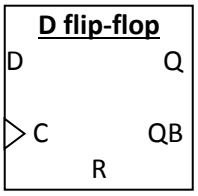
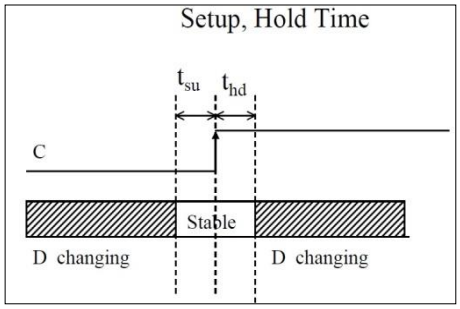
Delay and Other Non-Ideal Behavior in State Machines; Sequential MSI devices[1]

We're (hopefully) doing three things today.

1. Looking in a lot more detail at non-ideal behavior of state machines
   This is basically a way of saying "we need to build state machines out of real parts, and those real parts have real issues". We'll look at what those issues are and how to deal with them. In the process of doing that we'll review delay in combainational logic.
2. Spend some time on more Medium Scale Integrated (MSI) devices.
   Registers, Counters, Shift Registers.
3. Look at how to do sequential logic in Verilog.



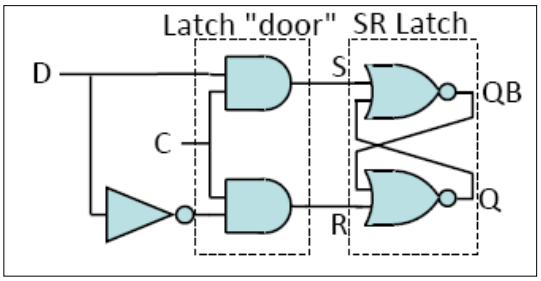
Non-ideal behavior—setup and hold time.



First, let us try for some intuition. Consider a D flip-flop. We know that Q becomes the value of D when the clock has a rising edge. But what if D is changing at the same time as that rising edge? Then it's hard to know what value we'll get on Q. So clearly we need D to remain steady for some time before and after C changes.

- We call the time period *before* the rising edge where D must remain constant the "***setup time***"
- We call the time period *after* the rising edge where D must remain constant the "***hold time***"

Looking at a flip-flop is a bit complex, so let's look at a D latch (which D flip-flops are made from). What might happen if D goes high "right before" C goes low?



- To set, S should be held at 1 for at least 2 NOR delays
  - Note that S = D *and* C
- D must change to 1 at least 2 NOR delays before C goes to 0 (closes the latch)

(Notice for the latch the setup time is before the *falling* edge. Why is that?)

As we can see if D becomes a "1" less than 2 NOR gate delays *before* clock goes to zero, we may have a problem. It may be that D gets missed.

- It may be that Q and QB go "metastable" (oscillate or, more likely, hit some value between "0" and "1" for a while.)

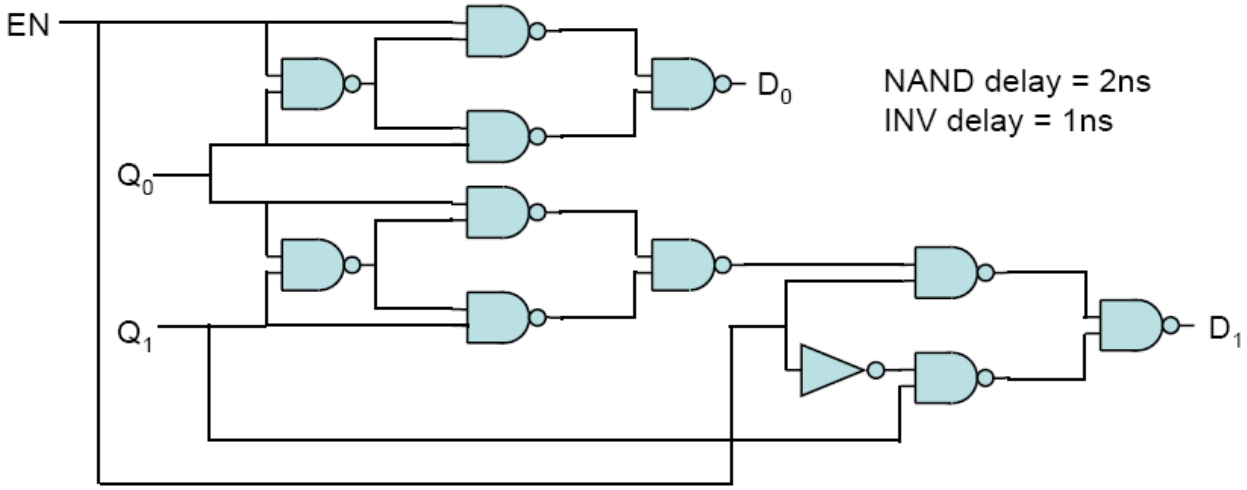
Further, we might find we have to wait even longer if D is changing from 1 to 0 as the reset path is a bit longer…

So we can see that the set-up time might be as much as the delays of 2 NORs and an inverter. The hold time might be pretty much 0 here…

[1] A significant part of this lecture taken from Dr. Karem Sakallah, one image from http://vlsi-expert.blogspot.com

Non-ideal behavior—Looking (again) at delay in combinational logic

Next, let's tackle something else: analyzing worst case (a.k.a." max" or "slow") paths and the best case (a.k.a. "min" or "fast") paths



	D_0	D_1
EN		
Q_0		
Q_1		

Min Delays

	D_0	D_1
EN		
Q_0		
Q_1		

Max Delays

Now redo if NAND delay is 1ns to 3ns and inverter is 0.5ns to 1ns

	D_0	D_1
EN		
Q_0		
Q_1		

Min Delays

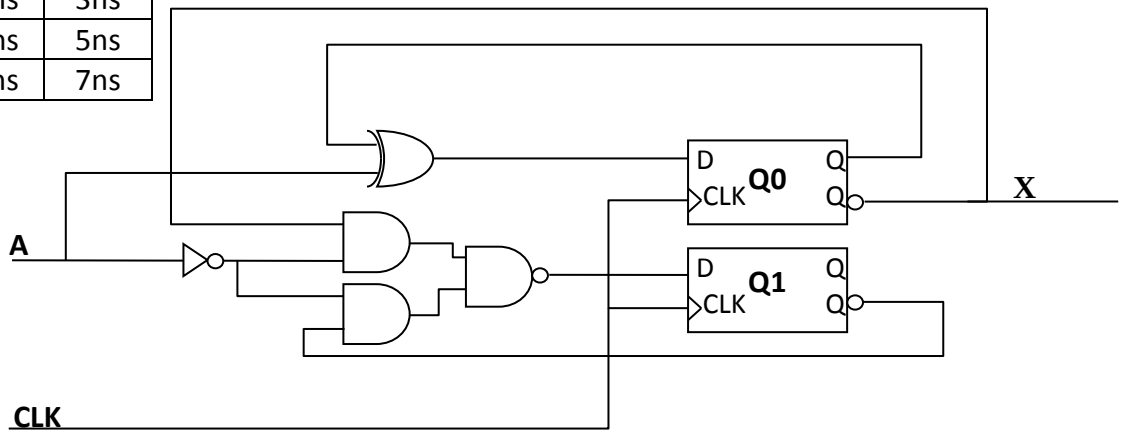
	D_0	D_1
EN		
Q_0		
Q_1		

Max Delays

Non-ideal behavior—putting it all together

Say you have the following values associated with the process you are using:

Device	Min	Max
DFF:		
Clock to Q	1ns	4ns
Set-up time	4ns	
Hold time	5ns	
OR/AND	2ns	6ns
NOT	1ns	3ns
NAND/NOR	2ns	5ns
XOR	3ns	7ns



Assume that the input A is coming from a flip-flop that has the same properties as the flip-flops that are shown and is clocked by the same clock.

a. Add inverter pairs as needed to the above figure to avoid any "fast path" problems. Do so in a way that has least impact on the worst-case delay (as a first priority) and which keeps the number of inverter pairs needed to a minimum (as a second priority).
b. *After y*ou've made your changes in part a, compute the maximum ***frequency*** at which this device can be safely clocked.

Verilog

Everything we've done so far in lab has been combinational logic using assign statements. In lab 4 we'll need a way to implement sequential logic AND we'll want something a bit more flexible for combinational logic.

Verilog is a very powerful language with a large mess of complexity that can be gotten into. In EECS 270 we greatly restrict the form of the language you can use. We do this partly for pedagogical reasons, but also to restrict the number of ways you can shoot yourself in the foot. Take EECS 470 if you want to learn more about Verilog (but even it is probably only the equivalent of EECS 280 for C++).

The "always @" statement

One way to model sequential logic is to say "only reevaluate this logic at certain times". The always @ statement does exactly that.

```
always @ (<sensitivity list>)
begin
    <body of block>
end
```

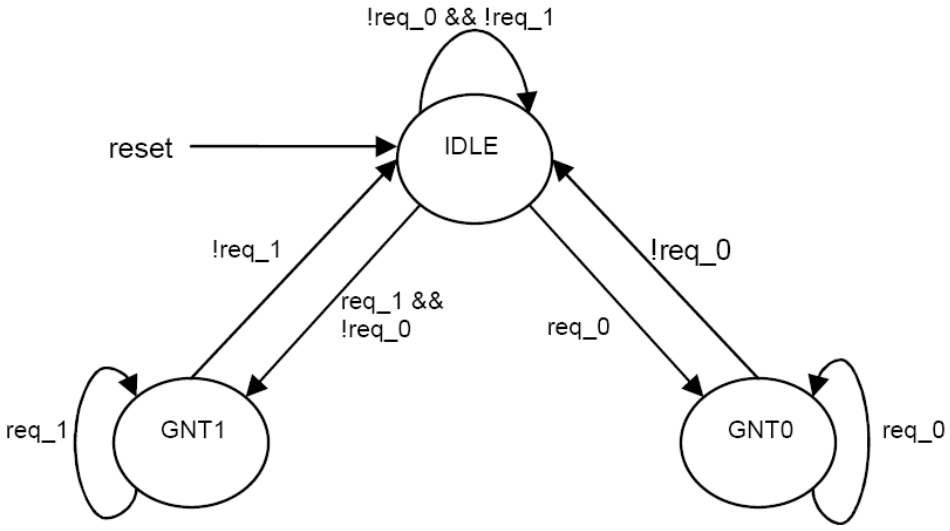
We always will use "@(posedge clock)" for sequential logic and "@*" for combinational logic. There are other options. See the section on sensitivity lists (below).

As the above figure indicates, we use only two forms of this. One is "evaluate only on the positive edge of the clock" and the other is "constantly reevaluate". That last one is another way to do combinational logic. Why do we need two ways of doing combinational logic? Well, always @* blocks are much more flexible but much easier to mess up. So my rule is to use an assign statement when you can.

The begin and end here are just like { and } in C++.

Example

The following is take from a lab document which builds the following FSM in Verilog.



In our @* block we can use case statements, if statements and a few other things. That makes it all a LOT more flexible. That said, it's really just asking Verilog to do the work. Let's consider this encoding (one-hot) with state and next_state having 3 bits.

```
parameter   IDLE=3'b001;
parameter   GNT0=3'b010;
parameter   GNT1=3'b100;

reg [2:0] state;
reg [2:0] next_state;
```

Next state logic via always @* block

```
always @*
begin
 case(state)
 IDLE :    if (req_0 == 1'b1)
               next_state = GNT0;
           else if (req_1 == 1'b1)
               next_state= GNT1;
           else
               next_state = IDLE;
 GNT0 :    if (req_0 == 1'b1)
               next_state = GNT0;
           else
               next_state = IDLE;
 GNT1 :    if (req_1 == 1'b1)
               next_state = GNT1;
           else
               next_state = IDLE;
 default:  next_state = IDLE;
 endcase
end
```

Next state logic via assign statements?

```
assign next_state[0] = (~req_0 & ~req_1 & state[0]) |
                       (~req_0 & state[1]) | (~req_1 & state[2]);
assign next_state[1] = req_0 & (state[0] | state[1]);
assign next_state[2] = req_1 & (state[0] | state[2]);
```

What are the pros and cons of each scheme?

Sequential logic in Verilog

We say "always @ (posedge clock)" to build something that is only updated at the positive edge of the clock (like a flip-flop). The following implements our 3 flip-flops.

```
always @ (posedge clock)        // Sequential logic-implements flip-flops (with
begin                           // reset) to store value of current state; reset
  if (reset == 1'b1)            // returns machine to initial state
       state <= IDLE;
  else
       state <= next_state;
end          // end of always @ (posedge clock)
```

Notice how we've created a reset state. Also notice that we use <= to assign in a sequential logic block.

Final comments:

- If we assign to something with an assign statement, it needs to be a "wire".
- If we assign to something in an always block it needs to be a "reg".
- If you assign to a variable in an always@* block you must be sure you assign to it on all paths. (why do you think that is?)
- You must never assign a value to the same variable in more than one block.
    - You may however assign to it more than once in a *given* block. The last one wins.

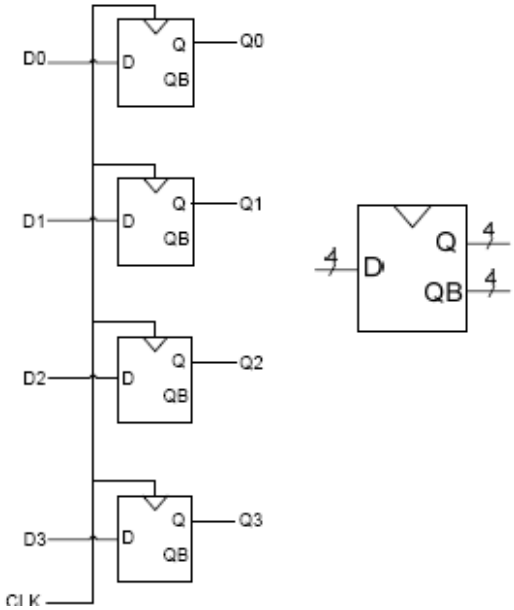
Appendix A: Complete code example

```
module arbiter (clock, reset, req_0, req_1, gnt_0, gnt_1);

input clock, reset, req_0, req_1;       // Input declarations
output      gnt_0, gnt_1;               // Output declarations
reg         gnt_0, gnt_1;

parameter  IDLE=3'b001;         // State definitions
parameter  GNT0=3'b010;         // This example uses a one-hot encoding for its three
parameter  GNT1=3'b100;         // states.  You can use a different encoding scheme.

reg [2:0] state;                // Sequential variable to store the current state
reg [2:0] next_state;           // Combinational variable used to calculate the next state

always @*                       // Combinational logic block
begin
  case(state)
  IDLE :    if (req_0 == 1'b1)          // Implements transitions in state diagram
                next_state = GNT0;      // (see Fig. 2 for state diagram)
            else if (req_1 == 1'b1)
            next_state= GNT1;
            else
            next_state = IDLE;
  GNT0 :    if (req_0 == 1'b1)
            next_state = GNT0;
            else
            next_state = IDLE;
  GNT1 :    if (req_1 == 1'b1)
                next_state = GNT1;
            else
                next_state = IDLE;
  default:  next_state = IDLE;
  endcase
end         // end of always @*

always @ (posedge clock)        // Sequential logic-implements flip-flops (with
begin                           // reset) to store value of current state; reset
  if (reset == 1'b1)            // returns machine to initial state
      state <= IDLE;
  else
      state <= next_state;
end         // end of always @ (posedge clock)

always @*       // Output logic-determines outputs from current state
begin
  case(state)
  IDLE :    begin
                gnt_0 = 1'b0;
                gnt_1 = 1'b0;
            end
  GNT0 :    begin
                gnt_0 = 1'b1;
                gnt_1 = 1'b0;
           end
  GNT1 :    begin
                gnt_0 = 1'b0;
                gnt_1 = 1'b1;
           end
  default : begin
             gnt_0 = 1'b0;
                gnt_1 = 1'b0;
             end
  endcase
end         // end of always @ (state)
endmodule
```

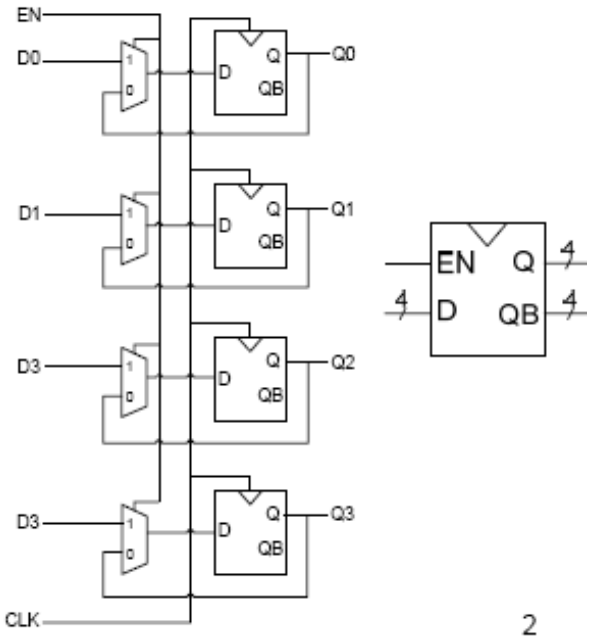
Sequential MSI devices.

Registers are a collection of 2 or more D flip-flops working in parallel.

- A collection of two or more D flip-flops with a common clock is called a **register**
- Used to store a collection of bits



- Adding a mux at the input of each D FF allows for loading new value or storing current value



2

Counters

A simple counter is a device that increments its value every clock cycle. (So 0, 1, 2, etc.). It generally has an enable (to count or stay where it is) and a reset (to go back to zero on the next rising edge.

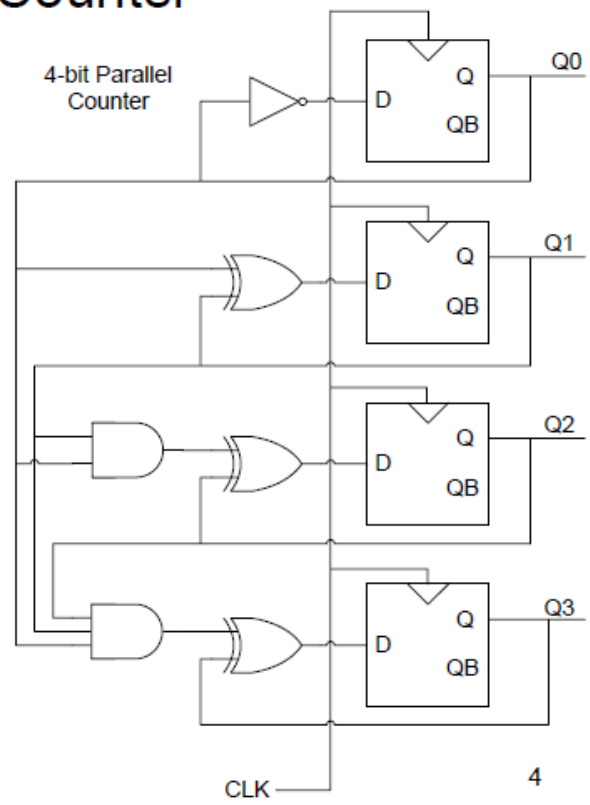
1. Design a 4-bit (count 0 to 15) counter using standard MSI devices.
2. Design a “count from 0 to 9” counter using your 4-bit counter.

Parallel Counter

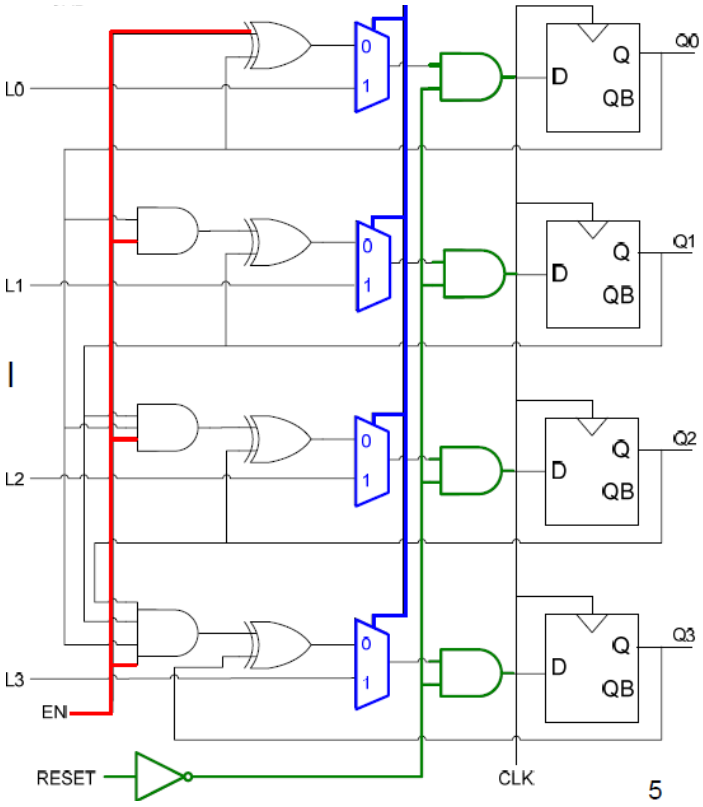
- Would like *all* state variables to run on the same clock signal
- Examining the binary code yields an easy implementation:

b_2	b_1	b_0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1
0	0	0

- Under what condition does each bit toggle?
- *When all bits of lesser significance are 1!*
- This is true no matter how many bits in the counter



What does this one do?



Let's review what a T Flip-flop does. Can you build a counter out of T flip-flops?

What are the problems with this "ripple" counter? What's nice about it?

Terminology

- **Counters** rotate through a set of values. Most count from 0 to N.
  - Some can count both up and down (generally called **up/down** counters)
  - Most have **reset** and **enable**.
  - One question is what to do with a counter that reaches the maximum (or minimum) value.
    - **Saturating** counters stay at that value
    - **Modulo** counters wrap around.
  - Question: Say you are given a 5-bit, saturating up counter with synchronous reset:
    - What numbers does this count through? (0 to what?)
    - Use that counter and gates to design a modulo-6 counter (counts 0 to 5 then wraps around) with synchronous reset.
  - We can have **ripple** or **parallel** counters.
    - Ripple is simple, but has some issues.
    - Parallel is more complex, but better behaved.

- **Shift registers** move the data one way or the other.
  - We can have shift registers that shift either way, load or reset all with the aid of counters.
  - Let's design a 4-bit shift register that takes two inputs X[1:0]. If X=3 we shift left, if X=2 we shift right, if X=1 we hold our value, and behavior is undefined if X=0.