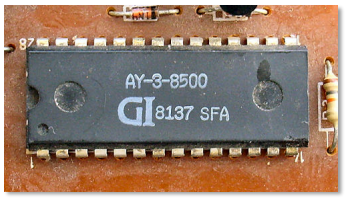
Lecture 7

LPset6 is due Thursday October 3

Lab 3 is Due next Tuesday October 1

Pong in History:

- http://www.pong-story.com/gi.htm

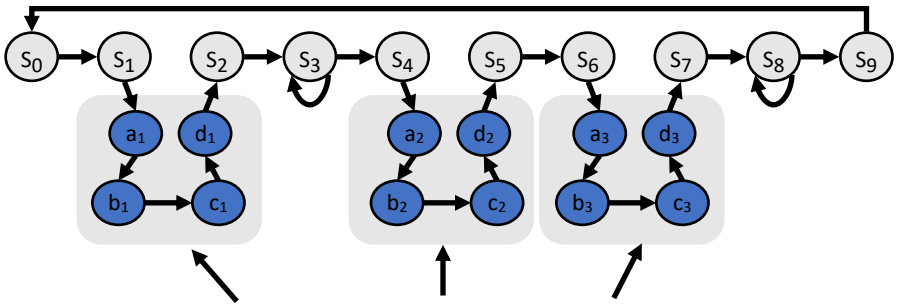


AY-3-8500 "Ball-and-Paddle" chip

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

Toward FSM Modularity

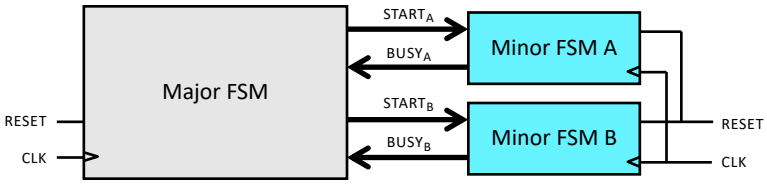
- Consider the following abstract FSM:



- Suppose that each set of states a_x...d_x is a "sub-FSM" that produces exactly the same outputs.
- Can we simplify the FSM by removing equivalent states?
  No! The outputs may be the same, but the next-state transitions are not.
- This situation closely resembles a procedure call or function call in software...how can we apply this concept to FSMs?

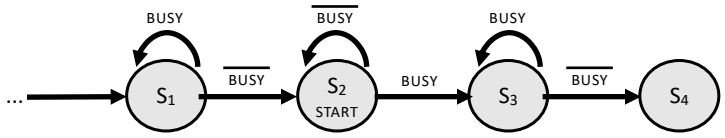
Acknowledgements: Rex Min

The Major/Minor FSM Abstraction



- Subtasks are encapsulated in minor FSMs with common reset and clock
- Simple communication abstraction:
  - START: tells the minor FSM to begin operation (the call)
  - BUSY: tells the major FSM whether the minor is done (the return)
- The major/minor abstraction is great for...
  - Modular designs (*always* a good thing)
  - Tasks that occur often but in different contexts
  - Tasks that require a variable/unknown period of time
  - Event-driven systems

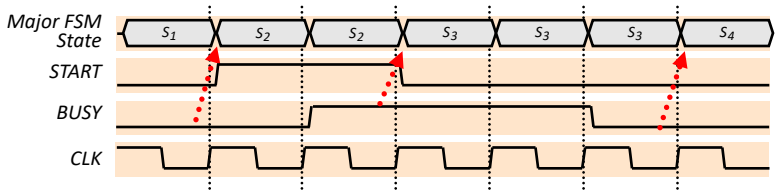
Inside the Major FSM



1. Wait until the minor FSM is ready

2. Trigger the minor FSM (and make sure it's started)

3. Wait until the minor FSM is done

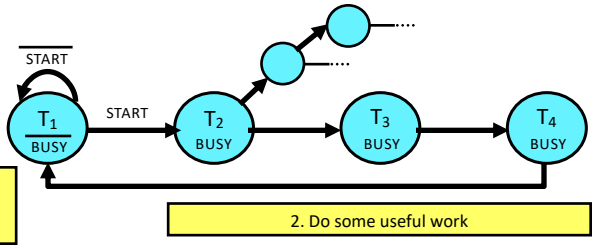


Variations:

- Usually don't need both Step 1 and Step 3
- One cycle "done" signal instead of multi-cycle "busy"

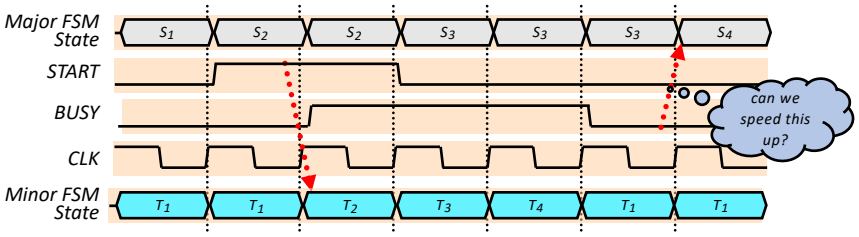
Inside the Minor FSM

1. Wait for a trigger from the major FSM



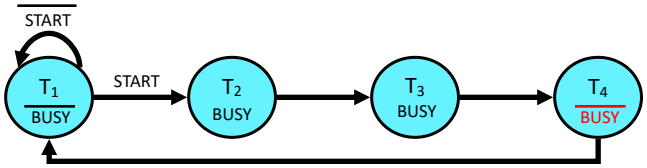
3. Signal to the major FSM that work is done

2. Do some useful work



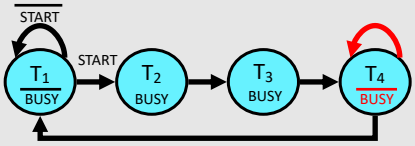
Optimizing the Minor FSM

Good idea: de-assert BUSY one cycle early



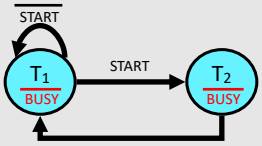
Bad idea #1:

T_4 may not immediately return to T_1

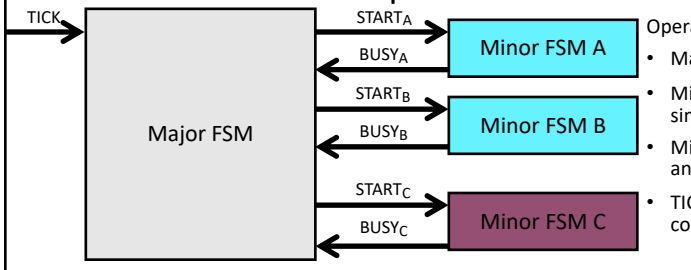


Bad idea #2:

BUSY never asserts!

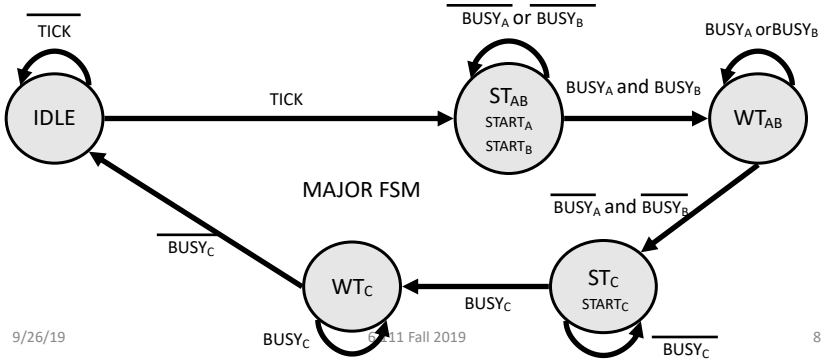


A Four-FSM Example

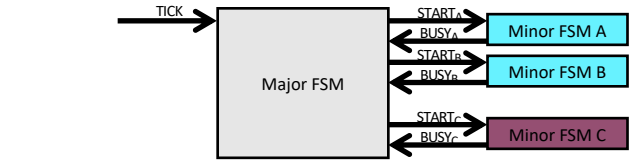


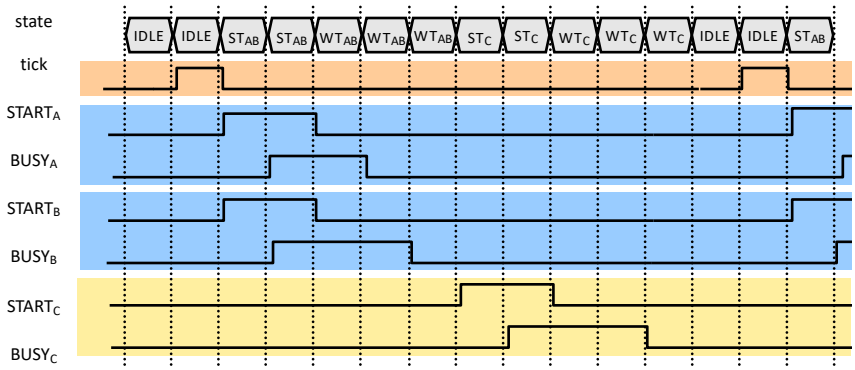
Operating Scenario:

- Major FSM is triggered by TICK
- Minors A and B are started simultaneously
- Minor C is started once both A and B complete
- TICKs arriving before the completion of C are ignored



Four-FSM Sample Waveform





Division (an example of an algorithm that takes an unknown amount of time)

```
1  def divider (dividend, divisor):
2    count = 0
3    while dividend>divisor:
4      dividend -= divisor
5      count += 1
6    return (count, dividend)
```

Super efficient divider \s

A Divider

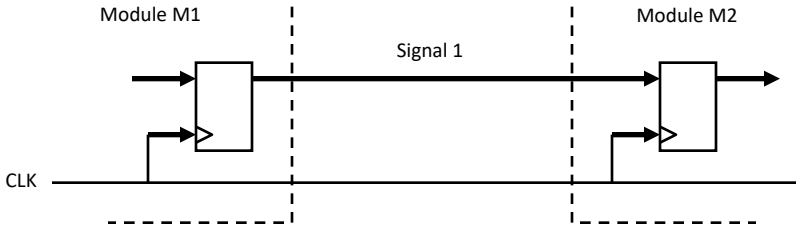
- This is a Verilog FSM example of the algorithm on the previous page which will run an unknown number of times given a set of inputs
- This is how the functionality of a while loop could be developed in your modules
- Ugh has a few problems though I'm just seeing now.

```
module divider (input clk_in, input rst_in, input[15:0] dividend_in,
                input[15:0] divisor_in, input data_valid_in,
                output logic[15:0] quotient_out,
                output logic[15:0] remainder_out, output logic busy_out);
    parameter RESTING = 0;
    parameter DIVIDING = 1;

    logic [15:0] quotient;
    logic [15:0] dividend;
    logic [15:0] divisor;

    logic state;

    assign busy_out = state;

    always_ff @(posedge clk_in)begin
      if (rst_in)begin
        quotient  <= 16'b0;
        dividend  <= 16'b0;
        divisor   <= 16'b0;
        remainder_out <= 16'b0;
        busy_out  <= 0;
        state <=
      end else begin
        case state
          RESTING: begin
            if (data_valid_in)begin
              state <= DIVIDNG;
              quotient <= 16'b0;
              dividend <= dividend_in;
              divisor  <= divisor_in;
            end
          end
          DIVIDING: begin
            if (dividend < divisor) begin
              state <= RESTING;
              remainder_out <= dividend;
              quotient_out <= dividend;
            end else begin
              quotient <= quotient + 1'b1;
              dividend <= dividend-divisor;
            end
          end
        endcase
      end
    end
endmodule
```

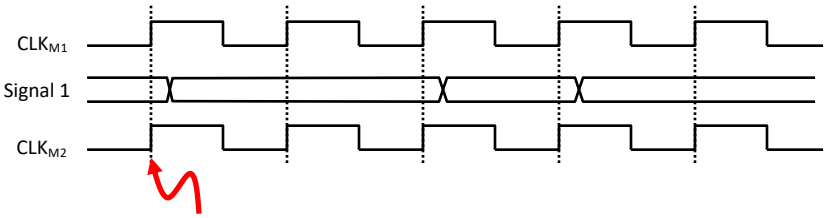

Clocking Issues

Thinking about a Few More Things Involving Clocks

Clocking and Synchronous Communication



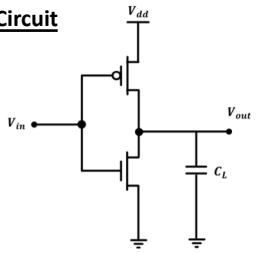
Ideal world:

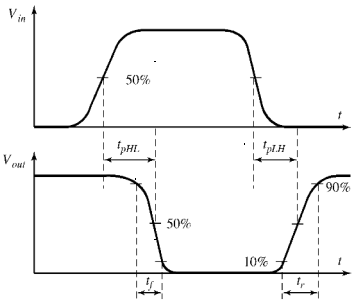


M1 and M2 clock edges aligned in time

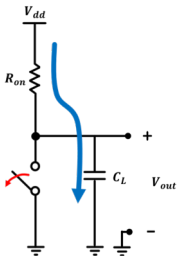
Delay Estimation: Simple RC Networks

Simple CMOS Circuit

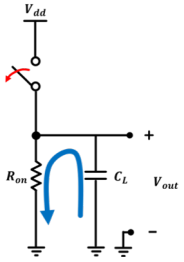




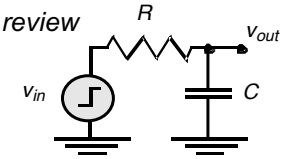
Low-to-High



High-to-Low



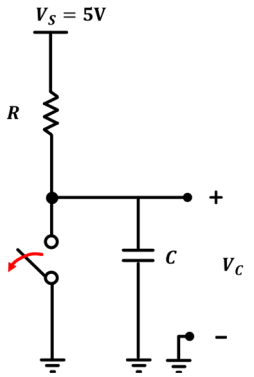
review



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

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

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

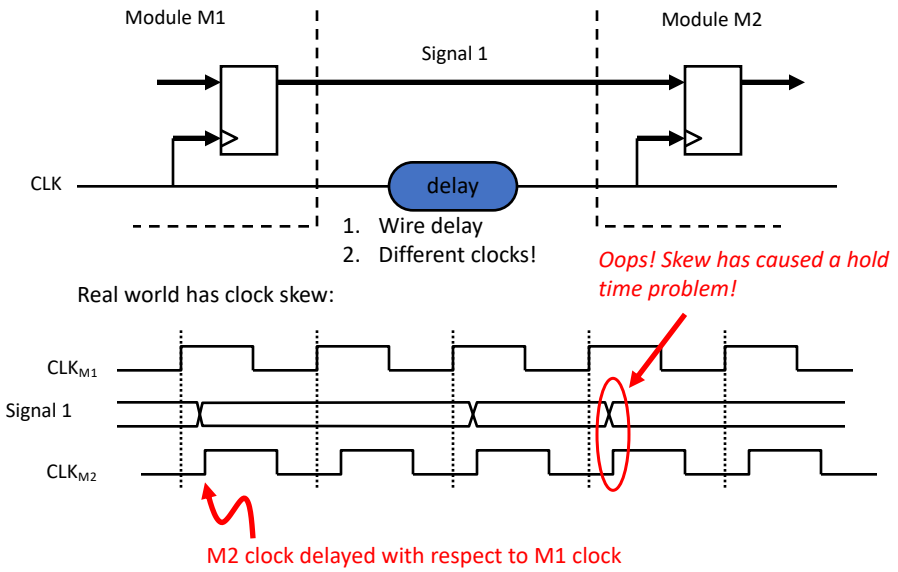
$V_s = V_R + V_C$

$V_s = i_R R + V_C \quad i_R = C\frac{dV_c}{dt}$

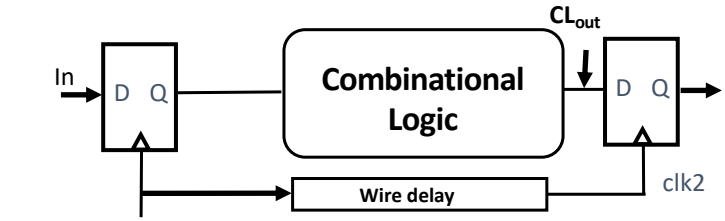
$V_s = RC\frac{dV_c}{dt} + V_c$

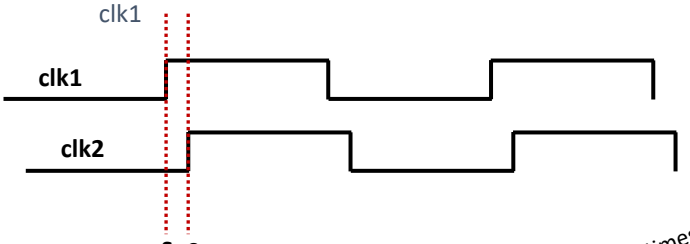
$$V_c = V_s\left(1 - e^{-\frac{t}{RC}}\right)$$

Clock Skew



Clocks are Not Perfect: Clock Skew

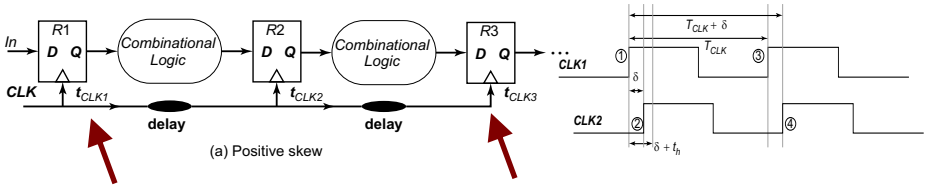




$t_{skew} = t_{clk2} - t_{clk1}$

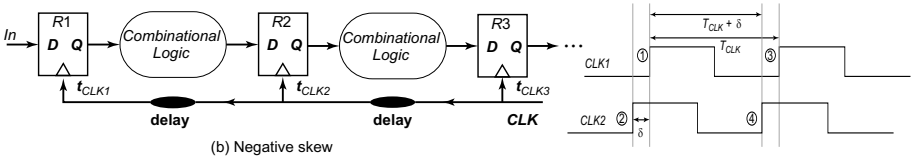
Based off of times of rising edges. Not periods!

Positive and Negative Skew



(a) Positive skew

Launching edge arrives before the receiving edge (positive skew)

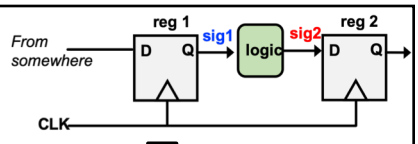


(b) Negative skew

Receiving edge arrives before the launching edge (negative skew)

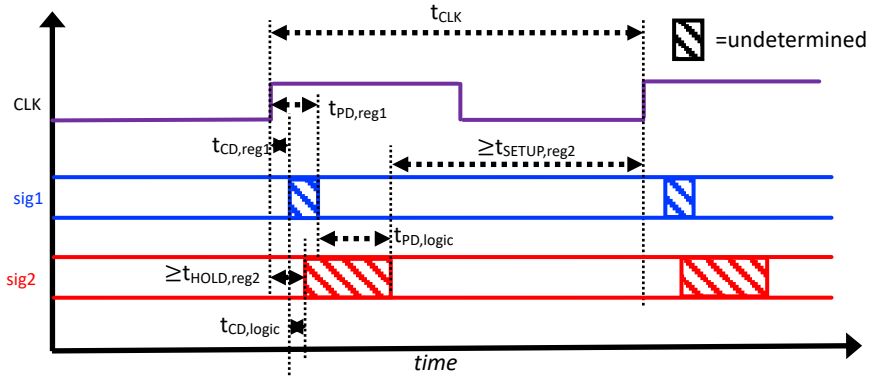
- Adapted from J. Rabaey, A. Chandrakasan, B. Nikolic, "Digital Integrated Circuits: A Design Perspective" Copyright 2003 Prentice Hall/Pearson.

This diagram again!?



=determined state

=undetermined state

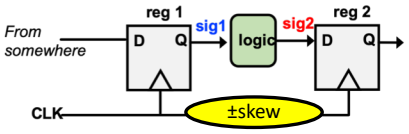


Two Requirements/ Conclusions:

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

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

D-Register Timing With Skew



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:

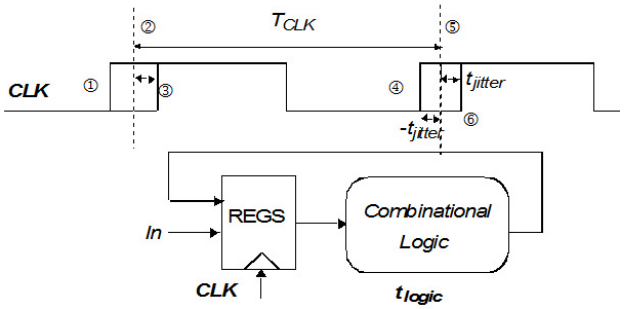
$t_{Rn,clk1}+t_{CD,reg1}+t_{CD,logic} \geq t_{Rn,clk2}+t_{HOLD,reg2}$

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

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)?

Clocks Are Not Perfectly Periodic either Ugggh: Jitter

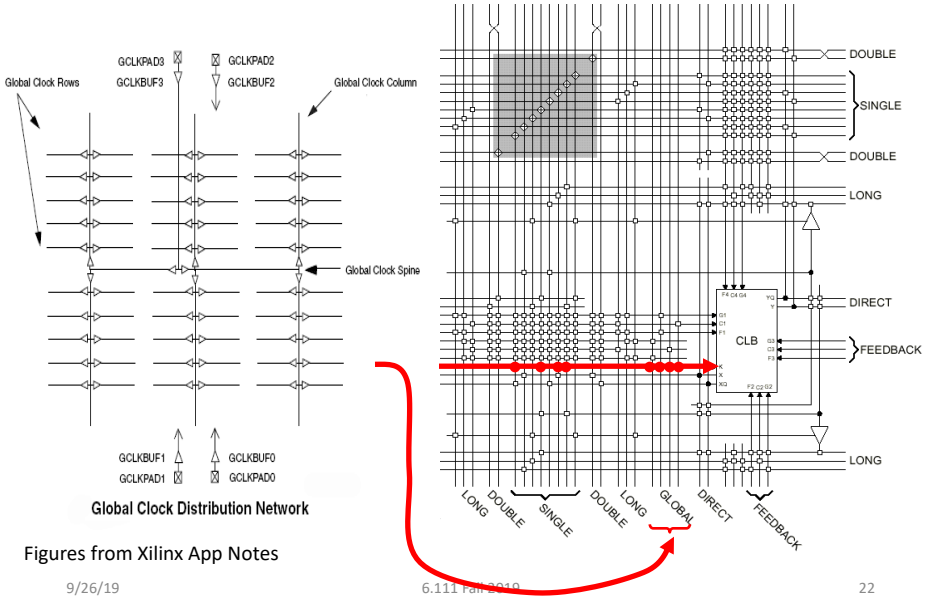


t_{pd}, t_{su}, t_{hold}

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

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

Low-skew Clocking in FPGAs



Figures from Xilinx App Notes

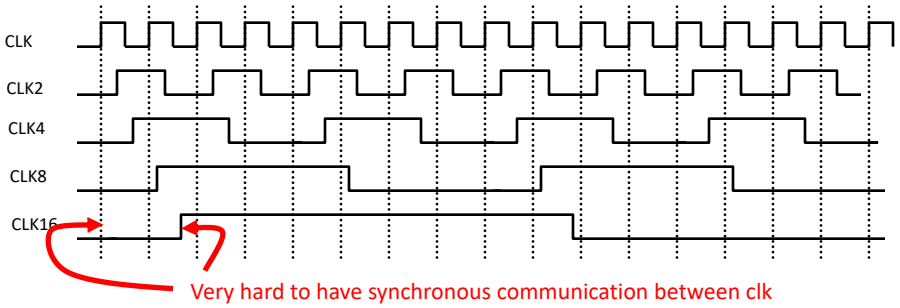
Goal: use as few clock domains as possible

Suppose we wanted clocks at f/2, f/4, f/8, etc.:

No! don't do it this way

```
reg clk2,clk4,clk8,clk16;
always @(posedge clk) clk2 <= ~clk2;
always @(posedge clk2) clk4 <= ~clk4;
always @(posedge clk4) clk8 <= ~clk16;
always @(posedge clk8) clk16 <= ~clk16;
```

No vsync!



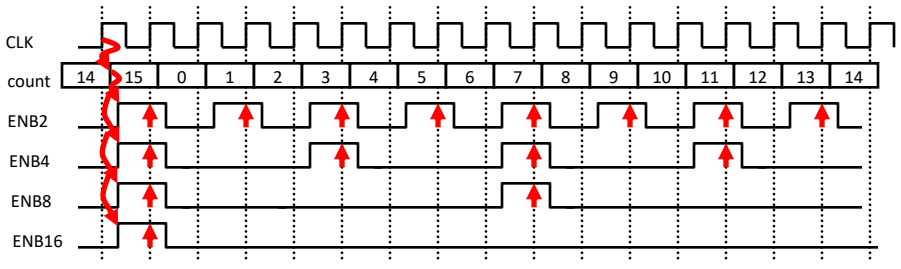
Very hard to have synchronous communication between clk and clk16 domains

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

```
logic [3:0] count;
always_ff @(posedge clk) count <= count + 1;   // counts 0..15
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);
```

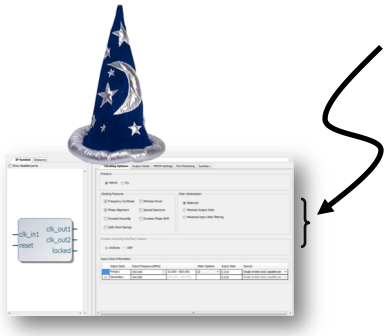
```
always_ff @(posedge clk)
  if (enb2) begin
    // get here every 2nd cycle
  end
```



= clock edge selected by enable signal

Generating Other Clock Frequencies (again)

The Nexys4 board has a 100MHz crystal (10ns period). Use "clock wizard" to generate other frequencies e.g., 65MHz to generate 1024x768 VGA video.



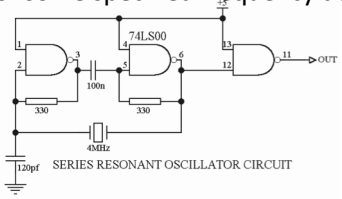
Clock Wizard can also synthesize certain multiples/fractions of the CLKIN frequency (100 MHz):

$$f_{CLKFX} = \left(\frac{M}{D}\right) f_{CLKIN}$$

Where do we get frequencies?

16MHz Crystal

- Most frequencies come from Crystal Oscillators made of quartz
- Equivalent to very High-Q LRC tank circuits
- https://en.wikipedia.org/wiki/Crystal_oscillator_frequencies
- Incorporate into circuit like that below and boom, you've got a square wave of some specified frequency dependent largely on the crystal



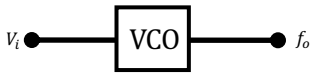
http://www.z80.info/uexosc.htm
https://en.wikipedia.org/wiki/Crystal_oscillator

High Frequencies

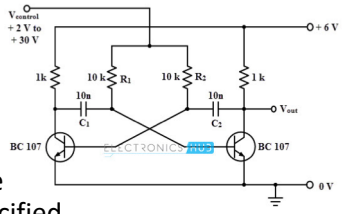
- Very hard to get a crystal oscillator to operate above ~200 MHz (7th harmonic of resonance of crystal itself, which usually is limited to about 30 MHz due to fabrication limitations)
- Where does the 2.33 GHz clock of my iPhone come from then?
- Frequency Multipliers!
- Talk about Phase Locked Loops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

Voltage Controlled Oscillator

- It is very easy to make voltage-controlled oscillators that run up to 1GHz 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 specified right?
- Same reason we don't see op amps in open loop out in the wild…they are too unstable…gotta place them in negative feedback

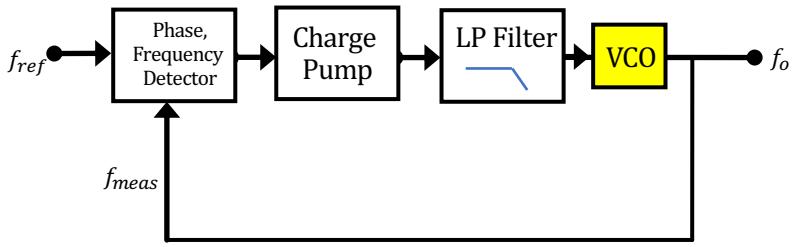


A simple VCO (not type found in FPGA)

http://www.electronicshub.org/voltage-controlled-oscillators-vco/

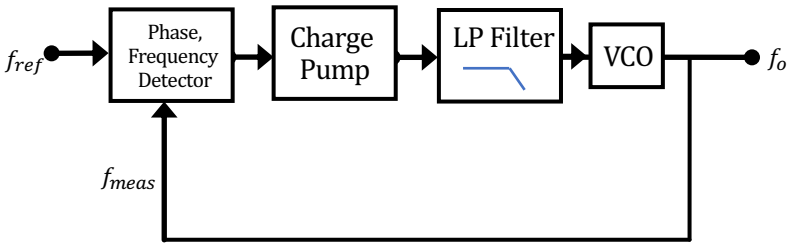
Phase Locked Loop

- Place the unstable, but capable VCO in a feedback loop.
- This type of circuit is a phase-locked loop variant

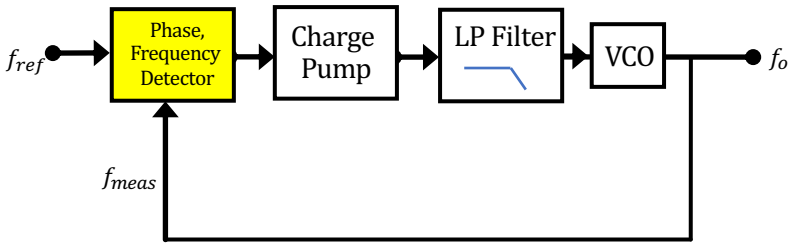


Phase Locked Loop

- Circuit that can track an input phase of a system and reproduce it at the output

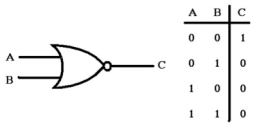


Phase, Frequency Detector

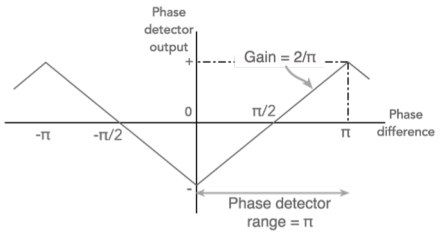


Phase Detector

- Can be a simple XOR gate



A	B	C
0	0	1
0	1	0
1	0	0
1	1	0



- 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

- Detects both change and which clock signal is consistently leading the other one
- Using MOSFETs you charge/discharge a capacitor accordingly which also with some resistors low-pass filter's the signal
- The output voltage is then roughly proportional to the frequency error!

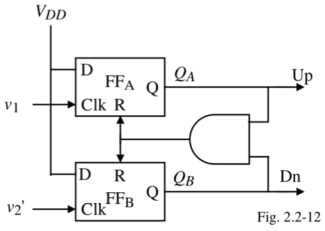


Fig. 2.2-12

http://www.globalspec.com/reference/72819/203279/2-7-phase-detectors-with-charge-pump-output

Phase Frequency Detection

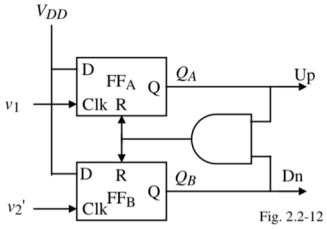


Fig. 2.2-12

- Clock 1 and clock 2 are constantly competing with one another to generate up and down signals
- The up signals charge up a capacitors through a pair of transistors…the down signal discharges the capacitor

PFD State Diagram:

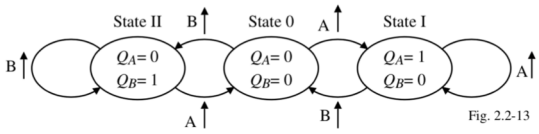
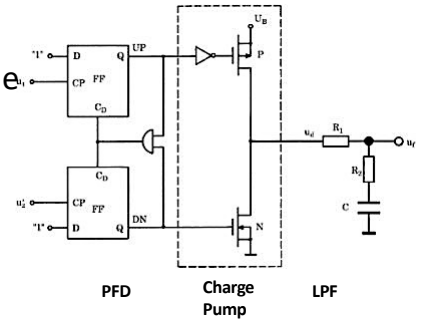


Fig. 2.2-13

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

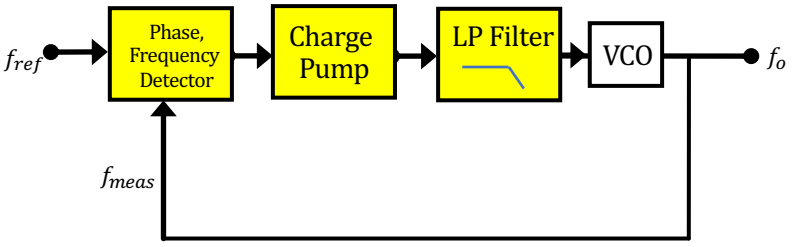
Phase-Frequency Detection

- Detects both change and which clock signal is consistently leading the other one
- Using MOSFETs you charge/discharge a capacitor accordingly which also with some resistors low-pass filter's the signal
- The output voltage is then roughly proportional to the frequency error!



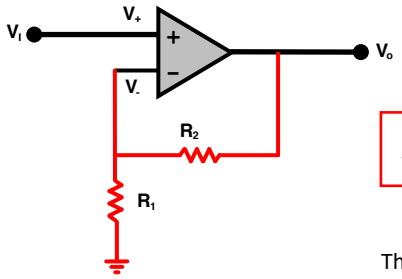
http://www.globalspec.com/reference/72819/203279/2-7-phase-detectors-with-charge-pump-output

PFD, Charge Pump, LP Filter



- So this circuit can make $f_0 = f_{ref}$ That doesn't help us!
- 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!

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

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

The gain A_v of this circuit is therefore:

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

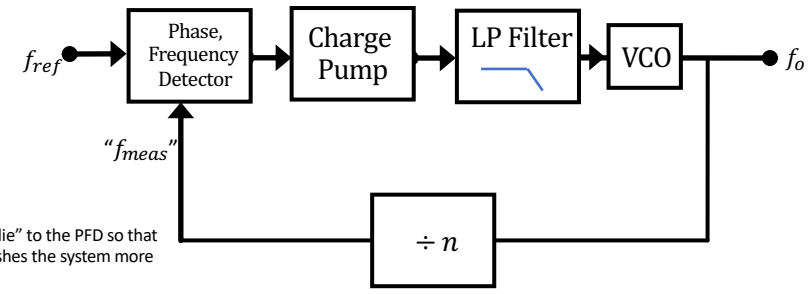
The gain of a "non-inverting amplifier"

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

Same Idea with Phase Locked Loops!

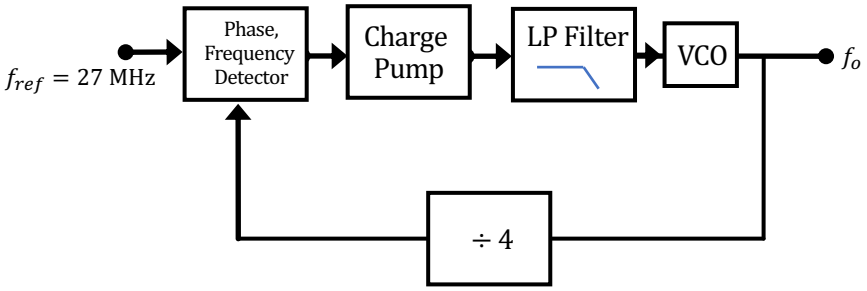
Use a Clock Divider in Feedback Path!

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



We "lie" to the PFD so that it pushes the system more

Use a Clock Divider in Feedback Path!



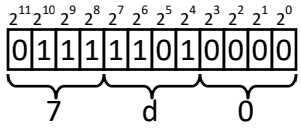

```
reg clk2,clk4,clk8,clk16;
always @(posedge clk) clk2 <= ~clk2;
always @(posedge clk2) clk4 <= ~clk4;
always @(posedge clk4) clk8 <= ~clk16;
always @(posedge clk8) clk16 <= ~clk16;
```


Number Representation

Encoding numbers

It is straightforward to encode positive integers as a sequence of bits. Each bit is assigned a weight. Ordered from right to left, these weights are increasing powers of 2. The value of an n-bit number encoded in this fashion is given by the following formula:

$$v = \sum_{i=0}^{n-1} 2^i b_i$$



Oftentimes we will find it convenient to cluster groups of bits together for a more compact notation. Two popular groupings are clusters of 4 bits and ever so rarely, 3 bits.

03720

Octal - base 8

000 - 0
001 - 1
010 - 2
011 - 3
100 - 4
101 - 5
110 - 6
111 - 7

0x7d0

Hexadecimal - base 16

0000 - 0 1000 - 8
0001 - 1 1001 - 9
0010 - 2 1010 - a
0011 - 3 1011 - b
0100 - 4 1100 - c
0101 - 5 1101 - d
0110 - 6 1110 - e
0111 - 7 1111 - f

Binary Representation of Numbers

How to represent negative numbers?

- Three common schemes:
  - sign-magnitude, ones complement, twos complement
- Sign-magnitude: MSB = 0 for positive, 1 for negative
  - Range: $-(2^{N-1} - 1)$ to $+(2^{N-1} - 1)$
  - Two representations for zero: 0000… & 1000…
  - Simple multiplication but complicated addition/subtraction
- Ones complement: if N is positive then its negative is $\overline{N}$
  - Example: 0111 = 7, 1000 = -7
  - Range: $-(2^{N-1} - 1)$ to $+(2^{N-1} - 1)$
  - Two representations for zero: 0000… & 1111…
  - Subtraction is addition followed by end-around carry (subtraction is different from addition unit)

Basically flip every bit of the number to negate it

Representing negative integers

To keep our arithmetic circuits simple, we'd like to find a representation for negative numbers so that we can use a single operation (binary addition) when we wish to find the sum of two integers, independent of whether they are positive or negative.

We certainly want A + (-A) = 0. Consider the following 8-bit binary addition where we only keep 8 bits of the result:

```
  11111111
+ 00000001
  --------
  00000000
```

which implies that the 8-bit representation of -1 is 11111111. More generally

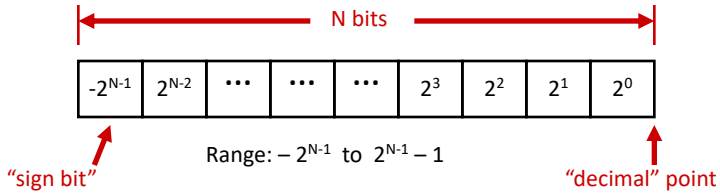
Negation: Complement and add 1

-A = 0 - A
= (-1 + 1)- A
= (-1 - A) + 1
= ~A + 1

	1	1	1	1	1	1	1	1
–	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
	$\overline{A_7}$	$\overline{A_6}$	$\overline{A_5}$	$\overline{A_4}$	$\overline{A_3}$	$\overline{A_2}$	$\overline{A_1}$	$\overline{A_0}$

~ means bit-wise complement

Signed integers: 2's complement



8-bit 2's complement example:

$11010110 = -2^7 + 2^6 + 2^4 + 2^2 + 2^1 = -128 + 64 + 16 + 4 + 2 = -42$

If we use a two's complement representation for signed integers, the same binary addition mod 2^n procedure will work for adding positive and negative numbers (don't need separate subtraction rules). The same procedure will also handle unsigned numbers!

By moving the implicit location of "decimal" point, we can represent fractions too:

$1101.0110 = -2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} = -8 + 4 + 1 + 0.25 + 0.125 = -2.625$

Sign extension

Consider the 8-bit 2's complement representation of:

```
42 = 00101010      -5 = ~00000101 + 1
                        =  11111010 + 1
                        =  11111011
```

What is their 16-bit 2's complement representation?

42 = 0000000000101010

-5 = 1111111111111011

Extend the MSB (aka the "sign bit") into the higher-order bit positions

Using Signed Arithmetic in Verilog

"<<<" and ">>>" tokens result in arithmetic (signed) left and right shifts: multiple by 2 and divide by 2.

Right shifts will maintain the sign by filling in with sign bit values during shift

```
wire signed [3:0] value = 4'b1000;  //  -8

value >>  2   // results in 0010  or 2
value >>> 2   // results in 1110 or -2
```

Using Signed Arithmetic in Verilog

ALL OF THE FOLLOWING ARE TREATED AS ***UNSIGNED*** IN VERILOG!!!

- *Any* operation on two operands, unless **both operands are signed**
- Based numbers (e.g. 12'd10), unless the explicit "s" modifier is used)
- Bit-select results `a[5]`
- Part-select results `a[4:2]`
- Concatenations

```
logic [15:0] a; // Unsigned
logic signed [15:0] b;
logic signed [16:0] signed_a;
logic signed [31:0] a_mult_b;

assign signed_a = a;//Convert to signed
assign a_mult_b = signed_a * b
```

Example of multiplying signed by unsigned

http://billauer.co.il/blog/2012/10/signed-arithmetics-verilog/

For example:

```
module test_one;
    logic signed [3:0] x;
    logic  [3:0] y;
    logic signed [8:0] z;
    initial begin
        x = -2;
        y=3;
        z = x*y;
        $display(x, y, z);
        $finish;
    end
endmodule
```

Result:

```
-2 3 42
```

```
module test_two;
    logic signed [3:0] x;
    logic signed [3:0] y;
    logic signed [8:0] z;
    initial begin
        x = -2;
        y=3;
        z = x*y;
        $display(x, y, z);
        $finish;
    end
endmodule
```

Result:

```
-2 3 -6
```

Not really synthesizable here ($finish, $display, etc)…but shows what Verilog is thinking

Signed Numbers

- Once you start using signed Verilog, just make everything you're using signed. If you do that, you should be ok.
- Make sure everything upstream of a calculation has been done in only a signed environment (held in signed logics and used with signed logics.
- Signed/Unsigned bugs are some of the hardest to find so be cautious