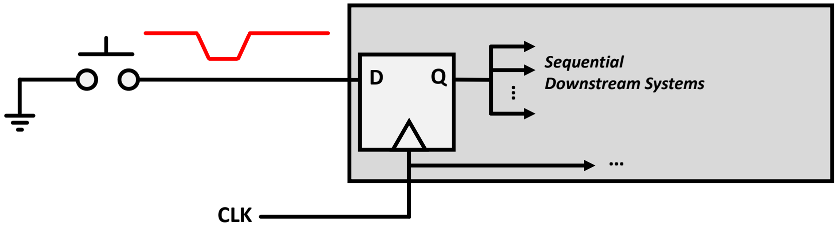
Output is metastable for an indeterminate amount of time.

Q: Which cases are problematic?

Asynchronous Inputs in Sequential Systems

- **All of them can be,** if more than one happens simultaneously within the same circuit.
- Guidelines: Ensure that external signals feed **exactly one** flip-flop





Metastability

- D-registers have metastable regions with all that feedback and stuff going on. Can go metastable

Figure 2. Effects of Violating t_{SU} & t_H Requirements

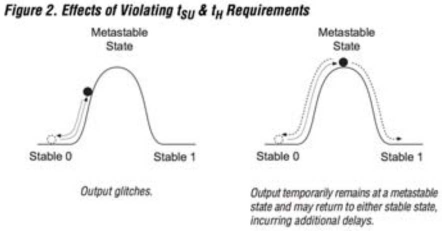
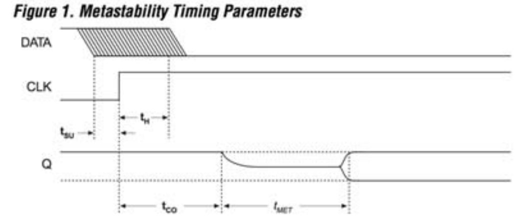


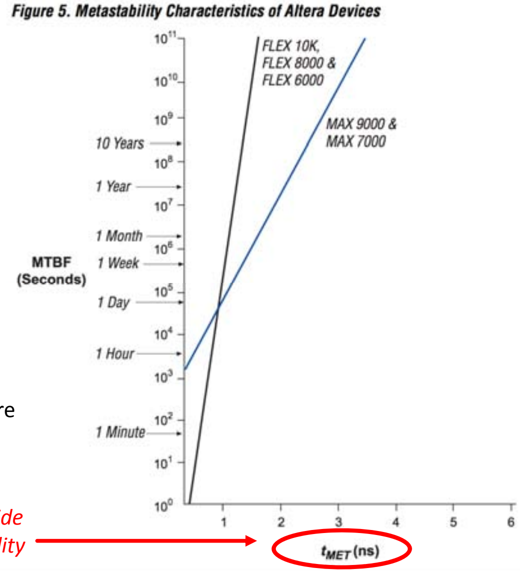
Figure 1. Metastability Timing Parameters



Metastability in Altera Devices
Altera Application Note 42 (1999)

MTBF

Figure 5. Metastability Characteristics of Altera Devices



MTBF: Mean Time Between Failure

Set by user (how much extra time do you provide per cycle for metastability to dissipate)

Metastability in Altera Devices
Altera Application Note 42 (1999)

Handling Metastability

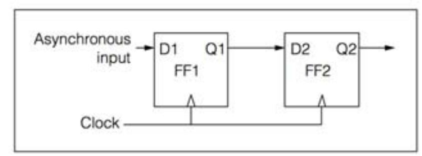
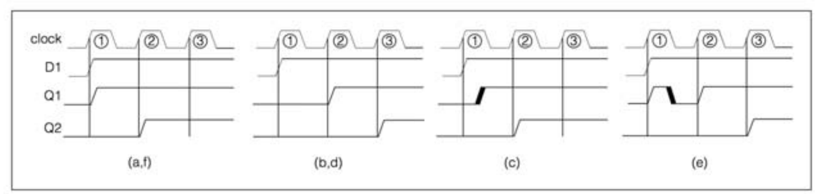


Figure 8. Two-flip-flop synchronization circuit.

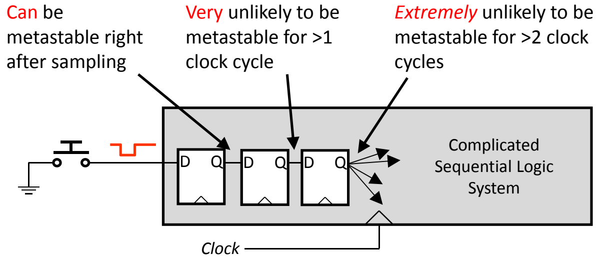


- FF2 (D-reg2) might go a clock cycle late, but it will almost never go metastable

"Metastability and Synchronizers: A Tutorial"
Ran Ginosar, Technion Israel Institute of Technology

Handling Metastability

- Preventing metastability turns out to be an impossible problem
- High gain of digital devices makes it likely that metastable conditions will resolve themselves quickly
- Solution to metastability: allow time for signals to stabilize



How many registers are necessary?

- Depends on many design parameters (clock speed, device speeds, …)
- In 6.111, a pair of synchronization registers is sufficient