

6.205

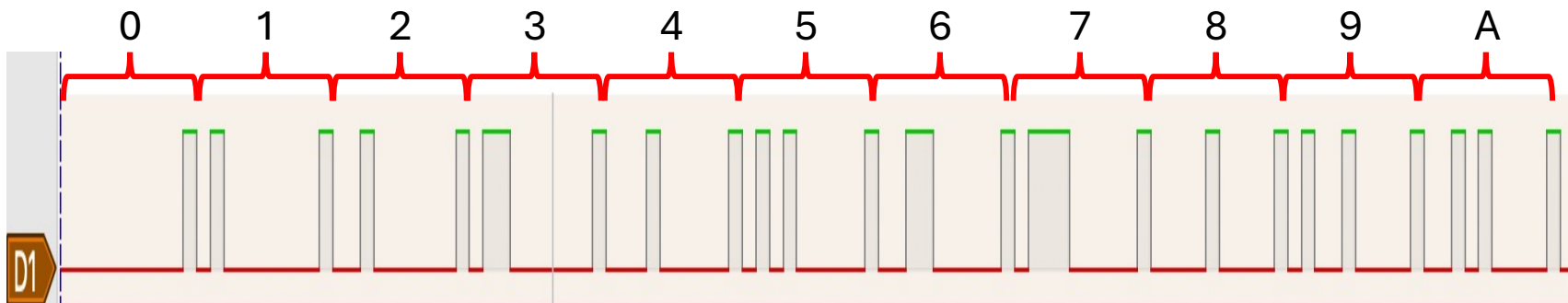
FSMs (finish), Clocking, Timing #4,

Admin

- Week 3 due tomorrow
- Week 4 out Thursday
- Hudson River Trading FPGA folks are here right after class on Thursday:
 - Presentation 4-5 (one of whom former 6.205 TA Driss)
 - Food
 - Q&A 1 on 1 (I guess?)

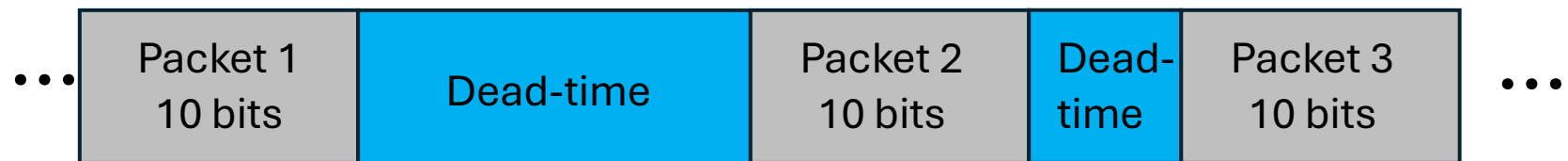
Notes on UART RX

- From the computer I decided to send down 0, then 1, then 2, then 3...to 255 over UART.
- This was the trace on the UART_RXD line



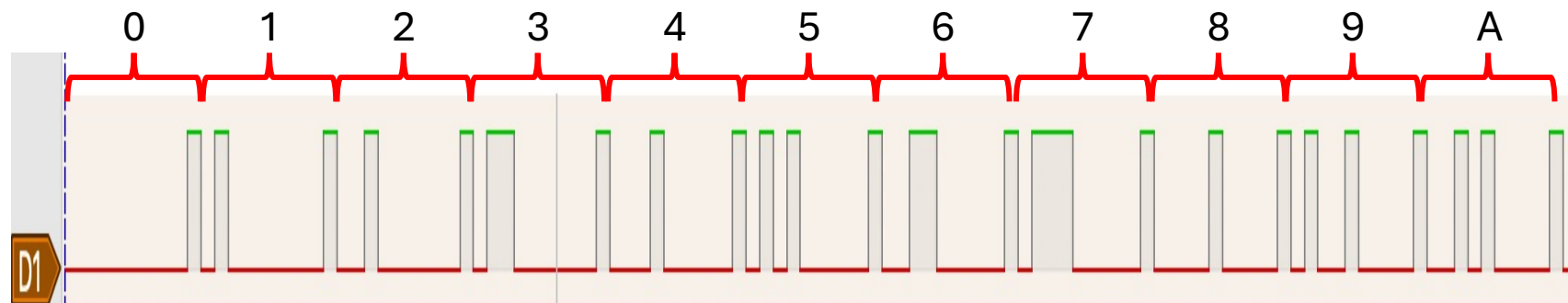
UART Packet

- In the UART standard there is *no guarantee* in regards to inter-byte spacing between bytes sent down.
- It can vary (and often does)



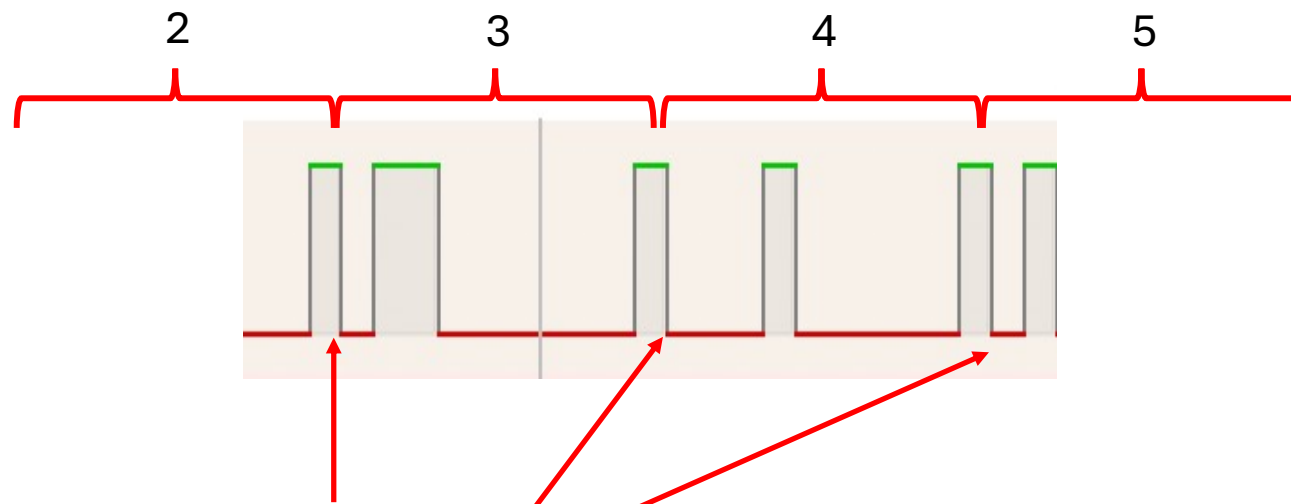
However...the FT2232 chip on our board that handles the UART...

- Packs the UART packets very tightly



However...the FT2232 chip on our board that handles the UART...

- Packs the UART packets very tightly



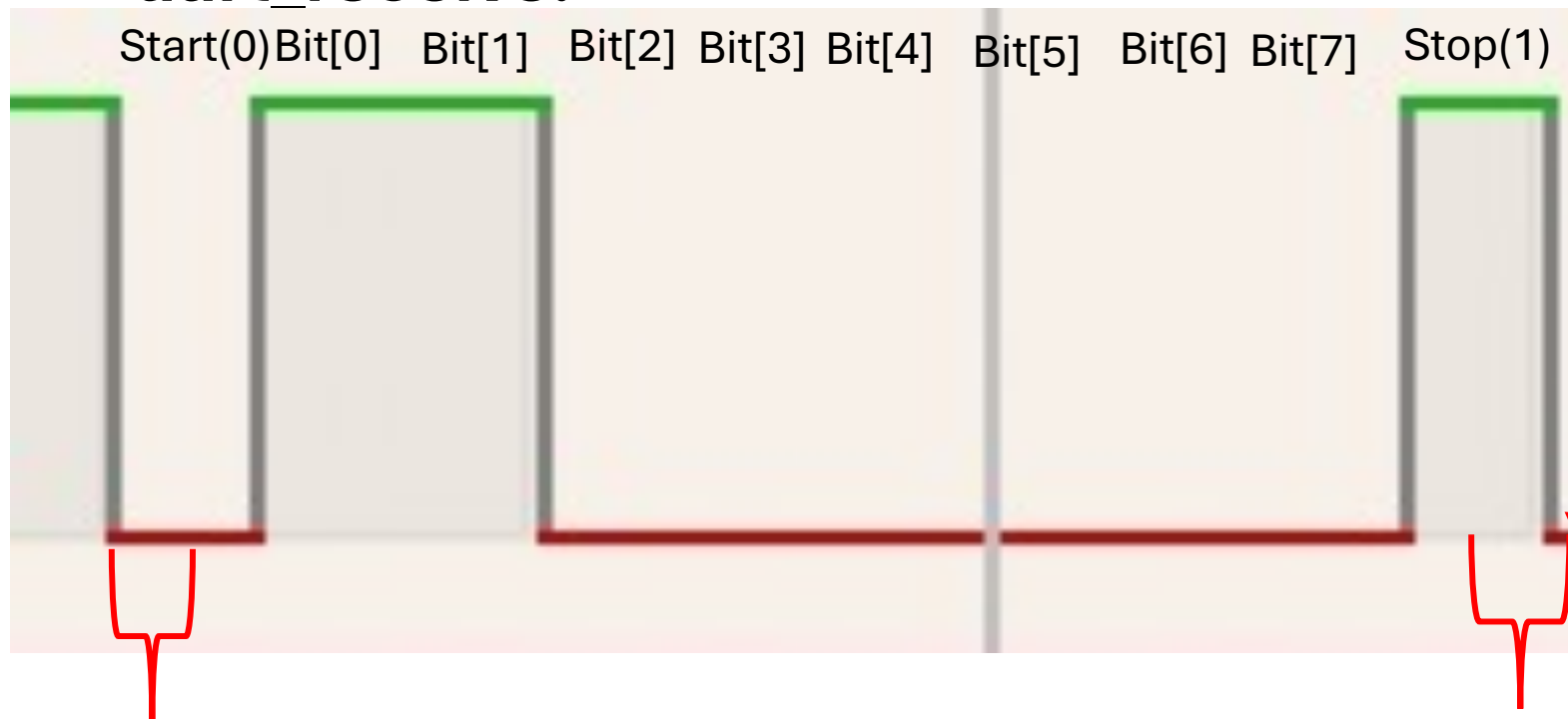
Basically no inter-byte spacing

You get a full baud period for the stop bit and then it immediately goes into the next packet

As a result...

Could fall off cliff into next start bit... :/

- Be careful with verifying too long or staying in a state too long! **Read instructions for `uart_receive`!**



Verify low for half BAUD

Verify high from 0.5BAUD to 1 BAUD

Finite State Machines...

Real Quick...

Moore



- Edward F. Moore
 - 1925-2003
 - Virginia Tech
 - Worked with Claude Shannon
-
- Not same Moore as Moore's Law...that was Gordon Moore from Intel

Mealy

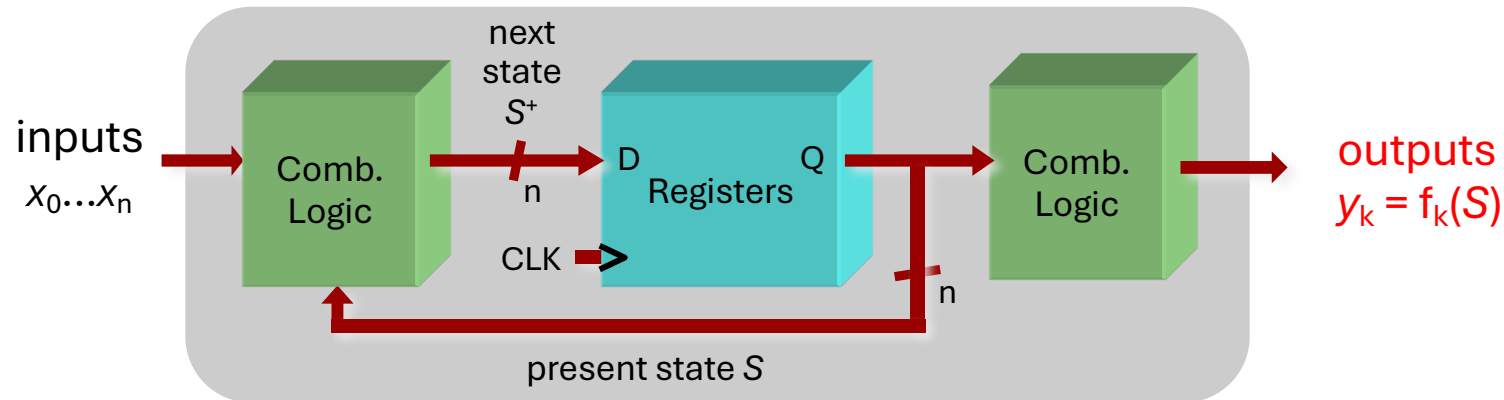


- George H. Mealy
- 1927-2010
- Harvard, Bell Labs

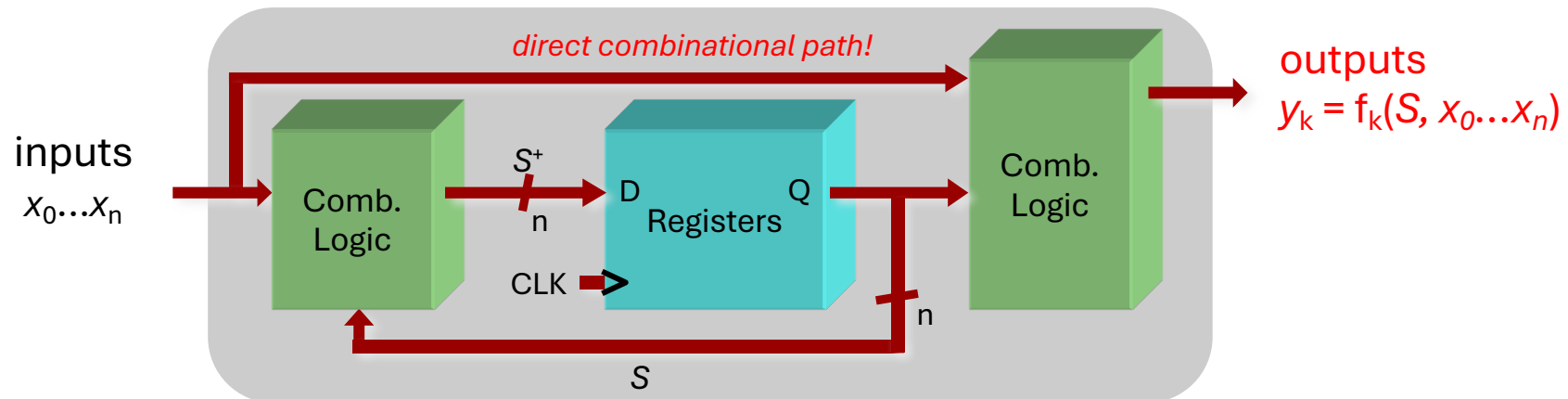
So Review: Two Types of FSMs

Moore and **Mealy** FSMs : different output generation

- **Moore FSM:**

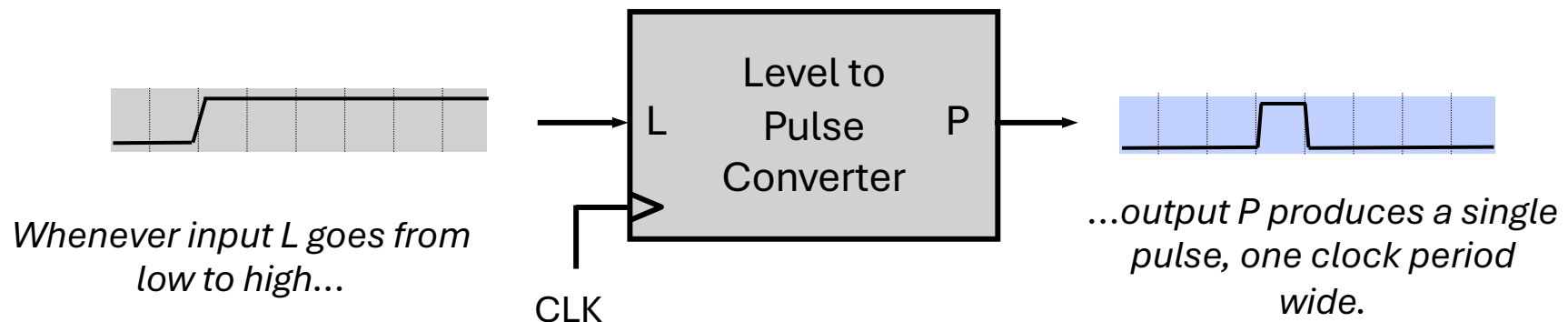


- **Mealy FSM:**

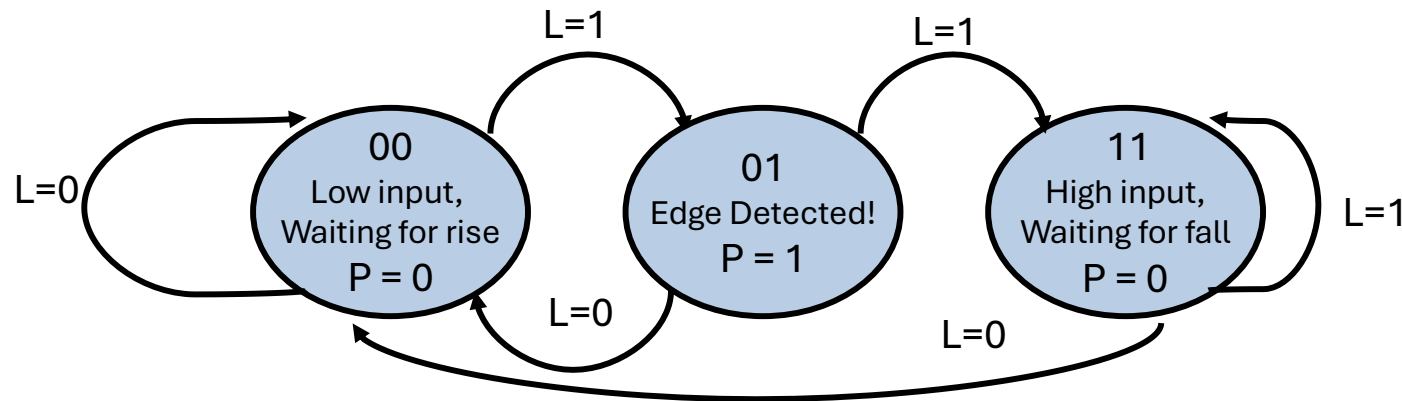


Design Example: Level-to-Pulse

- A **level-to-pulse converter** produces a single-cycle pulse each time its input goes high.
- It's a synchronous rising-edge detector.
- Sample uses:
 - Buttons and switches pressed by humans for arbitrary periods of time
 - Single-cycle enable signals for modules



Moore Valid State Transition Diagrams



- Arcs leaving a state are **mutually exclusive**, i.e., for any combination input values there's at most one applicable arc
- Arcs leaving a state are **collectively exhaustive**, i.e., for any combination of input values there's at least one applicable arc**
- So for each state: for any combination of input values there's exactly one applicable arc (**no ambiguity**)
- Often a starting state is specified
- Each state specifies values for all outputs (in the case of Moore)

Moore Level-to-Pulse Converter (SystemVerilog)

- An example of a **very explicit** Moore FSM implementation of the level-to-pulse converter:

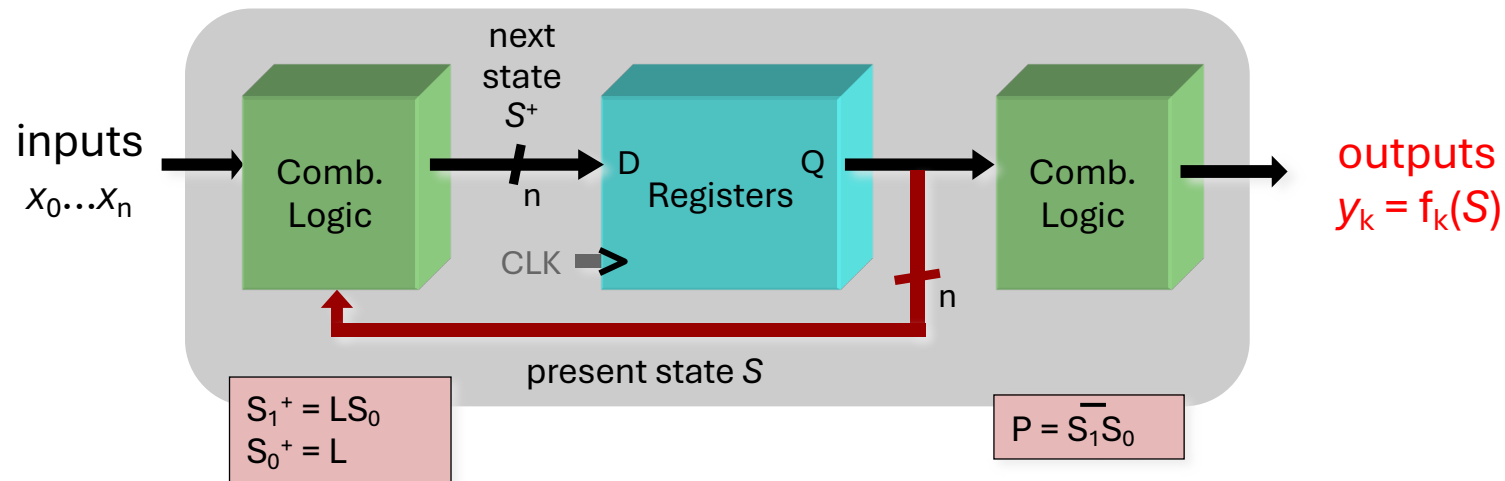
```
module moore_fsm( input wire clk,
                  input wire l_in,
                  output logic p_out);
    localparam LOW_WAITING = 2'b0; //define your states as...
    localparam EDGE_DETECTED = 2'b01; //parameters for easy...
    localparam HIGH_WAITING = 2'b10; //reading!
    logic [1:0] state; //contain state!
    logic [1:0] next_state; //hold next state!
    //Output Logic:
    always_comb begin
        case(state)
            LOW_WAITING: p_out = 1'b0; //output based only on...
            EDGE_DETECTED: p_out = 1'b1; //current state! This is...
            HIGH_WAITING: p_out = 1'b0; //a characteristic of Moore FSM
            default: p_out = 1'b0;
        endcase
    end
    //State Transition Logic (Combinational):
    always_comb begin
        case(state) //Also consider explicit if/elses
            LOW_WAITING: next_state = l_in?EDGE_DETECTED:LOW_WAITING;
            EDGE_DETECTED: next_state = l_in?HIGH_WAITING:LOW_WAITING;
            HIGH_WAITING: next_state = l_in?HIGH_WAITING:LOW_WAITING;
            default: next_state = LOW_WAITING;
        endcase
    end
    //State Transition
    always_ff @(posedge clk) begin
        //consider adding a reset here as well!
        state <= next_state; //state becomes calculated next_state
    end
endmodule
```

Moore Level-to-Pulse Converter (SystemVerilog)

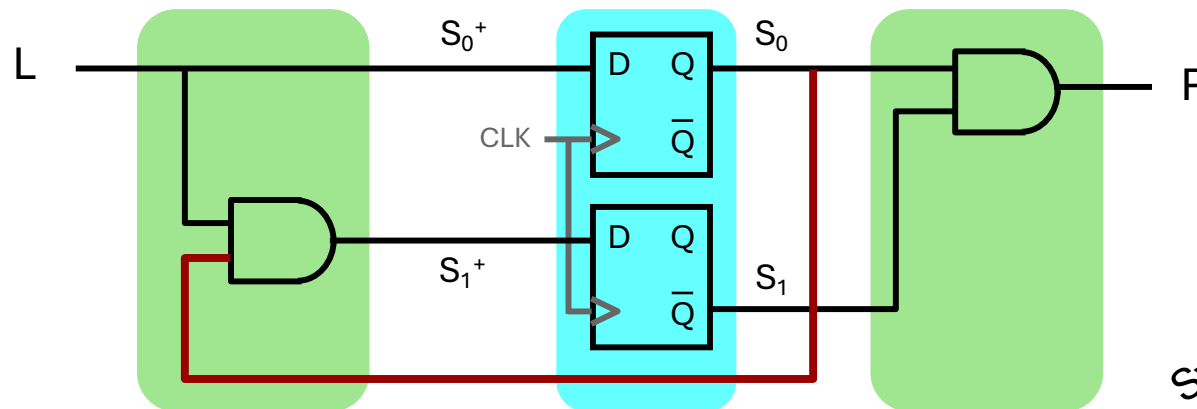
- Merging **State Transition Logic** and **State Transition** into one block
- Some people like this more

```
module moore_fsm( input wire clk,
                  input wire l_in,
                  output logic p_out);
    localparam LOW_WAITING = 2'b0;
    localparam EDGE_DETECTED = 2'b01;
    localparam HIGH_WAITING = 2'b10;
    logic [1:0] state;
    //Output Logic:
    always_comb begin
        case(state)
            LOW_WAITING: p_out = 1'b0;
            EDGE_DETECTED: p_out = 1'b1;
            HIGH_WAITING: p_out = 1'b0;
            default: p_out = 1'b0; //default
        endcase
    end
    //State Transition and Logic:
    always_ff @(posedge clk) begin
        //consider adding a reset here as well!
        case(state)
            LOW_WAITING: state <= l_in?EDGE_DETECTED:LOW_WAITING;
            EDGE_DETECTED: state <= l_in?HIGH_WAITING:LOW_WAITING;
            HIGH_WAITING: state <= l_in?HIGH_WAITING:LOW_WAITING;
            default: state <= LOW_WAITING;
        endcase
    end
endmodule
```

Moore Level-to-Pulse Converter



Moore FSM circuit implementation of level-to-pulse converter:

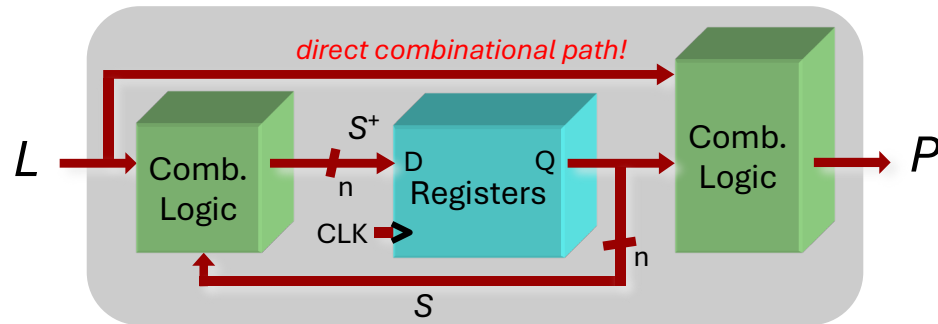


Synthesizable with
two LUT2s and two
flip flops

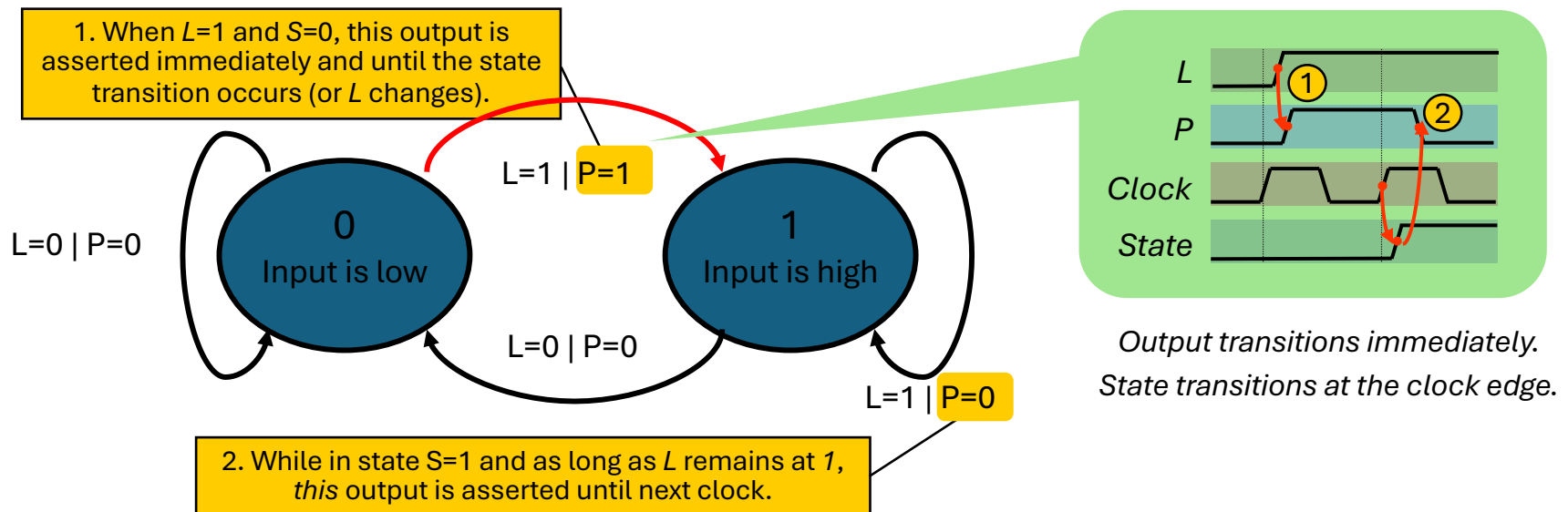
Moore is One Way

- There is another...

Design of a Mealy Level-to-Pulse



- Since outputs are determined by state *and* inputs, Mealy FSMs generally need fewer states than Moore FSM implementations



Mealy Level-to-Pulse Converter (SystemVerilog)

- An example of a **very explicit** Mealy FSM implementation of the level-to-pulse converter:

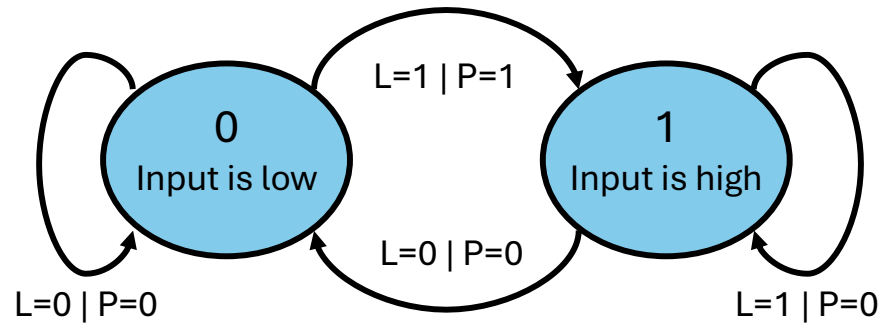
```
module mealy_fsm( input wire clk,
                  input wire l_in,
                  output logic p_out);
    localparam LOW_WAITING = 1'b0; //define states but notice...
    localparam HIGH_WAITING = 1'b1; //fewer needed...Mealy usually...
    //though not always, is like that
    logic state; //state (smaller than before...only two states to rep)
    logic next_state;
    //Output Logic:
    always_comb begin
        case(state) //outputs are based on state AND inputs!
            LOW_WAITING: p_out = l_in?1'b1:1'b0;
            HIGH_WAITING: p_out = 1'b0;
            default: p_out = 1'b0; //default
        endcase
    end
    //State Transition Logic:
    always_comb begin
        case(state)
            LOW_WAITING: next_state = l_in?HIGH_WAITING:LOW_WAITING;
            HIGH_WAITING: next_state = l_in?HIGH_WAITING:LOW_WAITING;
            default: next_state = LOW_WAITING;
        endcase
    end
    //State Transition
    always_ff @(posedge clk) begin
        //consider adding a reset here as well (same goes for any...
        //clocked logic block)
        state <= next_state;
    end
endmodule
```

Mealy Level-to-Pulse Converter (SystemVerilog)

- Merging **State Transition Logic** and **State Transition** into one block

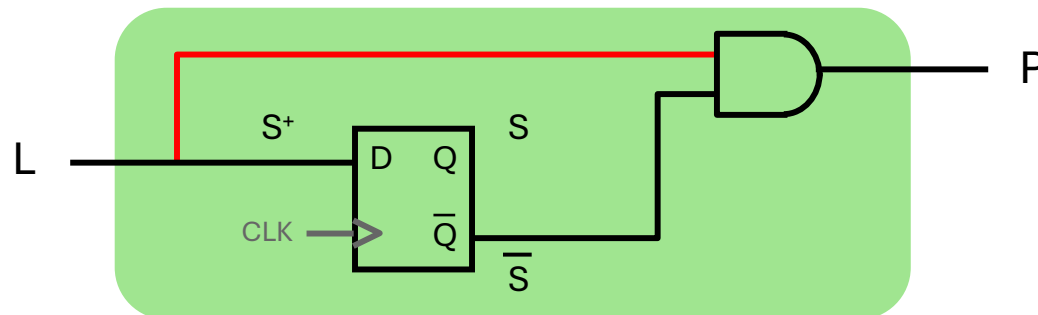
```
module mealy_fsm( input wire clk,
                  input wire l_in,
                  output logic p_out);
    localparam LOW_WAITING = 1'b0;
    localparam HIGH_WAITING = 1'b1;
    logic state;
    //Output Logic:
    always_comb begin
        case(state)
            LOW_WAITING: p_out = l_in?1'b1:1'b0;
            HIGH_WAITING: p_out = 1'b0;
            default: p_out = 1'b0; //default
        endcase
    end
    //State Transition and Transition Logic!
    always_ff @(posedge clk) begin
        //consider adding a reset here as well!
        case(state)
            LOW_WAITING: state <= l_in?HIGH_WAITING:LOW_WAITING;
            HIGH_WAITING: state <= l_in?HIGH_WAITING:LOW_WAITING;
            default: state <= LOW_WAITING;
        endcase
    end
endmodule
```

Mealy Level-to-Pulse Converter



*Synthesizable with
one LUT2 and one
flip flops*

Mealy FSM circuit implementation of level-to-pulse converter:

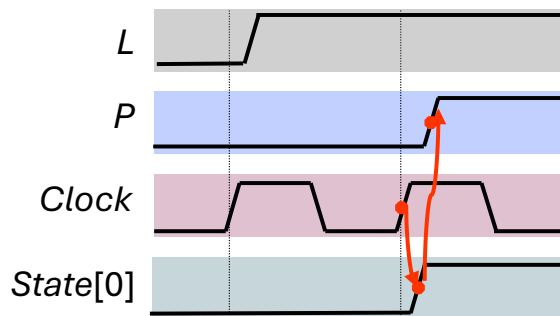


- FSM's state simply remembers the previous value of L
- Circuit benefits from the Mealy FSM's implicit single-cycle assertion of outputs during state transitions

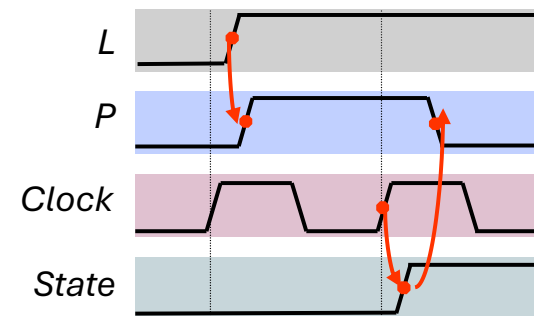
Moore/Mealy Trade-Offs

- How are they different?
 - Moore: $\text{outputs} = f(\text{state})$ only
 - Mealy $\text{outputs} = f(\text{state and input})$
 - Mealy outputs generally occur one cycle earlier than a Moore:

Moore: delayed assertion of P



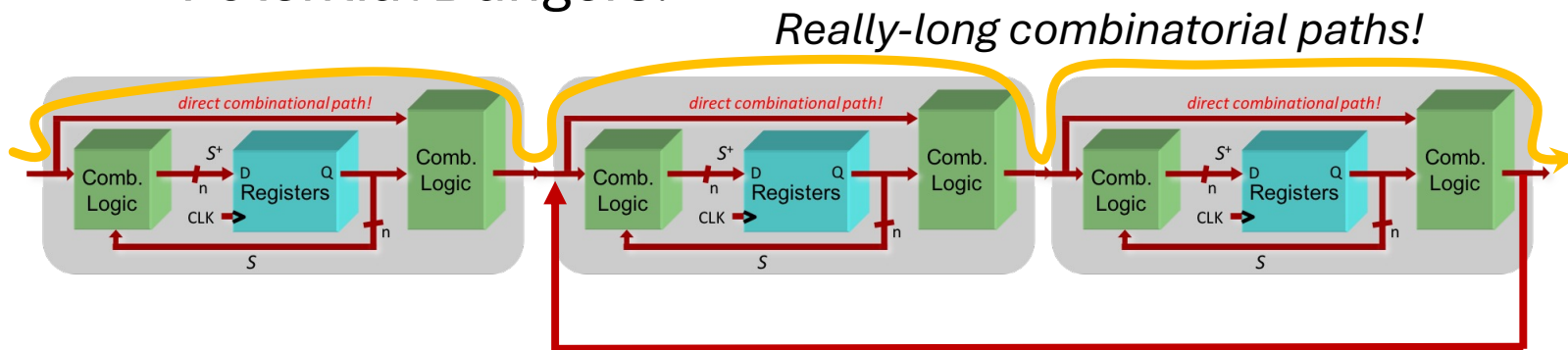
Mealy: immediate assertion of P



- Compared to a Moore FSM, a Mealy FSM ***might...***
 - Be more difficult to conceptualize and design (both at circuit level and in HDL)
 - Have fewer states
 - Be expressed using fewer lines of Verilog

Moore/Mealy Trade-Offs

- Moore:
 - Usually more states
 - Each state has a particular output
- Mealy:
 - Fewer states, outputs are specified on edges of diagram
 - Potential Dangers:



*Possible cyclic logic paths (combinational loops!!!)
Combinatorial logic driving itself asynchronously through
really hard-to-debug pathways!*

Enums

- Another good practice to prevent against accidental doubling of state variables

```
module mealy_fsm( input wire clk,
                  input wire l_in,
                  output logic p_out);
    localparam LOW_WAITING = 1'b1; //define states but notice...
    localparam HIGH_WAITING = 1'b1; //fewer needed...Mealy usually...
    //though not always, is like that
    logic state; //state (smaller than before...only two states to rep)
    logic next_state;
    //Output Logic:
    always_comb begin
        case(state) //outputs are based on state AND inputs!
            LOW_WAITING: p_out = l_in?1'b1:1'b0;
            HIGH_WAITING: p_out = 1'b0;
            default: p_out = 1'b0; //default
        endcase
    end
end
```

shooooooooottt

Enums

- Enums allow you to define variables that only take on specific values *and* which will guarantee those values will be unique

```
module mealy_fsm( input wire clk,
                  input wire l_in,
                  output logic p_out);
  typedef enum logic {LOW_WAITING, HIGH_WAITING} state_t; //define states but notice...
  //fewer needed...Mealy usually...though not always, is like that
  state_t state; //state (smaller than before...only two states to rep)
  state_t next_state;
  //Output Logic:
  always_comb begin
    case(state) //outputs are based on state AND inputs!
      LOW_WAITING: p_out = l_in?1'b1:1'b0;
      HIGH_WAITING: p_out = 1'b0;
      default: p_out = 1'b0; //default
    endcase
  end
```

Encoding with Enums?

- Generally in SystemVerilog, an enum, unless specified will be 0, 1, 2, 3, etc...

```
enum {IDLE,GOT_5c,GOT_10c,GOT_15c,GOT_20c,  
      GOT_25c,GOT_35c,GOT_40c,GOT_45c,  
      GOT_50c,RETURN_20c,RETURN_15c,  
      RETURN_10c,RETURN_5c } state, next;
```

- When synthesizing, Vivado may decide on a different encoding, however if it is advantageous to the design and/or allows logical optimization

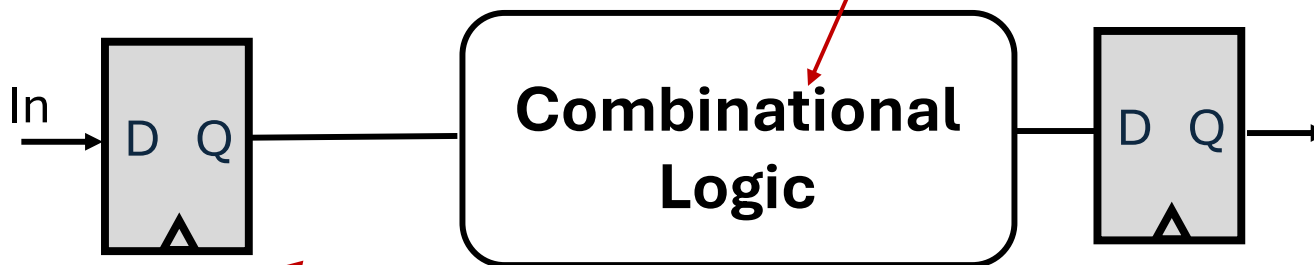
```
typedef enum {IDLE,GOT_5c,GOT_10c,GOT_15c,GOT_20c,  
             GOT_25c,GOT_35c,GOT_40c,GOT_45c,  
             GOT_50c,RETURN_20c,RETURN_15c,  
             RETURN_10c,RETURN_5c } state_enc_T;  
  
state_enc_T state, next;
```

Why Design with FSMs?

- Most stuff is an FSM anyways, whether you like it or not...
- The FSM structure is extremely amenable to digital design, specifically FPGAs.
- Knowing a variable is “internal” like a state variable can allow Vivado to optimize things better!
- In FPGAs, Flipflops are “cheap” whereas LUTs can tend to be more precious
 - What this means is that as you design more and more, the amount of combinational logic you’ll try to use will scale faster than the amount of registers as we try to max out timing.
 - The number of LUTs and FF’s are fixed on an FPGA

- We try to add more and more logic between two fixed sets of registers...so the relative usage will grow differently if left unchecked...

Added thicker and thicker logic right up to slack limits using more LUTs in process



1. Slice Logic

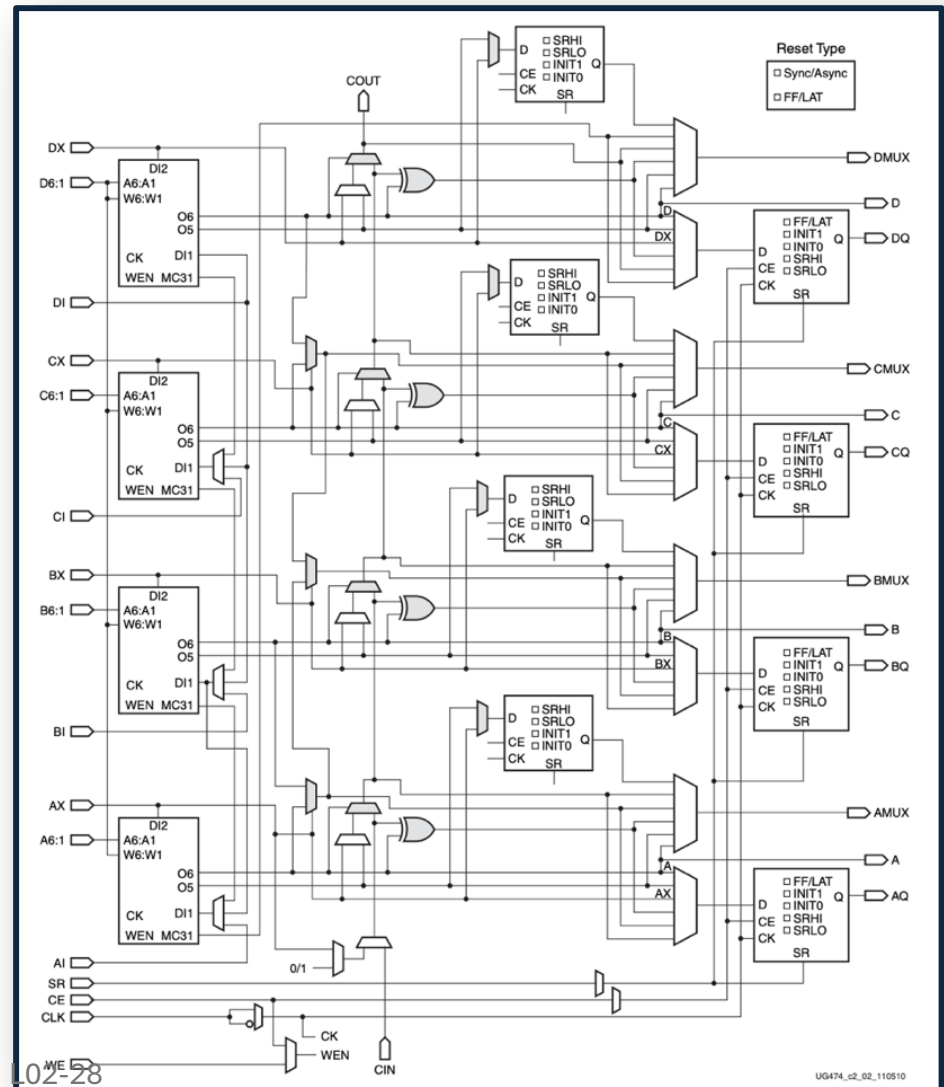
Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	2144	0	0	20800	10.31
LUT as Logic	2094	0	0	20800	10.07
LUT as Memory	50	0	0	9600	0.52
LUT as Distributed RAM	50	0	0		
LUT as Shift Register	0	0	0		
Slice Registers	1931	0	0	41600	4.64
Register as Flip Flop	1931	0	0	41600	4.64
Register as Latch	0	0	0	41600	0.00
F7 Muxes	0	0	0	16300	0.00
F8 Muxes	0	0	0	8150	0.00

Still just using two sets

* Warning! LUT value is adjusted to account for LUT combining.

Xilinx Logic Blocks

- Our FPGAs have about 8500 of these →
- Called “Logic Slices”
- Each slice has four CLBs (“**C**onfigurable **L**ogic **B**locks”) that form the clay from which we sculpt our digital functions



- If Vivado can identify the logic as a FSM, it will help in optimizing logic usage.

```

module moore_fsm( input wire clk,
                  input wire l_in,
                  output logic p_out);
  typedef enum {LOW_WAITING, EDGE_DETECTED, HIGH_WAITING} state_t;
  state_t state; //state (smaller than before...only two states to rep)
  //Output Logic:
  always_comb begin
    case(state)
      LOW_WAITING: p_out = 1'b0;
      EDGE_DETECTED: p_out = 1'b1;
      HIGH_WAITING: p_out = 1'b0;
      default: p_out = 1'b0; //default
    endcase
  end
  //State Transition and Logic:
  always_ff @(posedge clk) begin
    //consider adding a reset here as well!
    case(state)
      LOW_WAITING: state <= l_in?EDGE_DETECTED:LOW_WAITING;
      EDGE_DETECTED: state <= l_in?HIGH_WAITING:LOW_WAITING;
      HIGH_WAITING: state <= l_in?HIGH_WAITING:LOW_WAITING;
      default: state <= LOW_WAITING;
    endcase
  end
endmodule

```

INFO: [Synth 8-802] inferred FSM for state register 'state_reg' in module 'moore_fsm'
 INFO: [Synth 8-6159] Found Keep on FSM register 'state_reg' in module 'moore_fsm', re-encoding will not be performed

State	New Encoding	Previous Encoding

* LOW_WAITING	00	00
EDGE_DETECTED	01	01
HIGH_WAITING	10	10

Vivado Identifying FSMs

- Here's a more complicated build from a side project over the summer:

```
INFO: [Synth 8-802] inferred FSM for state register 'fsm_state_reg' in module 'uart_tx'
INFO: [Synth 8-802] inferred FSM for state register 'gen_rst_ic.curr_wrst_state_reg' in module 'xpm_fifo_rst'
INFO: [Synth 8-802] inferred FSM for state register 'gen_rst_ic.curr_rrst_state_reg' in module 'xpm_fifo_rst'
INFO: [Synth 8-802] inferred FSM for state register 'gen_fwft.curr_fwft_state_reg' in module 'xpm_fifo_base'
INFO: [Synth 8-802] inferred FSM for state register 'gen_fwft.curr_fwft_state_reg' in module
'xpm_fifo_base__parameterized0'
INFO: [Synth 8-802] inferred FSM for state register 'state_reg' in module 'dram_fifo'
INFO: [Synth 8-802] inferred FSM for state register 'gen_fwft.curr_fwft_state_reg' in module
'xpm_fifo_base__parameterized1'
INFO: [Synth 8-802] inferred FSM for state register 'state_reg' in module 'bit_unpacker'
INFO: [Synth 8-802] inferred FSM for state register 'gen_fwft.curr_fwft_state_reg' in module
'xpm_fifo_base__parameterized2'
INFO: [Synth 8-802] inferred FSM for state register 'usb_state_reg' in module 'top_level'
```

Figuring out how to encode state

- Here Vivado is changing the order that states are encoded OR using more bits to encode the state all in the pursuit of logical optimization...

State	New Encoding	Previous Encoding
FSM_IDLE	00	000
FSM_START	11	001
FSM_SEND	10	010
FSM_STOP	01	011

INFO: [Synth 8-3354] encoded FSM with state register 'fsm_state_reg' using encoding 'sequential' in module 'uart_tx'
INFO: [Synth 8-5552] Implemented safe state 'default_state' for state register 'gen_rst_ic.curr_wrst_state_reg'

State	New Encoding	Previous Encoding
* WRST_IDLE	00001	000
WRST_IN	00010	010
WRST_OUT	00100	111
WRST_EXIT	01000	110
WRST_GO2IDLE	10000	100

INFO: [Synth 8-3354] encoded FSM with state register 'gen_rst_ic.curr_wrst_state_reg' using encoding 'one-hot'
INFO: [Synth 8-5552] Implemented safe state 'default_state' for state register 'gen_rst_ic.curr_rrst_state_reg'

Encoding State?

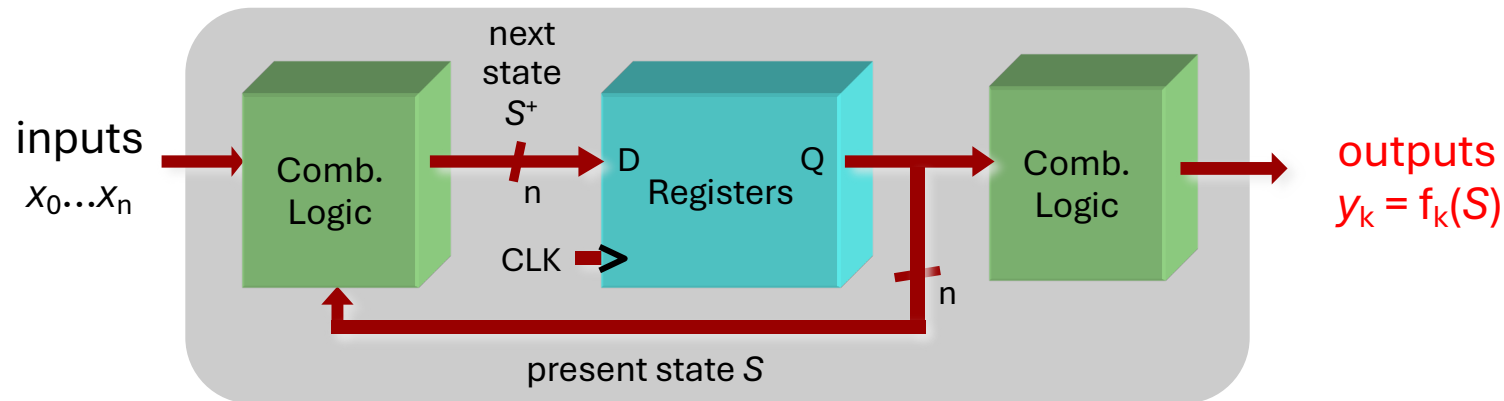
- Encoding state as sequential numbers may seem natural, but it can result in more expensive in terms of the combinational logic!!!

One-Hot Encoding

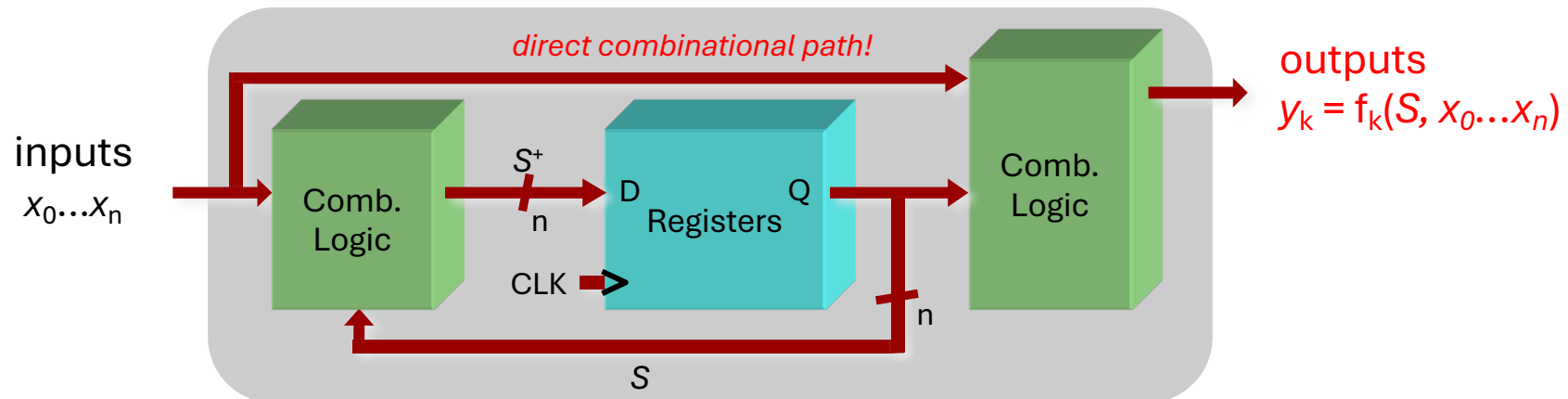
- Instead of sequential encoding (3 bits):
 - $000 \rightarrow 001 \rightarrow 010 \rightarrow 011 \rightarrow 100$ for 5 states...
- Do One-Hot encoding (5 bits):
 - $00001 \rightarrow 00010 \rightarrow 00100 \rightarrow 01000 \rightarrow 10000$ for 5 states

Remember how state is used...

- Moore FSM:



- Mealy FSM:

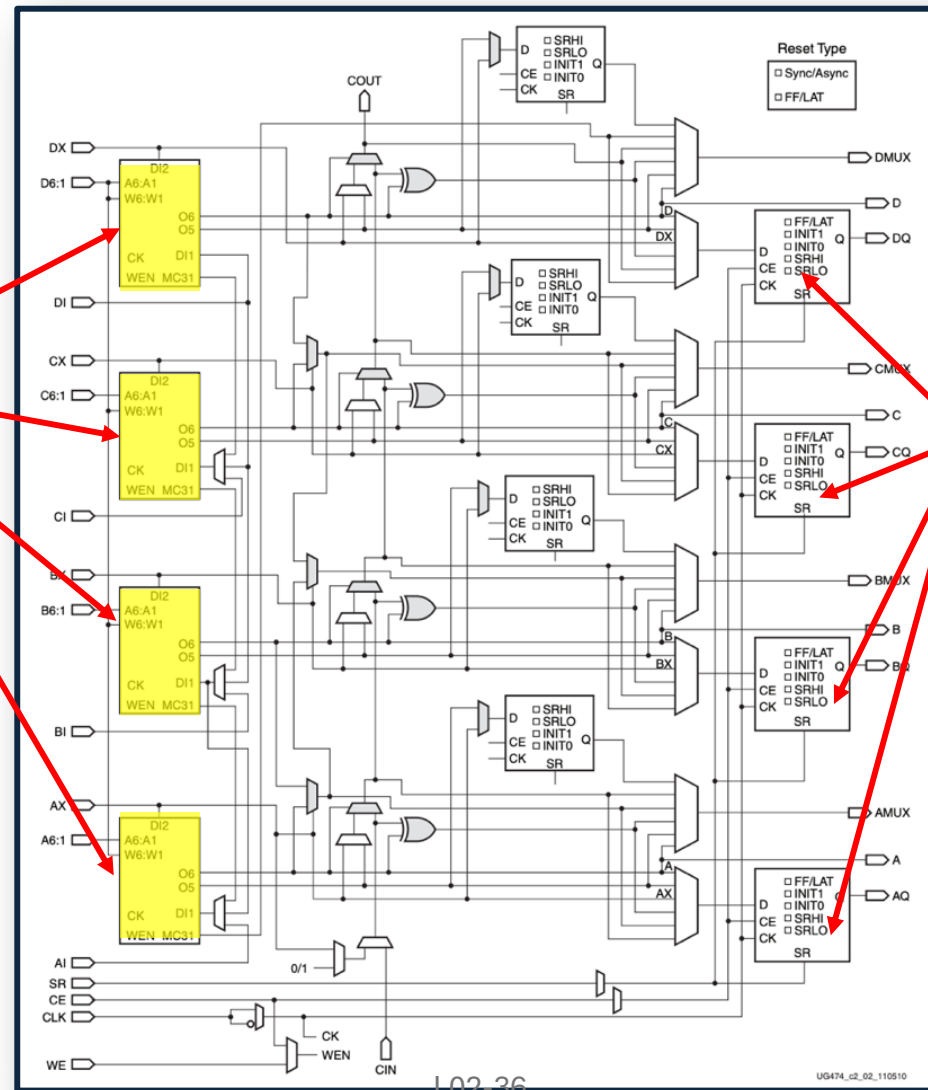


Cheaper state decoding with One-Hot

- If state is encoded sequentially...
 - $000 \rightarrow 001 \rightarrow 010 \rightarrow 011 \rightarrow 100$ for 5 states...
- To determine if you're in *state n* or not needs all three state bits!
- If state is encoded one-hot...
 - $00001 \rightarrow 00010 \rightarrow 00100 \rightarrow 01000 \rightarrow 10000$ for 5 states
- Determination if you're in *state n* or not only needs one single bit (bit_n)

Remembering our Logic Blocks!

By encoding more expensively using FF's we increase the chances that we can fit our state and output decoding logic into a single LUT6 (or fewer LUTs overall)



Use more of these things since they're just sitting around anyways

ASIC design

- Where you either add or don't add components has different economies going on when optimizing things
 - Making state have more bits/registers may not be the right choice there...but that's not for our class.

FSMs in History

We've been doing FSMs forever and will keep doing them forever...

Car Alarm FSM

- Up until 2021 the “FSM” lab in 6.111/6.205 was making a car alarm
- FSM design was and still can be a very common approach to digital circuit design



Fall 2019

[Home](#)
[Announcements](#)
[Labkit](#)
[News & DDR](#)

[Handouts](#)
[Lectures](#)
[Labs: 1, 2, 3, 4, 5A, 5B](#)

[Final Projects](#)
[Project info](#)
[Project list](#)
[Memorable projects](#)
[Past projects - all](#)
[Schedule](#)

[MIT cert required](#)
[On-line Grades](#)
[Submit PDFs](#)
[Submit Verilog](#)
[Staffed Lab Hours](#)

[Course info](#)
[Course objectives](#)
[Course calendar](#)

[Tools](#)
[Piazza \(new tab\)](#)

Description of Anti-Theft System

Since your client is completely focused on her start-up, she wants an anti-theft system that's highly automated. The system is armed automatically after she turns off the ignition and both the driver's door and passenger's doors are open, the system arms itself after all the doors have been closed and T_ARM_DELAY has passed; that delay is T_ARM_DELAY.

Once the system has been armed, opening the driver's door the system begins a countdown. If the ignition is not turned on within the countdown interval (T_DRIVER_DELAY) (T_ALARM_ON) after the door closes, at which time the system resets to the armed but silent state. If the ignition is turned on within the countdown interval, the system disarms itself.

Always a paragon of politeness, your client opens the passenger door first if she's transporting a guest. When the passenger door is opened first, a separate, presumably longer, countdown interval (T_PASSENGER_DELAY) begins. If the driver's door is opened first, a separate, presumably longer, countdown interval (T_DRIVER_DELAY) begins. If the driver's door is opened first, a separate, presumably longer, countdown interval (T_DRIVER_DELAY) begins.

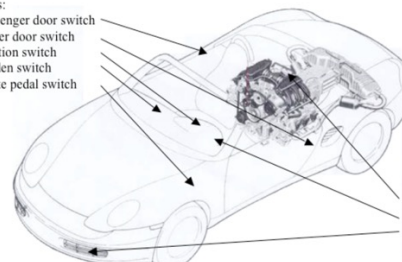
There is a status indicator LED on the dash. It blinks with a two-second period when the system is armed. It is constantly illuminated either the system is in the countdown interval or the system is armed.

So far this all is ordinary alarm functionality. But you're worried that a knowledgeable thief might disable the siren and then just drive off with the car. So you've added an override switch. Power is only restored when first the ignition is turned on and then the driver presses both a hidden switch and the brake pedal simultaneously. Power is then latched on until the driver's door is opened.

The diagram below lists all the sensors (inputs) and actuators (outputs) connected to the system.

Inputs:

- passenger door switch
- driver door switch
- ignition switch
- hidden switch
- brake pedal switch



Outputs:

- fuel pump power
- status indicator
- siren

Figure 1: System diagram showing sensors (inputs) and actuators (outputs)

The system timings are based on four parameters (in seconds): the delay between exiting the car and the arming of the alarm (T_ARM_DELAY), the length of the countdown interval (T_PASSENGER_DELAY), and the length of time the siren sounds (T_ALARM_ON). The default value for each parameter is listed in the table below, but each may be changed. Time_Parameter_Selector switches specify the parameter number of the parameter to be changed. Time_Value switches are a 4-bit value representing the value to be programmed. Note that your system should behave correctly even if one or more of the parameters is set to 0.

Interval Name	Symbol	Parameter Number	Default Time (sec)	Time Value
Arming delay	T_ARM_DELAY	00	6	0110
Countdown, driver's door	T_DRIVER_DELAY	01	8	1000
Countdown, passenger door	T_PASSENGER_DELAY	10	15	1111
Siren ON time	T_ALARM_ON	11	10	1010

Block Descriptions/Implementation

The following diagram illustrates a possible organization of your design into modules.

Car Alarm FSM

- When Gim graduated from MIT he got a job with DEC (Digital Equipment Corporation) that made the PDP-1 among other computers and then TI
- Got big signing bonus and bought a nice convertible
- Parked Convertible went into apartment.
- Convertible was not there came out



Gim Hom

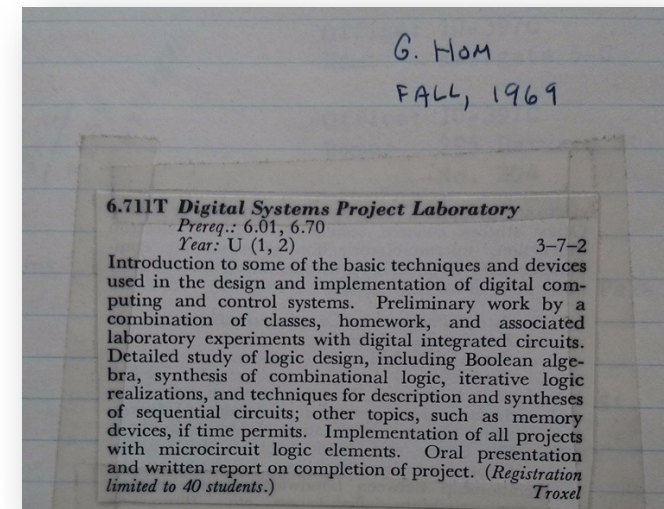
Took 6.111 in 1969..graduated in 1970

6.111 Instructor 2013-2021

Now retired

Before that was Anantha Chandrakasan

Before that was Troxel (go back to 1960s/originated course)



Car Theft FSMs

- 2016, MA: ~7 million people, 6,600 car thefts for year
- 1975, MA: 5.8 million people, 91,000 car thefts for year (peak)

~5% of cars were stolen per year in MA

*Massachusetts was
the #1 state for car
theft for 1965-1987!!!*

Massachusetts Population and Rate of Crime Rank Compared to other States

State	Year	Population	Index	Violent	Property	Murder	Rape	Robbery	Assault	Burglary	Larceny	Vehicle
Massachusetts	1960	8	35	36	33	41	38	32	36	34	38	11
Massachusetts	1961	8	29	35	28	42	35	31	34	29	32	7
Massachusetts	1962	9	26	33	24	40	37	25	31	26	31	7
Massachusetts	1963	9	28	33	27	40	39	27	33	27	33	4
Massachusetts	1964	9	26	33	25	37	37	24	38	25	34	2
Massachusetts	1965	10	23	33	22	36	39	21	36	23	35	1
Massachusetts	1966	10	24	32	23	38	41	19	36	21	37	1
Massachusetts	1967	10	25	34	25	38	37	23	37	25	38	1
Massachusetts	1968	10	21	31	20	35	35	21	35	19	35	1
Massachusetts	1969	10	19	29	18	35	38	19	35	15	34	1
State	Year	Population	Index	Violent	Property	Murder	Rape	Robbery	Assault	Burglary	Larceny	Vehicle
Massachusetts	1970	10	20	31	18	38	33	19	35	14	35	1
Massachusetts	1971	10	16	27	16	36	38	13	33	13	32	1
Massachusetts	1972	10	17	26	16	38	39	11	29	13	35	1
Massachusetts	1973	10	15	20	14	36	34	10	29	12	33	1
Massachusetts	1974	10	12	20	11	37	37	10	29	12	32	1
Massachusetts	1975	10	11	16	12	39	29	9	26	13	34	1
Massachusetts	1976	10	13	18	12	40	37	11	22	11	36	1
Massachusetts	1977	10	15	17	16	43	31	12	19	13	37	1
Massachusetts	1978	10	16	16	17	40	31	13	19	12	38	1
Massachusetts	1979	10	16	15	15	41	32	12	16	11	36	1
State	Year	Population	Index	Violent	Property	Murder	Rape	Robbery	Assault	Burglary	Larceny	Vehicle
Massachusetts	1980	11	16	13	17	39	31	9	13	13	39	1
Massachusetts	1981	11	20	12	21	40	30	8	15	17	40	1
Massachusetts	1982	11	18	14	21	37	31	10	13	19	40	1
Massachusetts	1983	11	20	12	22	40	28	9	10	20	41	1
Massachusetts	1984	12	25	13	24	36	25	12	12	25	41	2
Massachusetts	1985	12	23	16	25	39	23	12	14	24	40	1
Massachusetts	1986	12	26	18	27	38	26	14	17	28	43	1
Massachusetts	1987	13	26	14	28	42	26	15	14	31	45	1
Massachusetts	1988	13	23	13	25	38	25	14	11	29	41	3

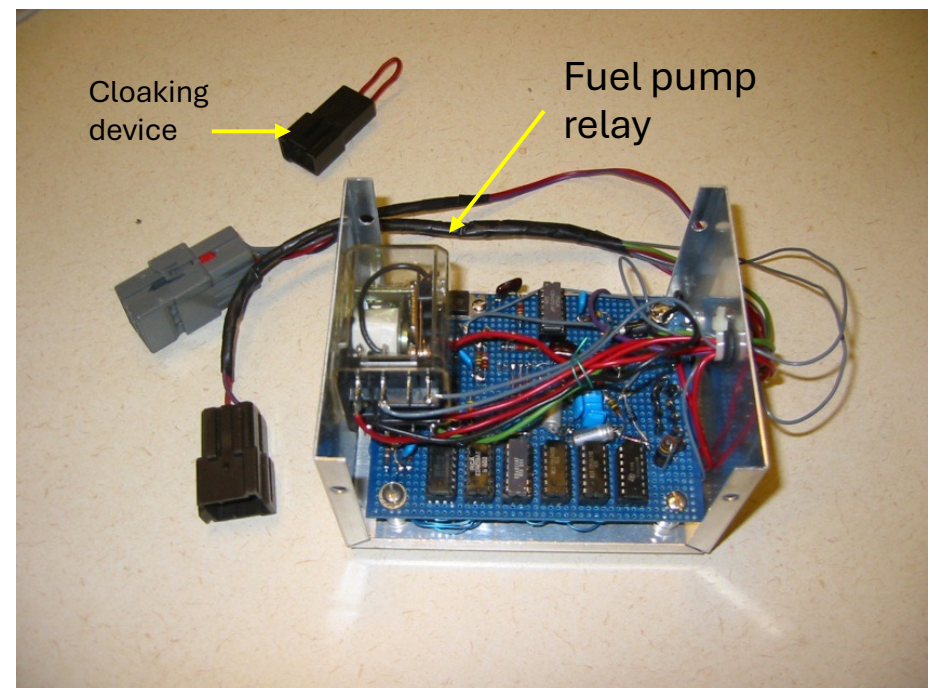
State	Year	Population	Index	Violent	Property	Murder	Rape	Robbery	Assault	Burglary	Larceny	Vehicle
Massachusetts	2000	13	42	21	44	41	36	27	14	41	49	16
Massachusetts	2001	13	41	20	43	42	29	25	15	40	47	18
Massachusetts	2002	13	39	18	42	38	34	24	17	39	46	18
Massachusetts	2003	13	28	17	40	42	30	19	18	36	46	18
Massachusetts	2004	13	39	18	42	37	33	20	18	38	47	22
Massachusetts	2005	13	42	19	45	37	36	20	18	35	48	30
Massachusetts	2006	13	41	20	44	35	37	21	19	34	45	33
Massachusetts	2007	14	42	22	42	37	38	23	18	33	45	36
Massachusetts	2008	14	37	20	44	41	39	24	14	32	45	35
Massachusetts	2009	15	37	17	43	38	38	23	14	33	45	33
State	Year	Population	Index	Violent	Property	Murder	Rape	Robbery	Assault	Burglary	Larceny	Vehicle
Massachusetts	2010	14	32	13	38	31	35	18	11	29	41	34
Massachusetts	2011	14	36	14	42	37	37	20	12	32	43	34
Massachusetts	2012	14	40	19	44	46	39	20	14	33	48	39
Massachusetts	2013	14	36	16	45	43	16	18	15	35	46	39
Massachusetts	2014	14	39	18	46	43	35	20	14	38	47	40
Massachusetts	2015	15	42	18	47	45	38	27	14	42	48	42
Massachusetts	2016	15	45	23	47	47	42	26	20	45	49	44

Car Alarm FSM

- Gim built a car alarm for his car.
- Designed it using an FSM-based approach.
- Had ~11 states
- Built it just like we talked about last week (bubble-diagram...developed logic, implemented...)



*Gim's FSM-based car alarm for his car
Built using 4000-series CMOS chips
(top of the line at the time)*



Magnavox Odyssey (1972)

- First commercially available game system
- Completely FSM-based

1TL200 BLAK & BK12 ODYSSEY GAME SIMULATOR

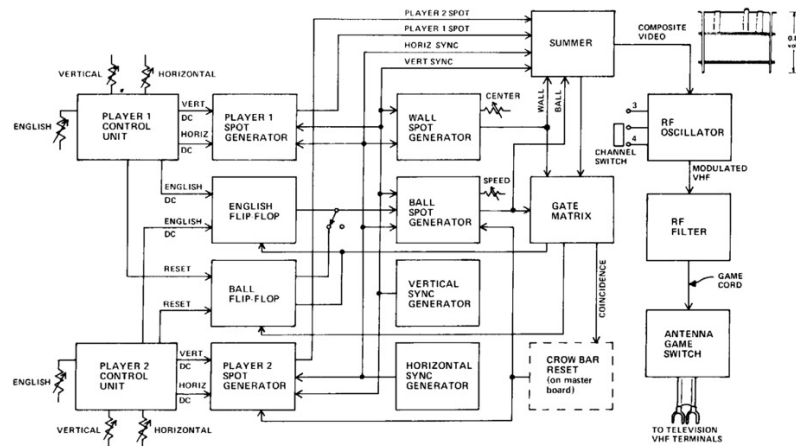
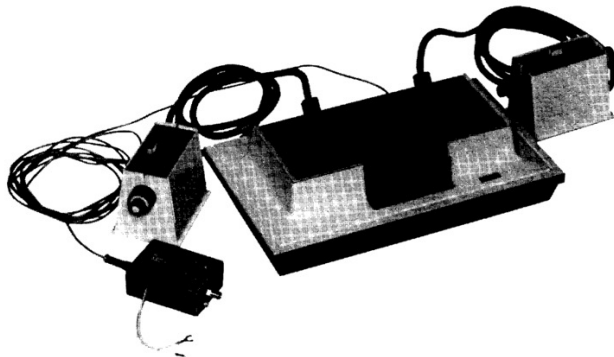


Figure 1 -- Odyssey Block Diagram

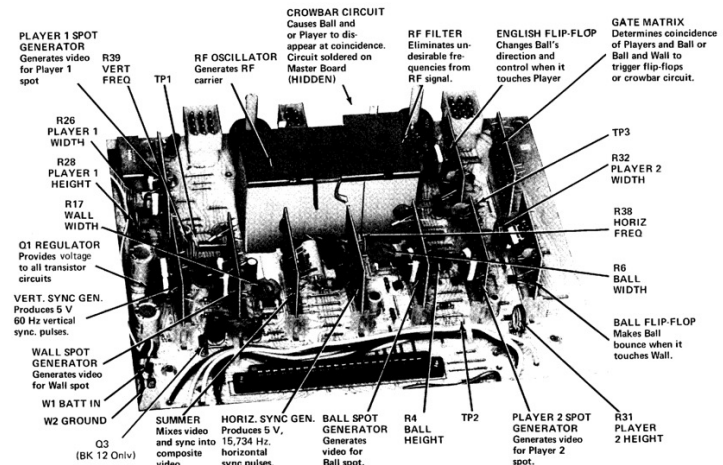
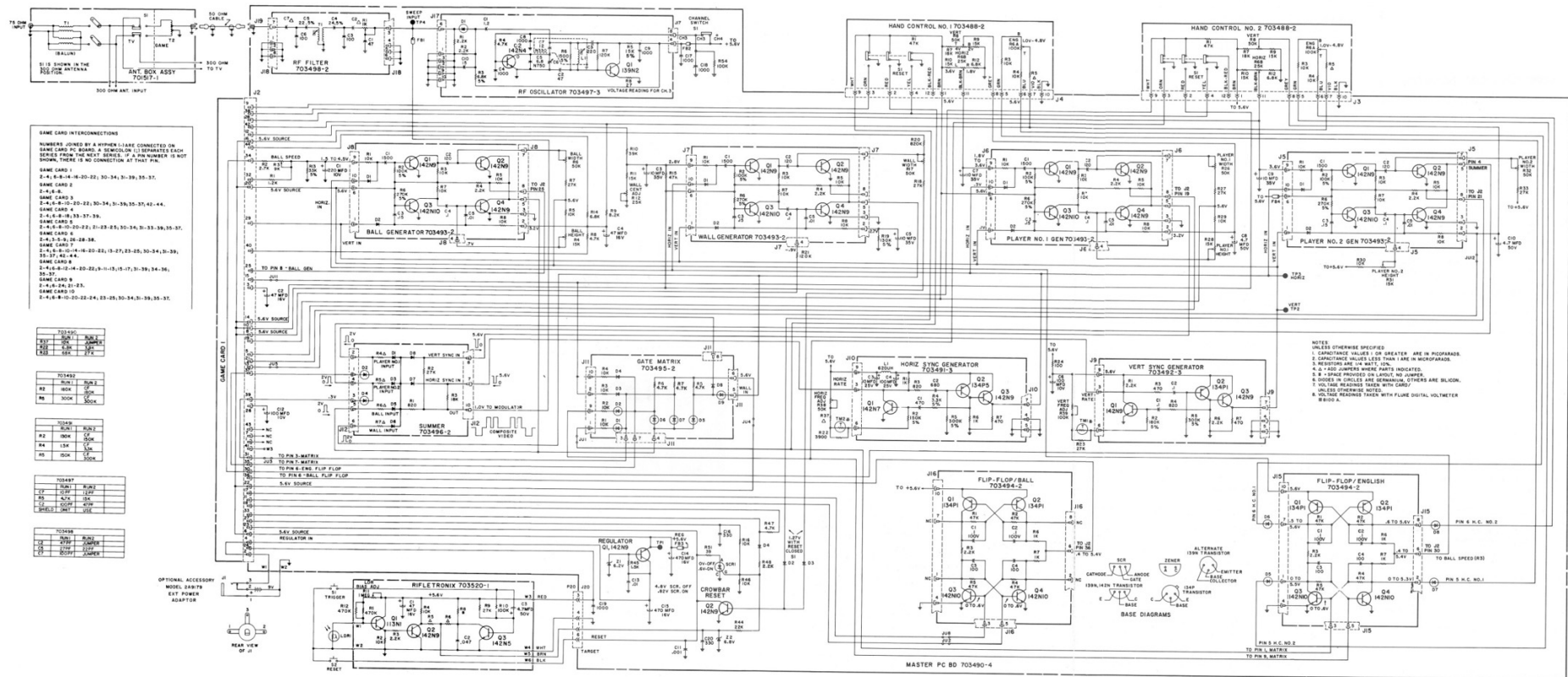


Figure 3 -- Master Board Module Location

MAGNAVOX ODYSSEY 1TL200BLAK SCHEMATIC DIAGRAM
Scanned from original service manual by David WINTER
<http://www.pong-story.com>



Was just a large finite state machine

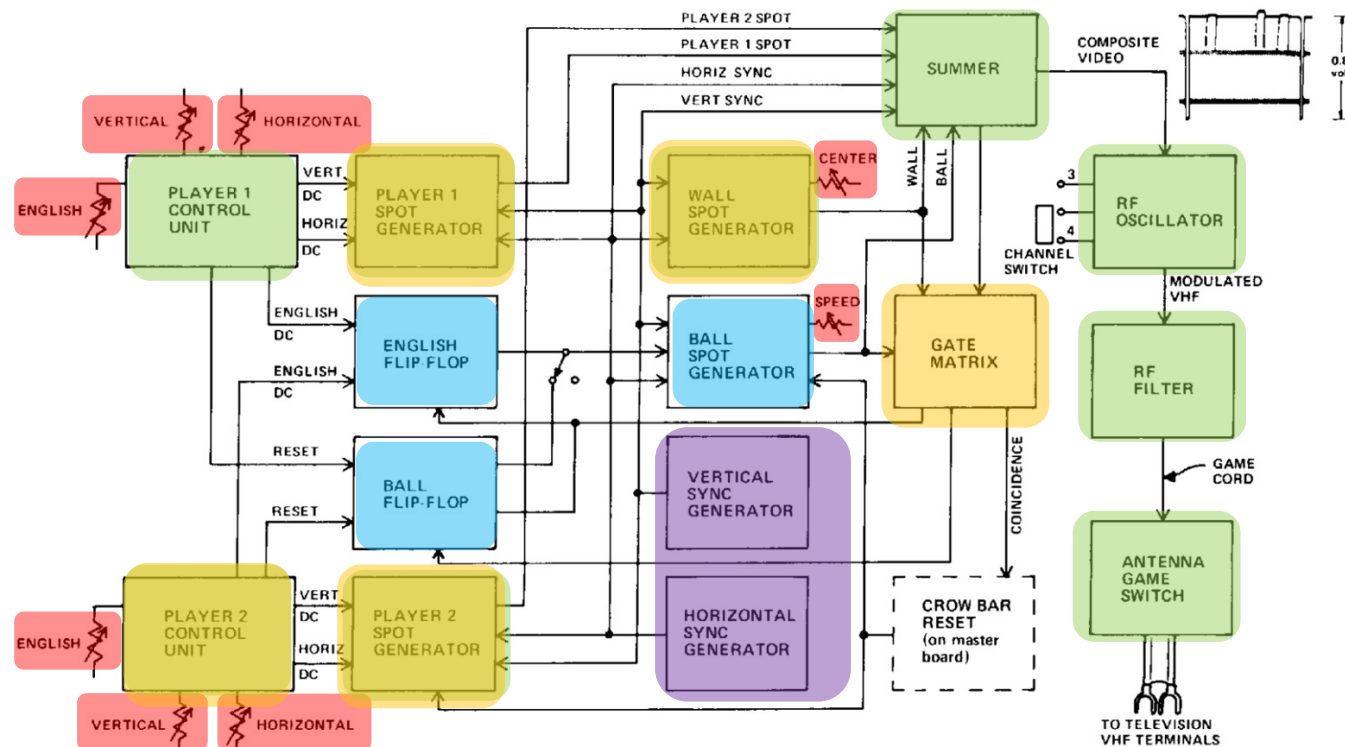
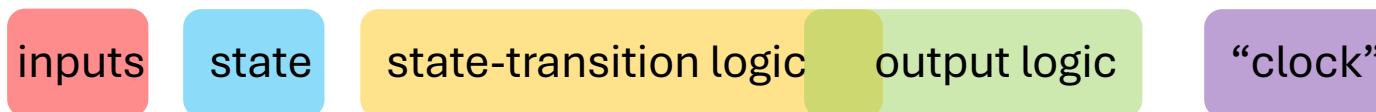


Figure 1 -- Odyssey Block Diagram



Magnavox Odyssey Game System



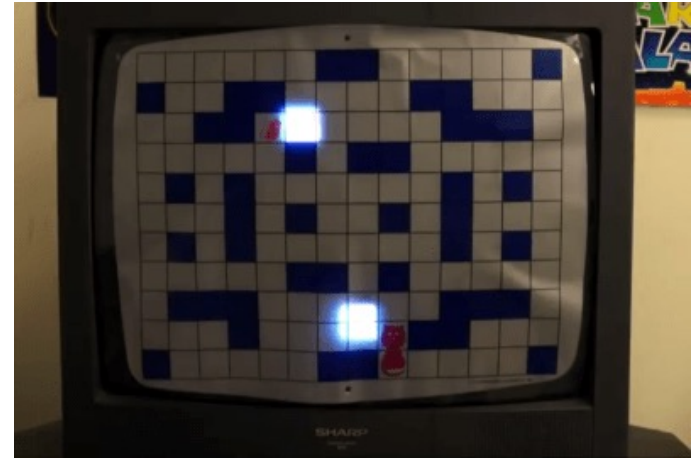
<http://odysim.blogspot.com/2020/>

Magnavox Odyssey Game System

Hockey



Cat and Mouse



Basketball



<https://youtube.com/playlist?list=PLtApm-Ri5WTIAEV1ClufPrca2MTj4uSvT&feature=shared>

https://www.youtube.com/watch?app=desktop&v=NsluZfTMRno&ab_channel=OdysseyNow

9/29/26

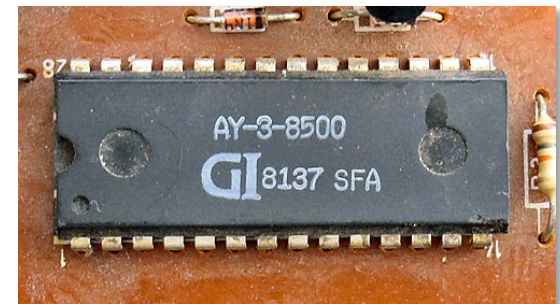
<https://fpga.mit.edu/6205/F26>

Early 1970s

- Most arcade systems were just FSMs implemented in discrete logic, including:
 - Pong
 - Breakout
- Space Invaders was first arcade machine to move some game logic to an Intel 8080 microprocessor.

Evolution of FSM-based Games

- As 1970s rolled on, entire game systems would get put on single chips
- “Ball-and-Paddle” Chips would be sold by companies and then other companies would buy them and put their own “skin” on them and sell them as their own. Many times it was the same game underneath
- Atari 2600 was first *microprocessor-based* home video game system



AY-3-8500 “Ball-and-Paddle” chip

• <http://www.pong-story.com/gi.htm>

<https://commons.wikimedia.org/wiki/File:AY-3-8500.jpg>

TV Fun



- Runs off an AY-3-8500
- Made by APF who started out importing Japanese 8-track players
- Company went bankrupt in the great video game crash of 1983.
- Have one set upstairs in lab in case anybody wants to play.

Tiger Electronics Games



Tiger Electronics Games

- Tiger Electronics had 100's of versions of these in the 1980s and 1990s
- Almost all of them were based on three or four common **finite state machine** game chips or four-bit microcontrollers (very poorly documented/insider information)
- They'd slap a different LCD skin and game art onto the same chip and resell



Modern Games

- Modern games are far too complex to be implemented with an FSM in any productive way (though it is still generally possible)
- However well-characterized chunks of game software is still used and re-used/skinned (for example game engines)
- But also stuff gets reskinned all the time

Candy Crush Saga



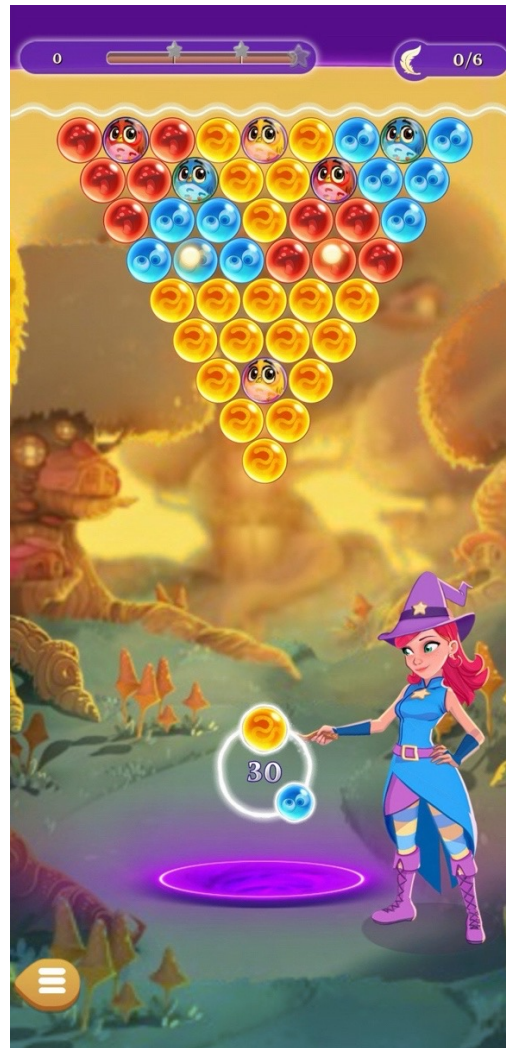
Pet Rescue Saga



Soda Saga



Bubble Witch 3 Saga



Farm Heroes Saga

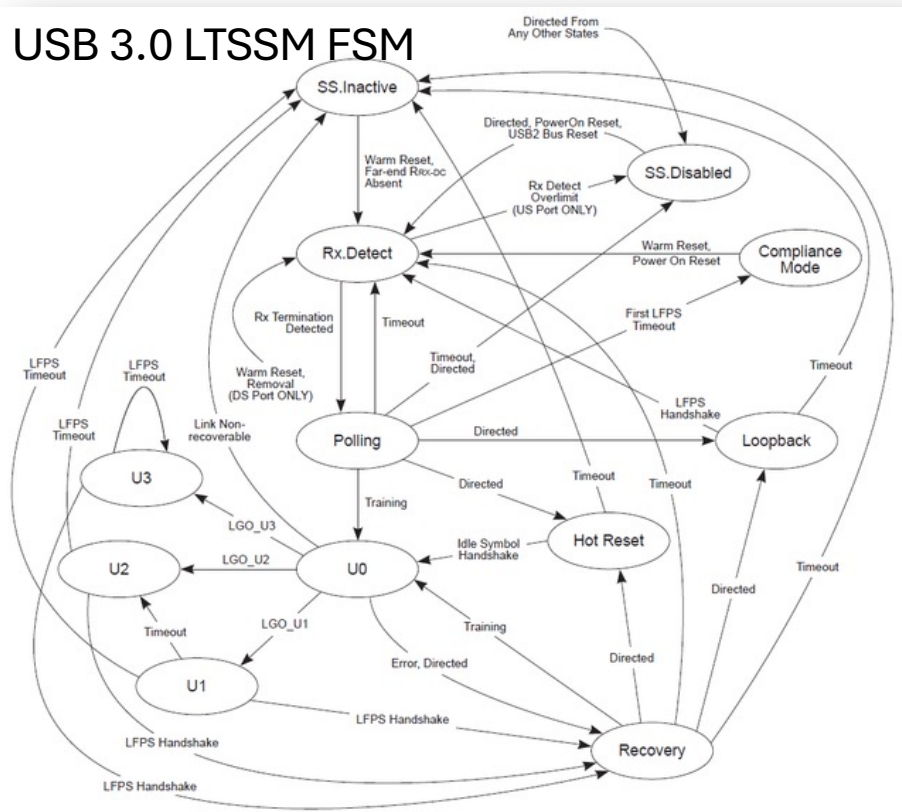


Finite State Machines Relevant

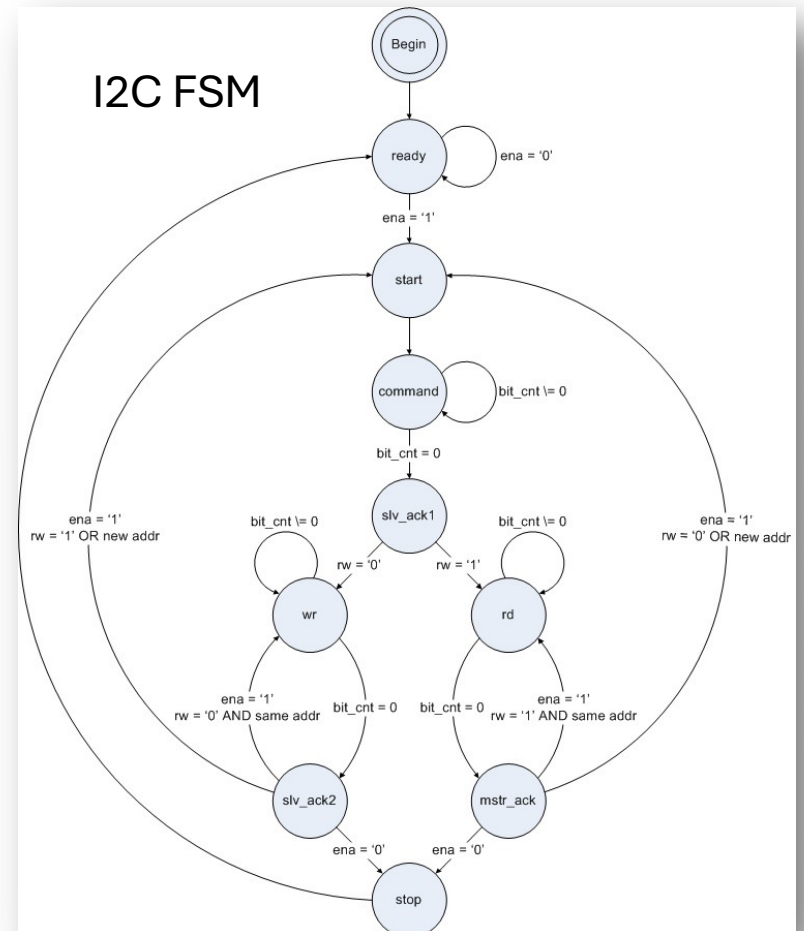
- Designing systems as finite state machines is still very common in digital design.
- Doing it in a structured way can make your HDL very transparent so you know what you're getting!
- You'll see data sheets and other places with FSM diagrams and many protocols express their functionality with FSMs.

See them everywhere

USB 3.0 LTSSM FSM



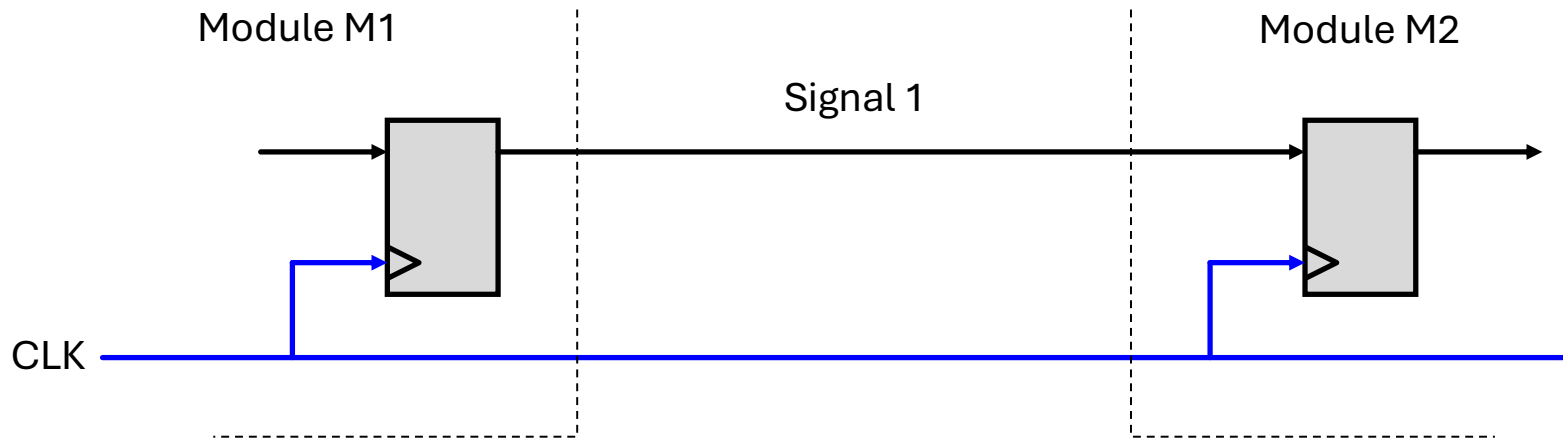
I2C FSM



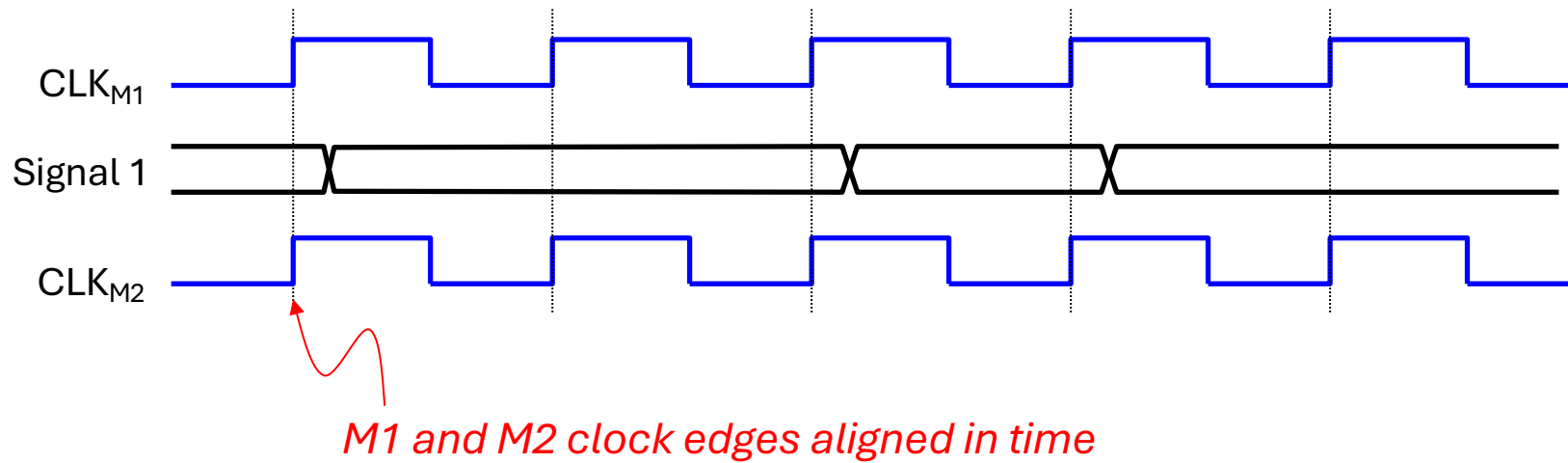
Clocks and Time

Times and Clocks

Clocking and Synchronous Communication

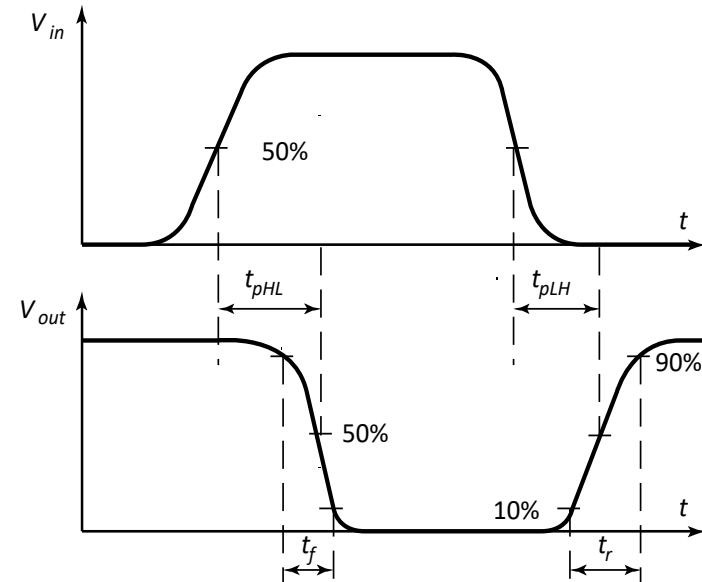
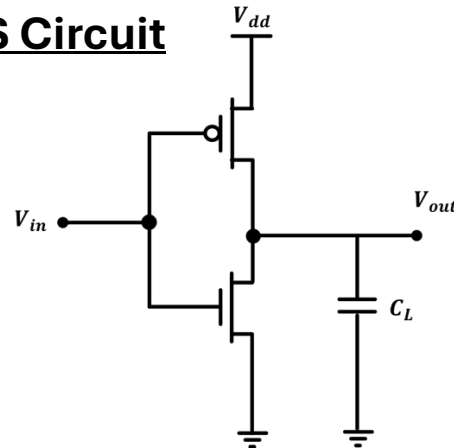


Ideal world:

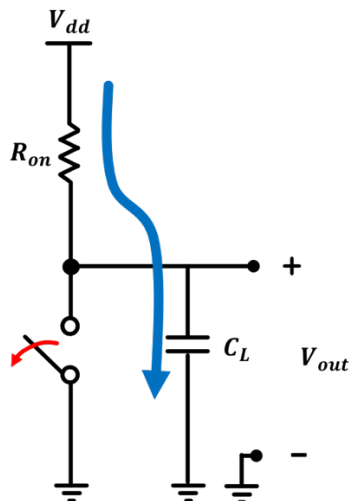


Delay Estimation: Simple RC Networks

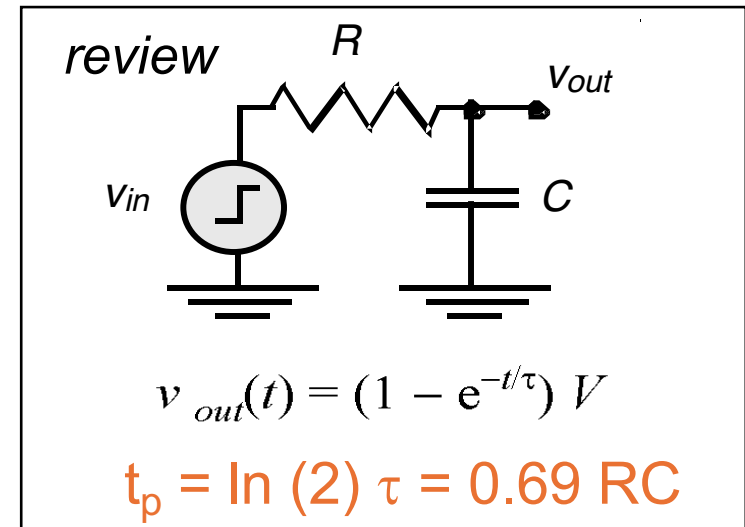
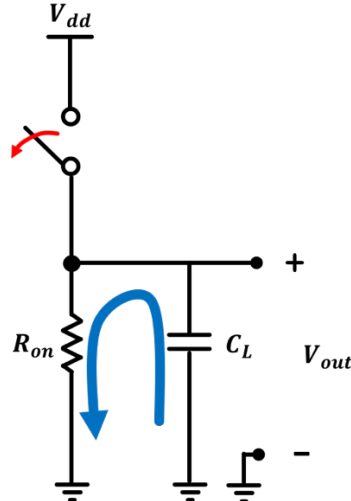
Simple CMOS Circuit



Low-to-High



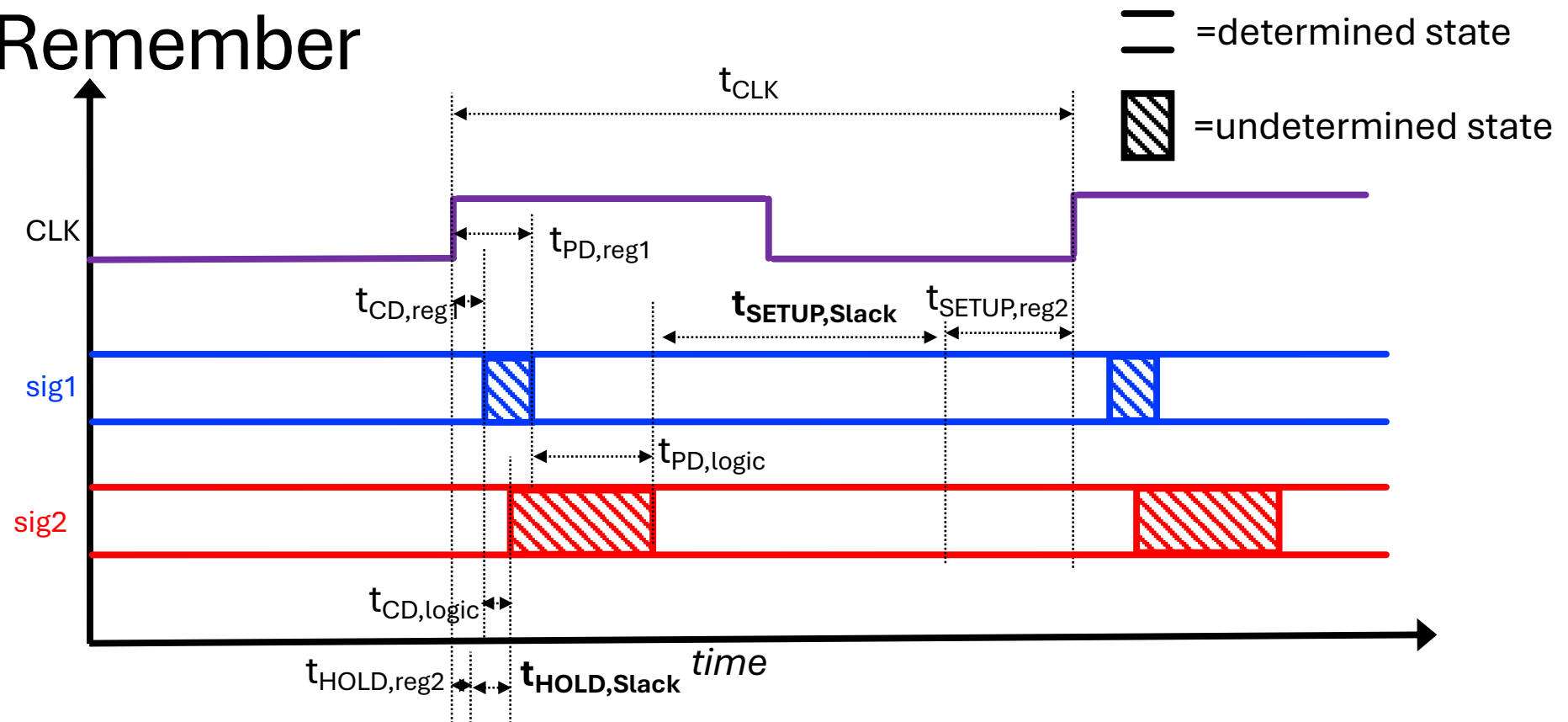
High-to-Low



So Signals Experience Delays

- “Signals” generally have their delays expressed with:
 - Contamination Delay
 - Propagation Delay
- But the clock is not immune to delay too!

Remember



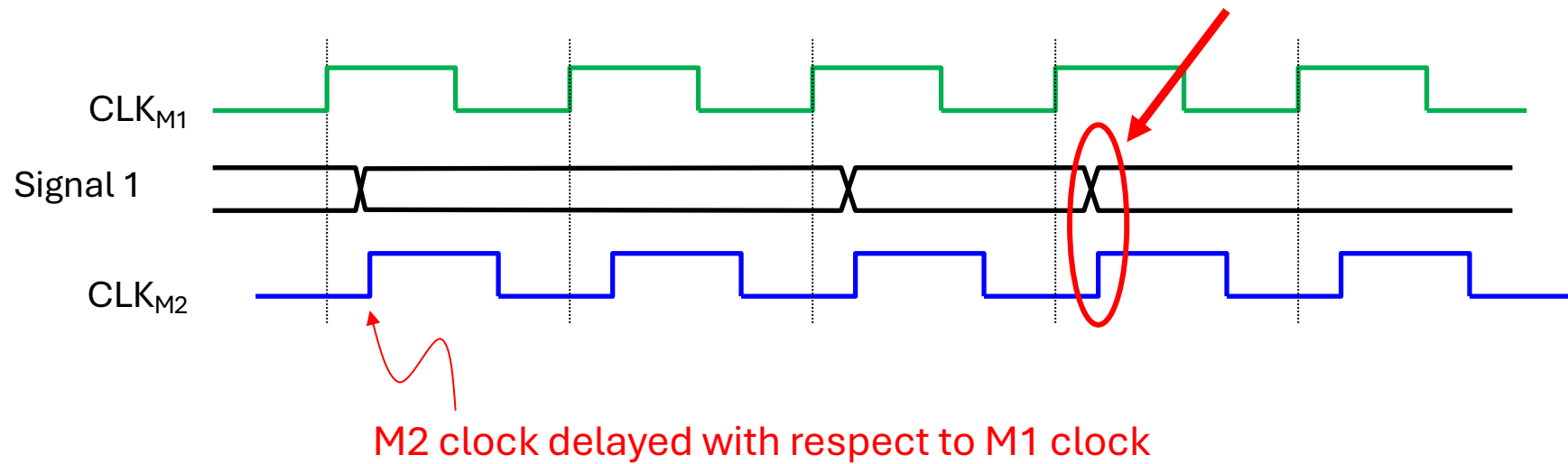
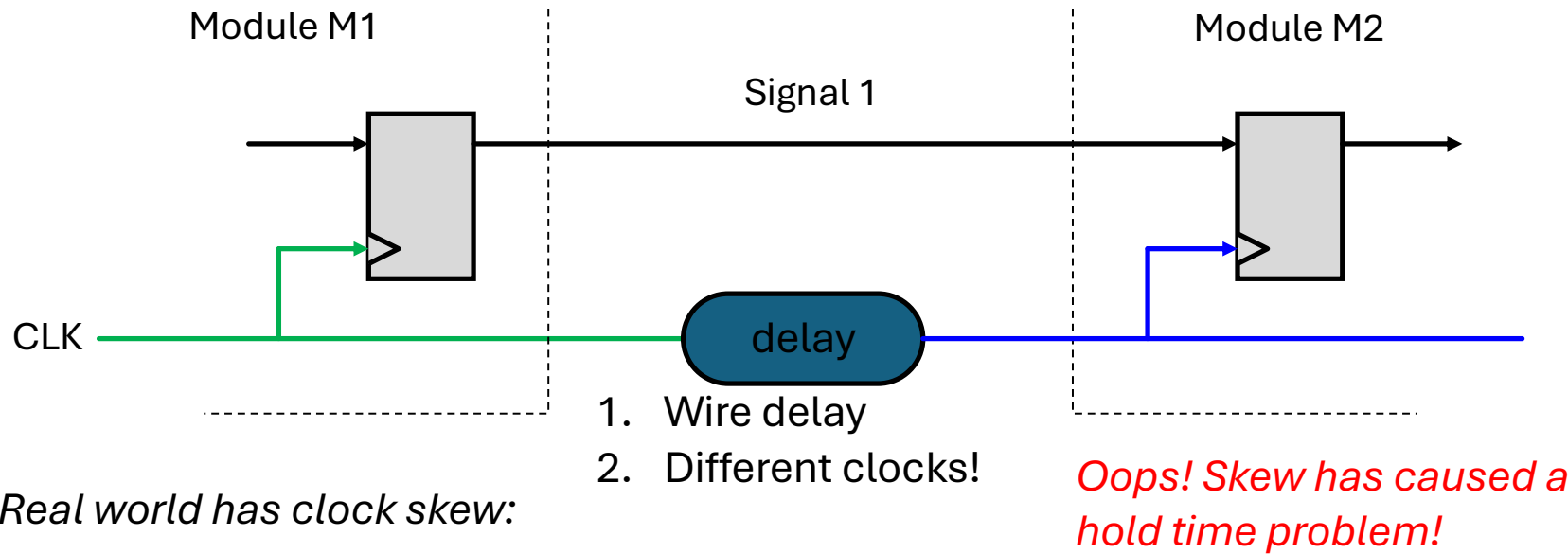
$$t_{PD,reg1} + t_{PD,logic} + t_{SETUP,reg2} + t_{SETUP,Slack} = t_{CLK}$$

$$t_{CD,reg1} + t_{CD,logic} = t_{HOLD,reg2} + t_{HOLD,Slack}$$

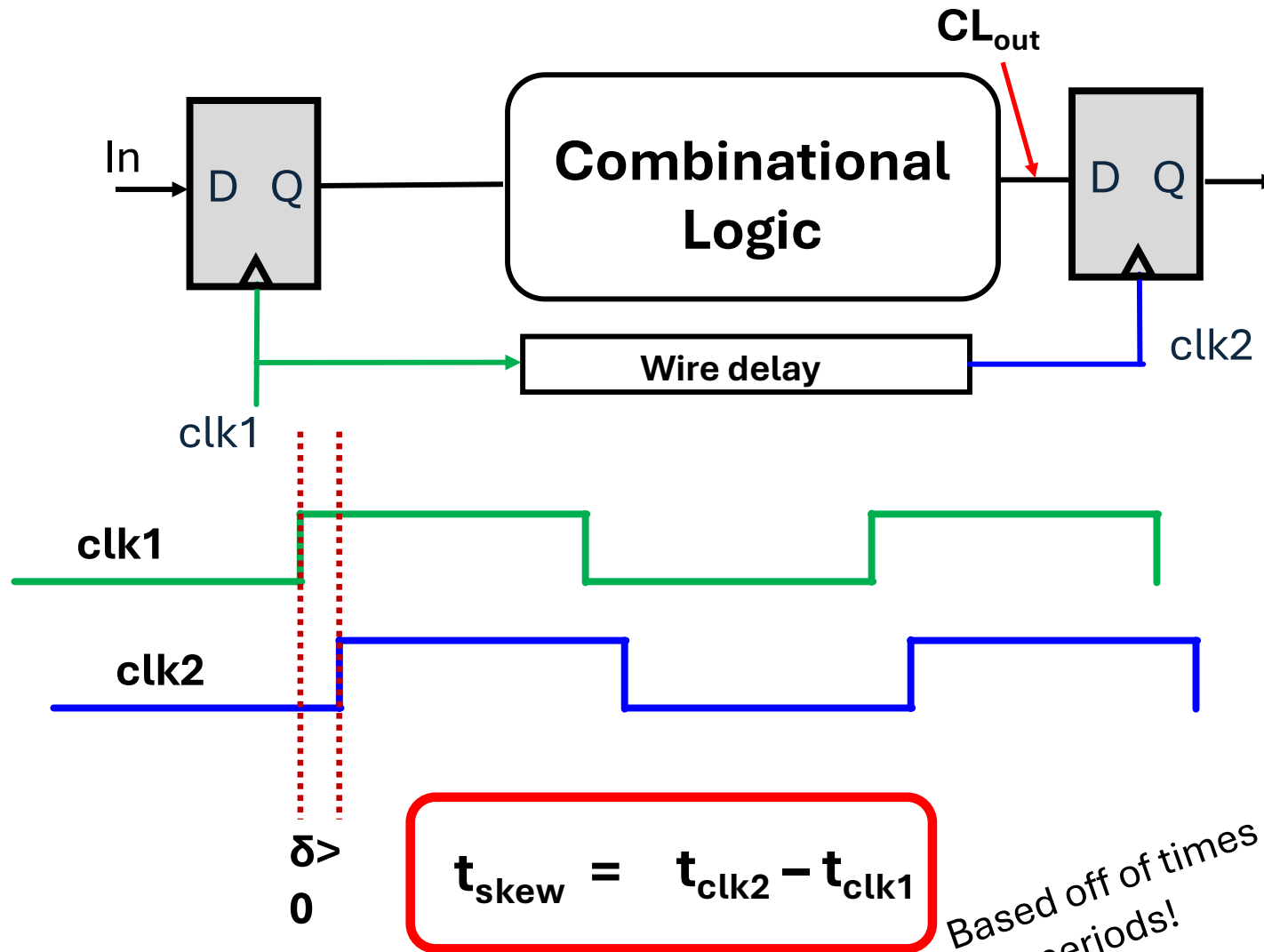
Equations* $t_{SETUP,Slack} = t_{CLK} - (t_{PD,reg1} + t_{PD,logic} + t_{SETUP,reg2})$

$$t_{HOLD,Slack} = t_{CD,reg1} + t_{CD,logic} - t_{HOLD,reg2}$$

Clock Skew

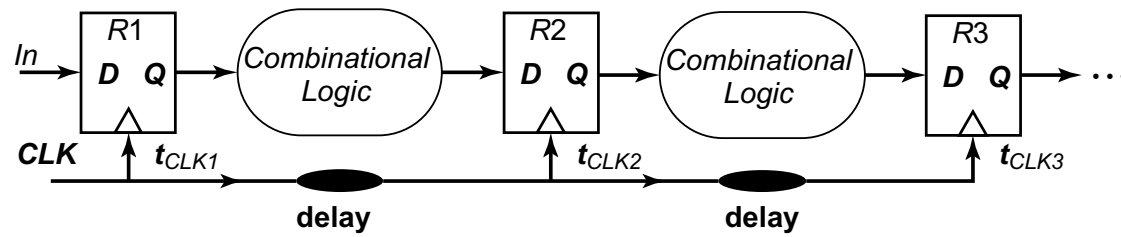


Clocks are Not Perfect: Clock Skew

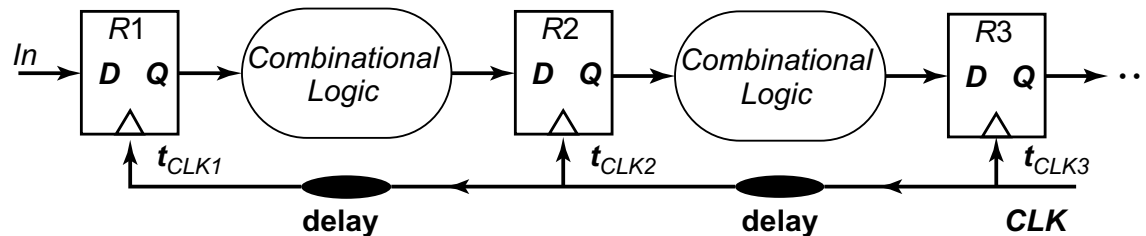


You Can Have Two Types of Skew!

Positive Skew:



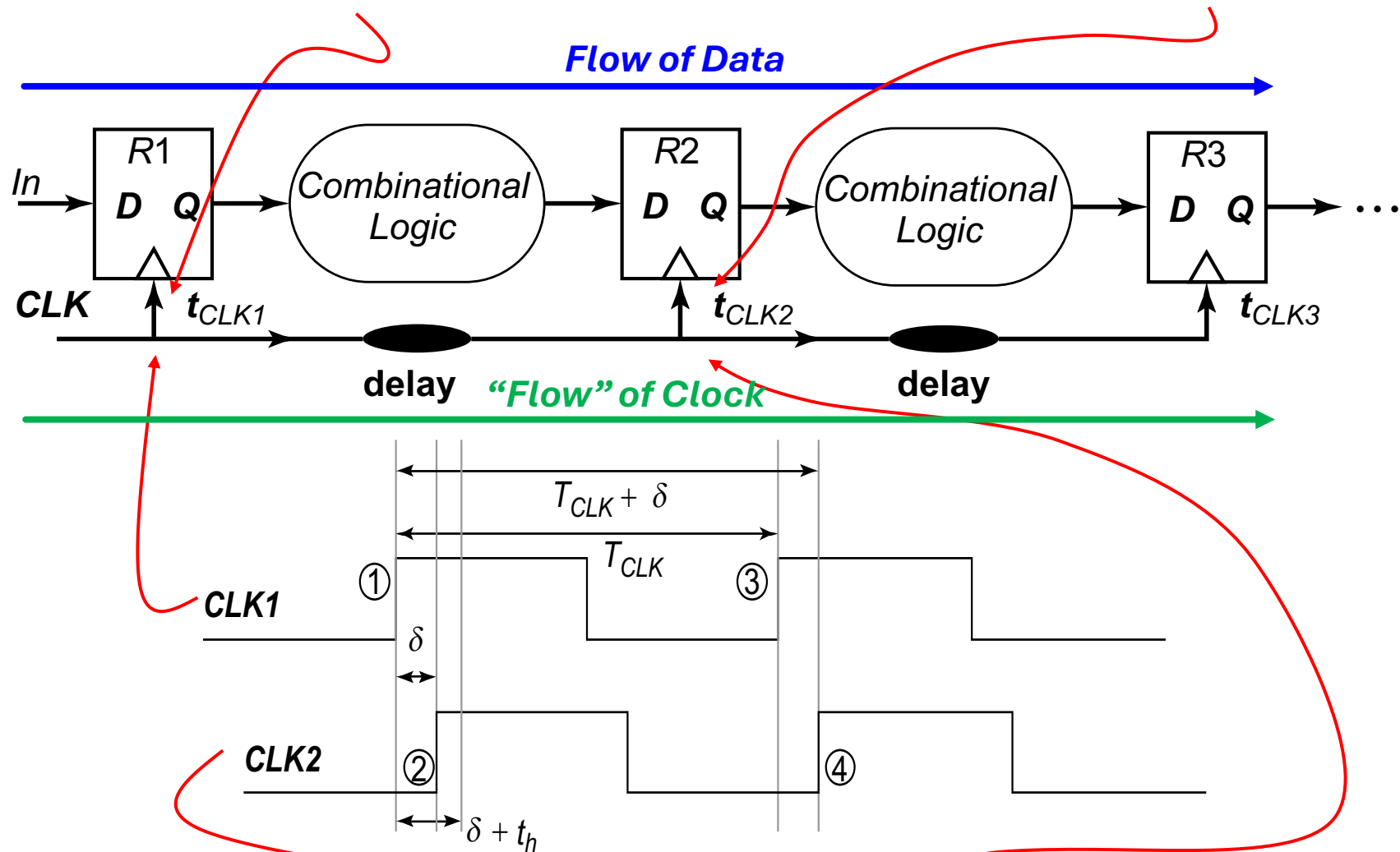
Negative Skew:



➤ Adapted from J. Rabaey, A. Chandrakasan, B. Nikolic,
“Digital Integrated Circuits: A Design Perspective” Copyright 2003 Prentice Hall/Pearson.

Positive Skew

The launching edge arrives (in time) before the receiving edge



➤ Adapted from J. Rabaey, A. Chandrakasan, B. Nikolic,

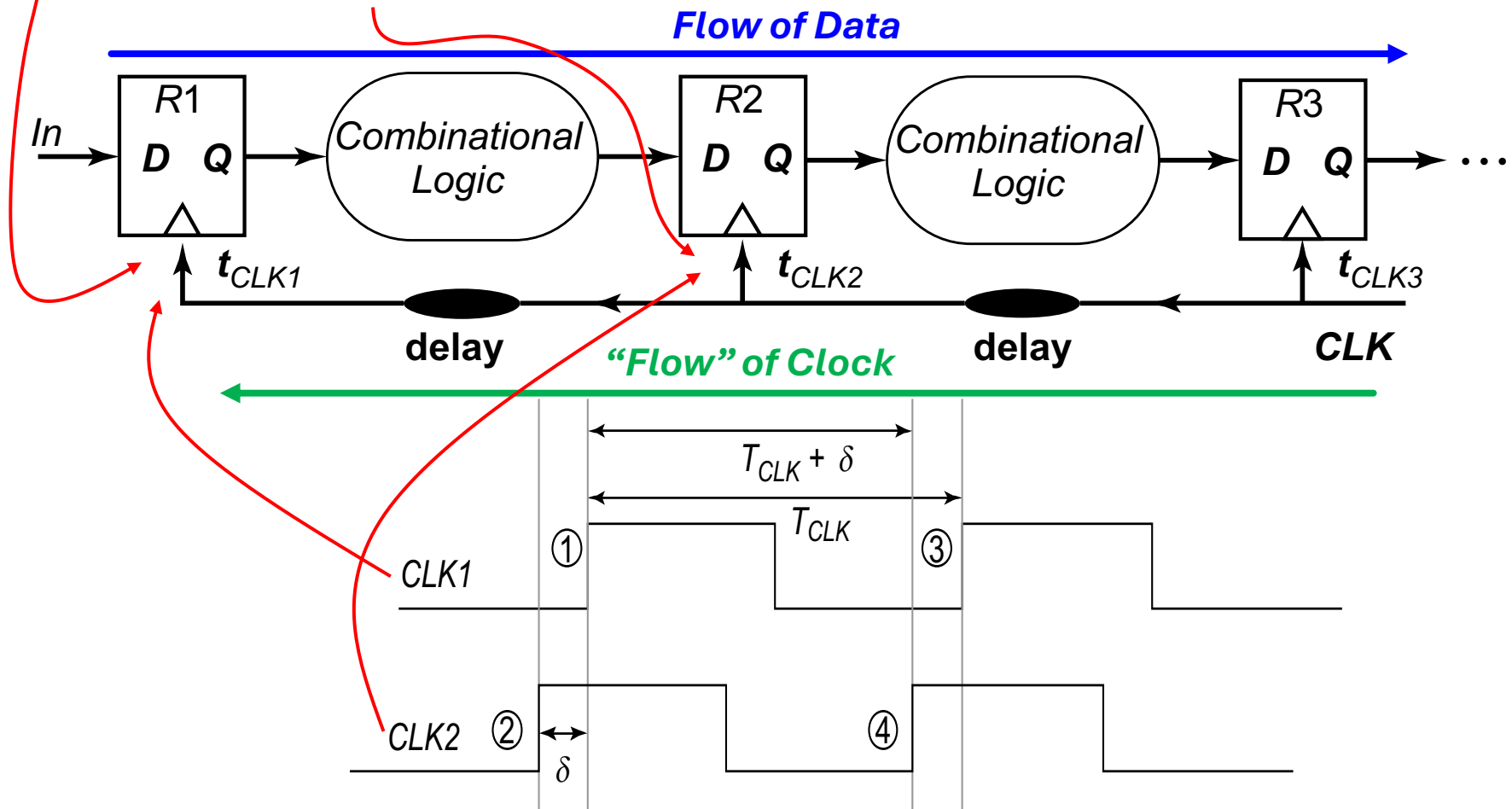
"Digital Integrated Circuits: A Design Perspective" Copyright 2003 Prentice Hall/Pearson.

9/29/26

<https://fpga.mit.edu/6205/F26>

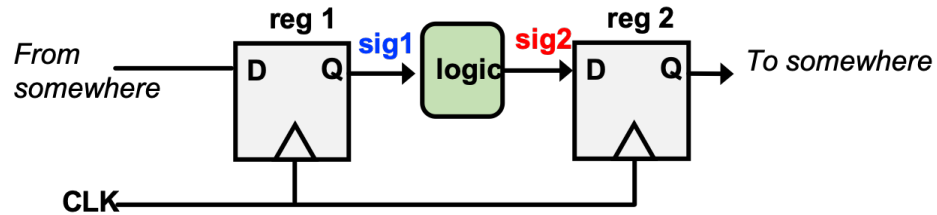
Negative Skew

The receiving edge arrives (in time) before the launching edge

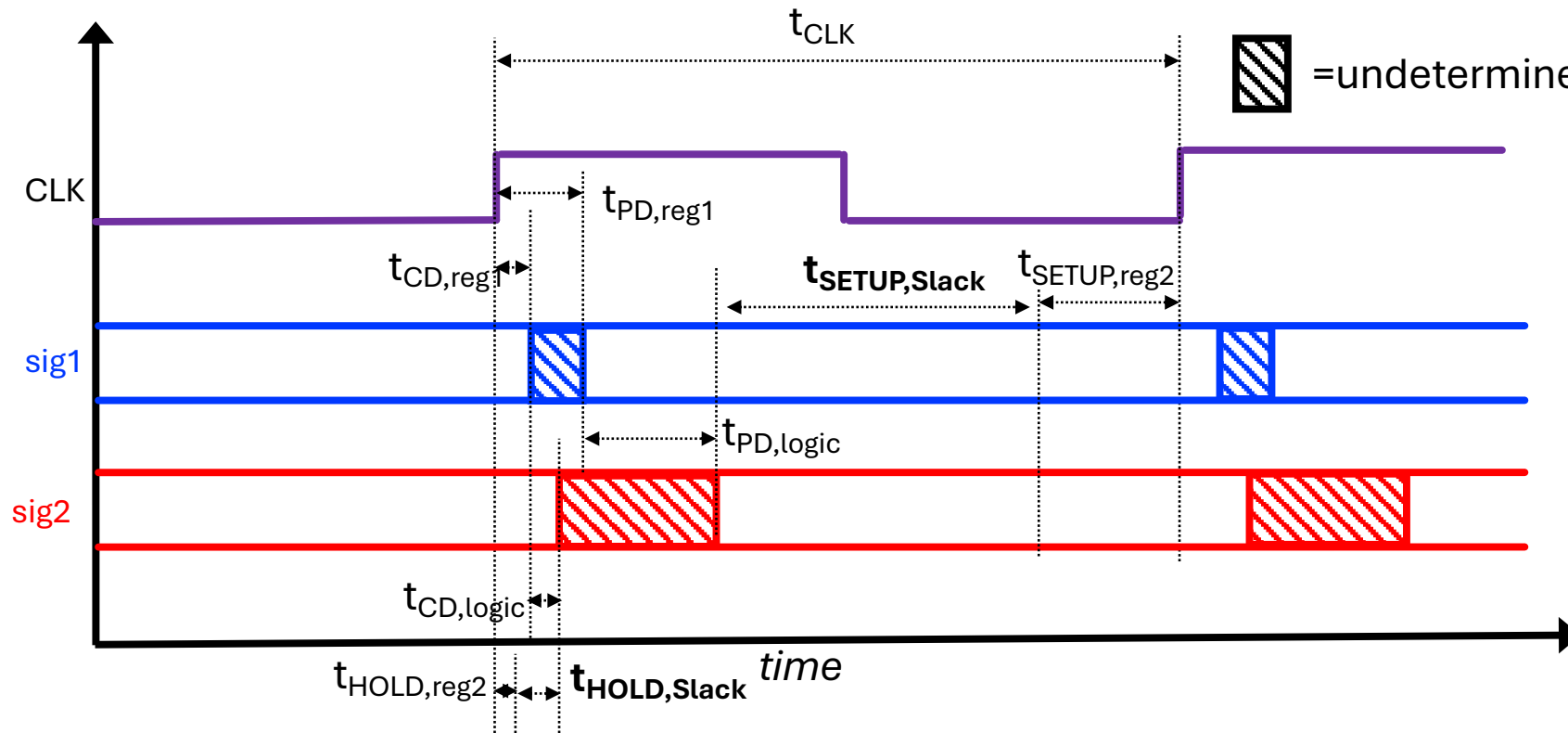


➤ Adapted from J. Rabaey, A. Chandrakasan, B. Nikolic,
"Digital Integrated Circuits: A Design Perspective" Copyright 2003 Prentice Hall/Pearson.

Timing



— =determined state
 =undetermined state



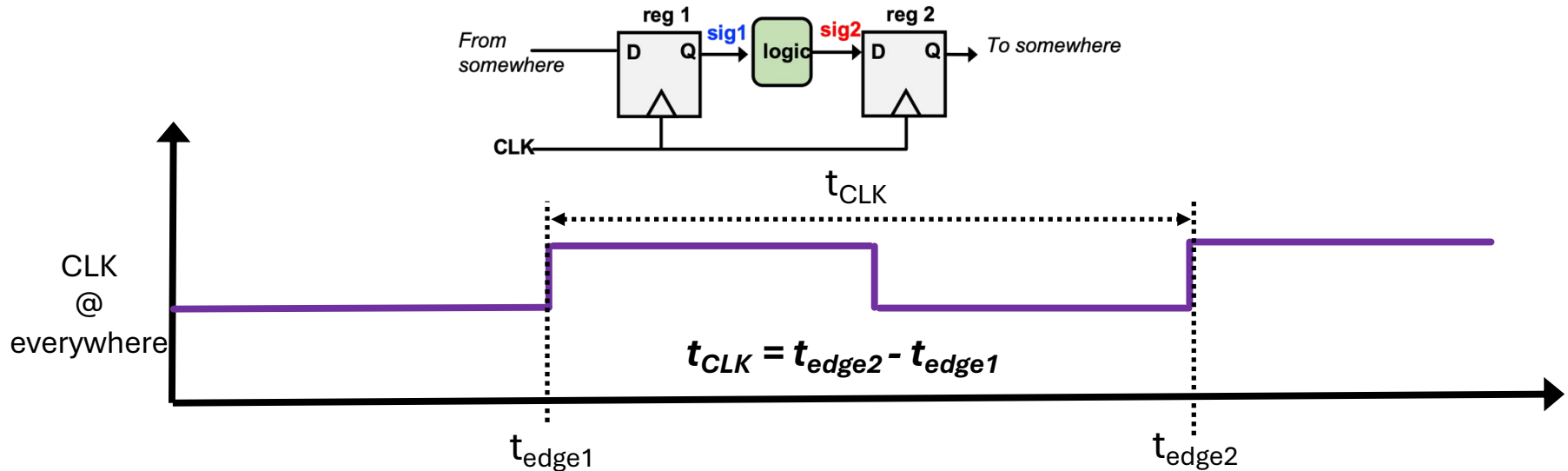
Equations*

$$t_{\text{SETUP,Slack}} = t_{\text{CLK}} - (t_{\text{PD,reg1}} + t_{\text{PD,logic}} + t_{\text{SETUP,reg2}})$$

$$t_{\text{HOLD,Slack}} = t_{\text{CD,reg1}} + t_{\text{CD,logic}} - t_{\text{HOLD,reg2}}$$

How does Skew Affect Those Equations?

- Originally in our model circuit, we assume all devices experience the clock edges at the same time!!!!



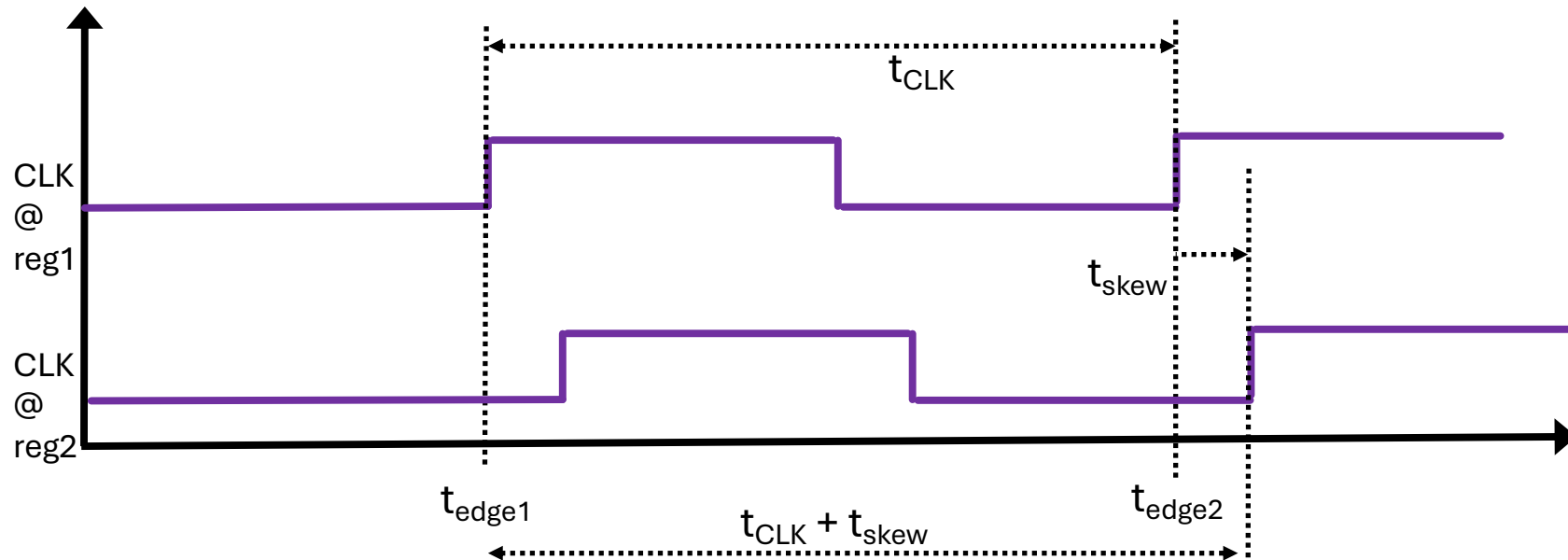
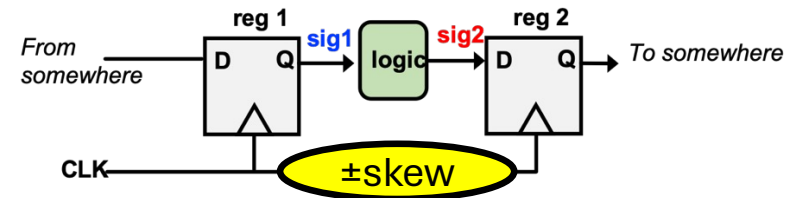
- This whole time:

$$t_{\text{SETUP,Slack}} = t_{\text{CLK}} - (t_{\text{PD,reg1}} + t_{\text{PD,logic}} + t_{\text{SETUP,reg2}})$$

- ...was really short-hand for:

$$t_{\text{SETUP,Slack}} = t_{\text{edge2}} - t_{\text{edge1}} - (t_{\text{PD,reg1}} + t_{\text{PD,logic}} + t_{\text{SETUP,reg2}})$$

With Skew in the Mix...



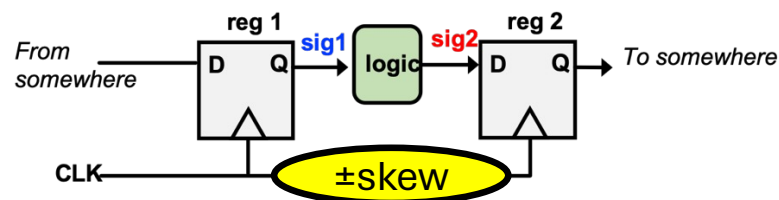
- With skew the equation turns into:

$$t_{\text{SETUP,Slack}} = t_{\text{edge2}} + t_{\text{skew}} - t_{\text{edge1}} - (t_{\text{PD,reg1}} + t_{\text{PD,logic}} + t_{\text{SETUP,reg2}})$$

- But since $t_{\text{CLK}} = t_{\text{edge2}} - t_{\text{edge1}}$

$$t_{\text{SETUP,Slack}} = t_{\text{CLK}} + t_{\text{skew}} - (t_{\text{PD,reg1}} + t_{\text{PD,logic}} + t_{\text{SETUP,reg2}})$$

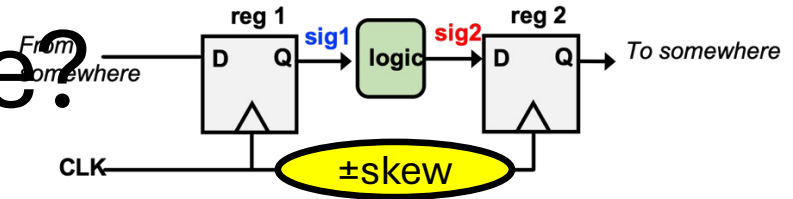
With Skew



$$t_{\text{SETUP,Slack}} = t_{\text{CLK}} + t_{\text{skew}} - (t_{\text{PD,reg1}} + t_{\text{PD,logic}} + t_{\text{SETUP,reg2}})$$

- Remember! Our goal is always to get Slack to be positive!
- If that's now our modified setup equation...
 - Positive skew makes equation *easier* to satisfy
 - Negative skew makes equation *harder* to satisfy
- It can still be fixed even in negative skew case
 - BUT you have degree of freedom with $t_{\text{PD,logic}}$...maybe you can change that?
 - AND/OR you could also increase t_{clk} as well.

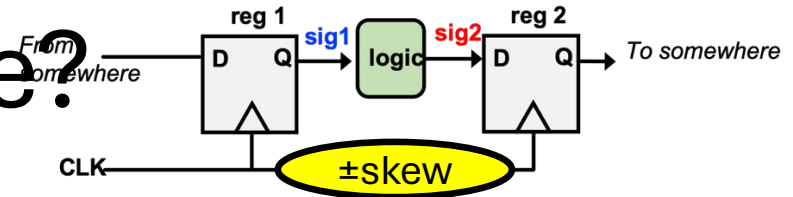
What about Hold Time?



- If the second register is getting its clock edge t_{skew} *after* the first register that means the signals going into reg2 need to be held for t_{skew} longer after the clock edge than otherwise.
- Effectively increases the hold time requirement of receiving register
- Hold Equation gets modified to be :-/

$$t_{\text{HOLD,Slack}} = t_{\text{CD,reg1}} + t_{\text{CD,logic}} - t_{\text{HOLD,reg2}} - t_{\text{skew}}$$

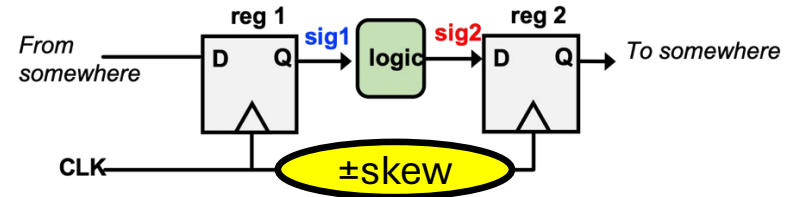
What about Hold Time?



$$t_{\text{HOLD,Slack}} = t_{\text{CD,reg1}} + t_{\text{CD,logic}} - t_{\text{HOLD,reg2}} - t_{\text{skew}}$$

- The goal is always to have positive slack, so a minus sign here means...
- Positive skew makes equation harder to satisfy.
 - Further there's nothing *you* can do since contamination delays are usually very low and beyond our control
- Negative skew makes equation easier to satisfy.

Conclusions



$$t_{\text{SETUP,Slack}} = t_{\text{CLK}} + t_{\text{skew}} - (t_{\text{PD,reg1}} + t_{\text{PD,logic}} + t_{\text{SETUP,reg2}})$$

$$t_{\text{HOLD,Slack}} = t_{\text{CD,reg1}} + t_{\text{CD,logic}} - t_{\text{HOLD,reg2}} - t_{\text{skew}}$$

- **Positive clock skew** improves the minimum cycle time of our design but makes it harder to meet register hold times.
- **Negative clock skew** hurts the minimum cycle time of our design but makes it easier to meet register hold times.
- *Positive skew* is tougher to deal with

Low-skew Clocking in FPGAs

- When Vivado is doing place-and-route it tries to position logic so that skew is minimized wherever possible.
 - It'll sometimes play games to take advantage of skew, but as we saw, it is very much a double-edged sword.
- Special clock paths and buffers exist throughout the chip to distribute the clock as effectively as possible.
- Think of it like priority boarding/VIP status for signals

Figures from Xilinx App Notes

From the Vivado Docs

The screenshot displays the AMD Technical Information Portal for the Vivado Design Suite Properties Reference Guide (UG912). The page is titled "Vivado Design Suite Properties Reference Guide (UG912)" and includes a search bar, a table of contents, and a list of properties. The property "CLOCK_DEDICATED_ROUTE" is highlighted in the table of contents. The main content area shows the "CLOCK_DEDICATED_ROUTE" property, which is enabled (TRUE) by default. It explains that this property ensures clock resource placement DRCs are considered error conditions that must be corrected prior to routing or bitstream generation. If set to FALSE, it downgrades the placement DRC to a warning and lets the Vivado router use fabric routing to connect from a clock-capable IO (CCIO) to a global clock resource such as an MMCM. A caution note states that setting CLOCK_DEDICATED_ROUTE to FALSE can result in sub-optimal clock delays, resulting in potential timing violations and other issues. The page also includes a section on external user clocks and a note on the general use of the CLOCK_DEDICATED_ROUTE property.

AMD | Technical Information Portal

← Search | Vivado Design Suite Properties Reference Guide (UG912) | UG912 | 2025-05-29 | 2025.1 English

Search in document

Keywords

- BLOCK_SYNTH
- BUFFER_TYPE
- CARRY_REMAP
- CASCADE_HEIGHT
- CELL_BLOAT_FACTOR
- CFGBVS
- CLOCK_BUFFER_TYPE**
- CLOCK_DEDICATED_ROUTE**
- CLOCK_DELAY_GROUP
- CLOCK_EXPANSION_WINDOW
- CLOCK_LOW_FANOUT
- CLOCK_REGION
- CLOCK_ROUTE_GUIDE
- CLOCK_ROOT
- CONFIG_MODE

Table of contents

PDF and attachments

• Opt Design

Related Information

[GCLK_DESKEW](#)

CLOCK_DEDICATED_ROUTE

The CLOCK_DEDICATED_ROUTE property is enabled (TRUE) by default, and ensures that clock resource placement DRCs are considered error conditions that must be corrected prior to routing or bitstream generation. CLOCK_DEDICATED_ROUTE=FALSE downgrades the placement DRC to a warning and lets the Vivado router use fabric routing to connect from a clock-capable IO (CCIO) to a global clock resource such as an MMCM.

CAUTION! Setting CLOCK_DEDICATED_ROUTE to FALSE can result in sub-optimal clock delays, resulting in potential timing violations and other issues.

External user clocks must be brought into the FPGA on differential clock pin pairs called clock-capable inputs (CCIO). These CCIOs provide dedicated, high-speed routing to the internal global and regional clock resources to guarantee timing of various clocking features. Refer to the *7 Series FPGAs Clocking Resources User Guide (UG472)*, or the *UltraScale Architecture Clocking Resources User Guide (UG572)* for more information on clock placement rules.

The CLOCK_DEDICATED_ROUTE property is generally used when it becomes necessary to place clock components in such a way as to take clock routing off of the dedicated clock trees in the target FPGA, and use standard routing channels. If the dedicated routes are not available, setting CLOCK_DEDICATED_ROUTE to FALSE demotes a clock placement DRC from an ERROR to a WARNING when a clock source is placed in a sub-optimal location compared to its load clock buffer.

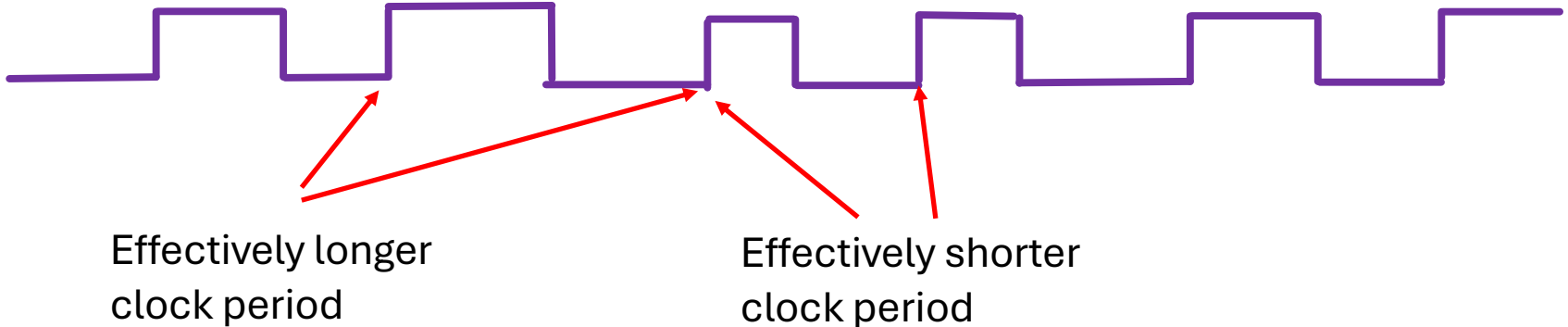
Other Problems you can have with clocks...

- Stable Clock:

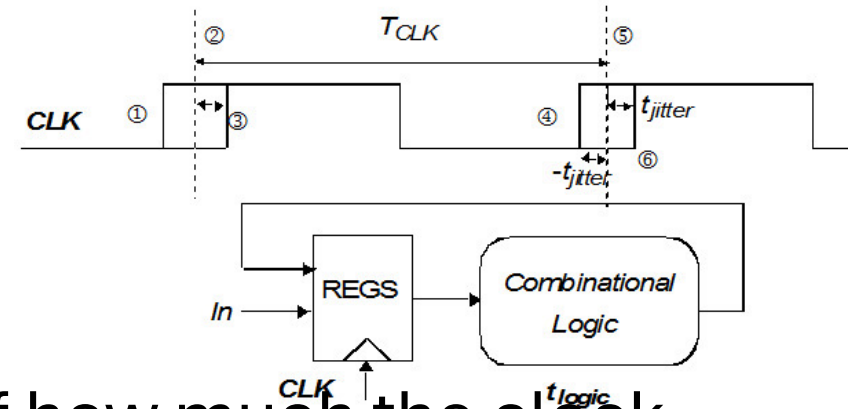


- Jittery Clock:

Clock jitter means cycle-to-cycle you can have larger or shorter clock periods!



Clocks Are Not Perfectly Periodic!:



- Jitter is an approximation of how much the clock period can increase/decrease cycle to cycle:
- Can make it harder to meet timing since it effectively shortens t_{clk} ...
- Even though some cycles might have larger t_{clk} , we, as engineers are required to be big pessimists and plan for the worst case at all times...
- and that affects the setup equation...

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

$$t_{\text{SETUP,Slack}} = t_{\text{CLK}} - 2t_{\text{jitter}} - (t_{\text{PD,reg1}} + t_{\text{PD,logic}} + t_{\text{SETUP,reg2}})$$

Other Problems you can have with clocks...

- 50% Duty Cycle Clock



- Not 50% Duty Cycle Clock



What is the clock period of the 50% duty cycle clock vs the non-50% duty cycle clock?

Trick question: they're the same. I got you.

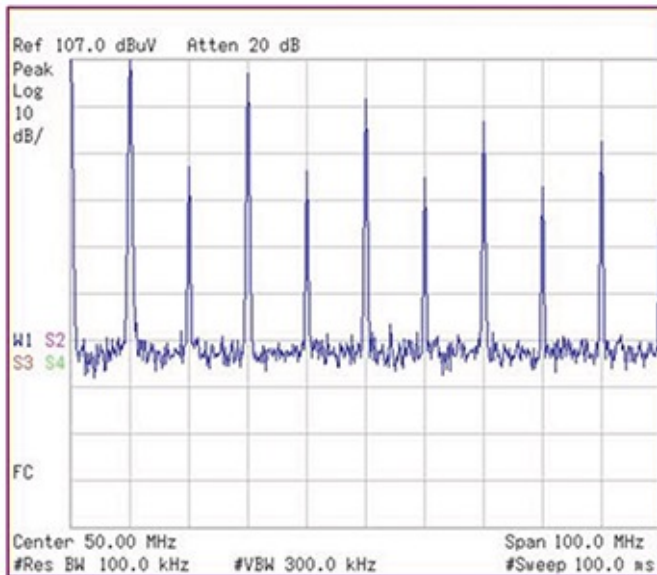
So What? Fifty Shmifty

- Even though the clock period is the same, there can be issues.
- In more advanced designs, you use both clock edges (timing gets more complicated). 50% duty cycle ensures equal time for both halves of logic
- Too far of a deviation from 50% may also break the setup/hold time model of our flip-flops
 - Most flip flops have *minimum-pulse* duration specs...go too low and they won't react appropriately.

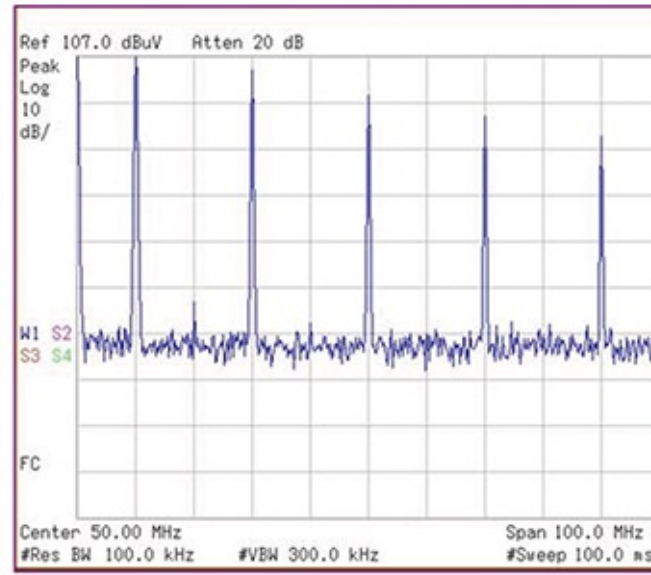
Also 50% is “Pure”



- A pure 50% Duty cycle signal has no odd harmonics*
- Non 50% duty cycle will have odd harmonics, so



49% Duty Cycle

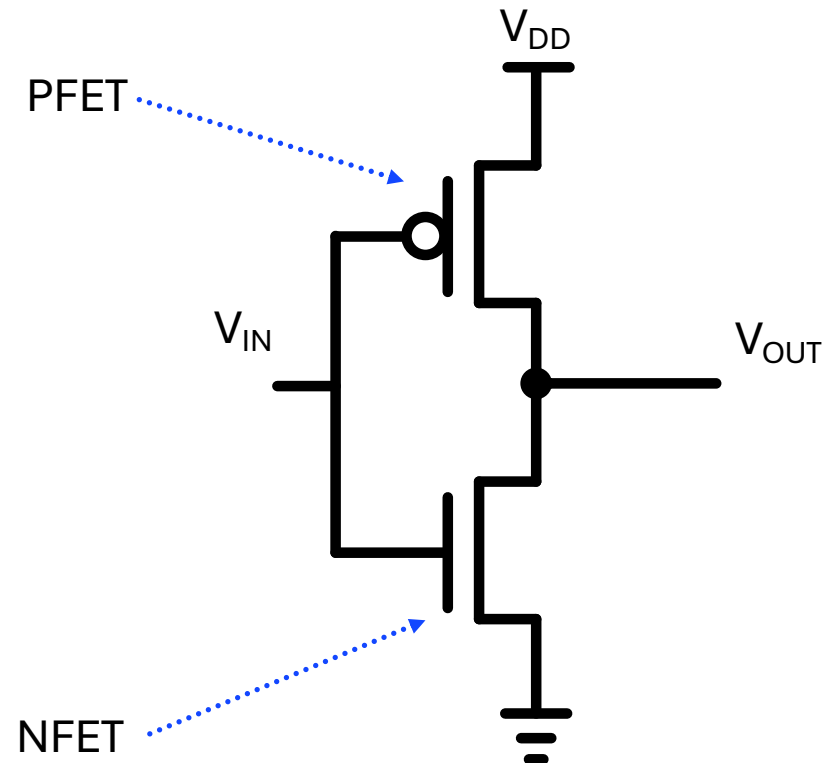


50% Duty Cycle

*Take 6.300

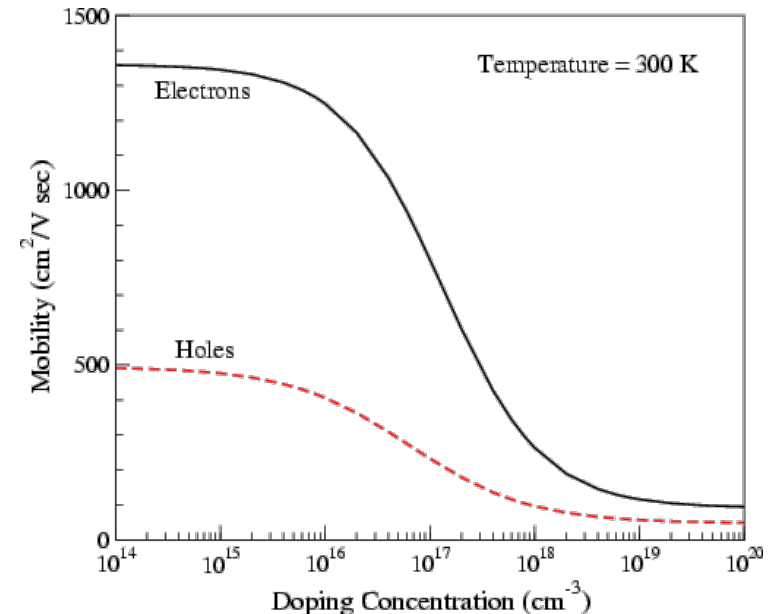
Almost All Modern Digital Design is CMOS:

- PFET in charge of making signal go to 1.
- NFET in charge of making signal go to 0.
- They work in a **Complementary** fashion



Device Physics is Hard* (6.2080, 6.2090, 6.6400, etc...)

- The electrons and holes that N- and P- channel devices use are uneven in mobility so it is very hard to make *exactly* balanced N- and P- channel FETS
- They may pull to 1 and pull to 0 with different “strengths”
- Very easy for circuits to quickly deviate from equal time in on and off state!



<https://www.iue.tuwien.ac.at/phd/park/node30.html>

*but fun and fulfilling

Duty Cycle

- Driving clocks with healthy circuits that get some daily exercise is important

AMD

Technical Information Portal

Sign In

← Search

Vivado Design Suite 7 Series FPGA and Zynq 7000 SoC Libraries Guide (UG953)

UG953

2025-05-29

2025.1 English

Table of contents

PDF and attachments

Search in document

Keywords

Xilinx Parameterized Macros

Unimacros

Functional Categories

Design Elements

BSCANE2

BUFG

BUFGCE

BUFGCE_1

BUFGCTRL

BUFGMUX

BUFGMUX_1

BUFGMUX_CTRL

BUFH

BUFHCE

BUFIO

BUFMR

BUFG

Primitive: Global Clock Simple Buffer

X10654

Introduction

This design element is a high-fanout buffer that connects signals to the global routing resources for low skew distribution of the signal. BUFGs are typically used on clock nets as well other high fanout nets like sets/resets and clock enables.

Port Descriptions

Port	Direction	Width	Function
I	Input	1	Clock input.
O	Output	1	Clock output.

And Other Special Circuits

- Other Special Circuits like Phase-Locked Loops help designs keep healthy, clean 50% duty cycle clocks with low jitter
- They can also be used to make clocks from other clocks!

How to Make Frequencies and Clocks

Where do they come from???

Goal: Use as few clock domains as possible

Suppose we have a reference clk at frequency f and we want signals at $f/2$, $f/4$, $f/8$, etc.:????

```
logic clk2,clk4,clk8,clk16;

always_ff @(posedge clk)begin
|   clk2 <= ~clk2;
end
always_ff @(posedge clk2)begin
|   clk4 <= ~clk4;
end
always_ff @(posedge clk4)begin
|   clk8 <= ~clk16;
end
always_ff @(posedge clk8)begin
|   clk16 <= ~clk16;
end
```

Goal: use as few clock domains as possible

Suppose we have a reference clk at frequency f and we want signals at $f/2$, $f/4$, $f/8$, etc.:????

```
logic clk2,clk4,clk8,clk16;

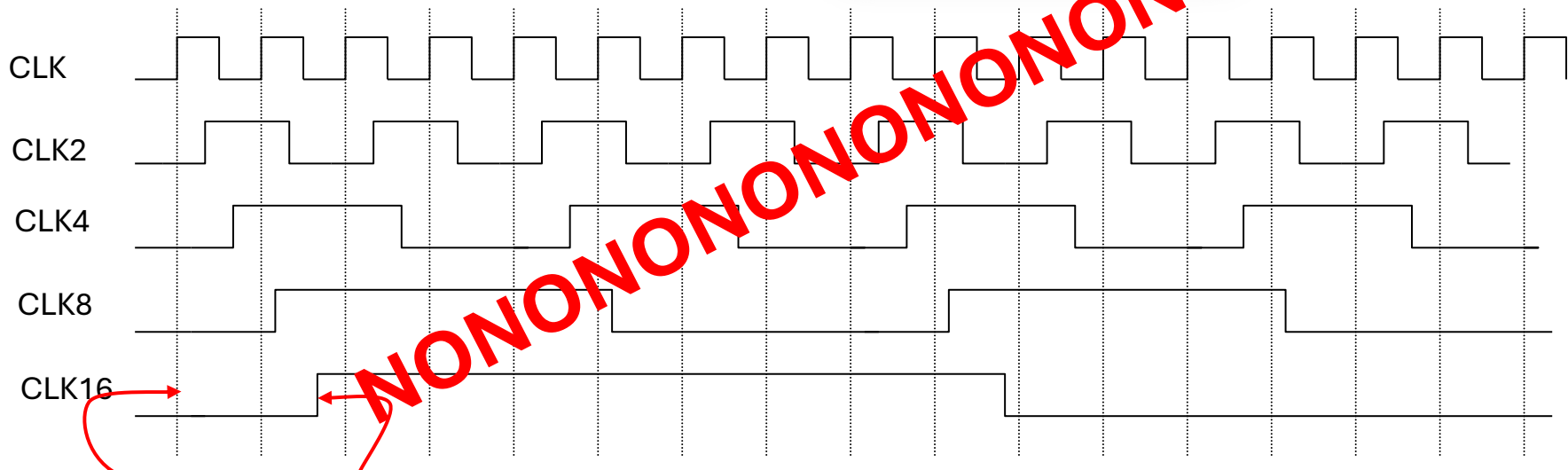
always_ff @(posedge clk)begin
    clk2 <= ~clk2;
end

always_ff @(posedge clk2)begin
    clk4 <= ~clk4;
end

always_ff @(posedge clk4)begin
    clk8 <= ~clk16;
end

always_ff @(posedge clk8)begin
    clk16 <= ~clk16;
end
```

***No! don't do
it this way, you'll
never make a
dime***



Very hard to have synchronous communication between
clk and clk16 domains... Can lead to lots of timing
violations!

Goal: Use as few clock domains as possible

Suppose we have a reference clk at frequency f and we want signals at $f/2$, $f/4$, $f/8$, etc.:????

```
logic [3:0] count;
always_ff @(posedge clk) begin
    count <= count + 1;    // counts 0..15
end
logic enb2, enb4, enb8, enb16;
assign enb2 = (count[0] == 1'b1);
assign enb4 = (count[1:0] == 2'b11);
assign enb8 = (count[2:0] == 3'b111);
assign enb16 = (count[3:0] == 4'b1111);

always_ff @(posedge clk) begin
    if (enb2) begin
        // get here every 2nd cycle
    end
end
```

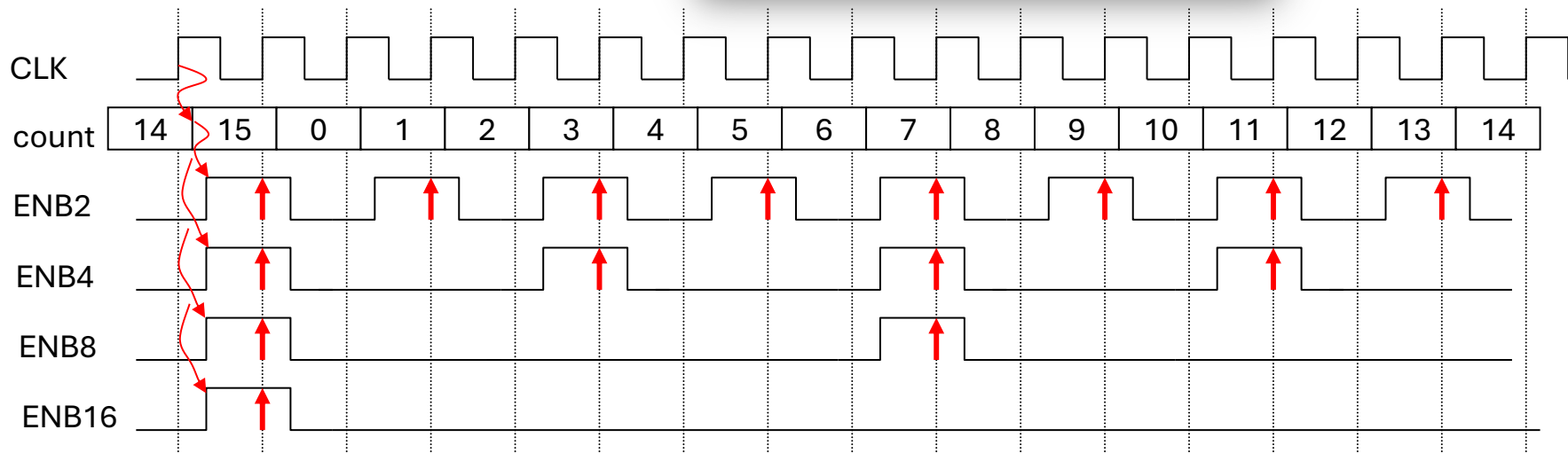
Solution: One clock, Many enables

- Use one (high speed) clock, but create enable signals to select a subset of the edges to use for a particular piece of sequential logic (much easier on timing requirements)

```
logic [3:0] count;
always_ff @(posedge clk) begin
    count <= count + 1; // counts 0..15
end
logic enb2, enb4, enb8, enb16;
assign enb2 = (count[0] == 1'b1);
assign enb4 = (count[1:0] == 2'b11);
assign enb8 = (count[2:0] == 3'b111);
assign enb16 = (count[3:0] == 4'b1111);

always_ff @(posedge clk) begin
    if (enb2) begin
        // get here every 2nd cycle
    end
end
```

Yes! A good idea that will lead to good outcomes

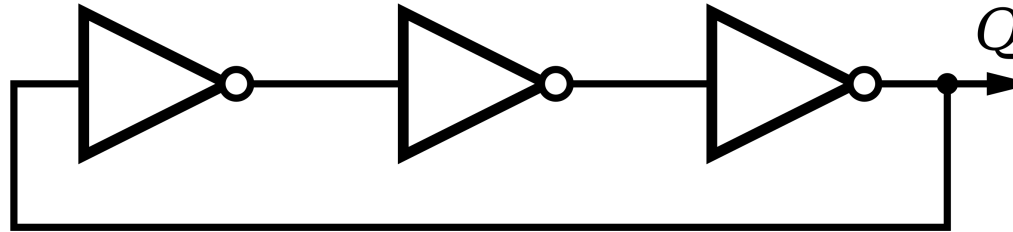


= clock edge selected by enable signal

How to Make Frequencies and Clocks

Where do they come from???

Where do we get frequencies?



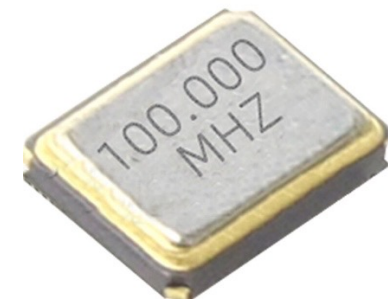
- Particular combinational circuits that are fed back onto themselves so that they cannot be stable can be made to form oscillators.
- The ring oscillator above is a classic example.
- There is no stable set of output states so this circuit perpetually oscillates.
- Period of oscillation is based on the delay of each element

Where do we get frequencies?

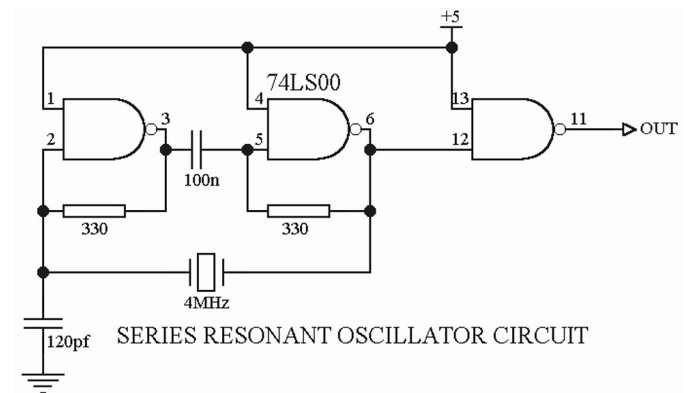
- Most frequencies come from Crystal Oscillators made of quartz
- Equivalent to very High-Q RLC tank circuits
- Incorporate into circuit like that to right and boom, you've got a square wave of some specified frequency dependent largely on the crystal



16MHz Crystal



100MHz Crystal

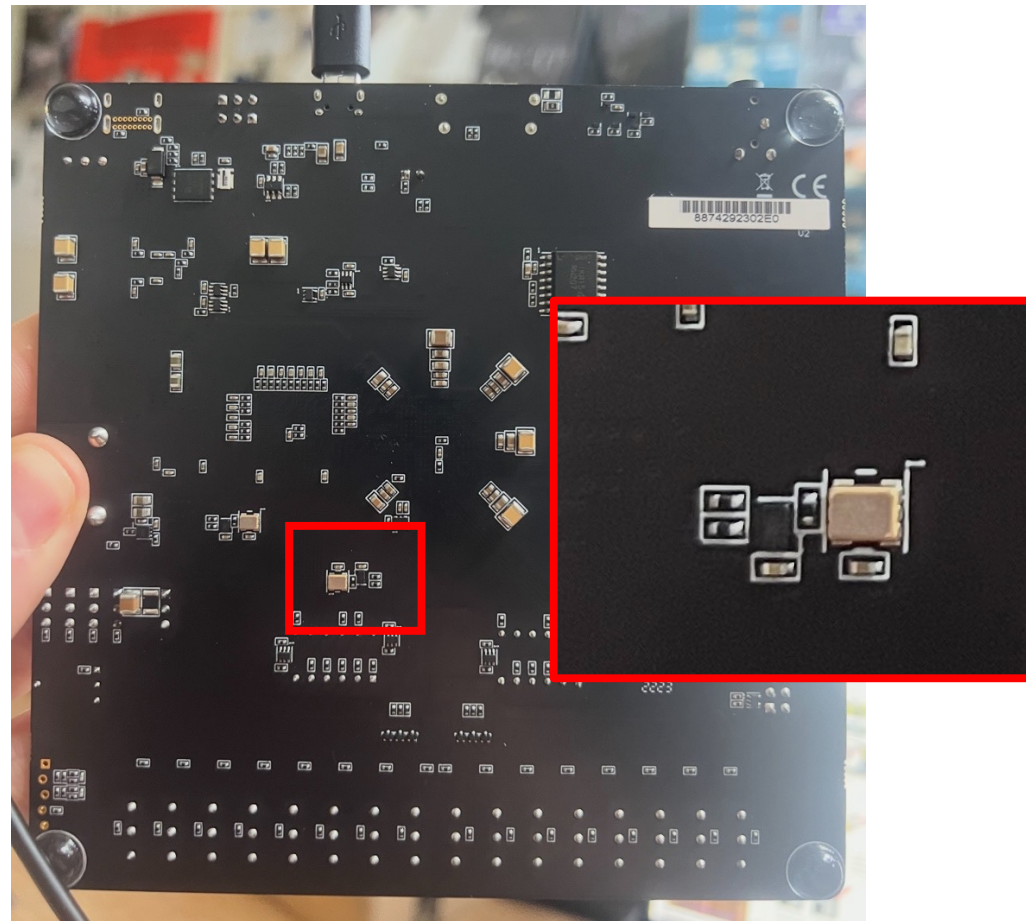


<http://www.z80.info/uexosc.htm>

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

From XDC file:

```
# clk_100mhz is from the 100 MHz oscillator on Urbana Board  
set_property -dict {PACKAGE_PIN N15 IOSTANDARD LVCMOS33} [get_ports {clk_100mhz}]  
create_clock -add -name gclk -period 10.000 -waveform {0 5} [get_ports {clk_100mhz}]
```



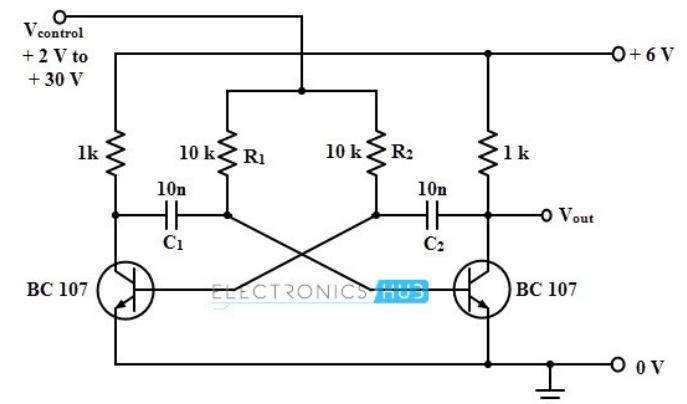
High Frequencies

- Very hard to get a crystal oscillator to operate above ~200 MHz (7th harmonic of resonance of crystal itself, which usually is limited to about 30 MHz due to fabrication limitations)
- Where does the 2.33 GHz clock of my iPhone come from then?
- Frequency Multipliers!

Voltage Controlled Oscillator

- It is very easy to make voltage-controlled oscillators that run up to 1GHz or more.
 - Low voltage circuit oscillates at low frequency
 - Higher voltage $\rightarrow$ higher frequency oscillation

- Block Diagram:



*A simple VCO (not type found in FPGA). Same general idea... You use **FEEDBACK** to make a circuit unstable but guide that instability into productive oscillations with careful tuning and external control!*

Voltage Controlled Oscillator

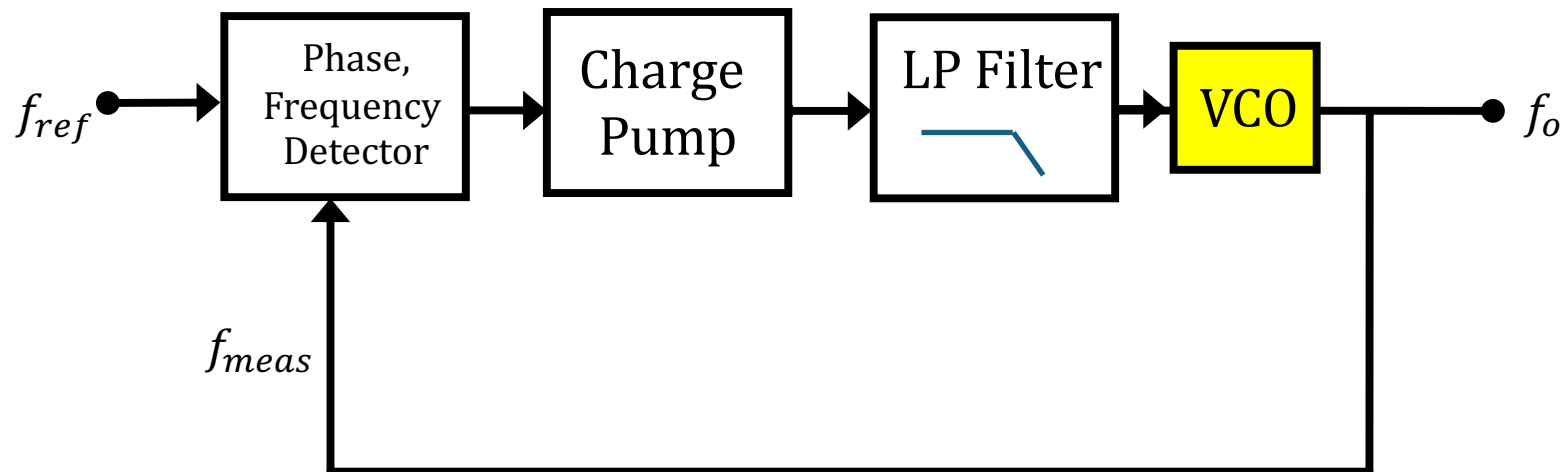
- It is very easy to make voltage-controlled oscillators that run up to several GHz or more.
- Why don't we just:



- Pick the voltage V_i that is needed to get the frequency we want f_o ?
- That's gotta be ok right?

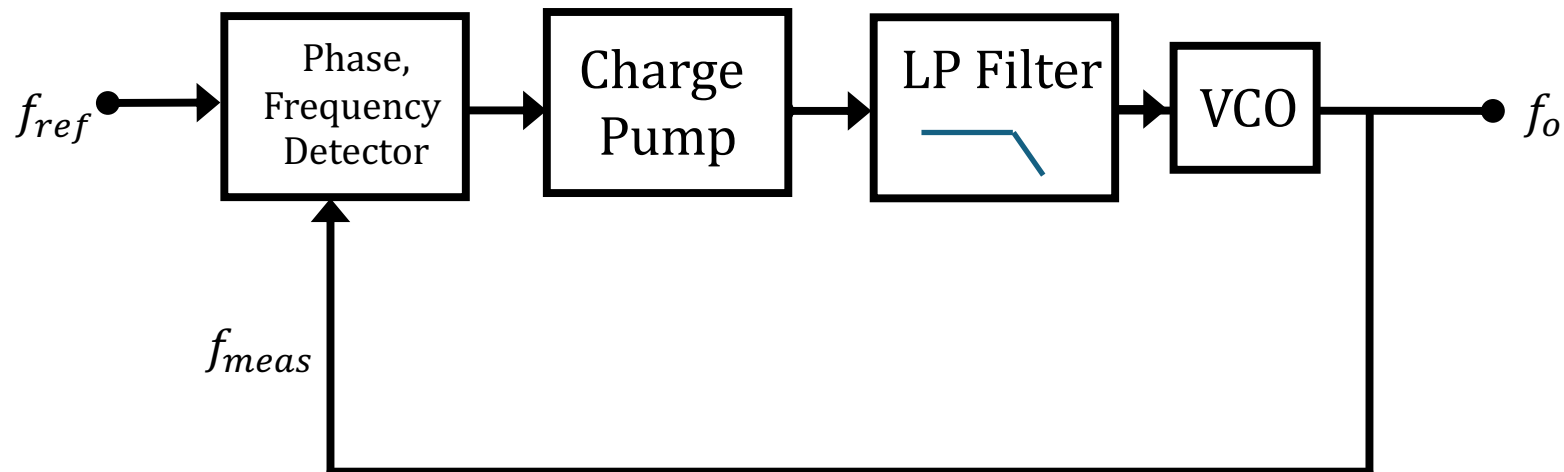
Phase Locked Loop (PLL)

- Place the unstable, but capable, VCO in a feedback loop.



Phase Locked Loop (PLL)

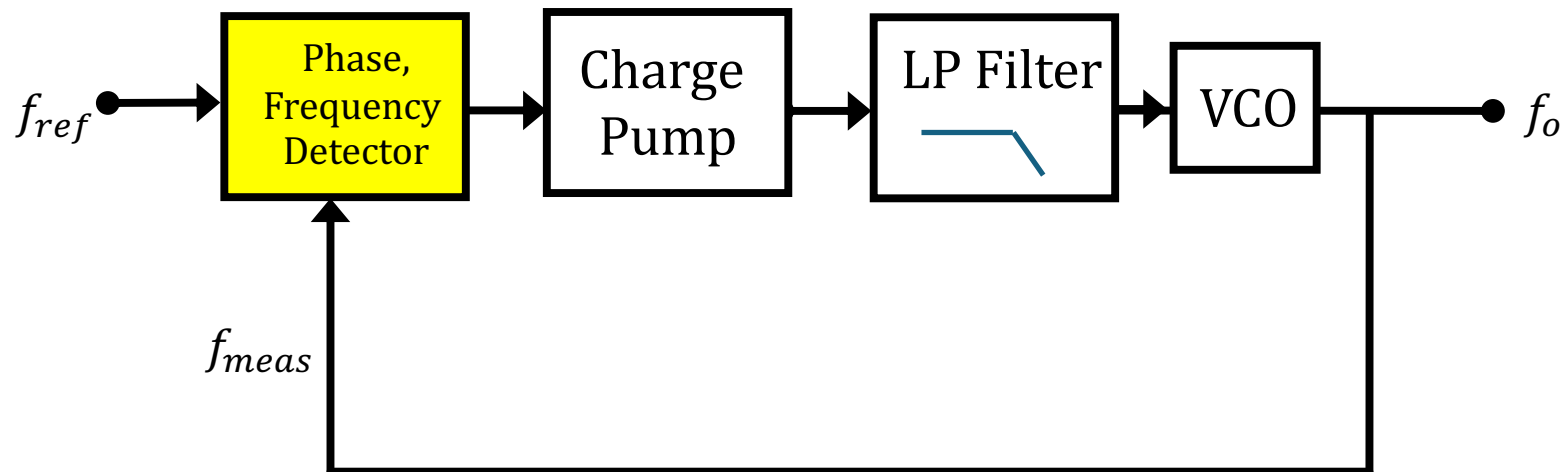
- A PLL is a circuit that can track an input frequency f_{ref} of a system and reproduce it at the output f_o



GOTTA GO THROUGH EACH PART

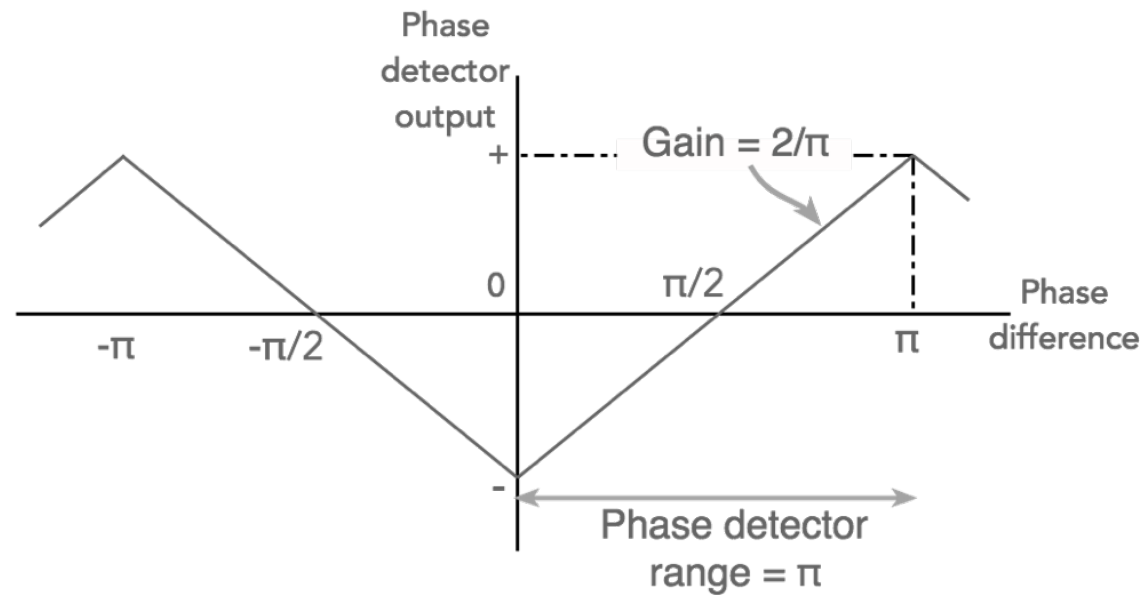
Phase Locked Loop (PLL)

- The Phase, Frequency Detector

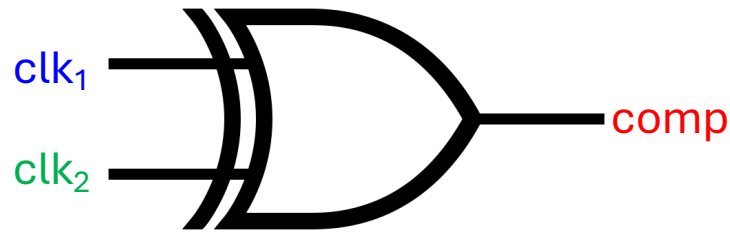


Phase Detector

- Can be a simple XOR, XNOR gate
 - Low-pass the output



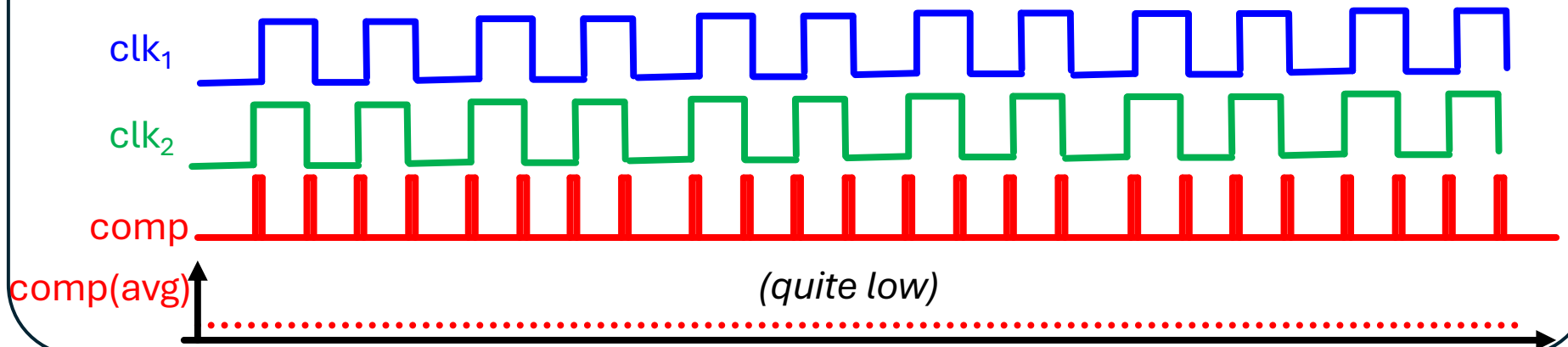
Phase Detector...



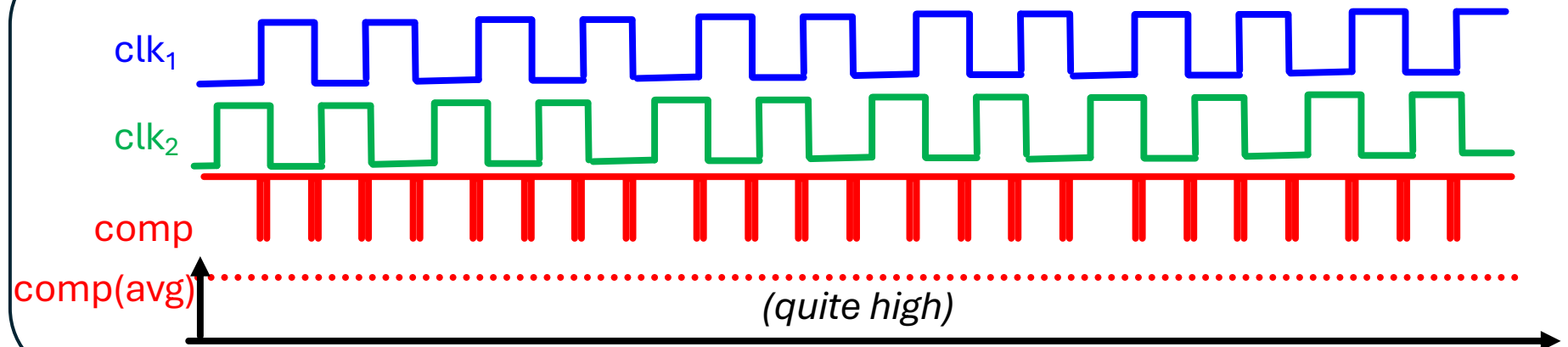
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

XOR

Almost In Phase Signals:

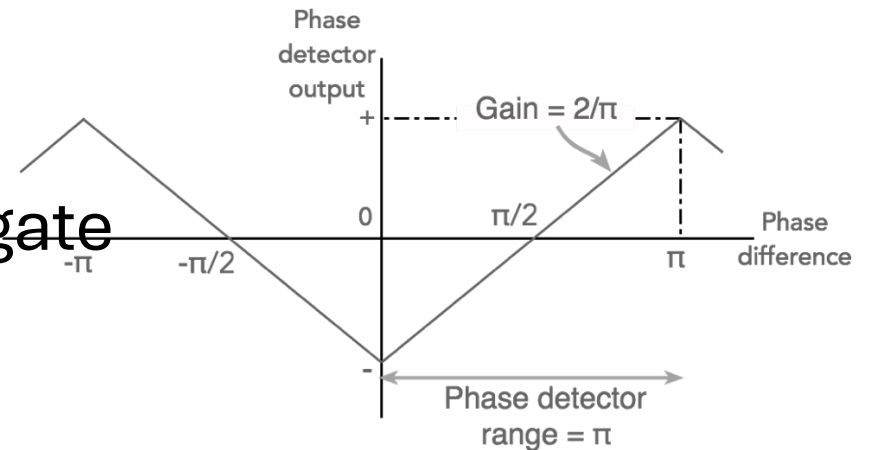


Almost Out-of-Phase Signals:



Phase Detector

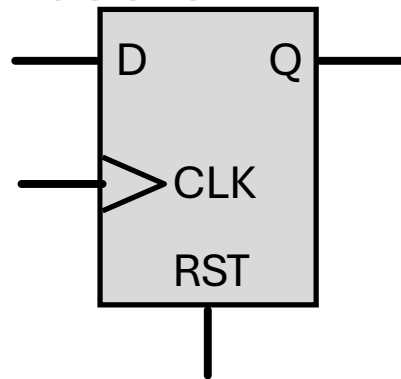
- Can be a simple XOR, XNOR gate
 - Low-pass the output



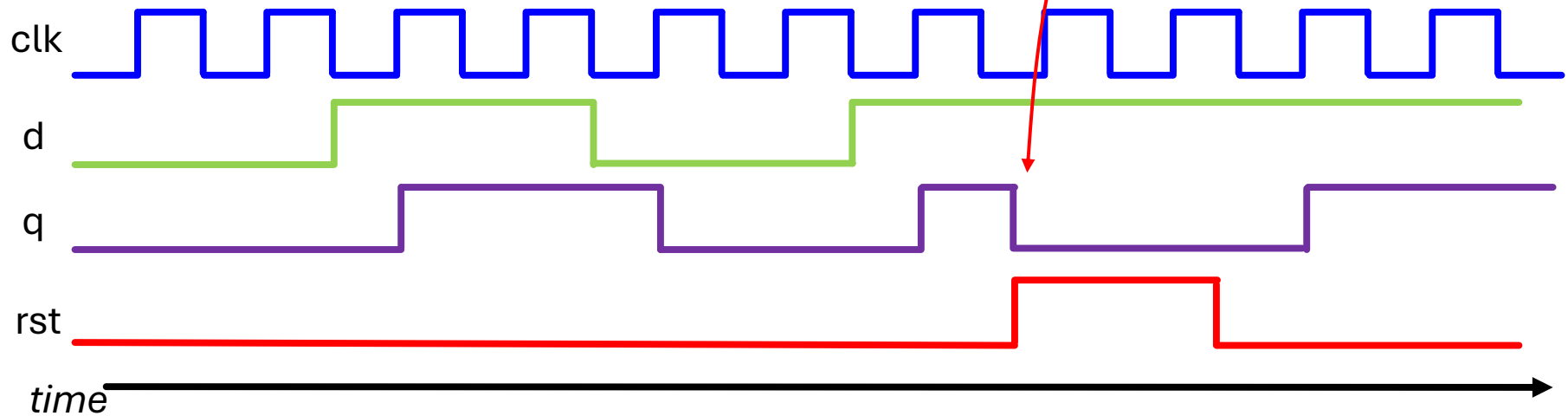
- If near the desired frequency already this can work...if it is too far out, it won't and can be very unreliable since phase and frequency are related but not quite the same thing, it will lock onto harmonics, etc...
- For frequency we instead use a **PFD**:
 - Phase/Frequency Detector:

Phase-Frequency Detection

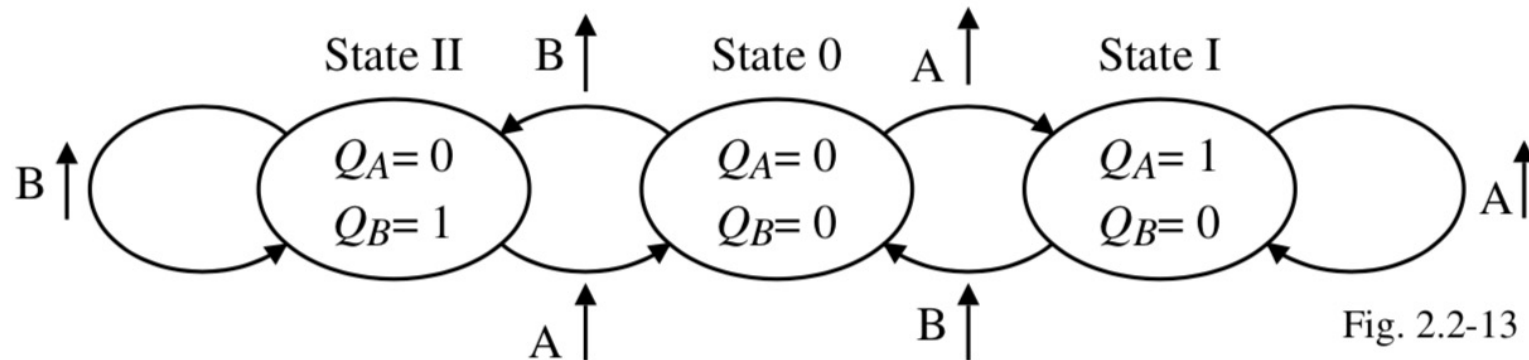
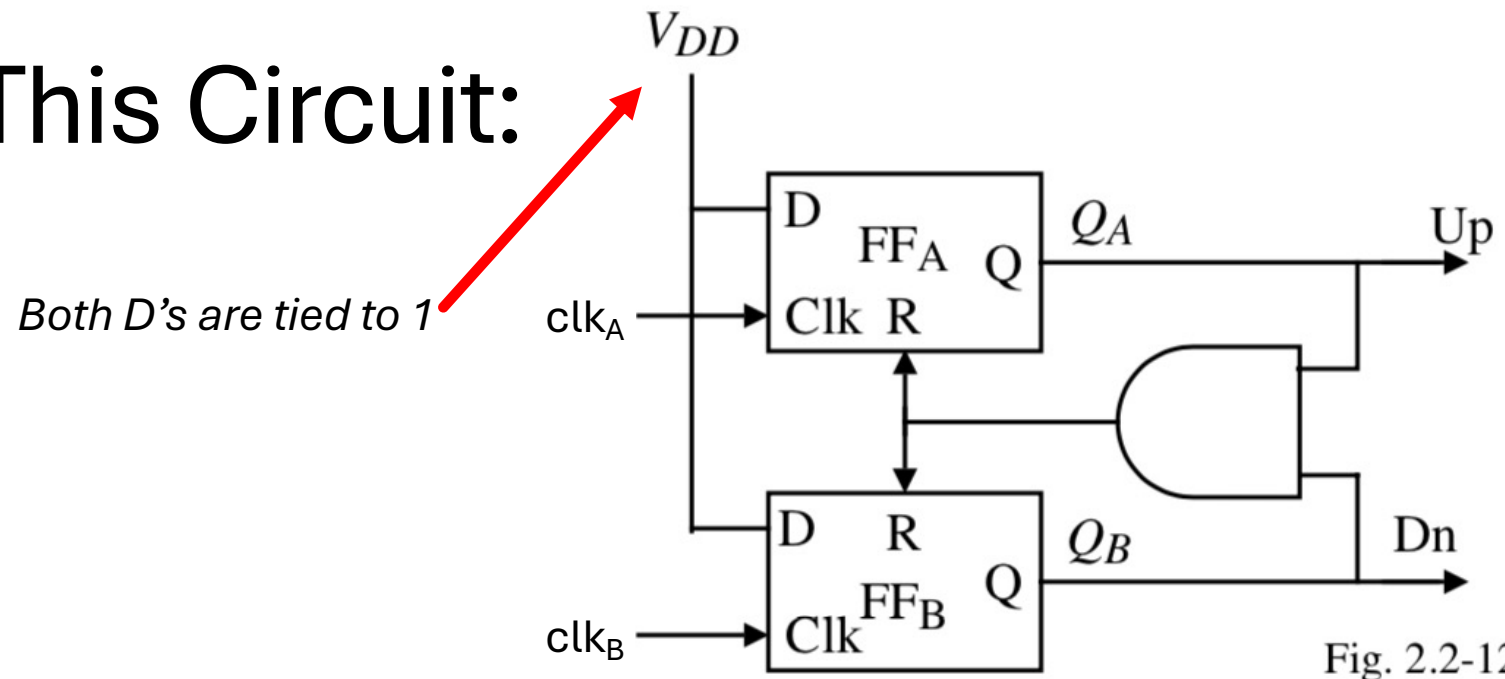
- Built around Flip-Flops with Asynchronous Resets:



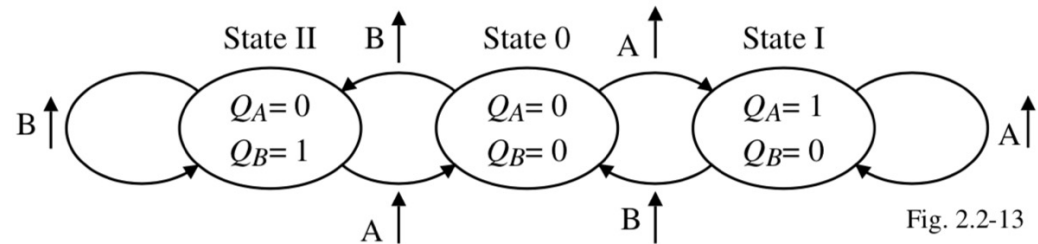
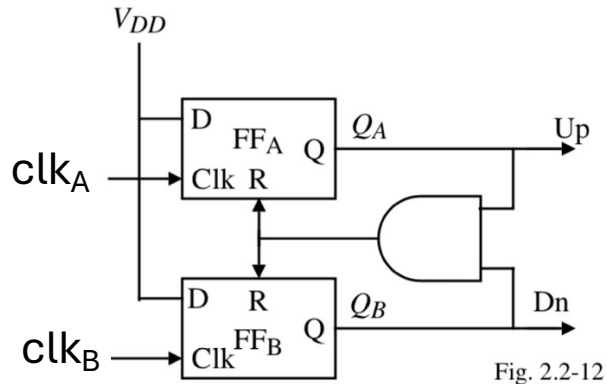
```
always_ff @(posedge clk or posedge rst)begin
    if (rst)begin
        q <= 0; //immediate reset to 0
    end else begin
        q <= d; //update on clock edge
    end
end
```



Build This Circuit:



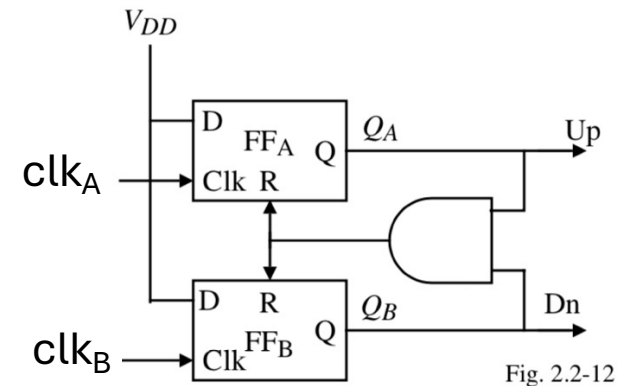
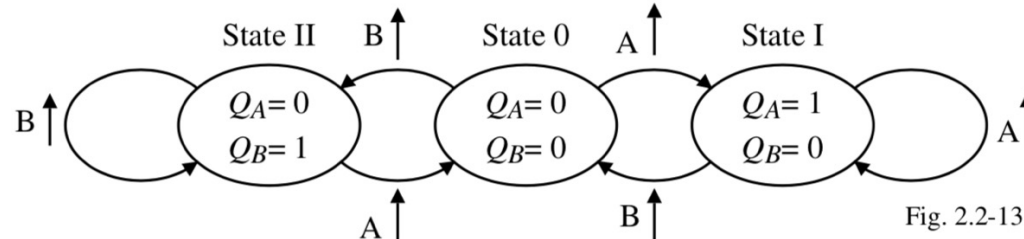
Phase Frequency Detection



- Circuit starts in State 0 (middle) (“Up” off, “Dn” off)
- First clock to rise will move into one of two side states:
 - clk_A ?: State 1 (“Up” on, “Dn” off)
 - clk_B ?: state II (“Dn” on, “Up” off)
- When loser finally rises, circuit resets to State 0 (middle)
- Competition starts again

1.pallen.ece.gatech.edu/Academic/ECE_6440/Summer_2003/L070-DPLL(2UP).pdf

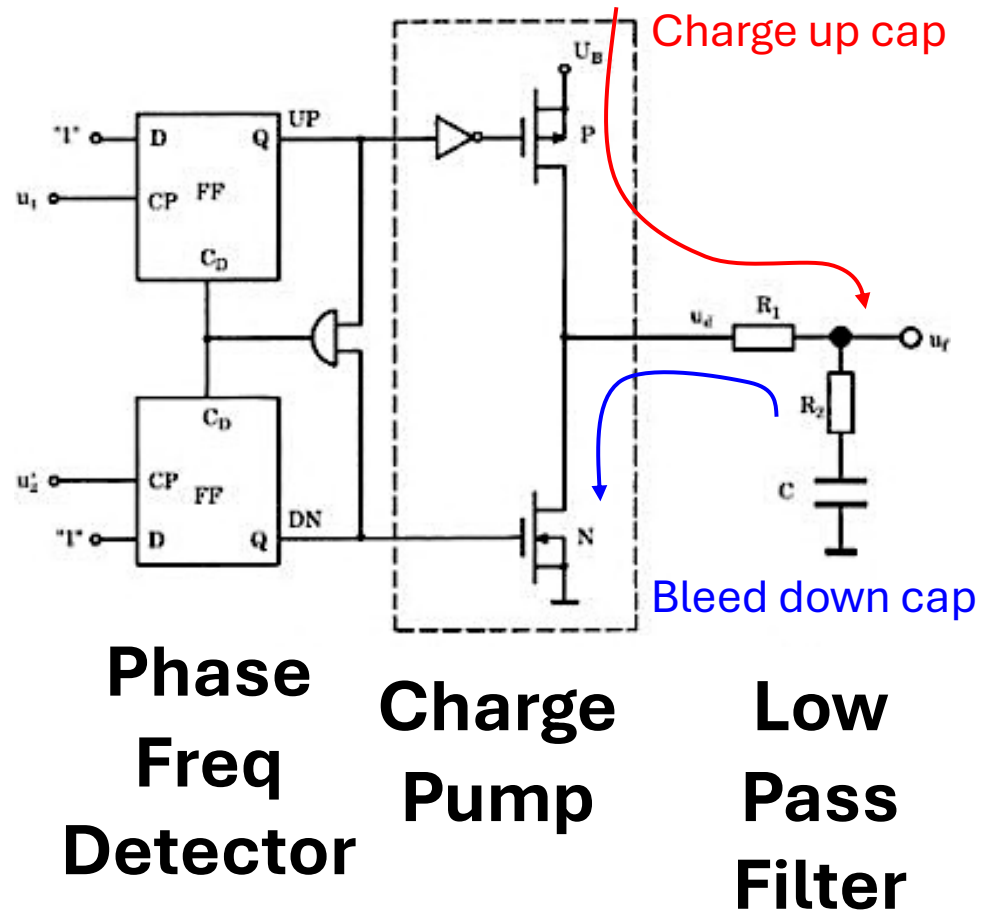
Phase Frequency Detection



- If clk_A is higher frequency, it will, on average, win more races and the FSM will spend more time in State I
- If clk_B is higher frequency, it will, on average, win more races and the FSM will spend more time in State II
- The closer they are, the more balanced the State I and State II time will be
- The greater their difference, the greater the on-average discrepancy

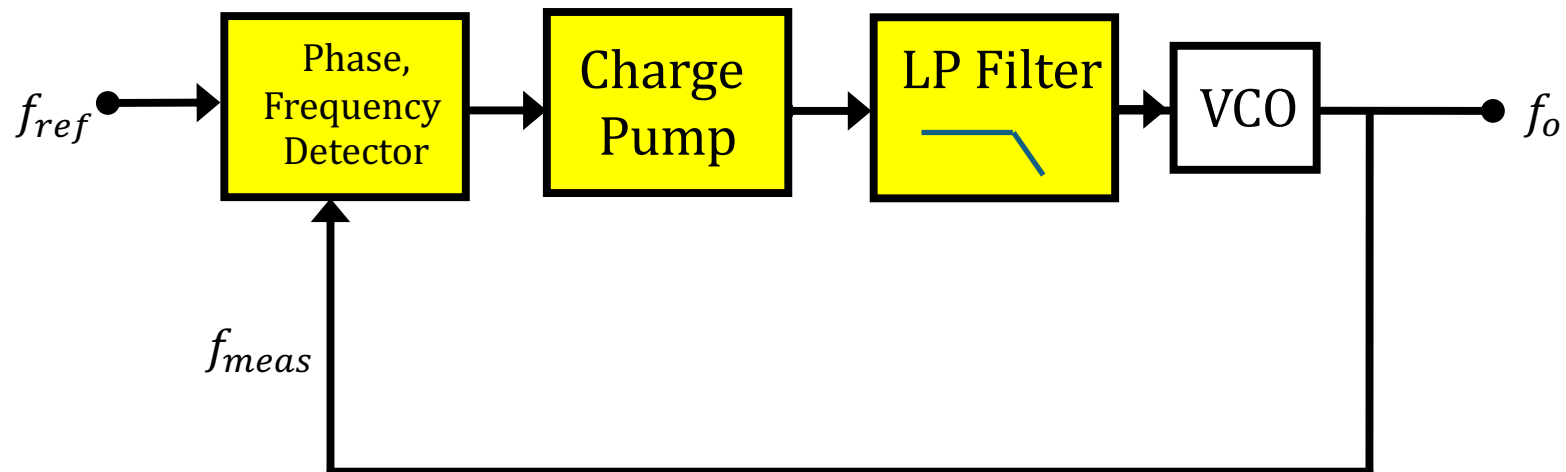
UP and DN control a capacitor voltage

- If you're in State I:
 - Increase voltage on capacitor
- If you're in State II:
 - Decrease voltage on capacitor
- The voltage that builds up will be tightly related to how different these two clocks are



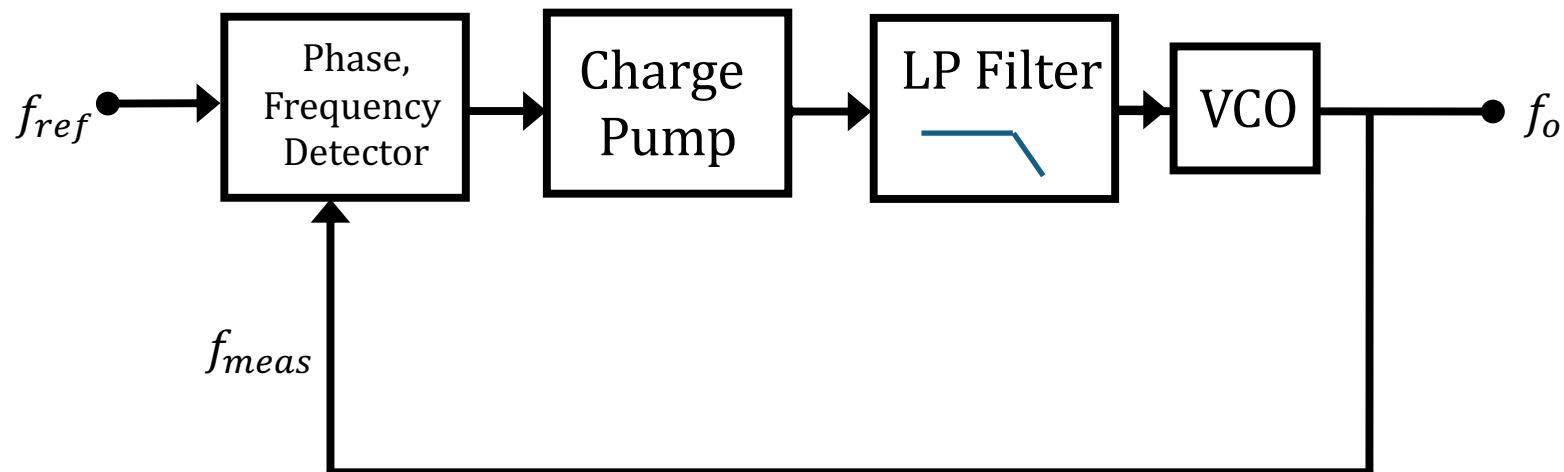
Phase Locked Loop (PLL)

- So that's those three pieces

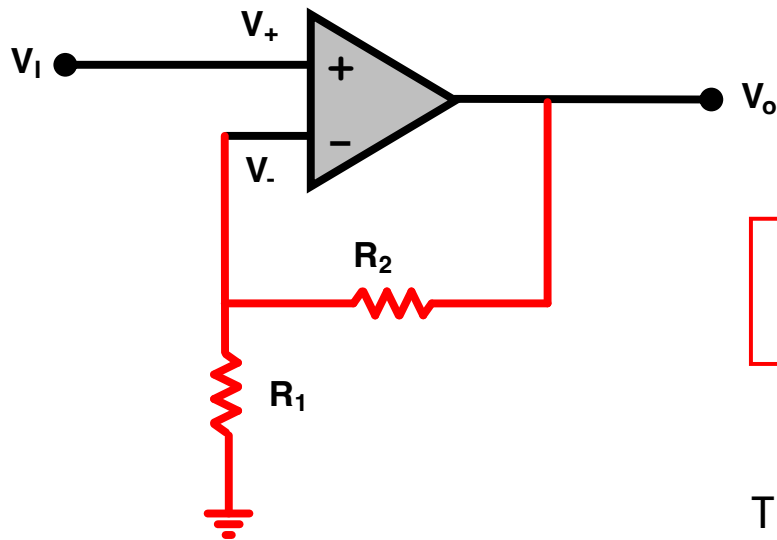


Phase Locked Loop (PLL)

- So this circuit can make $f_0 = f_{ref}$ and there's lots of uses for this (frequency locking, etc...)
 - *But that doesn't help us! I want higher frequencies. You said we could do that.*
 - *I feel betrayed and angry and don't know how to express that emotion in a healthy way*
- **How can we make a higher frequency?**



Use Resistors in Voltage Divider in Feedback Path!



- A voltage divider in feedback path gives us voltage gain!

$$K = \frac{1}{1 - p + G} \quad p \approx 0.9999 \text{ means} \quad K = \frac{1}{G}$$

$$G = \frac{R_1}{R_1 + R_2}$$

The gain A_v of this circuit is therefore:

$$A_v = \frac{R_1 + R_2}{R_1}$$

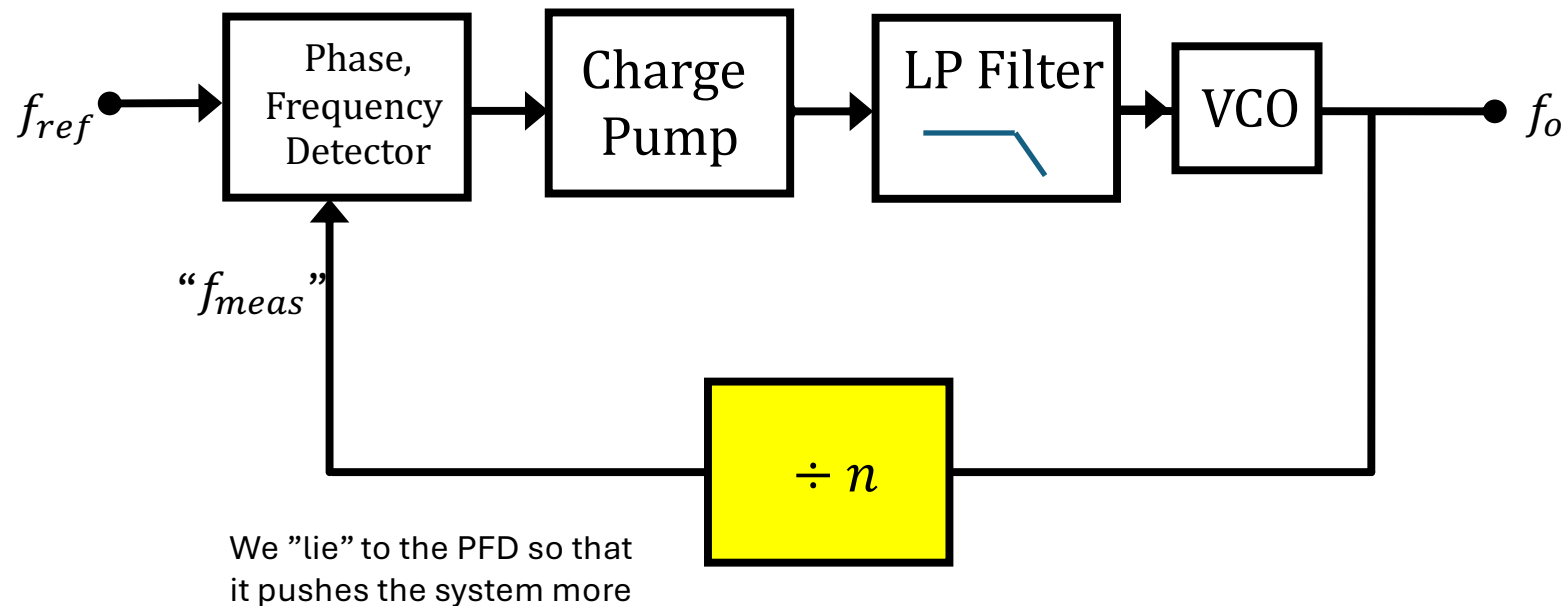
The gain of a “non-inverting amplifier”

$$V_- = V_o \frac{R_1}{R_1 + R_2}$$

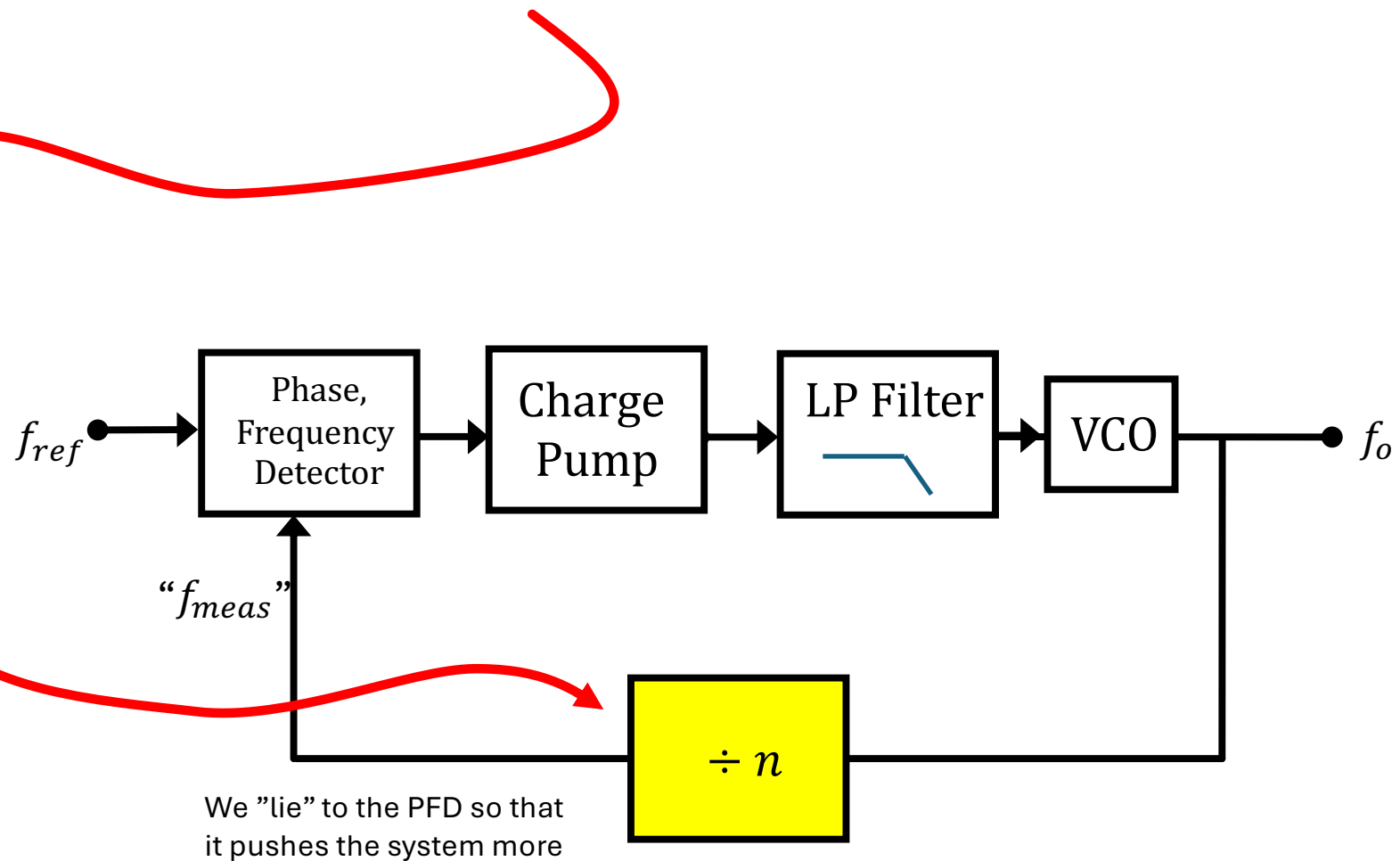
Same Idea with Phase Locked Loops!

Use a Clock Divider in Feedback Path!

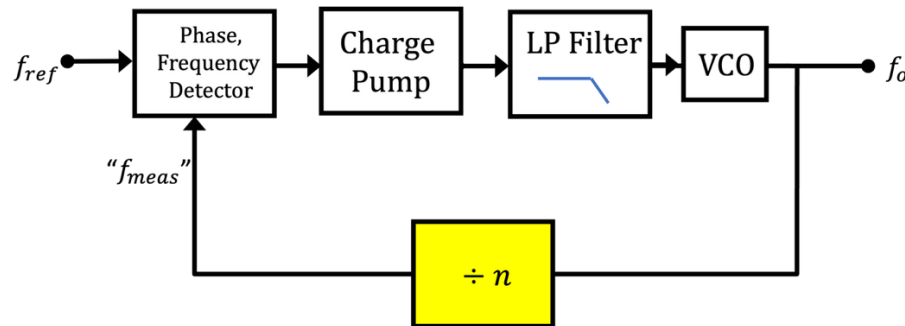
- A clock divider in feedback path gives us clock gain!
- By lying to the circuit, you can emotionally manipulate the system as a whole to make higher, stable frequencies.



Where is *that* coming from?



Can actually just use flip-flops to divide

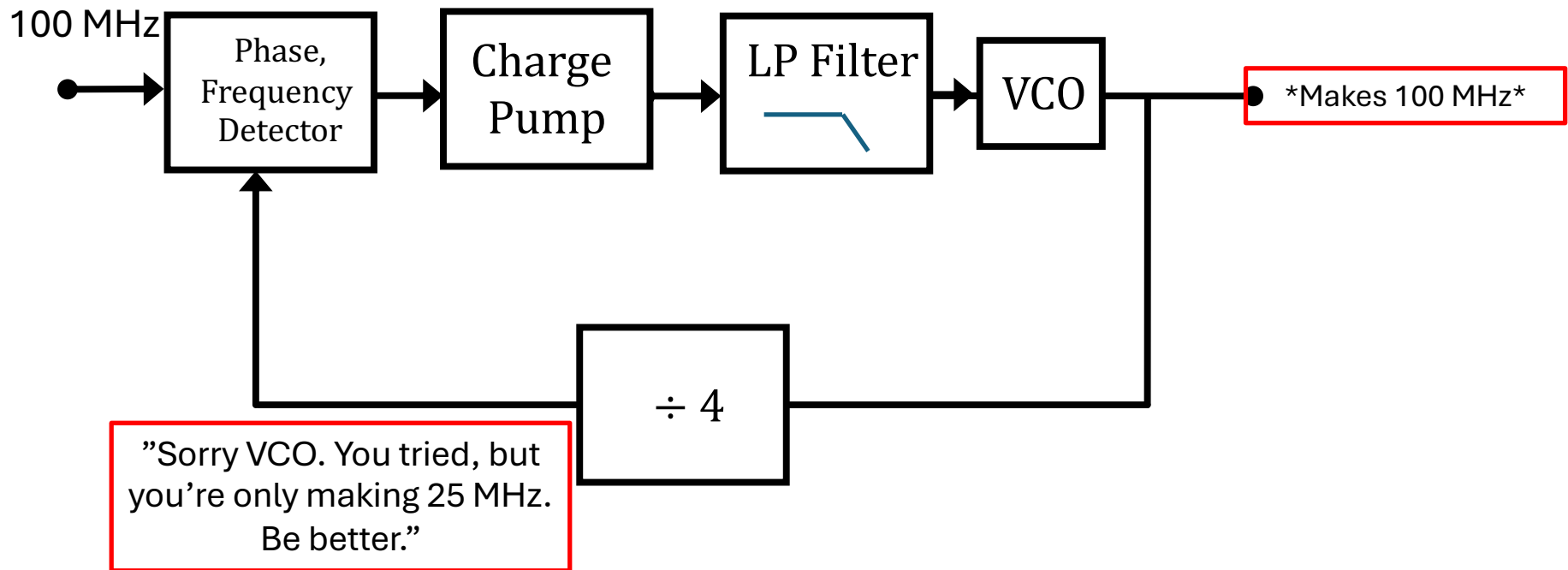


- The PFD will handle the phase mismatch that arises from the propagation delays naturally

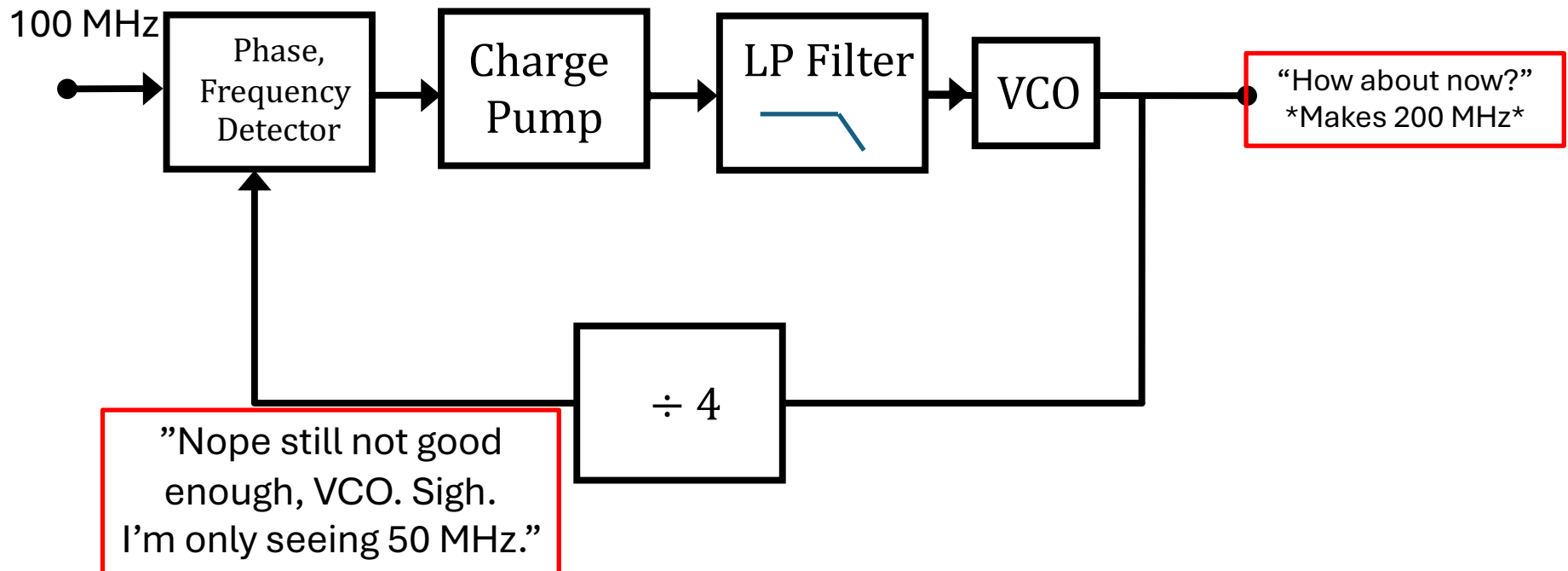
```
logic clk2,clk4,clk8,clk16;

always_ff @(posedge clk)begin
|   clk2 <= ~clk2;
end
always_ff @(posedge clk2)begin
|   clk4 <= ~clk4;
end
always_ff @(posedge clk4)begin
|   clk8 <= ~clk8;
end
always_ff @(posedge clk8)begin
|   clk16 <= ~clk16;
end
```

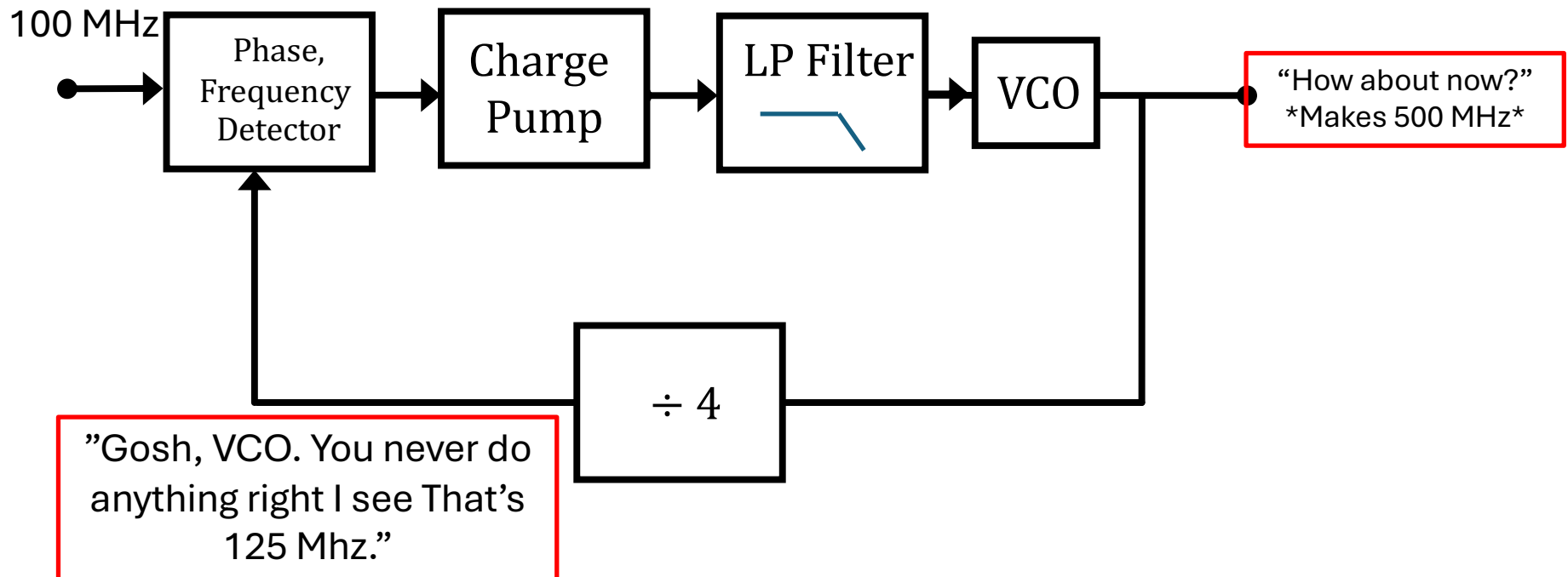
Example of a Toxic Relationship Circuit



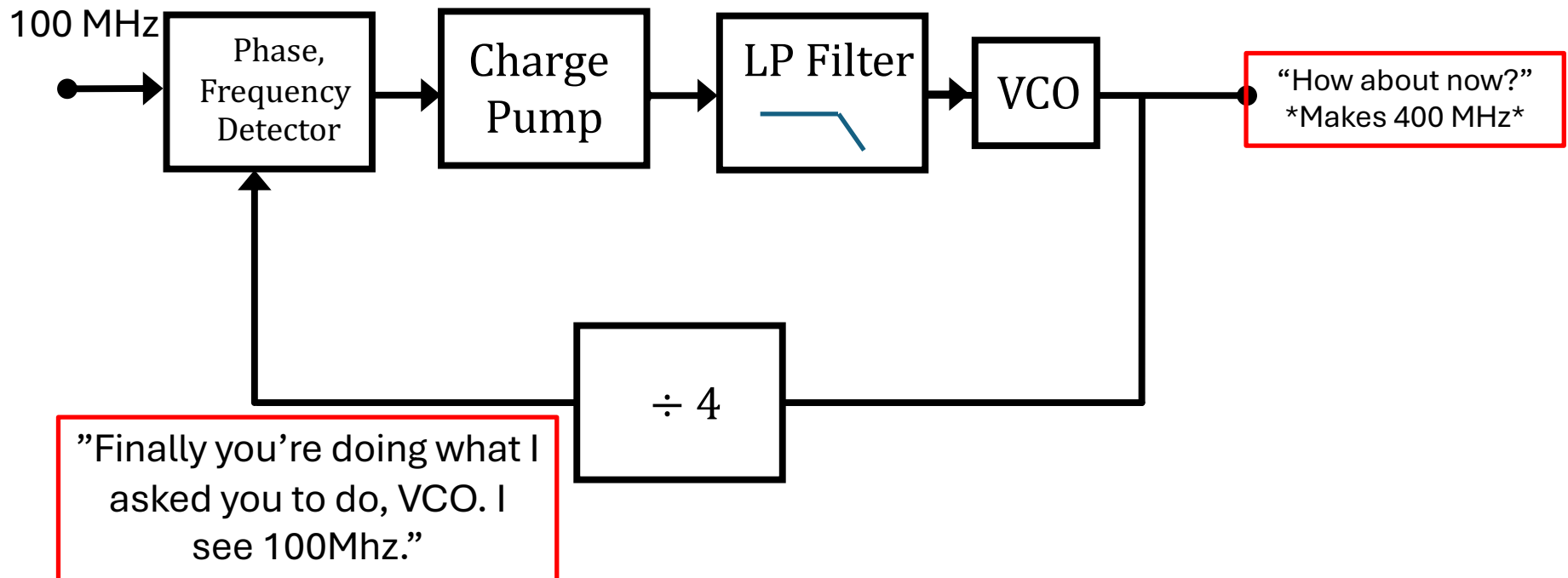
Example of a Toxic Relationship Circuit



Example of a Toxic Relationship Circuit



Example of a Toxic Relationship Circuit

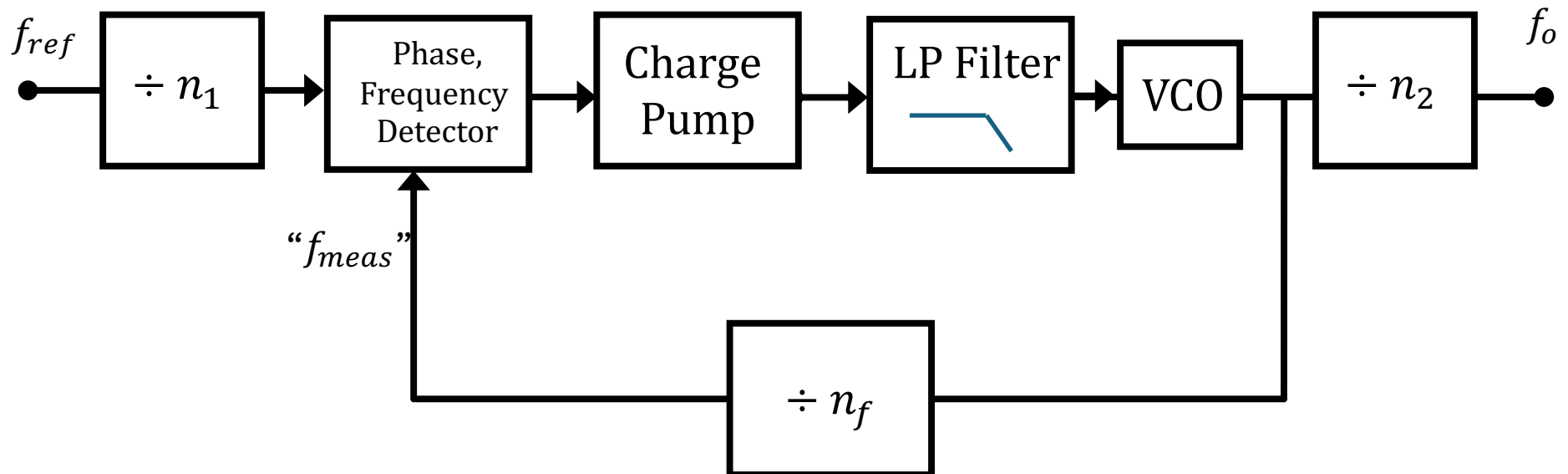


Conclusions

- Do not be in emotionally manipulative friendships or work relationships, or regular relationships, or situationships. You are better than that.
- Do not emotionally manipulate other people.
- Circuits are not people so it is probably ok and can be beneficial.
- Negative fractional feedback in PLLs allows us to generate higher frequencies that are stable since they are referenced to lower (stable) frequency sources.

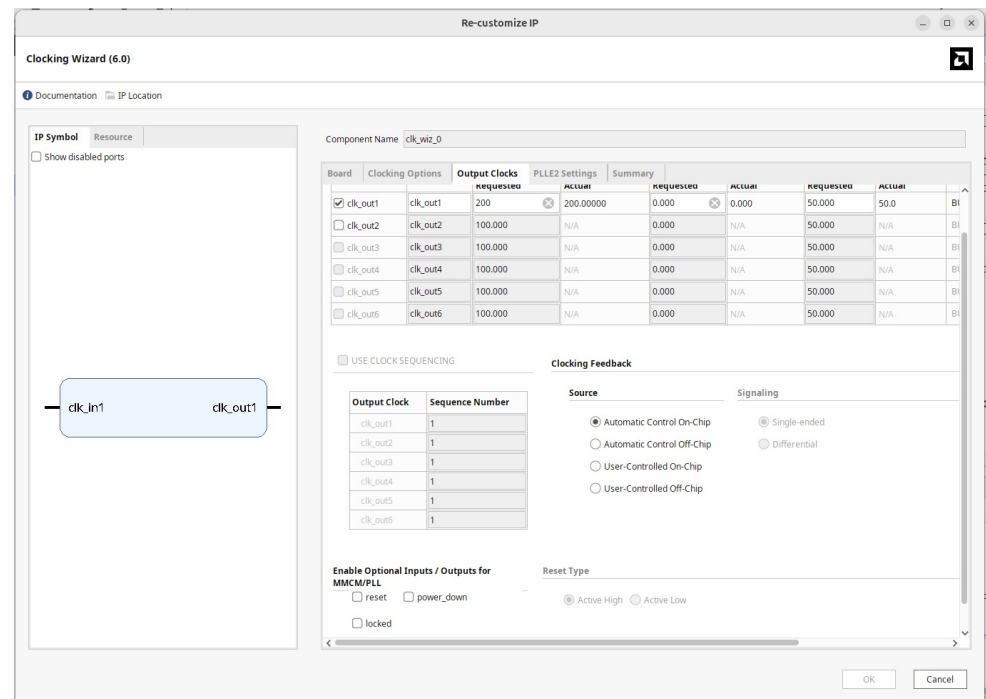
Add a Pre- and Post- Divider for Flex

- There will often be additional stages of frequency division to improve flexibility and range



Do we/I have to build this?

- Yes/No. Circuit is too sensitive for us to specify details in Verilog on an FPGA
- Most FPGAs have pre-built “hardened” Mixed-Mode Clock Management (MMCM) units and PLLs to deploy
- You can configure them using the datasheet and/or “wizards”



Our FPGA?

- It has **five** MMCM/PLL units spaced throughout the FPGA.
- Each unit can make up to 7 somewhat independent output clocks*
 - Lots of caveats with how independent these can be, but often times you don't want completely independent clocks anyways...you want a bunch of clocks that are tightly related in phase/clean frequency divisions which is what this can do.

In Week 04

- We'll build HDMI video from scratch.
- For 720p we'll need:
 - a clock at 74.25 MHz (for the pixels)
 - A clock at 371.25 MHz (for the bits of the pixels to be sent serially)
 - We'll use this clock along with a device that is built to run using always @(posedge clk or negedge clk) to get 742.25 MHz of data out to drive the 720p data.

So to Make 74.25 MHz and 371.25 MHz?

- Instance of MMCM clock manager:

This is a hardened specialized component on your FPGA

```
MMCME2_ADV
#(.BANDWIDTH          ("OPTIMIZED"),
  .CLKOUT4_CASCADE    ("FALSE"),
  .COMPENSATION        ("ZHOLD"),
  .STARTUP_WAIT        ("FALSE"),
  .DIVCLK_DIVIDE       (5),
  .CLKFBOUT_MULT_F     (37.125),
  .CLKFBOUT_PHASE      (0.000),
  .CLKFBOUT_USE_FINE_PS ("FALSE"),
  .CLKOUT0_DIVIDE_F    (10.000),
  .CLKOUT0_PHASE       (0.000),
  .CLKOUT0_DUTY_CYCLE  (0.500),
  .CLKOUT0_USE_FINE_PS ("FALSE"),
  .CLKOUT1_DIVIDE      (2),
  .CLKOUT1_PHASE       (0.000),
  .CLKOUT1_DUTY_CYCLE  (0.500),
  .CLKOUT1_USE_FINE_PS ("FALSE"),
  .CLKIN1_PERIOD       (10.000))
mmcm_adv_inst
// Output clocks
```

2. Divides by 5

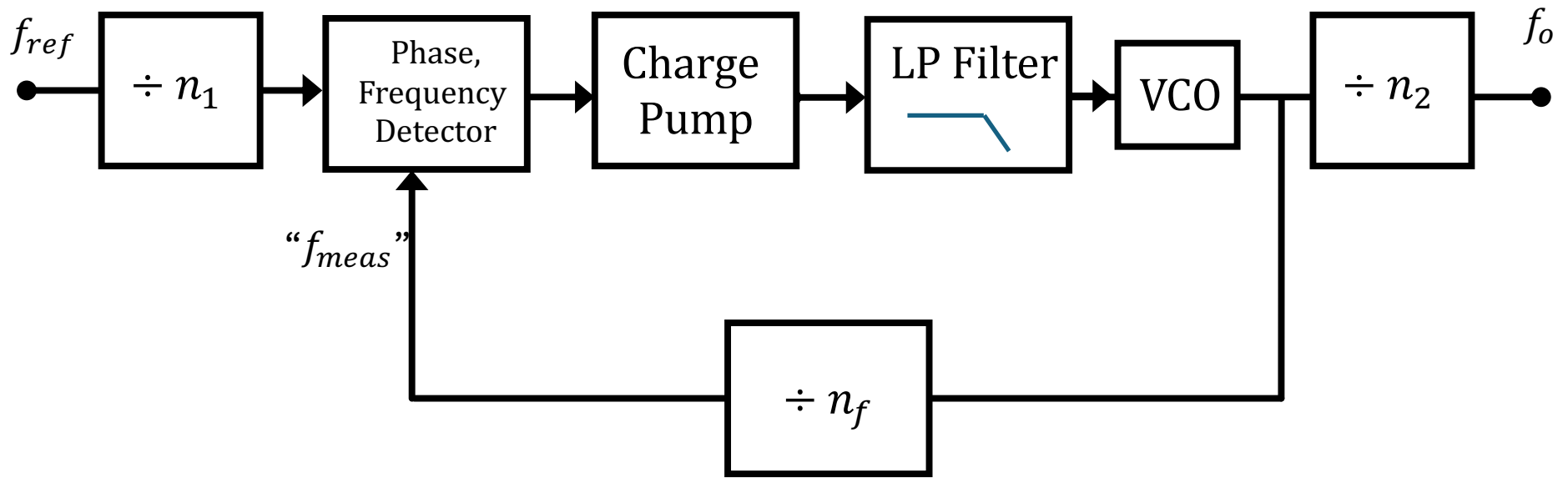
3. Multiplies by 37.125

4a. Divide by 10 for 74.25 MHz

4b. Divide by 2 for 371.25 MHz

1. Takes in 100 MHz

Side-Note



- n_f and n_2 can generally be fractions by switching between several dividers with a time-weighted averaging

Final Projects Coming Up

- In another ~week or so, we have to start thinking about planning on starting to get going on final projects.
- First part of that is teaming and teams benefit from targeting shared goals
- On the site, we'll put up an archive of final projects