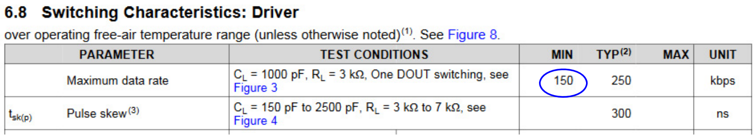
Sequential Logic

- Digital state: the D-Register
- Timing constraints for D-Registers
- Specifying registers in Verilog
- Blocking and nonblocking assignments
- Examples

Reminder: Lab #2 due Thursday

Lpset 2 Q1 - Datasheet Specs

- Parts are only guaranteed to meet *min/max specs – not typical.*
  - Lpset 2a: "As a design engineer using good engineering practice, what would you specify as the maximum data transfer rate to be published in a "labkit datasheet"? Ans: 150kps



6.8 Switching Characteristics: Driver

over operating free-air temperature range (unless otherwise noted)(1). See Figure 8.

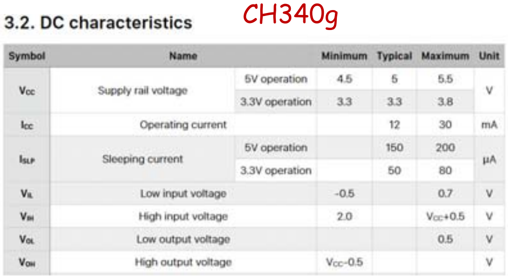
	PARAMETER	TEST CONDITIONS	MIN	TYP(2)	MAX	UNIT
	Maximum data rate	C_L = 1000 pF, R_L = 3 kΩ, One DOUT switching, see Figure 3	150	250		kbps
$t_{sk(p)}$	Pulse skew(3)	C_L = 150 pF to 2500 pF, R_L = 3 kΩ to 7 kΩ, see Figure 4		300		ns

- Typical values acceptable for prototyping, testing
  - Lpset 2b: 250kps

Lpset 2 Q1 - Datasheet Specs

- Understand voltage margins between families
  - Vcc max32222: 3.3 or 5v; CH340g: 3.3 or 5v
- Input/out voltages

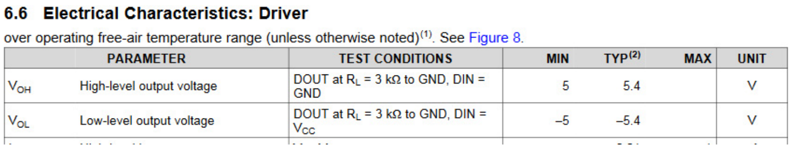
CH340g



3.2. DC characteristics

Symbol	Name		Minimum	Typical	Maximum	Unit
V_{CC}	Supply rail voltage	5V operation	4.5	5	5.5	V
		3.3V operation	3.3	3.3	3.8	
I_{CC}	Operating current			12	30	mA
I_{SLP}	Sleeping current	5V operation		150	200	μA
		3.3V operation		50	80	
V_{IL}	Low input voltage		-0.5		0.7	V
V_{IH}	High input voltage		2.0		V_{CC}+0.5	V
V_{OL}	Low output voltage				0.5	V
V_{OH}	High output voltage		V_{CC}-0.5			V

max3222



6.6 Electrical Characteristics: Driver

over operating free-air temperature range (unless otherwise noted)(1). See Figure 8.

	PARAMETER	TEST CONDITIONS	MIN	TYP(2)	MAX	UNIT
V_{OH}	High-level output voltage	DOUT at R_L = 3 kΩ to GND, DIN = GND	5	5.4		V
V_{OL}	Low-level output voltage	DOUT at R_L = 3 kΩ to GND, DIN = V_{CC}	–5	–5.4		V

Module Instantiation

Use Explicit Port Declarations

```
module mux32two
      (input [31:0] i0,i1,
       input sel,
       output [31:0] out);

   assign out = sel ? i1 : i0;
endmodule
```

```
mux32two   adder_mux(.i0(b), .i1(32'd1),
              .sel(f[0]), .out(addmux_out));
```

```
mux32two   adder_mux(b, 32'd1, f[0], addmux_out);
```

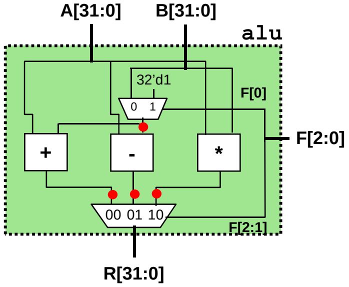
Order of the ports matters!

Top-Level ALU Declaration

- Given submodules:

```
module mux32two(i0,i1,sel,out);
module mux32three(i0,i1,i2,sel,out);
module add32(i0,i1,sum);
module sub32(i0,i1,diff);
module mul16(i0,i1,prod);
```

- Declaration of the ALU Module:



```
module alu
  (input [31:0] a, b,
  input [2:0] f,
  output [31:0] r);
  wire [31:0] submux_out;
  wire [31:0] add_out, sub_out, mul_out;

  mux32two   sub_mux(b, 32'd1, f[0], submux_out);
  add32      our_adder(a, addmux_out, add_out);
  sub32      our_subtracter(a, submux_out, sub_out);
  mul16      our_multiplier(a[15:0], b[15:0], mul_out);
  mux32three output_mux(add_out, sub_out, mul_out, f[2:1], r);

endmodule
```

intermediate output nodes ●

module names

(unique) instance names

corresponding wires/regs in module alu

Verilog Summary

- Verilog – Hardware description language – not software program.
- A convention: lowercase for variables, UPPERCASE for parameters

```
module blob
  #(parameter WIDTH = 64, // default width: 64 pixels
              HEIGHT = 64, // default height: 64 pixels
              COLOR = 3'b111) // default color: white
   (input [10:0] x,hcount, input [9:0] y,vcount, output reg [2:0] pixel);
 endmodule
```

- wires

```
wire a,b,z;               // three 1-bit wires
wire [31:0] memdata;      // a 32-bit bus
wire [7:0] b1,b2,b3,b4;   // four 8-bit buses
wire [WIDTH-1:0] input;   // parameterized bus
```

Examples

parameter MSB = 7; // defines msb as a constant value 7

parameter E = 25, F = 9; // defines two constant numbers

parameter BYTE_SIZE = 8,
BYTE_MASK = BYTE_SIZE - 1;

parameter [31:0] DEC_CONST = 1'b1; // value converted to 32 bits

parameter NEWCONST = 3'h4; // implied range of [2:0]

parameter NEWCONS = 4; // implied range of at least [31:0]

Something We Can't Build (Yet)

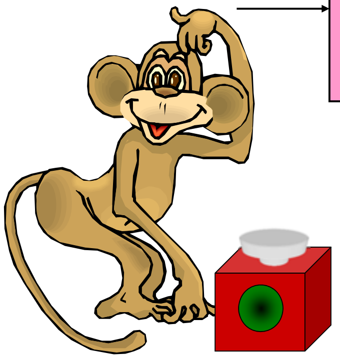
What if you were given the following design specification:

button

When the button is pushed:
1) Turn on the light if it is off
2) Turn off the light if it is on

The light should change state within a second of the button press

light

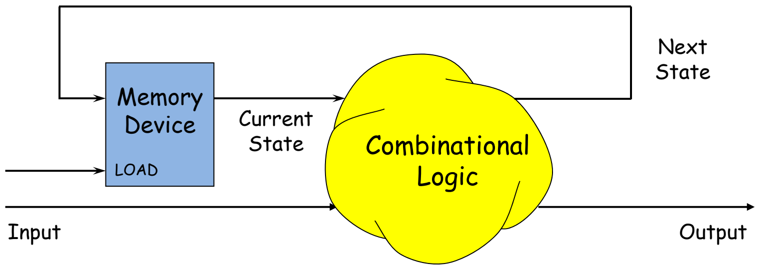


What makes this circuit so different from those we've discussed before?

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

Digital State

One model of what we'd like to build



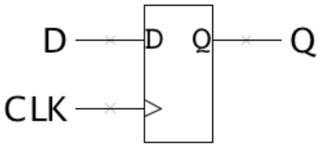
Plan: Build a Sequential Circuit with stored digital STATE –

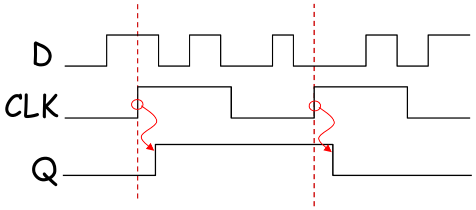
- Memory stores CURRENT state, produced at output
- Combinational Logic computes
  - NEXT state (from input, current state)
  - OUTPUT bit (from input, current state)
- State changes on LOAD control input

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.

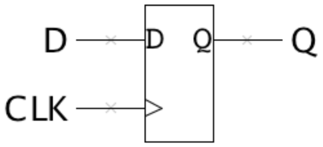
Our next building block: the D register

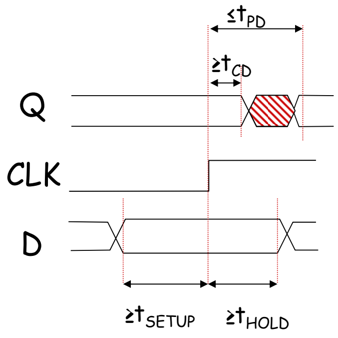
The edge-triggered D register: *on the rising edge of CLK*, the value of D is saved in the register and then shortly afterwards appears on Q.





D-Register Timing - I





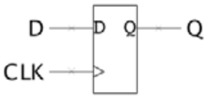
t_{PD}: maximum propagation delay, CLK →Q

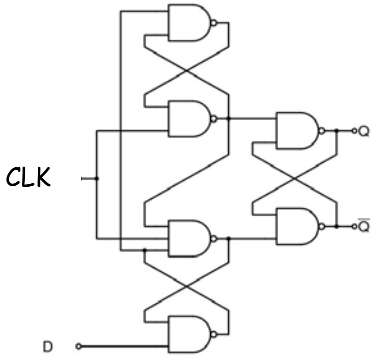
t_{CD}: minimum contamination delay, CLK →Q

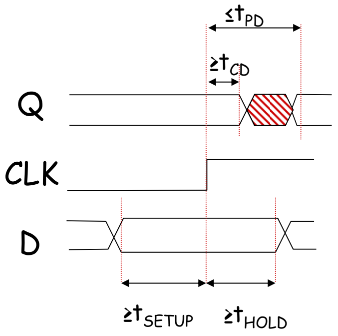
t_{SETUP}: setup time
How long D must be stable before the rising edge of CLK

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

D-Register Internals – 74LS74



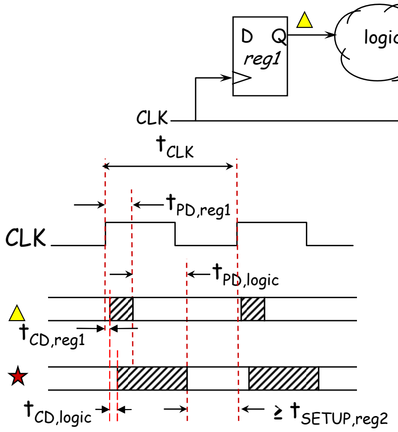




t_{SETUP} = 20ns t_{PD-HL} = 40ns

t_{HOLD} = 5ns t_{PD-LH} = 25ns

D-Register Timing - II



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

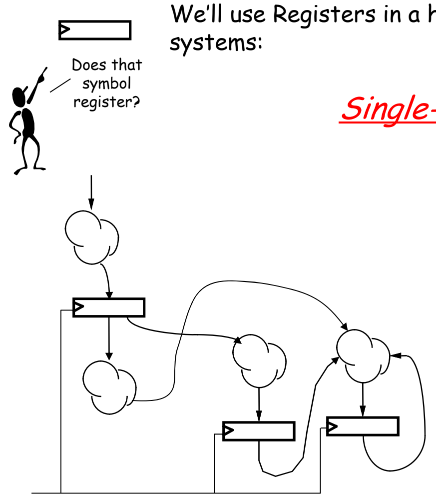
The good news: you can choose t_{CLK} so that this constraint is satisfied!

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

The bad news: you have to change your design if this constraint isn't met.

Single-clock Synchronous Circuits

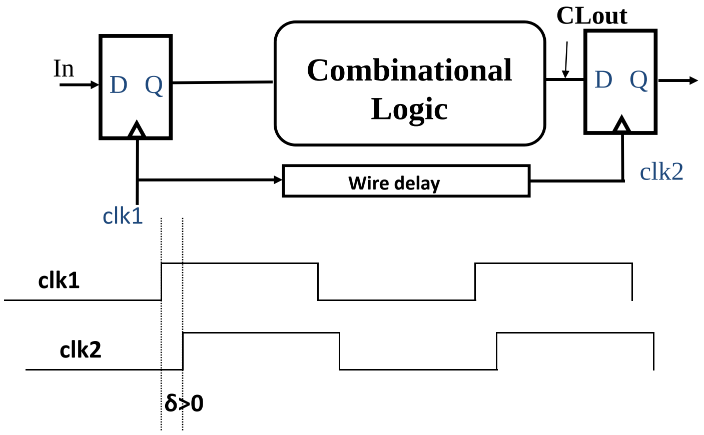
We'll use Registers in a highly constrained way to build digital systems:



Single-clock Synchronous Discipline

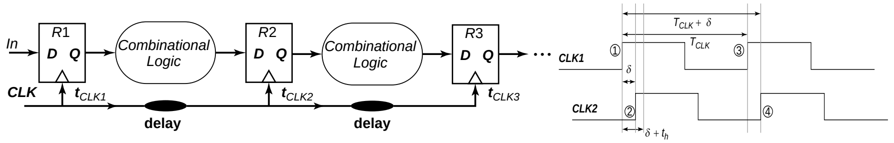
- No combinational cycles
- Single clock signal shared among all clocked devices (one *clock domain*)
- Only care about value of combinational circuits just before rising edge of clock
- Clock period greater than every combinational delay
- Change saved state after noise-inducing logic transitions have stopped!

Clocks are Not Perfect: Clock Skew



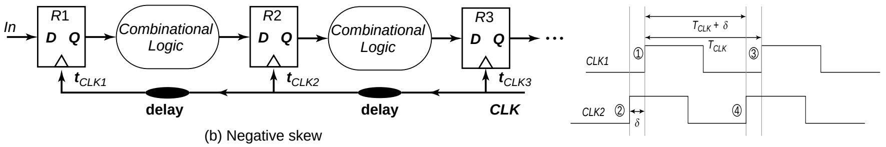
$t_{skew} = t_{clk2} - t_{clk1}$

Positive and Negative Skew



(a) Positive skew

Launching edge arrives before the receiving edge

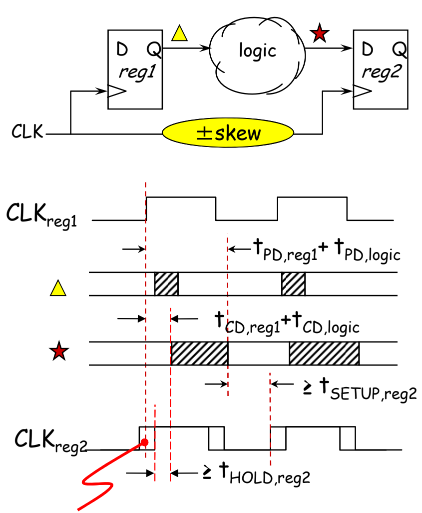


(b) Negative skew

Receiving edge arrives before the launching edge

➢**Adapted from J. Rabaey, A. Chandrakasan, B. Nikolic, "Digital Integrated Circuits: A Design Perspective" Copyright 2003 Prentice Hall/Pearson.**

D-Register Timing With Skew



CLK_{reg2} rising edge might fall anywhere in this region.

In the real world the clock signal arrives at different registers at different times. The difference in arrival times (pos or neg) is called the *clock skew t_{skew}*.

$t_{skew} = t_{Rn,clk2} - t_{Rn,clk1}$

We can update our two timing constraints to reflect the worst-case skew

Setup time: $t_{Rn,clk} = t_{Rn+1,clk}$

$t_{Rn,clk1}+t_{PD,reg1}+t_{PD,logic}+t_{SETUP,reg2} \leq t_{Rn+1,clk2}$

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

Hold time:

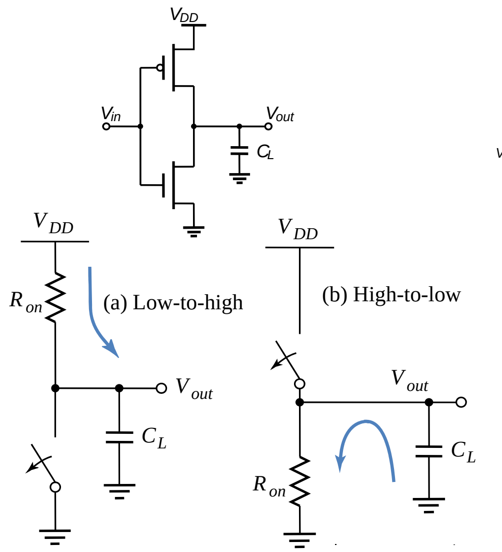
$t_{Rn,clk1}+t_{CD,reg1}+t_{CD,logic} \geq t_{Rn,clk2}+t_{HOLD,reg2}$

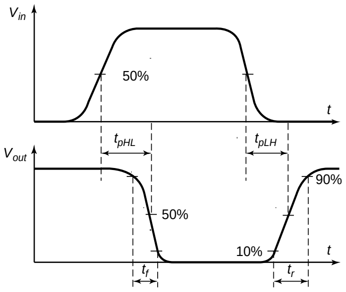
$t_{CD,reg1}+t_{CD,logic} \geq t_{HOLD,reg2}+t_{skew}$

Thus clock skew increases the minimum cycle time of our design and makes it harder to meet register hold times.

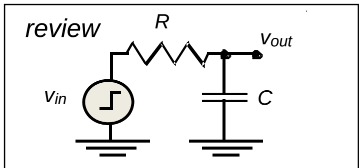
Which skew is tougher to deal with (pos or neg)?

Delay Estimation : Simple RC Networks





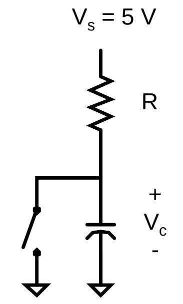
review



$v_{out}(t) = (1 - e^{-t/\tau})\,V$

$t_p = \ln(2)\,\tau = 0.69\ RC$

RC Equation



$$V_c = 5\left(1 - e^{-\frac{t}{RC}}\right)$$

V_s = 5 V

Switch is closed t<0

Switch opens t>0

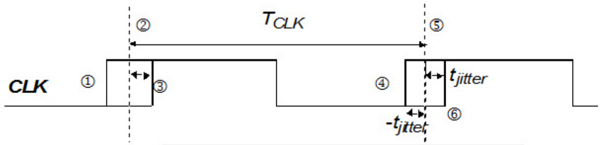
$V_s = V_R + V_C$

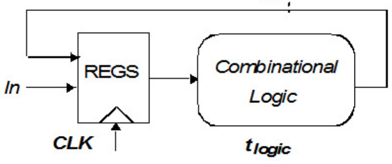
$V_s = i_R R + V_c \qquad i_R = C\frac{dV_c}{dt}$

$V_s = RC\frac{dV_c}{dt} + V_c$

$$V_c = V_s\left(1 - e^{-\frac{t}{RC}}\right)$$

Clocks are Not Perfect: Clock Jitter



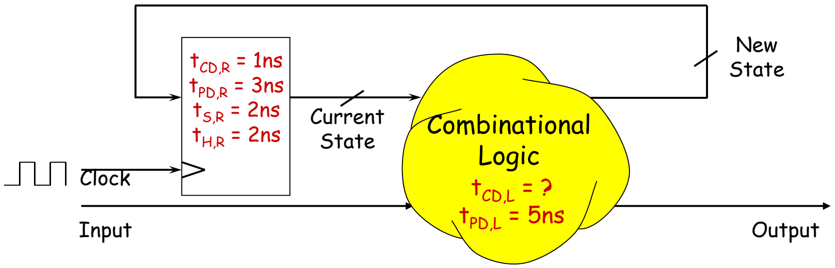


t_{pd}, t_{su}, t_{hold}

$t_{clk} - 2t_{jitter} > t_{pd} + t_{su} + t_{logic}$

Typical crystal oscillator
100mhz (10ns)
Jitter: 1ps

Sequential Circuit Timing



Questions:

Constraints on t_{CD} for the logic?	> 1 ns
Minimum clock period?	> 10 ns ($t_{PD,R}+t_{PD,L}+t_{SETUP,R}$)
Setup, Hold times for Inputs?	$t_{SETUP,Input} = t_{PD,L} + t_{SETUP,R}$ $t_{HOLD,Input} = t_{HOLD,R} - t_{CD,L}$

This is a simple *Finite State Machine* … more on next time!

The Sequential always Block

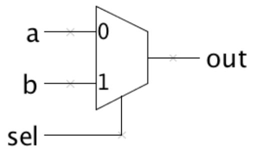
Edge-triggered circuits are described using a sequential always block

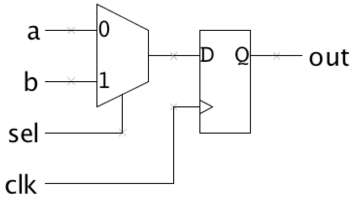
Combinational

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

Sequential

```
module seq(input a, b, sel, clk,
           output reg out);
  always @(posedge clk) begin
    if (sel) out <= b;
    else out <= a;
  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-Register with synchronous clear

```
module dff_sync_clear(
  input d, clearb, clock,
  output 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-Register with asynchronous clear

```
module dff_sync_clear(
  input d, clearb, clock,
  output 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 with same trigger execute concurrently…

Blocking vs. Nonblocking Assignments

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

```
always @(*) 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 <u>all</u> right-hand sides have been evaluated (*even those in other active* always *blocks*)

```
always @(*) 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!

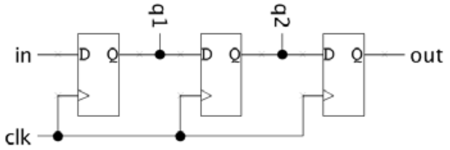
Blocking vs. Nonblocking Assignments

- Guaranteed question on job interviews with Verilog questions.
- *Blocking assignment (=):* evaluation and assignment are immediate; subsequent statements affected.
- *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*)

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

Assignment Styles for Sequential Logic

What we want:
Register Based
Digital Delay Line



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 in, clk,
  output reg out
);
  reg q1, q2;
  always @(posedge clk) begin
    q1 <= in;
    q2 <= q1;   // uses old q1
    out <= q2;  // uses old q2
  end

endmodule
```

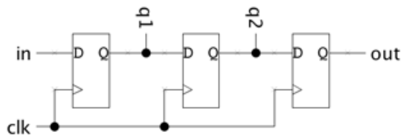
```
module blocking(
  input in, clk,
  output reg out
);
  reg q1, q2;
  always @(posedge clk) begin
    q1 = in;
    q2 = q1;   // uses new q1
    out = q2;  // uses new q2
  end

endmodule
```

Use Nonblocking for Sequential Logic

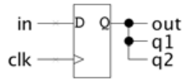
```
always @(posedge clk) begin
    q1 <= in;
    q2 <= q1;   // uses old q1
    out <= q2;  // uses old q2
  end
```

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

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



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



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

---

always block

- Sequential always block: always @(posedge clock) use <=
- Combinatorial always block: always @ * use =
- Results of operators (LHS) inside always block (sequential and combinatorial) must be declared as "reg"
- Equivalent Verilog

reg z
always @ *
z = x && y

← same as →
example of
combinatorial
always block

assign z = x && y
// z not a "reg"

- case statements must be used within an always block; include default case

Sequential always block style

```
// There are two styles for creating this sample divider. The
// first uses sequential always block for state assignment and
// a combinational always block for next-state. This style tends
// to result in fewer errors.
//
// An alternate approach is to use a single always block. An example
// of a divide by 5 counter will illustrate the differences
```

```
///////////////////////////////////
// Sequential always block with a
//   combinational always block

 reg [3:0] count1, next_count1;

 always @(posedge clk)
    count1 <= next_count1;

  always @*  begin
    if (reset) next_count1 = 0;
    else next_count1 =
         (count1 == 4) ? 0 : count1 + 1;
    end

 assign enable1 = (count1 == 4);
///////////////////////////////////
```

```
//////////////////////////////////
// Single always block
//

   reg [3:0] count2;

  always @(posedge clk) begin
    if (reset) count2 <= 0;
        else count2 <=
                  (count2 == 4) ? 0 : count2 + 1;
    end

    assign enable2 = (count2 == 4);

///////////////////////////////////
```

Coding Guidelines

The following helpful guidelines are from the Cummings paper. If followed, they ensure your simulation results will match what they synthesized hardware will do:

1. When modeling sequential logic, use nonblocking assignments.
2. When modeling latches, use nonblocking assignments.
3. When modeling combinational logic with an always block, use blocking assignments.
4. When modeling both sequential and "combinational" logic within the same always block, use nonblocking assignments.
5. Do not mix blocking and nonblocking assignments in the same always block.
6. Do not make assignments to the same variable from more than one always block.
7. Use $strobe to display values that have been assigned using nonblocking assignments.
8. Do not make assignments using #0 delays.

For more info see: http://www.sunburst-design.com/papers/CummingsSNUG2002Boston_NBAwithDelays.pdf

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

Guideline 4: Sequential and "combinatorial" logic in the same always block

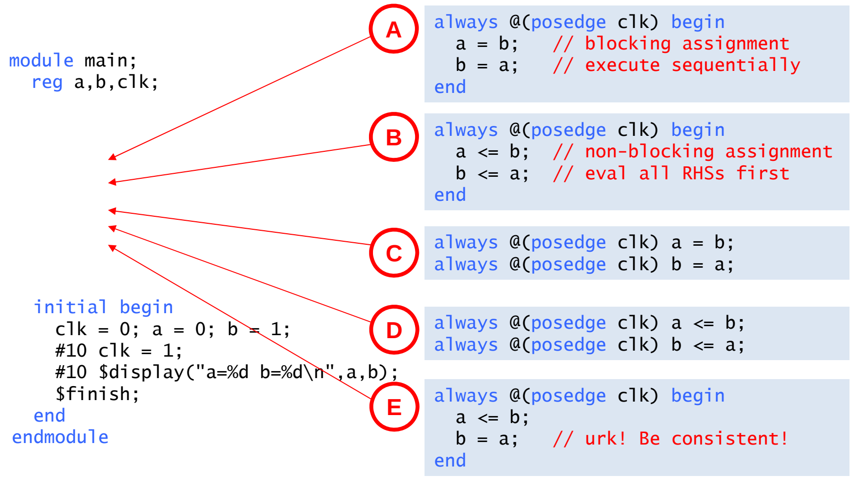
```
module nbex1
 (output reg q,
  input clk, rst_n,
  input a, b);

  reg y;
  always @(a or b)
  y = a ^ b;

  always @(posedge clk or
               negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else q <= y;

endmodule
```

Combinatorial logic

```
module nbex2
 (output q,
  input clk, rst_n,
  input a, b);

  reg q;
  always @(posedge clk or
               negedge rst_n)
    if (!rst_n) q <= 1'b0;
     else q <= a ^ b;

endmodule
```

Combinatorial logic

= vs. <= inside always



```
module main;
  reg a,b,clk;



  initial begin
    clk = 0; a = 0; b = 1;
    #10 clk = 1;
    #10 $display("a=%d b=%d\n",a,b);
    $finish;
  end
endmodule
```

A
```
always @(posedge clk) begin
  a = b;   // blocking assignment
  b = a;   // execute sequentially
end
```

B
```
always @(posedge clk) begin
  a <= b;  // non-blocking assignment
  b <= a;  // eval all RHSs first
end
```

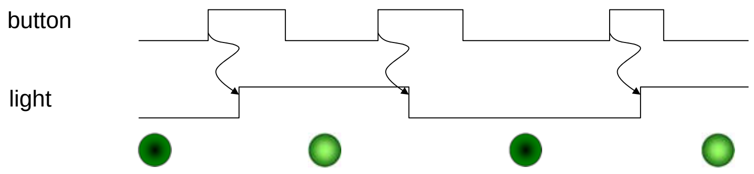
C
```
always @(posedge clk) a = b;
always @(posedge clk) b = a;
```

D
```
always @(posedge clk) a <= b;
always @(posedge clk) b <= a;
```

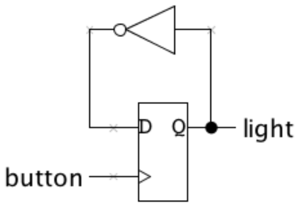
E
```
always @(posedge clk) begin
  a <= b;
  b = a;   // urk! Be consistent!
end
```

Rule: always change state using <= (e.g., inside `always @(posedge clk)`…)

Implementation for on/off button



```
module onoff(input button, output reg light);
  always @(posedge button) light <= ~light;
endmodule
```



Synchronous on/off button

When designing a system that accepts many inputs it would be hard to have input changes serve as the system clock (which input would we use?). So we'll use a single clock of some fixed frequency and have the inputs control what state changes happen on rising clock edges.

For most of our lab designs we'll use a 27MHz system clock (37ns clock period).

```
module onoff_sync(input clk, button,
                  output reg light);
  always @ (posedge clk) begin
    if (button) light <= ~light;
  end
endmodule
```

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.

```
module onoff_sync(input clk, reset, button,
                  output reg light);
  always @ (posedge clk) begin
    if (reset) light <= 0;
    else if (button) light <= ~light;
  end
endmodule
```

* Actually, our FPGAs will reset all registers to 0 when the device is programmed. 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.

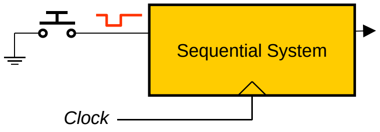
Clocks are fast, we're slow!

The circuit on the last slide toggles the light on every rising clock edge for which button is 1. But clocks are fast (27MHz!) and our fingers are slow, so how do we press the button for just one clock edge? Answer: we can't, but we can add some state that remembers what button was last clock cycle and then detect the clock cycles when button changes from 0 to 1.

```
module onoff_sync(input clk, reset, button,
                  output reg light);
  reg old_button;  // state of button last clk
  always @ (posedge clk) begin
    if (reset)
      begin light <= 0; old_button <= 0; end
    else if (old_button==0 && button==1)
      // button changed from 0 to 1
      light <= ~light;
    old_button <= button;
  end
endmodule
```

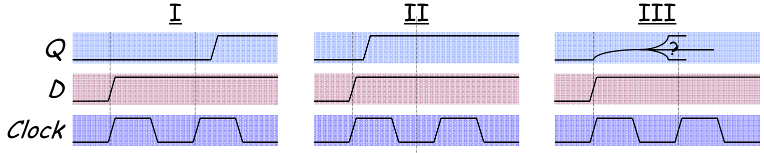
Asynchronous Inputs in Sequential Systems

What about external signals?



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

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



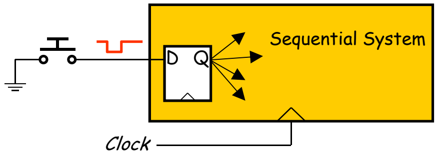
I	II	III
Transition is missed on first clock cycle, but caught on next clock cycle.	Transition is caught on first clock cycle.	Output is metastable for an indeterminate amount of time.

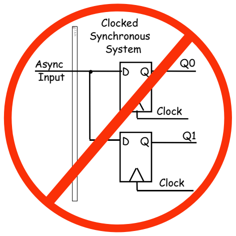
Q: Which cases are problematic?

Asynchronous Inputs in Sequential Systems

All of them can be, if more than one happens simultaneously within the same circuit.

Guideline: ensure that external signals directly feed exactly one flip-flop



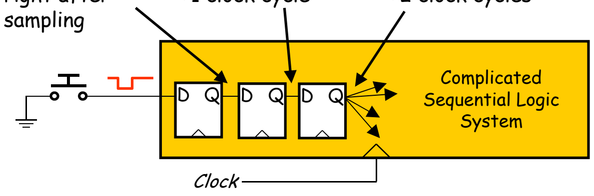


This prevents the possibility of I and II occurring in different places in the circuit, but what about metastability?

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

Can be metastable right after sampling	Very unlikely to be metastable for >1 clock cycle	*Extremely* unlikely to be metastable for >2 clock cycles



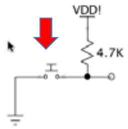
How many registers are necessary?

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

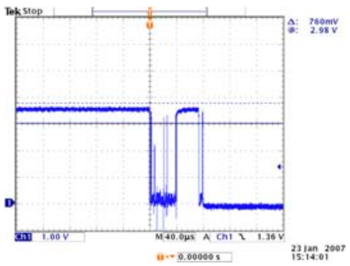
---

One last little problem…

Mechanical buttons exhibit contact "bounce" when they change position, leading to multiple output transitions before finally stabilizing in the new position:



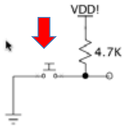
We need a debouncing circuit!



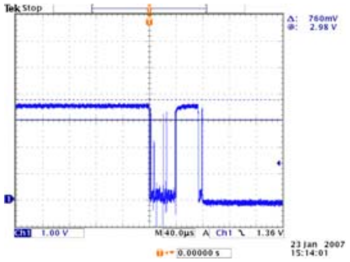
```
// Switch Debounce Module
// use your system clock for the clock input
// to produce a synchronous, debounced output
// DELAY = .01 sec with a 27Mhz clock
module debounce #(parameter DELAY=270000-1)
                (input reset, clock, bouncey,
                 output reg steady);

   reg [18:0] count;
   reg old;

   always @(posedge clock)

endmodule
```

One last little problem…

Mechanical buttons exhibit contact "bounce" when they change position, leading to multiple output transitions before finally stabilizing in the new position:



We need a debouncing circuit!



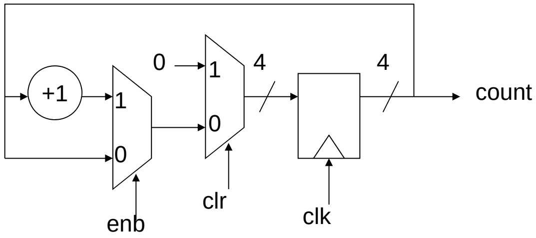
```
// Switch Debounce Module
// use your system clock for the clock input
// to produce a synchronous, debounced output
// DELAY = .01 sec with a 27Mhz clock
module debounce #(parameter DELAY=270000-1)
                (input reset, clock, bouncey,
                 output reg steady);

   reg [18:0] count;
   reg old;

   always @(posedge clock)
     if (reset)    // return to known state
       begin
         count <= 0;
         old <= bouncey;
         steady <= bouncey;
       end
     else if (bouncey != old)  // input changed
       begin
         old <= bouncey;
         count <= 0;
       end
     else if (count == DELAY) // stable!
         steady <= old;
     else                     // waiting…
       count <= count+1;

endmodule
```

On/off button: final answer

```
module onoff_sync(input clk, reset, button_in,
                  output reg light);
  // synchronizer
  reg button,btemp;
  always @(posedge clk)
    {button,btemp} <= {btemp,button_in};

  // debounce push button
  wire bpressed;
  debounce db1(.clock(clk),.reset(reset),
               .bouncey(button),.steady(bpressed));

  reg old_bpressed;  // state last clk cycle
  always @ (posedge clk) begin
    if (reset)
      begin light <= 0; old_bpressed <= 0; end
    else if (old_bpressed==0 && bpressed==1)
      // button changed from 0 to 1
      light <= ~light;
    old_bpressed <= bpressed;
  end
endmodule
```

Example: A Simple Counter

Isn't this a lot like Exercise 1 in Lab 2?



```
// 4-bit counter with enable and synchronous clear
module counter(input clk,enb,clr,
               output reg [3:0] count);
  always @(posedge clk) begin
    count <= clr ? 4'b0 : (enb ? count+1 : count);
  end
endmodule
```