

Memory II, FSMs, Pipelining

6.205 Fall 2025

Administrative

- Week 4 due last night
- Week 5 out after class
- More video and now camera too
- Final project details by tomorrow

Memory Example!

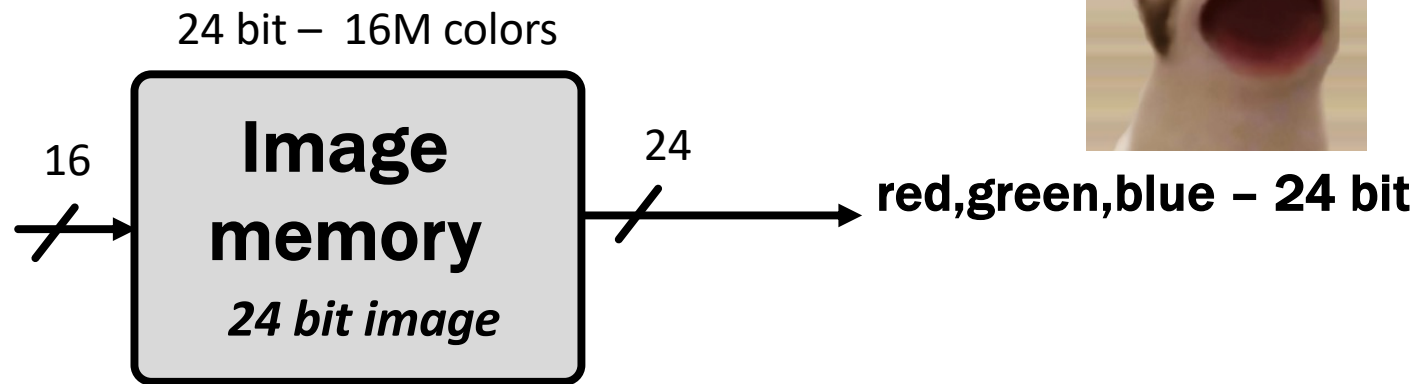
- In the first part of week 05 you're going to be displaying popcat on your FPGA over video.
- We need to store popcat
- Popcat is a 256X256 24 bit color image.
- How to encode with memories?



popcat

Popcat

- We could build a *24bit-wide* memory that has 256x256 (65,536) entries (deep) in it (one for each pixel)
- Math: $256 \times 256 \times 24 = 1,572,864$ bits
- That's greater than >50% of our memory on the FPGA....all for one popcat

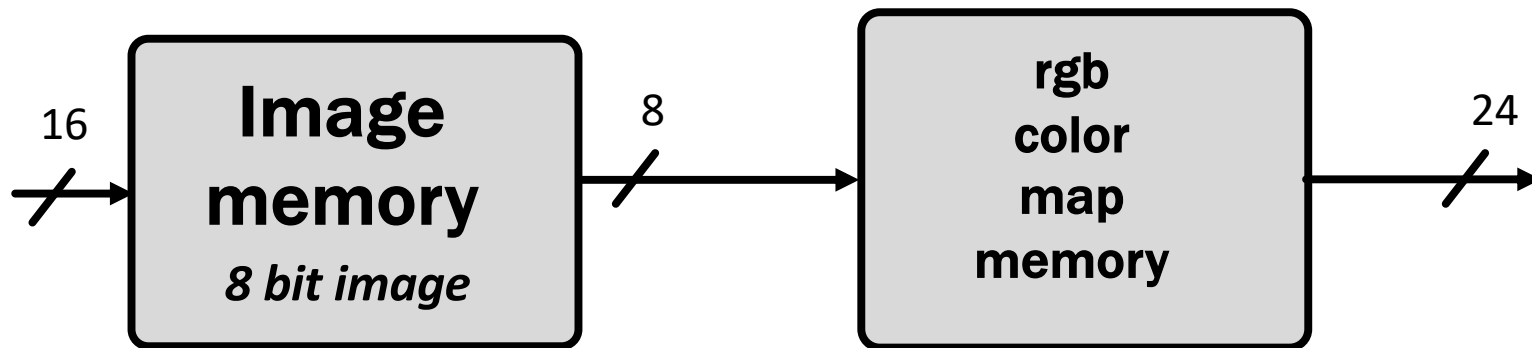


Strategy

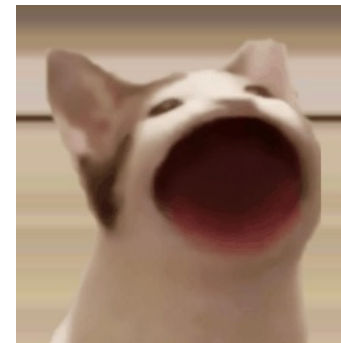
- Images are large and take up lots of memory
- Want to save space, and store image using less memory
- Many images don't express every one of the 2^{24} "true" colors.
- Why waste the space storing an unused possibility?
- So pick the N most popular and only display them:
 - You can encode each pixel using $\text{ceil}(\log_2(N))$ bits (save space)
 - Then use a color table to look up what full color (24 bit value) that corresponds to!

Color Lookup Table

- So use two memories:
 - 8-wide memory with 65536 entries for each pixel encoding one of 256 colors (using 8 bits)
 - 24-wide memory with 256 entries encoding those colors



Colors are still 24 bit, but the pixels are encoded using only 8 bits



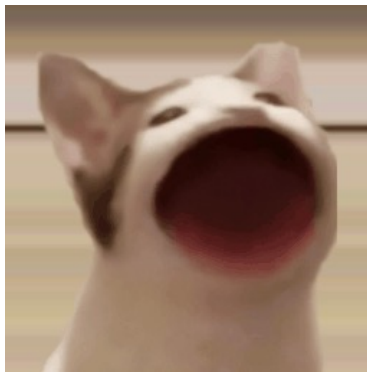
**red,
green,
blue – 24 bit**

Storage Savings



24 bit – 16M colors

256 X 256 image @ 24 bits per pixel is:
 $256 \times 256 \times 24 \text{ bits} = 1.572 \text{ Mbits}$
(196.6 kBytes)



8 bit – 256 colors

256 X 256 image with 8 bit color map:
 $256 \times 256 \times 8 \text{ bits} + 256 \times 24 = 0.530432 \text{ Mbits}$
(66.304 kBytes)

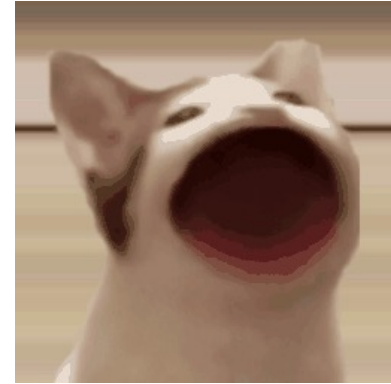
If need more savings...keep going as needed



24 bit – 16M colors



8 bit – 256 colors



6 bit – 16 colors



4 bit – 16 colors



2 bit – 4 colors



1 bit – 2 colors

Color Lookup Table

Open up image.mem

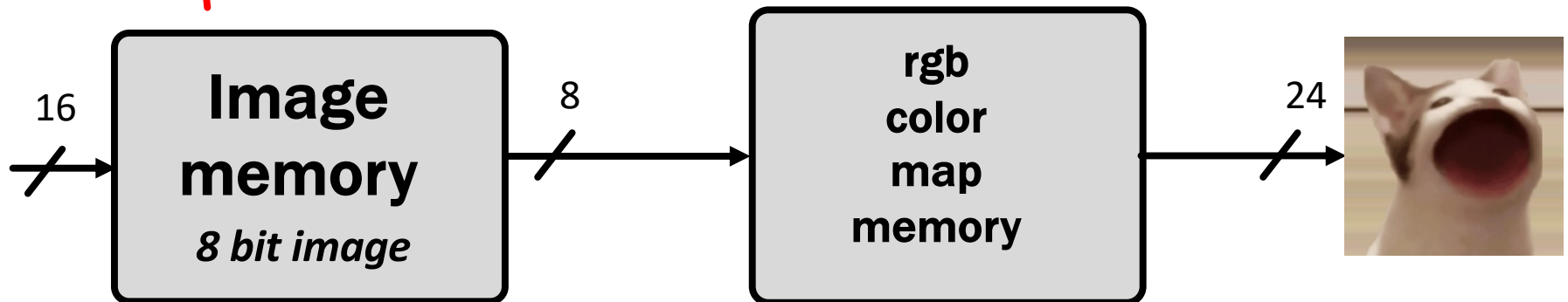
```
1 7d
2 d0
3 bf
4 bf
5 bf
6 bf
7 bf
8 bf
9 bf
10 bf
11 bf
12 bf
13 bf
14 bf
15 bf
16 bf
17 bf
18 bf
19 bf
20 bf
```

Pixel 10 is 0xbf

Open up palette.mem

```
181 997567
182 967565
183 9a785a
184 97785c
185 98755b
186 947665
187 847464
188 947759
189 937459
190 907662
191 8f7657
192 8a7660
193 897657
194 997261
195 95725f
196 937264
197 93715a
198 8f7165
199 8e715e
200 8e7159
201 8e7153
202 956d5d
203 906d5a
204 8c6d5f
```

*Color 0xbf is
0x8f7657*



Additional Tricks Can be Played

- Dithering, in particular can help with this problem of using limited colors more wisely, but we'll go into that in a future week. (form of “noise shaping”)

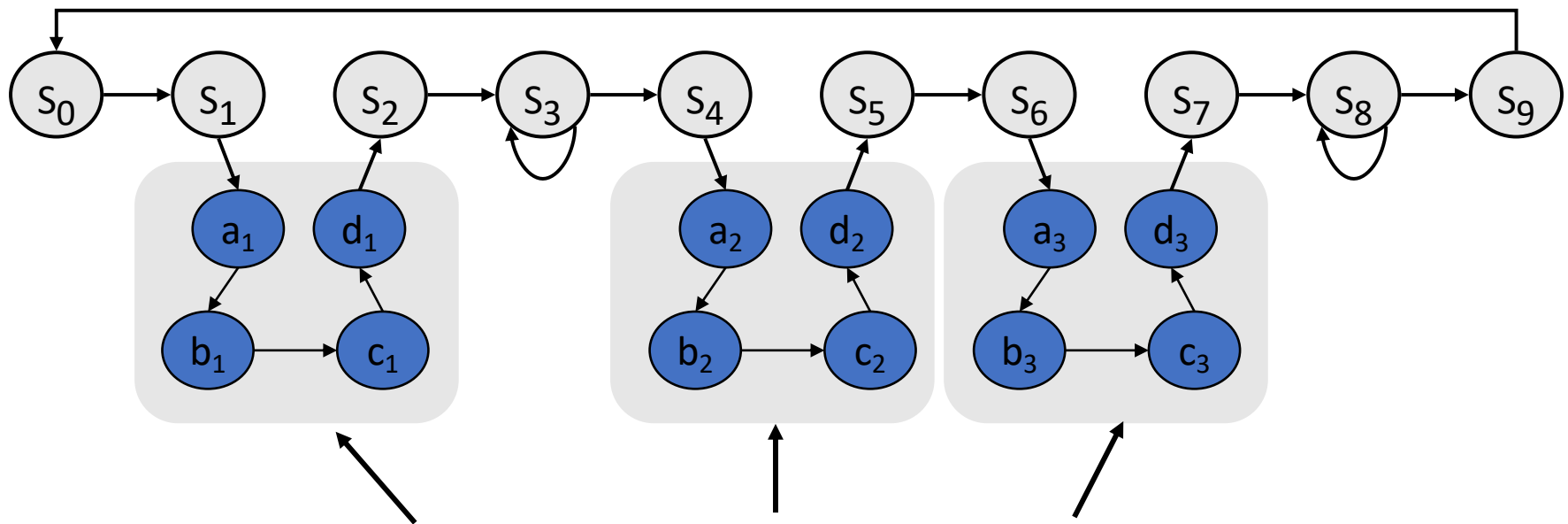
What about Memory Latency?

- Yes What about latency.
- These things we just described are memories!
- Memory of any scale usually has latency involved with it
- Xilinx BRAM has how many cycles latency on a read?
 - 1 technically, though...
 - 2 cycles recommended, dare I say mandated in 6.205
- Lab/Week 5 will investigate

FSM Modularity

Toward FSM Modularity

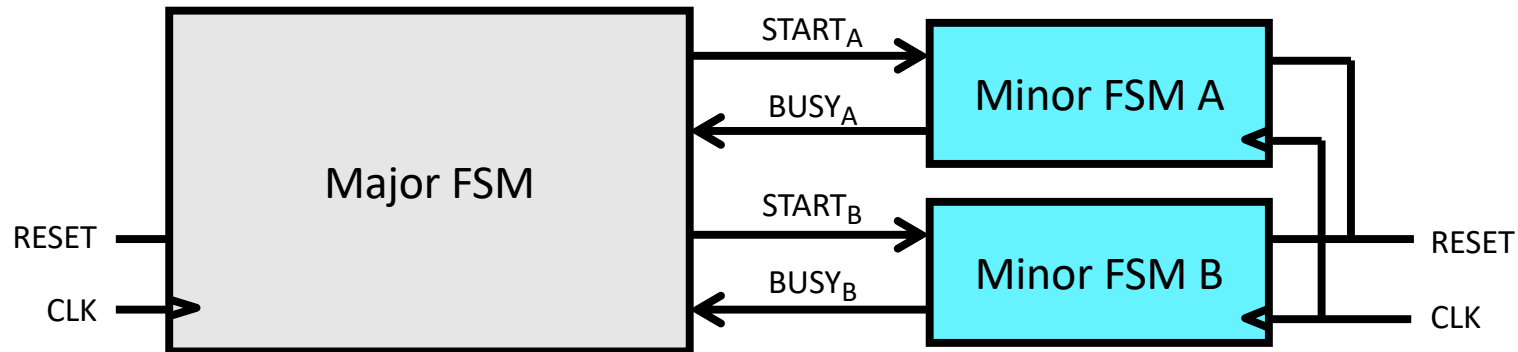
- Consider the following abstract FSM:



- Suppose that each set of states $a_x \dots d_x$ 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?

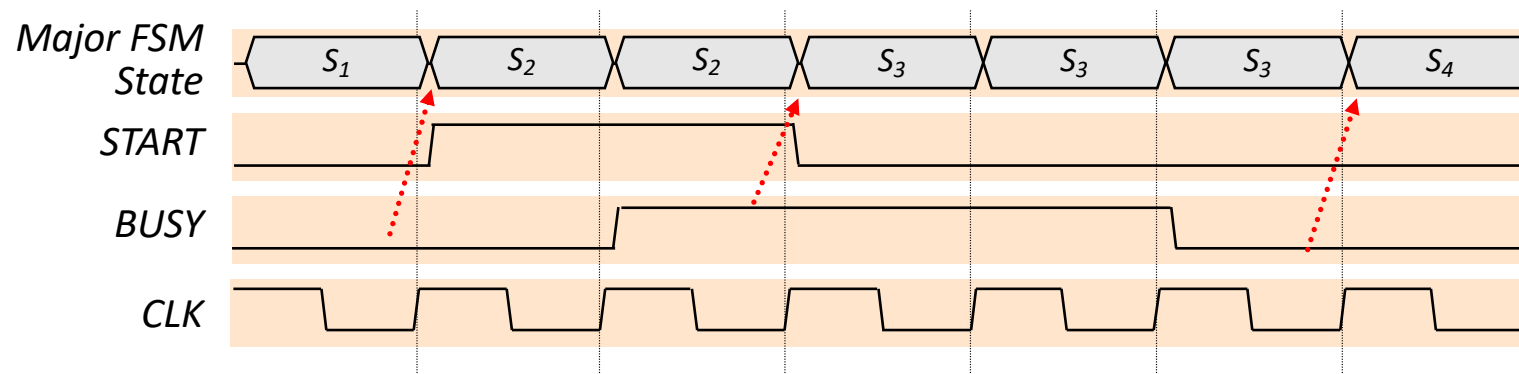
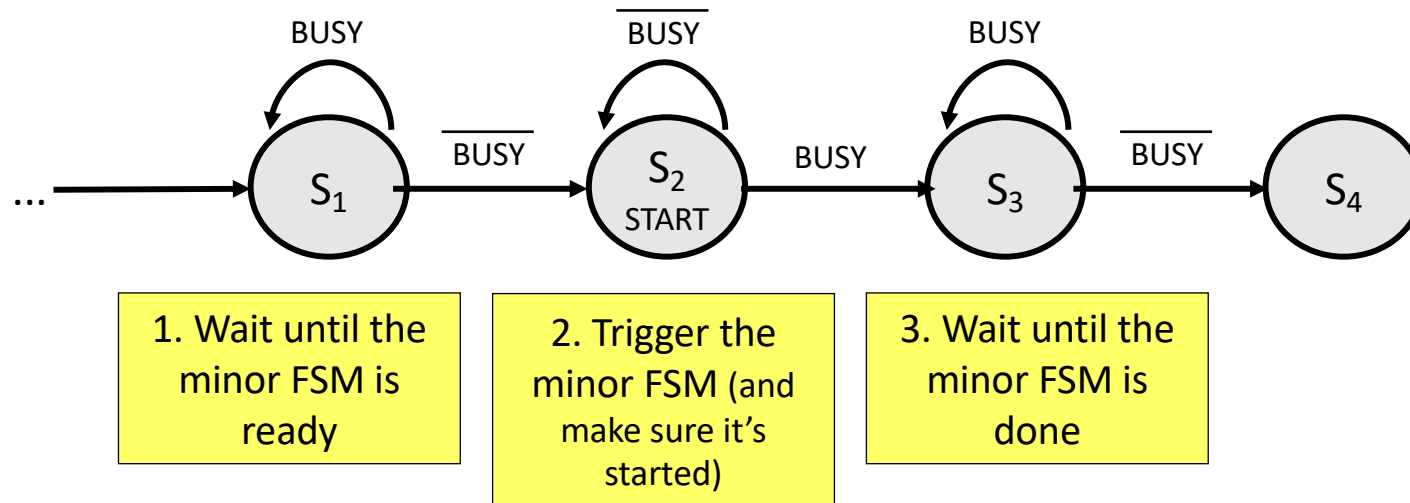
Acknowledgements: Rex Min

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

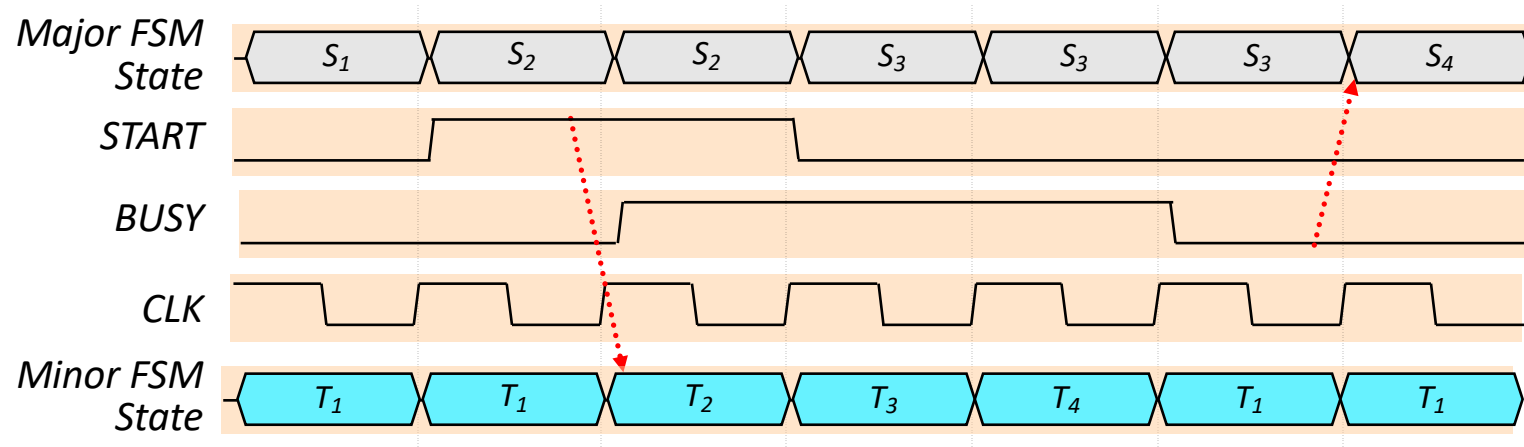
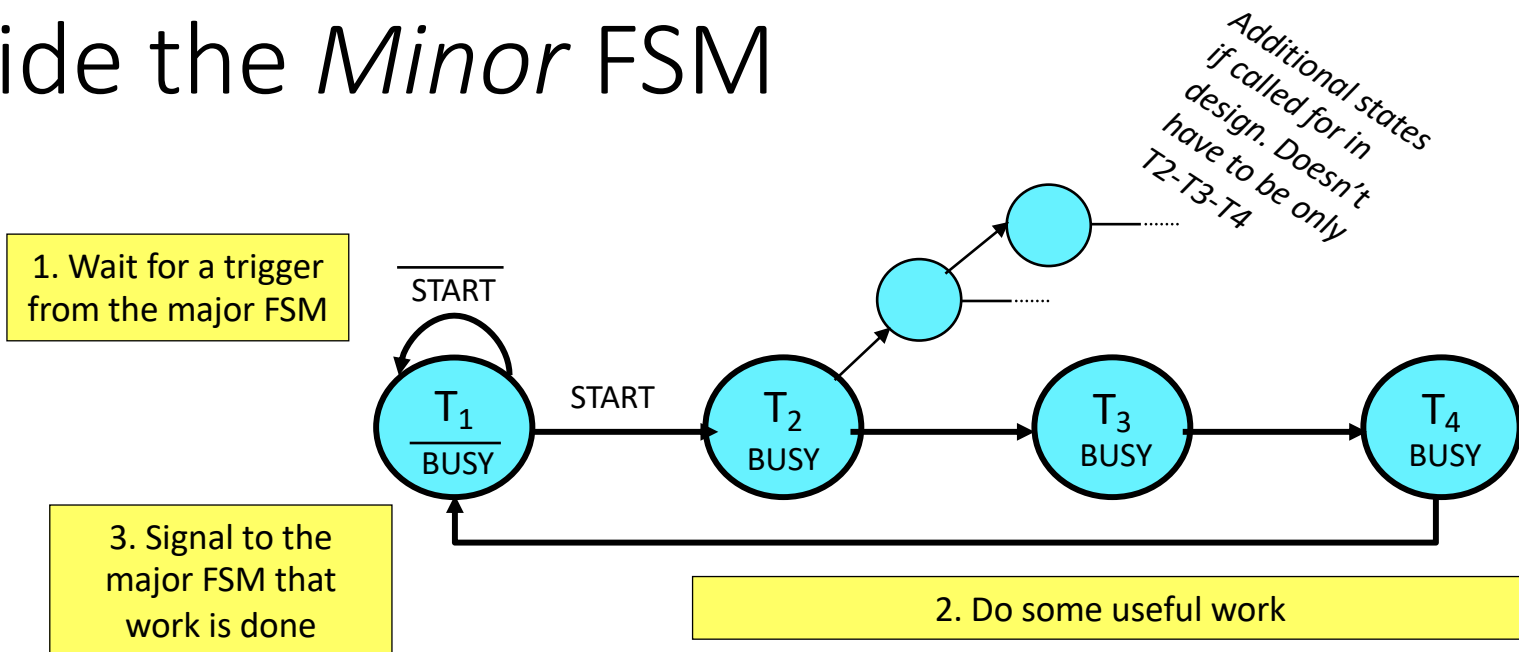
Inside the *Major* FSM



Variations:

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

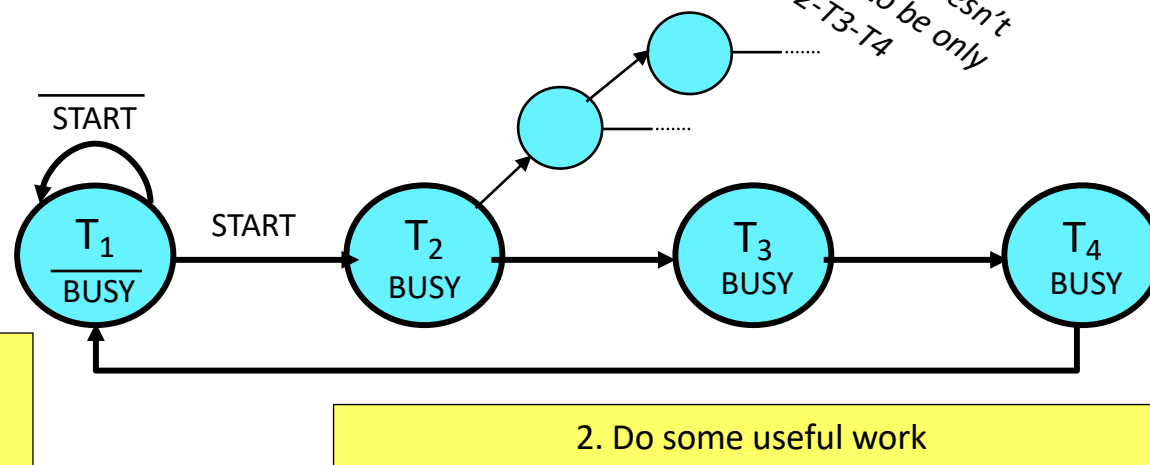
Inside the *Minor* FSM



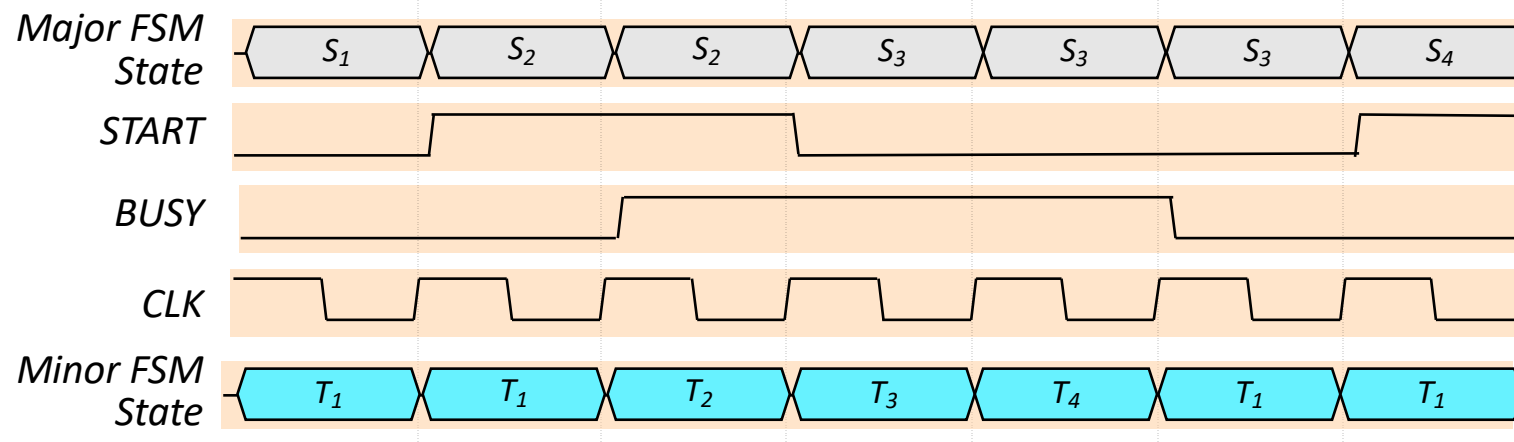
Issue with minor FSM?

1. Wait for a trigger from the major FSM

3. Signal to the major FSM that work is done

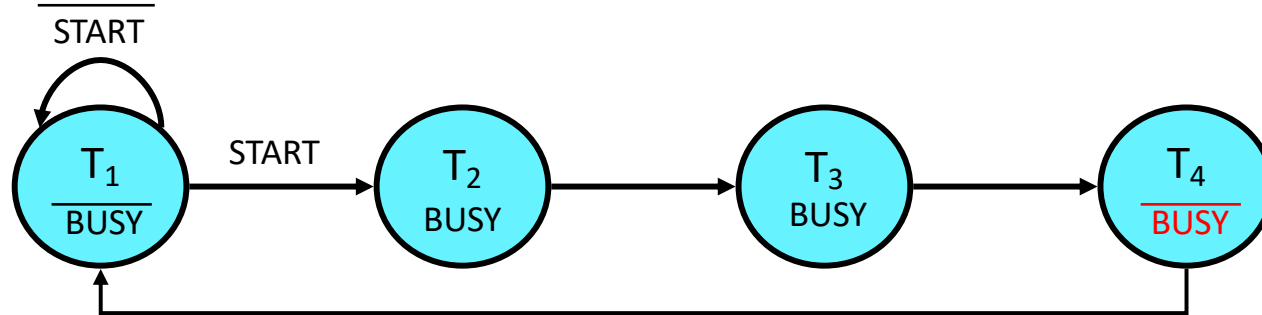


Major triggers another calculation



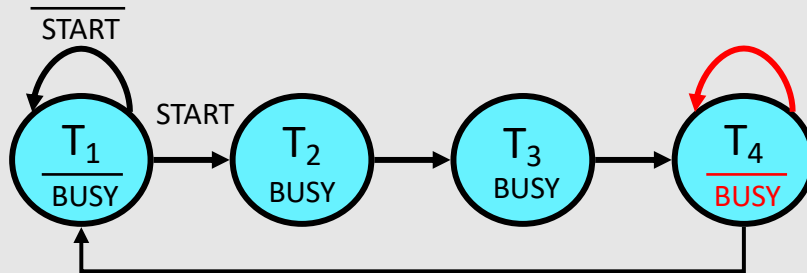
Optimizing the *Minor* FSM

Good idea: de-assert BUSY one cycle early with caveats!!!



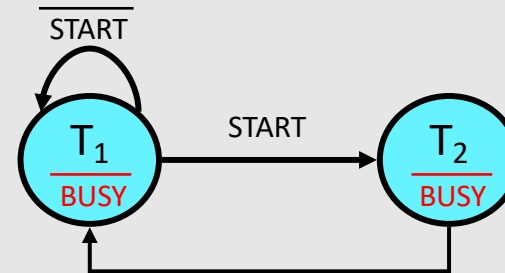
Bad idea #1:

T₄ may not immediately return to T₁



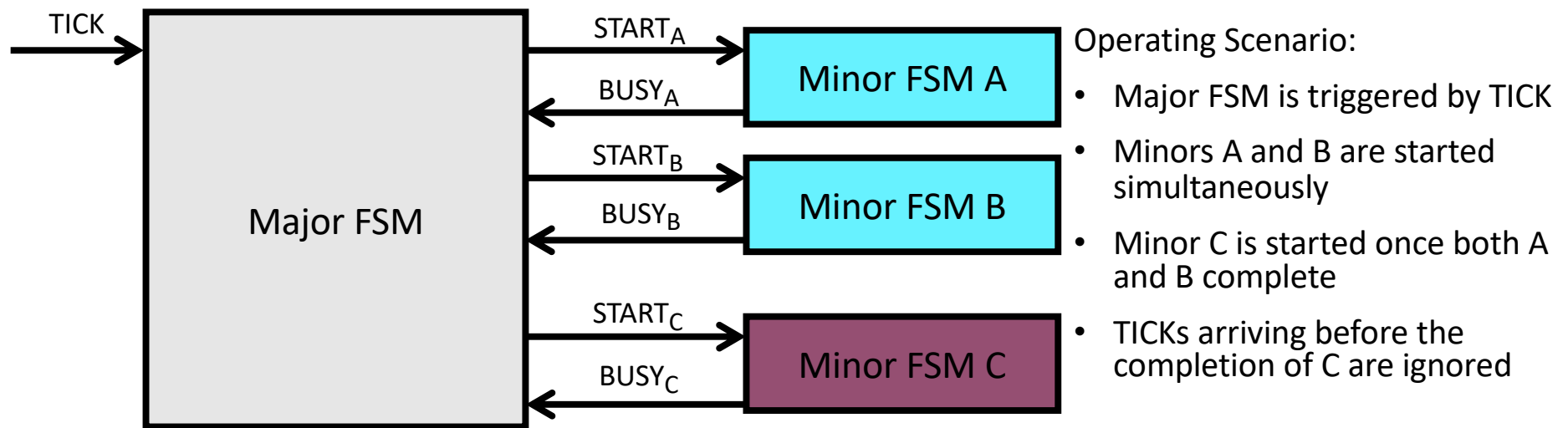
Bad idea #2:

BUSY never asserts!



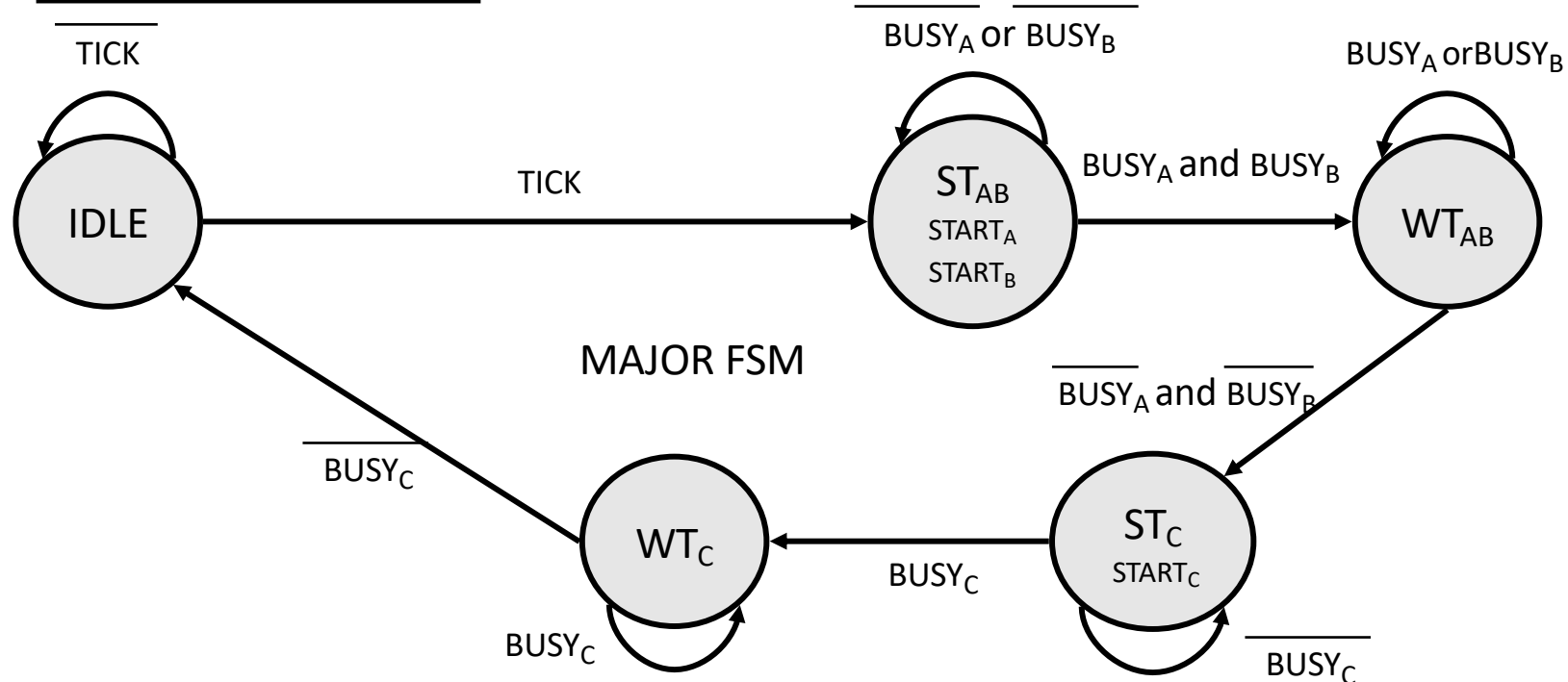
So make sure you if you do this, that last state always happens and always happens for one cycle

A Four-FSM Example

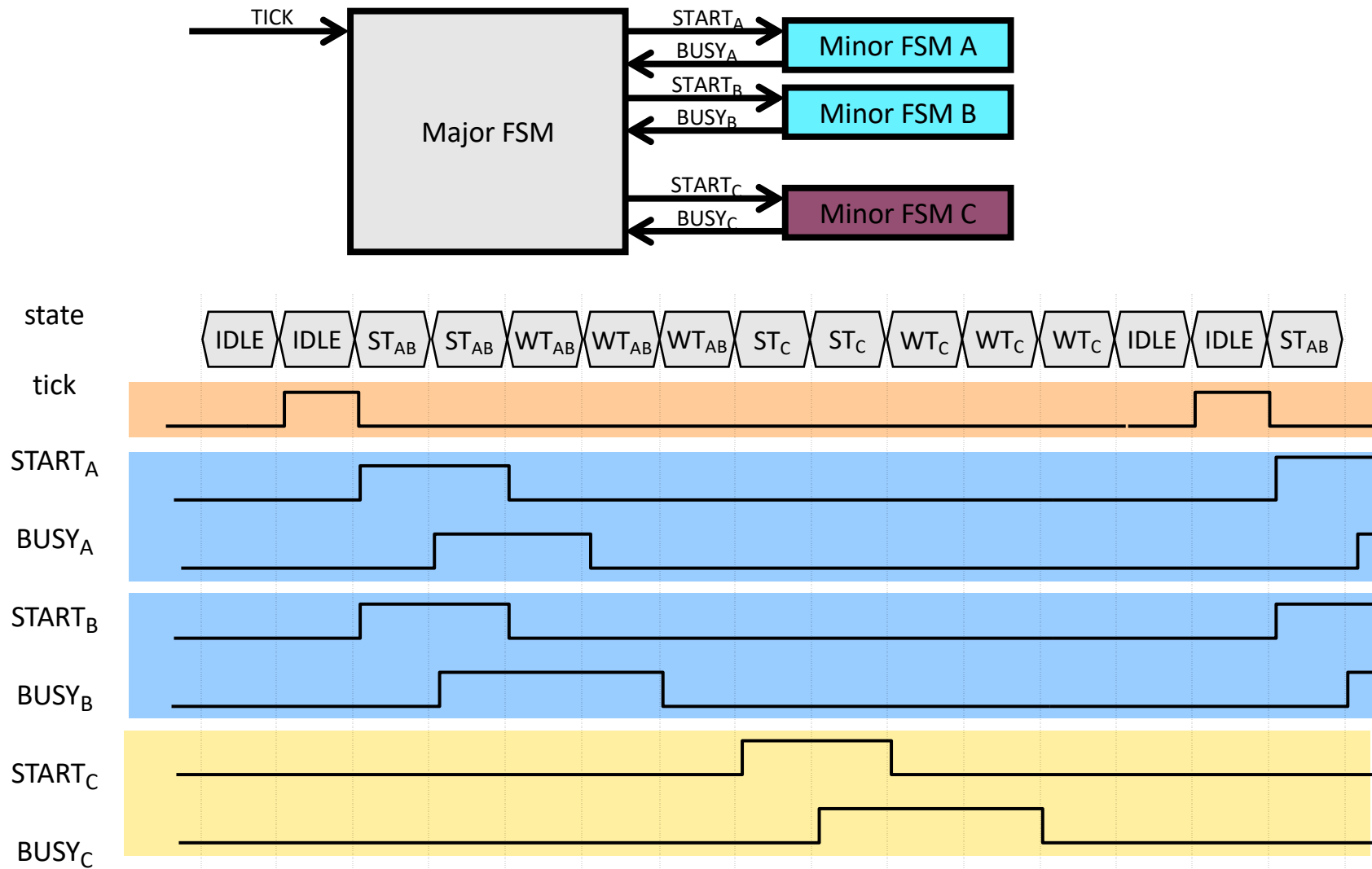


Operating Scenario:

- 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



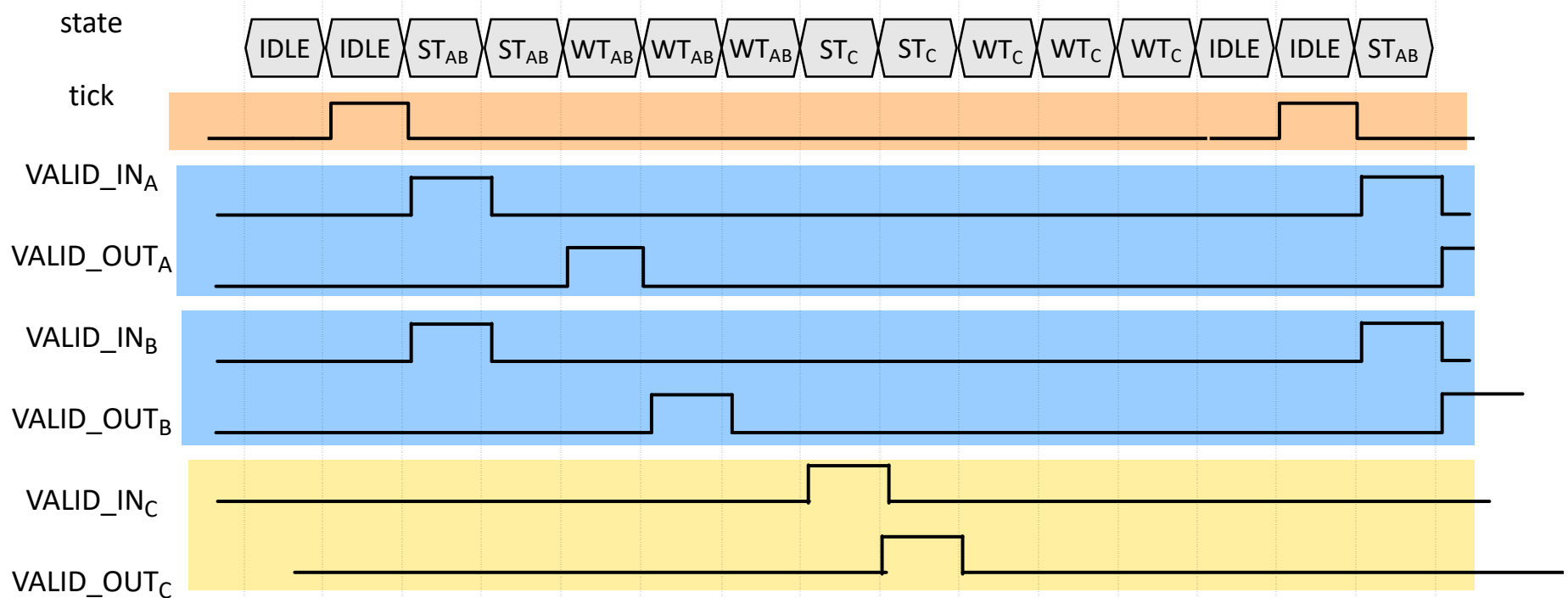
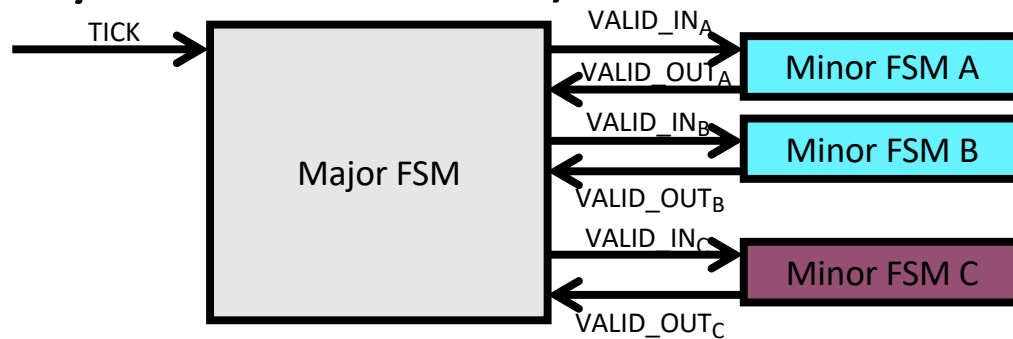
Four-FSM Sample Waveform



Alternative to Busy Signals

- As an alternative to busy signals sometimes just having a single-cycle “valid” signals is sufficient.
- You have an implied “business” until valid shows up
- If the downstream systems involved are stateful enough to be able to keep track of various system’s this can work
- Or you can do both. Depends on your design

Alternative to Busy Signals (Single-cycle asserts)



A Divider

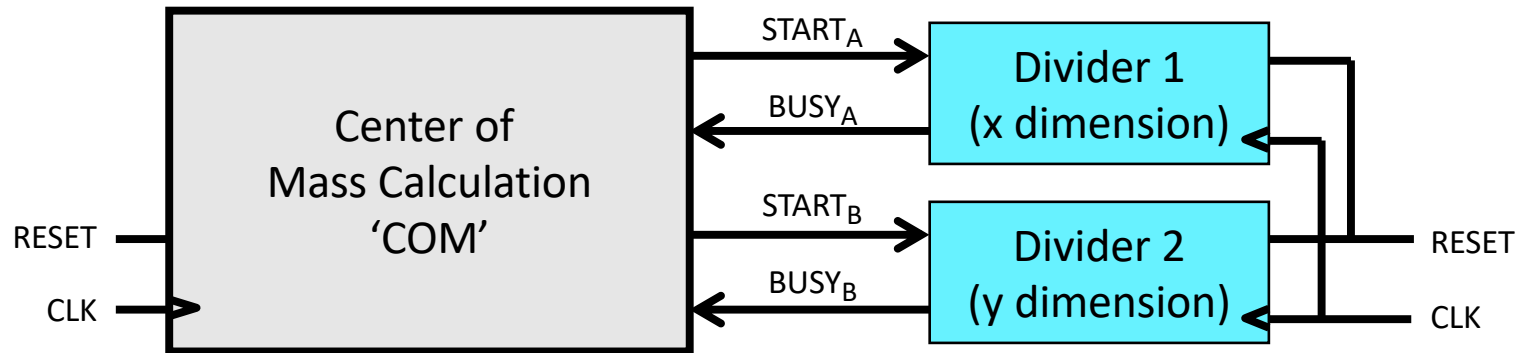
- The Divider from when we first talked about FSMs is an example of a system which might be a minor FSM in part of a larger major's algorithm
- Many things need division, but it would suck to have to rewrite it repeatedly.
- We want you to get practice with that in Week 5's lab

```
3
4 module divider #(parameter WIDTH = 32)
5 (
6     input wire clk,
7     input wire rst,
8     input wire[WIDTH-1:0] dividend,
9     input wire[WIDTH-1:0] divisor,
10    input wire data_in_valid,
11    output logic[WIDTH-1:0] quotient,
12    output logic[WIDTH-1:0] remainder,
13    output logic data_out_valid,
14    output logic error,
15    output logic busy
16);
17 logic [WIDTH-1:0] quotient_g;
18 logic [WIDTH-1:0] dividend_h;
19 logic [WIDTH-1:0] divisor_h;
20 enum {RESTING, DIVIDING} state;
21 always_ff @(posedge clk)begin
22     if (rst)begin
23         quotient_g <= 0;
24         dividend_h <= 0;
25         divisor_h <= 0;
26         remainder <= 0;
27         quotient <= 0;
28         busy <= 1'b0;
29         error <= 1'b0;
30         state <= RESTING;
31         data_out_valid <= 1'b0;
32     end else begin
33         case (state)
34             RESTING: begin
35                 if (data_in_valid)begin
36                     state <= DIVIDING;
37                     quotient_g <= 0;
38                     dividend_h <= dividend;
39                     divisor_h <= divisor;
40                     busy <= 1'b1;
41                     error <= 1'b0;
42                 end
43                 data_out_valid <= 1'b0;
44             end
45             DIVIDING: begin
46                 if (dividend_h <= 0)begin
47                     state <= RESTING; //similar to return statement
48                     remainder <= dividend_h;
49                     quotient <= quotient_g;
50                     busy <= 1'b0; //tell outside world i'm done
51                     error <= 1'b0;
52                     data_out_valid <= 1'b1; //good stuff!
53                 end else if (divisor_h == 0)begin
54                     state <= RESTING;
55                     remainder <= 0;
56                     quotient <= 0;
57                     busy <= 1'b0; //tell outside world i'm done
58                     error <= 1'b1; //ERROR
59                     data_out_valid <= 1'b1; //valid ERROR
60                 end else if (dividend_h < divisor_h) begin
61                     state <= RESTING;
62                     remainder <= dividend_h;
63                     quotient <= quotient_g;
64                     busy <= 1'b0;
65                     error <= 1'b0;
66                     data_out_valid <= 1'b1; //good stuff!
67                 end else begin
68                     //state staying in.
69                     state <= DIVIDING;
70                     quotient_g <= quotient_g + 1'b1;
71                     dividend_h <= dividend_h - divisor_h;
72                 end
73             end
74         endcase
75     end
76 end
77 endmodule
78 `default_nettype wire
79
```

Center of Mass Calculation in Lab05

- You will write a center-of-mass calculator that is best thought of as an FSM.
- For each frame of video:
 - Sum the x location and y location of every active pixel you come across
 - Keep track of how many pixels you've encountered
 - At end of frame (or beginning of next one, divide the two sums by the number of active pixels
 - This will give an average X,Y
- Division takes time!
- Need to create a major/minor FSM system

Lab 05 Center of Mass

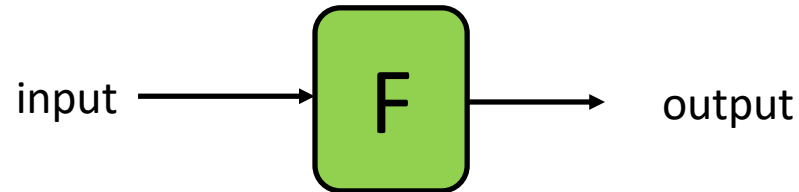


- C.O.M. will be in a “data collection state” during the active portion of a video frame
- When the frame’s active part is done, it needs to calculate the average x,y position of the “hot” pixels it has observed.
- To divide, the C.O.M. module hands the values it needs divided off to two separate dividers.
- C.O.M. waits on them monitoring their BUSY signals
- They can do division separately (in parallel)
- When done, they report back to the C.O.M with their result
- C.O.M. reports to outside world its calculation

Pipelining

How to make sure signals are balanced going through a sequence of operations.

Performance Metrics



- Latency (L):
 - time between arrival of new input and generation of corresponding output.
 - For purely combinational circuits this is just t_{pD} .
- Throughput (T):
 - Rate at which new outputs appear.
 - For purely combinational circuits this is just $1/t_{pD}$ or $1/L$.

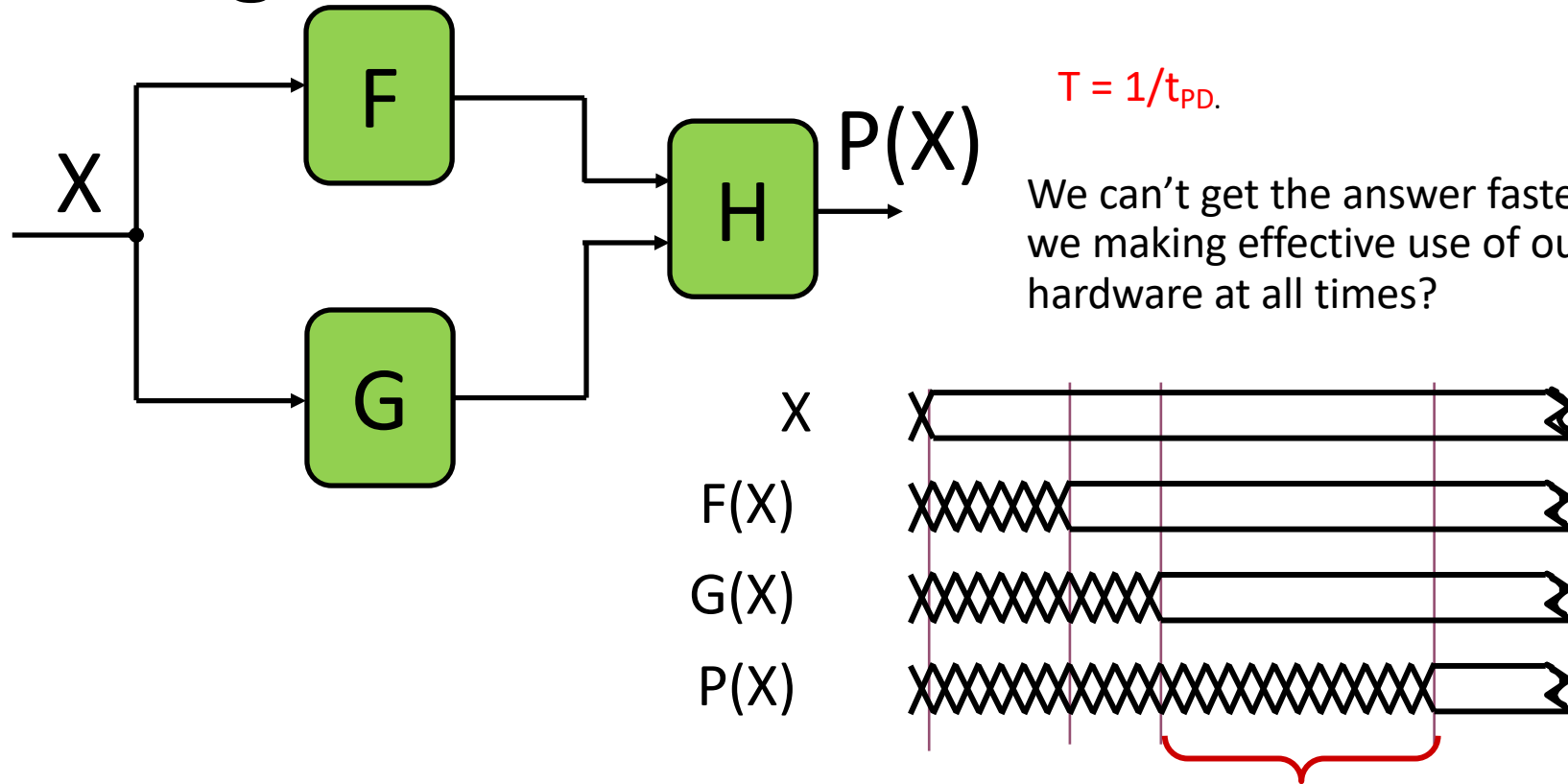
Performance of Combinational Logic

For combinational logic:

$$L = t_{PD},$$

$$T = 1/t_{PD}.$$

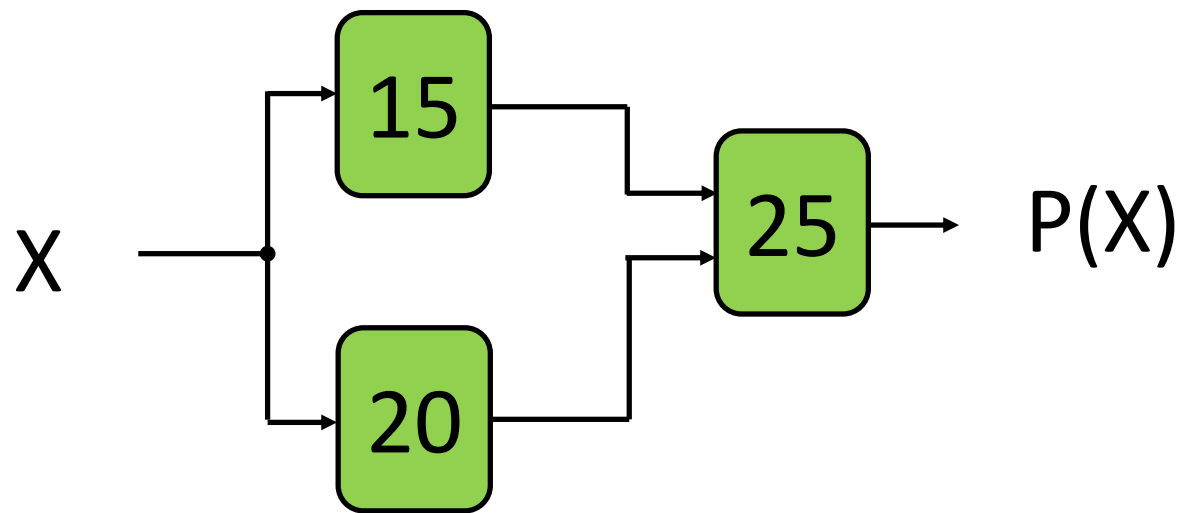
We can't get the answer faster, but are we making effective use of our hardware at all times?



F & G are “idle”, just holding their outputs stable while H performs its computation

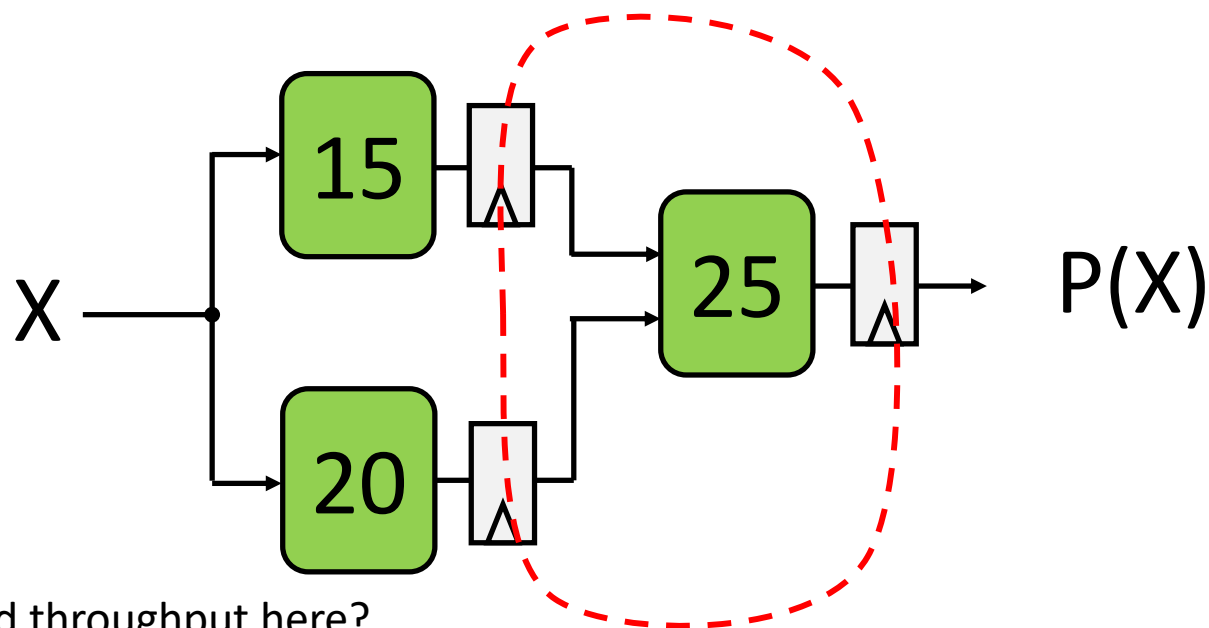
Retiming: A useful transform

Propagation delays indicated by numbers:



Retiming: A useful transform

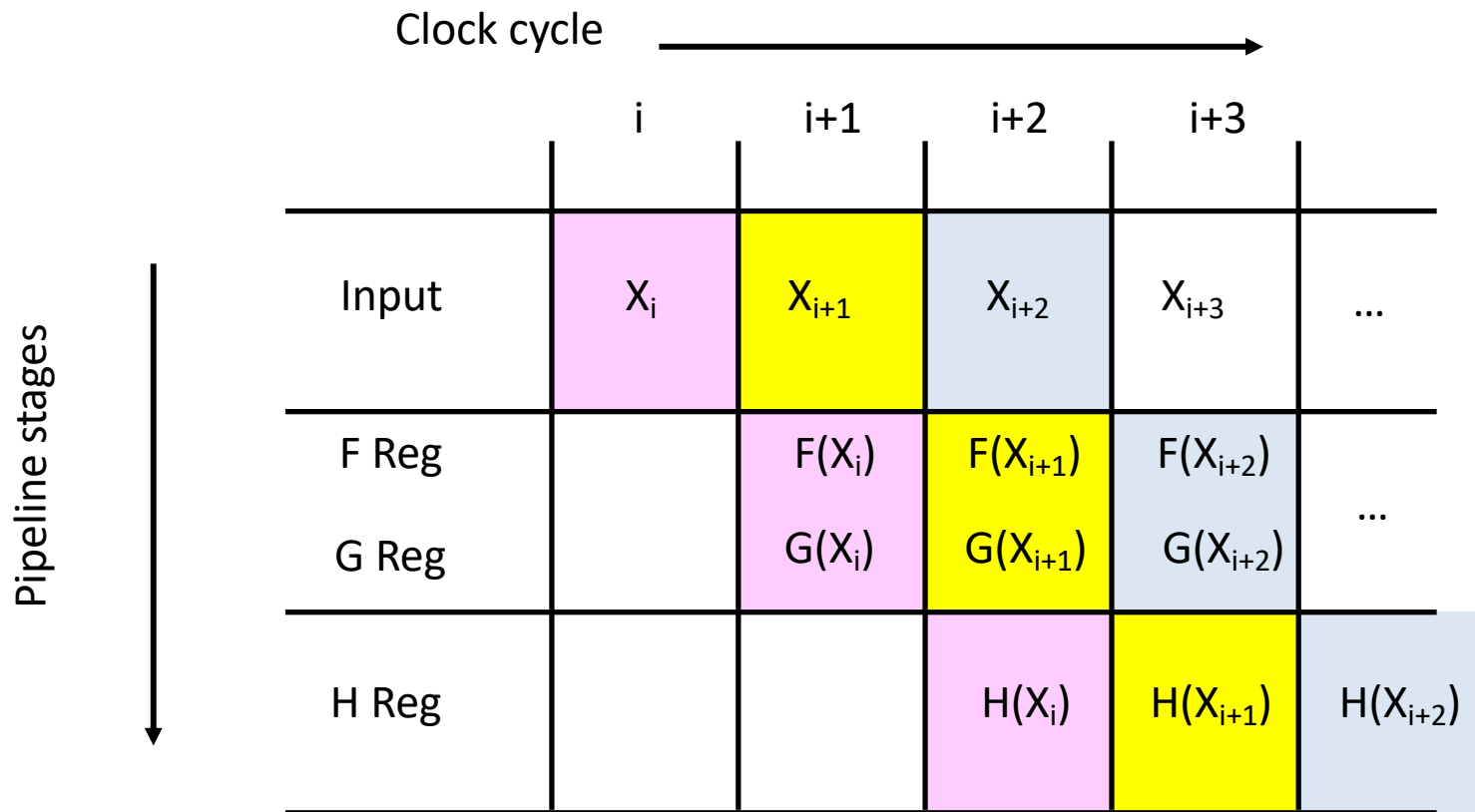
- Add in Flops
- Run clock as fast as possible given the presence of the flops



What is latency and throughput here?

*Assuming ideal registers:
i.e., $t_{PD} = 0$, $t_{SETUP} = 0$*

Pipeline Diagrams



The results associated with a particular set of input data moves diagonally through the diagram, progressing through one pipeline stage each clock cycle.

Pipeline Conventions

- a **K-Stage Pipeline** (“K-pipeline”) is an acyclic circuit having exactly K registers on every path from an input to an output.
- a COMBINATIONAL CIRCUIT is thus a 0-stage pipeline.

*acyclic means not cyclical/loop

Pipeline Conventions

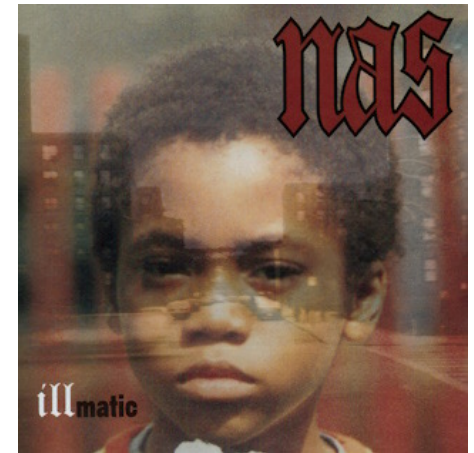
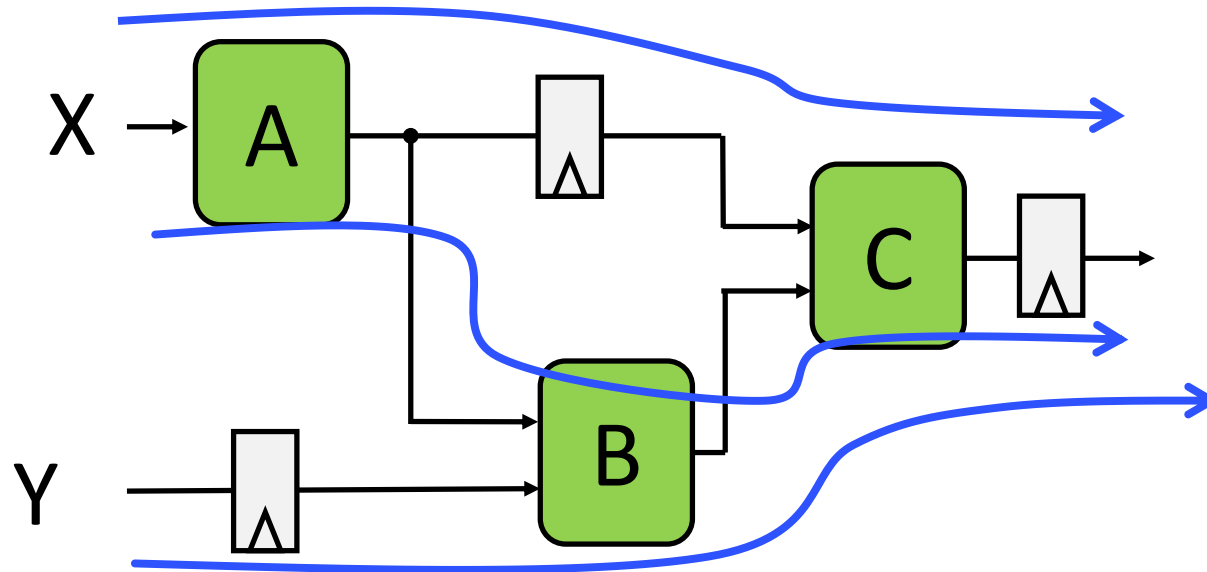
- CONVENTION:
 - Every pipeline stage, hence every K-Stage pipeline, has a register on its OUTPUT (not on its input).
- ALWAYS:
 - The CLOCK common to all registers must have a period sufficient to cover propagation over combinational paths PLUS (input) register t_{PD} PLUS (output) register t_{SETUP} .

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

The LATENCY of a K-pipeline is K times the period of the clock common to all registers.

The THROUGHPUT of a K-pipeline is the frequency of the clock.

ILL-formed Pipeline



For what value of K is the following circuit a K-Pipeline? none

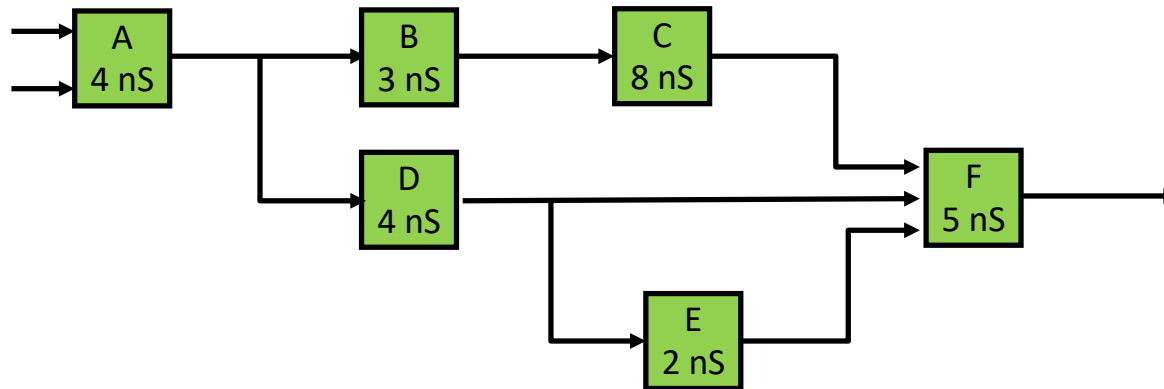
Problem:

Successive inputs get mixed: e.g., $B(A(X_{i+1}), Y_i)$. This happened because some paths from inputs to outputs have 2 registers, and some have only 1!

This CAN'T HAPPEN on a well-formed K pipeline!

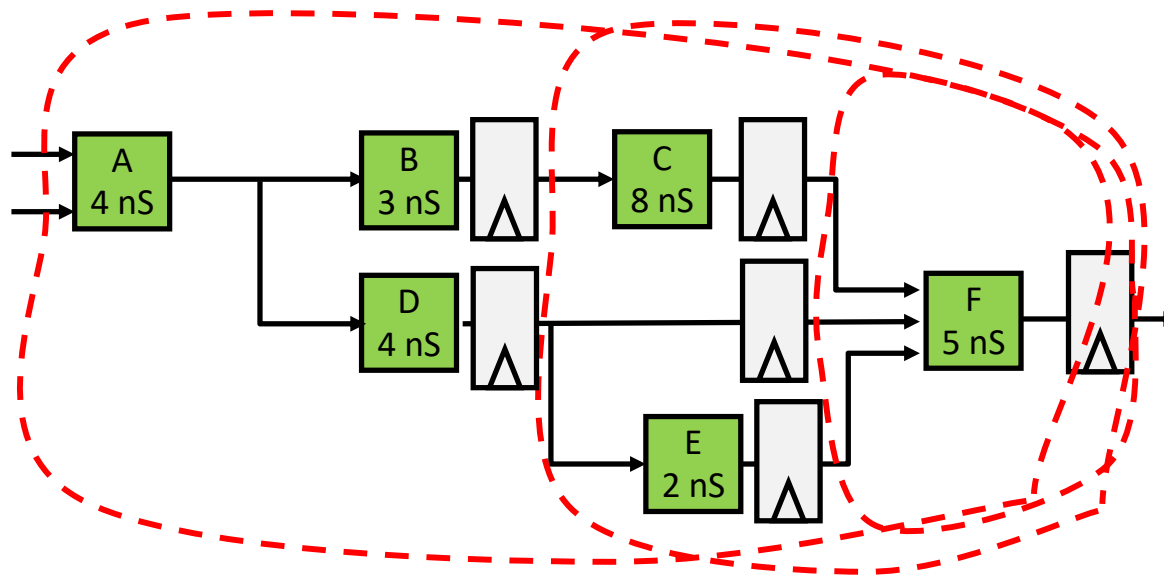
Pipelining

Let's say we want t_{clk} to be 8ns



Pipelining

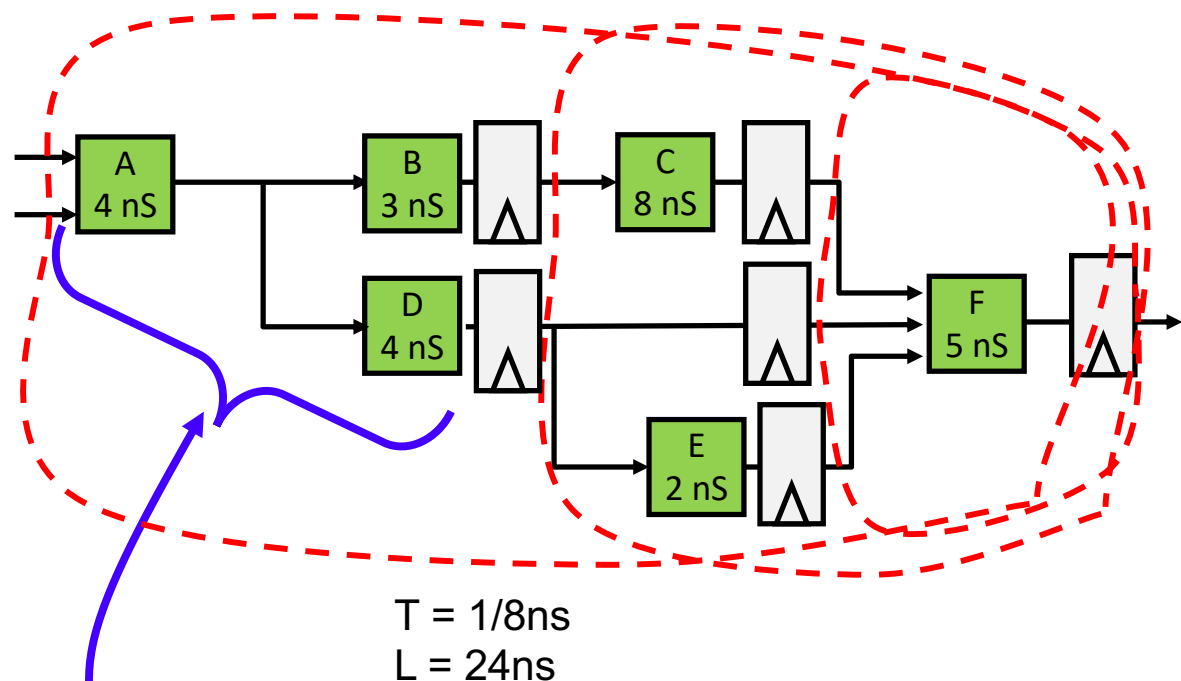
Let's say we want t_{clk} to be 8ns



- **Step 1:** Add a register on the output.
- **Step 2:** From register. Draw a contour backwards that includes as much of the circuit that will fit inside required period. Add registers. Make sure path for every signal is balanced
- Repeat until satisfied with T. Look for redundant registers

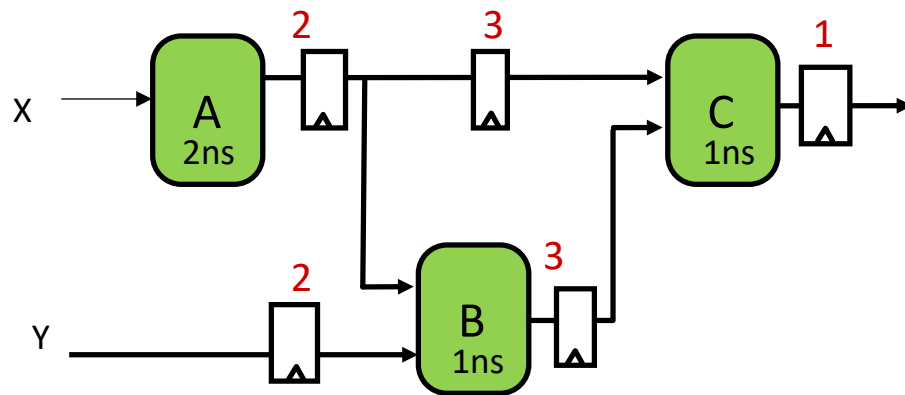
STRATEGY:

Focus your attention on placing pipelining registers around the slowest circuit elements (BOTTLENECKS).



Assuming this interfaces with other modules that have registered outputs the input will chain will be ok ($\leq 8\text{ns}$)

Another Pipeline Example



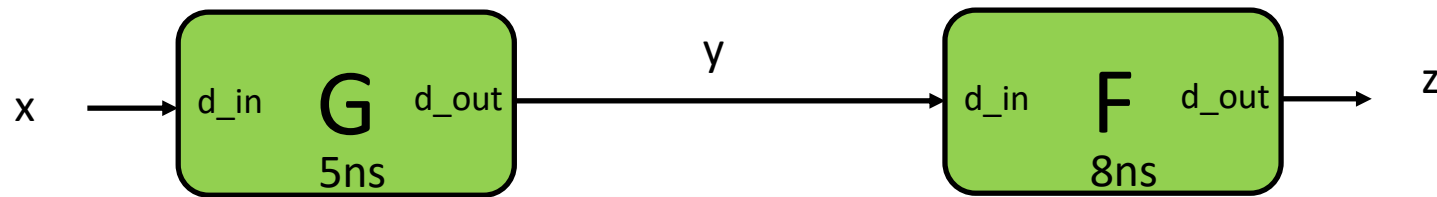
OBSERVATIONS:

- 1-pipeline improves neither L or T.
- T improved by breaking long combinational paths, allowing faster clock.
- Too many stages cost L, don't improve T.
- Back-to-back registers are often required to keep pipeline well-formed.

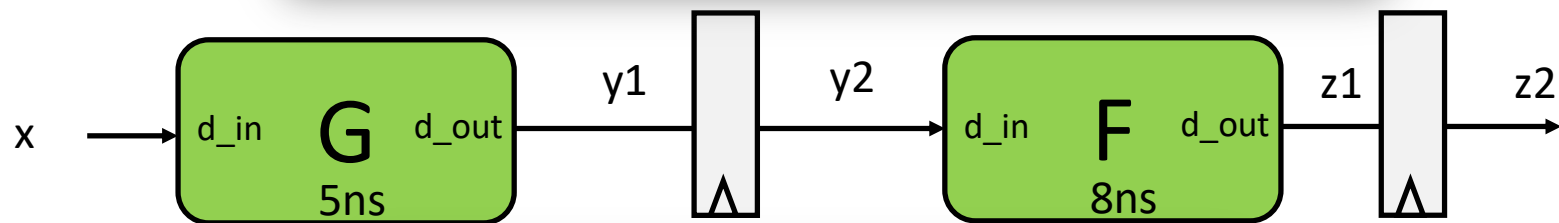
	LATENCY	THROUGHPUT
0-pipe:	4ns	1/4ns
1-pipe:	4ns	1/4ns
2-pipe:	4ns	1/2ns
3-pipe:	6ns	1/2ns

Better throughput here means we can run at higher clock rate

Pipelining in Verilog



```
1
2  logic x,y,z;
3  //G and F are purely combinational modules.
4  G myg (.d_in(x), .d_out(y));
5  F myf (.d_in(y), .d_out(z));
```



```
1
2  logic x,y1,y2,z1,z2;
3  //G and F are purely combinational modules.
4  G myg (.d_in(x), .d_out(y1));
5  F myf (.d_in(y2), .d_out(z1));
6
7  always_ff @(posedge clk)begin
8      y2 <= y1;
9      z2 <= z1;
10 end
11
```

How often should you be adding FlipFlops in your FPGA?

- This comes with experience and getting to know your system.
- AND by using the feedback that the tool gives you after place and route.
- Most of what you want to do really is some form of math.
- So knowing how much math you can do in a clock cycle is useful

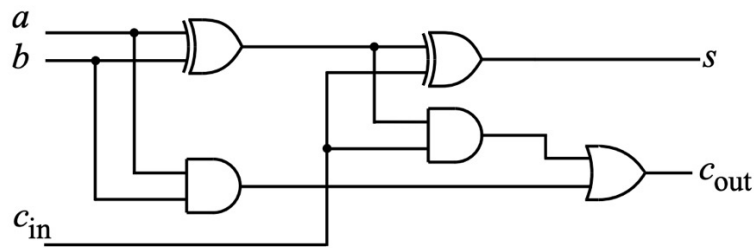
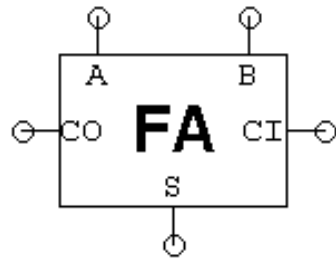
The Complexity of Math Operations

Let's look at some basic math circuits:

$+$, $-$, $*$, $/$

“Full Adder” building block

The “half adder” circuit has only the A and B inputs (no carry)
Full adders handle carry bits



A	B	CI	S	CO
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Adder: a circuit that does addition

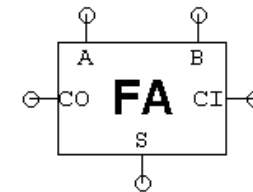
Here's an example of binary addition as one might do it by "hand":

Adding two N-bit numbers produces an (N+1)-bit result

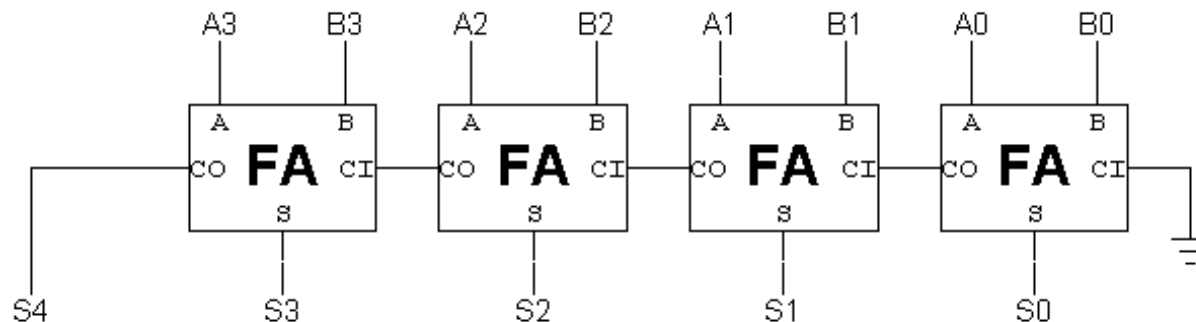
$$\begin{array}{r} 1101 \\ + 0101 \\ \hline 10010 \end{array}$$

Carries from previous column

If we build a circuit that implements one column:



we can quickly build a circuit to add two 4-bit numbers...



"Ripple-carry adder"

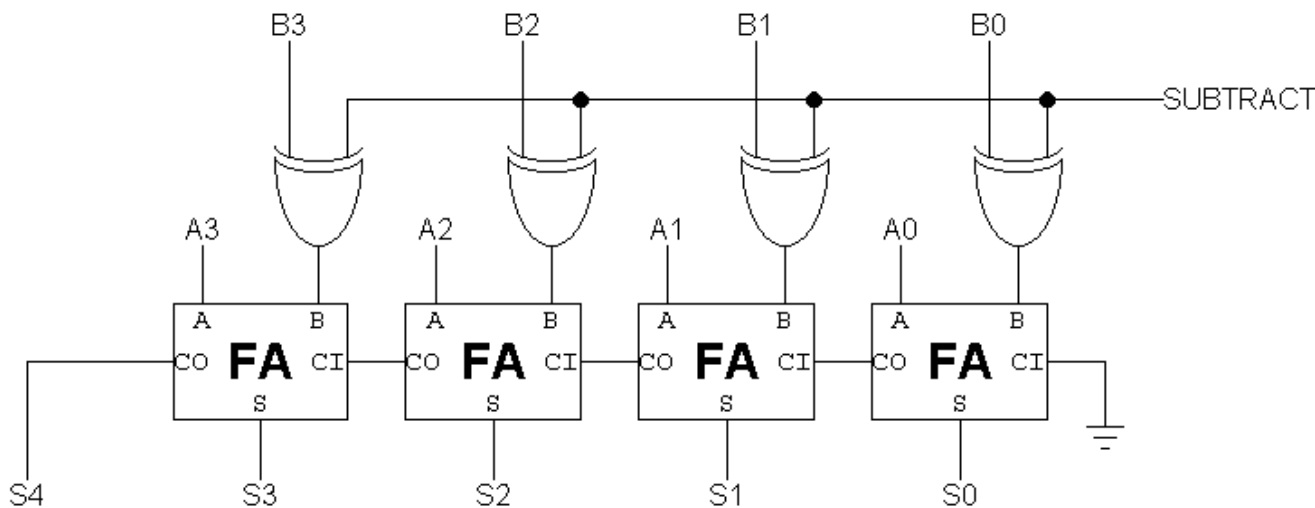
Subtraction: $A - B = A + (-B)$

Using 2's complement representation: $-B = \sim B + 1$

\sim = bit-wise complement



So let's build an arithmetic unit that does both addition and subtraction.
Operation selected by control input:

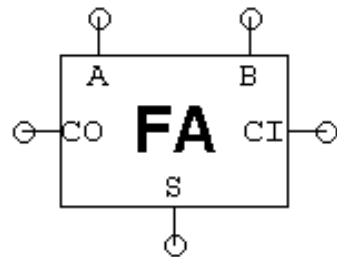


When SUBTRACT is 1:

- *Invert each bit*
- *Start with a Carry of 1 (same as adding 1)*

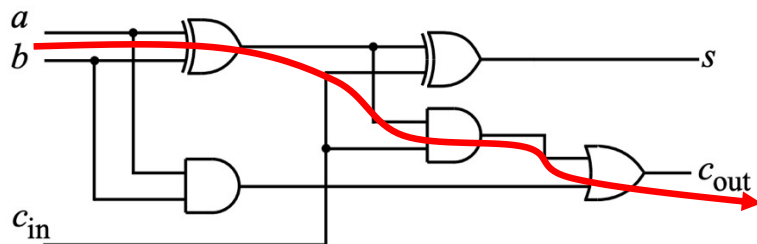
“Full Adder” building block

The “half adder” circuit has only the A and B inputs



A	B	CI	S	CO
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

What’s the critical path through this circuit?

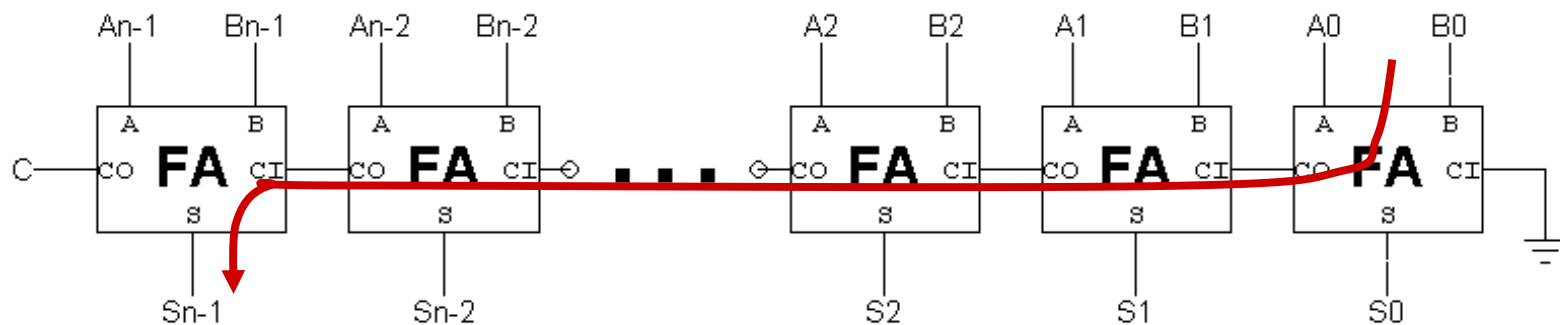


t_{pd} dictated by carry path!

Can also rewrite the carry path as: $c_{out} = (a \& c_{in}) | (b \& c_{in}) | (a \& b)$

Speed: t_{PD} of Ripple-carry Adder

$$C_O = AB + AC_I + BC_I$$



Worst-case path: carry propagation from LSB to MSB, e.g., when adding 11...111 to 00...001.

$$t_{PD} = (N-1) * \underbrace{(t_{PD,OR} + t_{PD,AND})}_{\text{CI to CO}} + \underbrace{t_{PD,XOR}}_{\text{CI}_{N-1} \text{ to } S_{N-1}} \approx \Theta(N)$$

$\Theta(N)$ is read “order N” : means that the latency of our adder grows at worst in proportion to the number of bits in the operands.

$$t_{\text{adder}} = (N-1)t_{\text{carry}} + t_{\text{sum}}$$

The Carry Path Becomes Limiting

- Solution is the Carry-Look-ahead Adder:

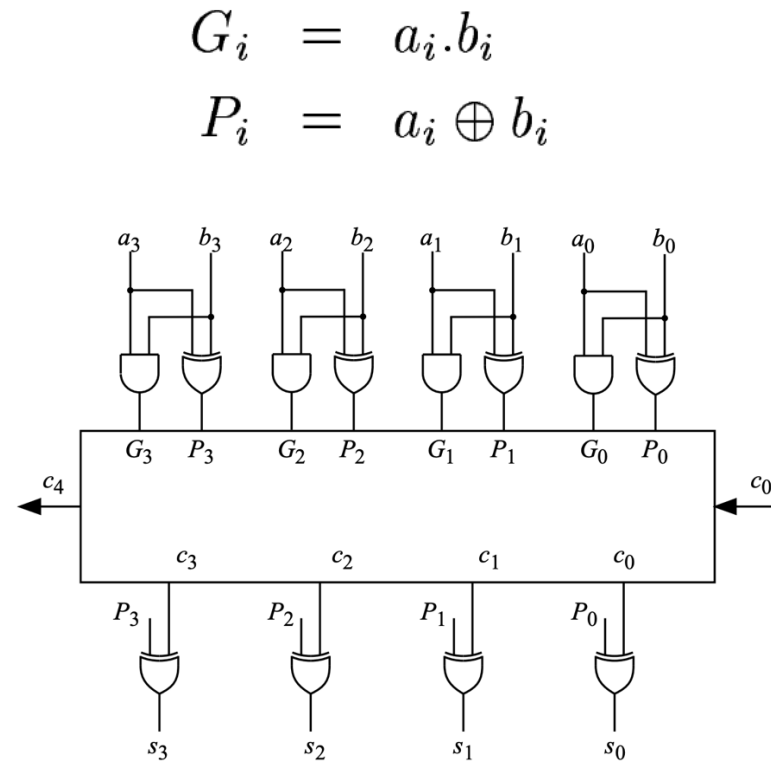
$$c_1 = G_0 + P_0.c_0$$

$$c_2 = G_1 + P_1.G_0 + P_1.P_0.c_0$$

$$c_3 = G_2 + P_2.G_1 + P_2.P_1.G_0 + P_2.P_1.P_0.c_0$$

$$c_4 = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3.P_2.P_1.G_0 + P_3.P_2.P_1.P_0.c_0$$

Can do some
factoring/redesign and cut-
down on t_{pd} of the carry path

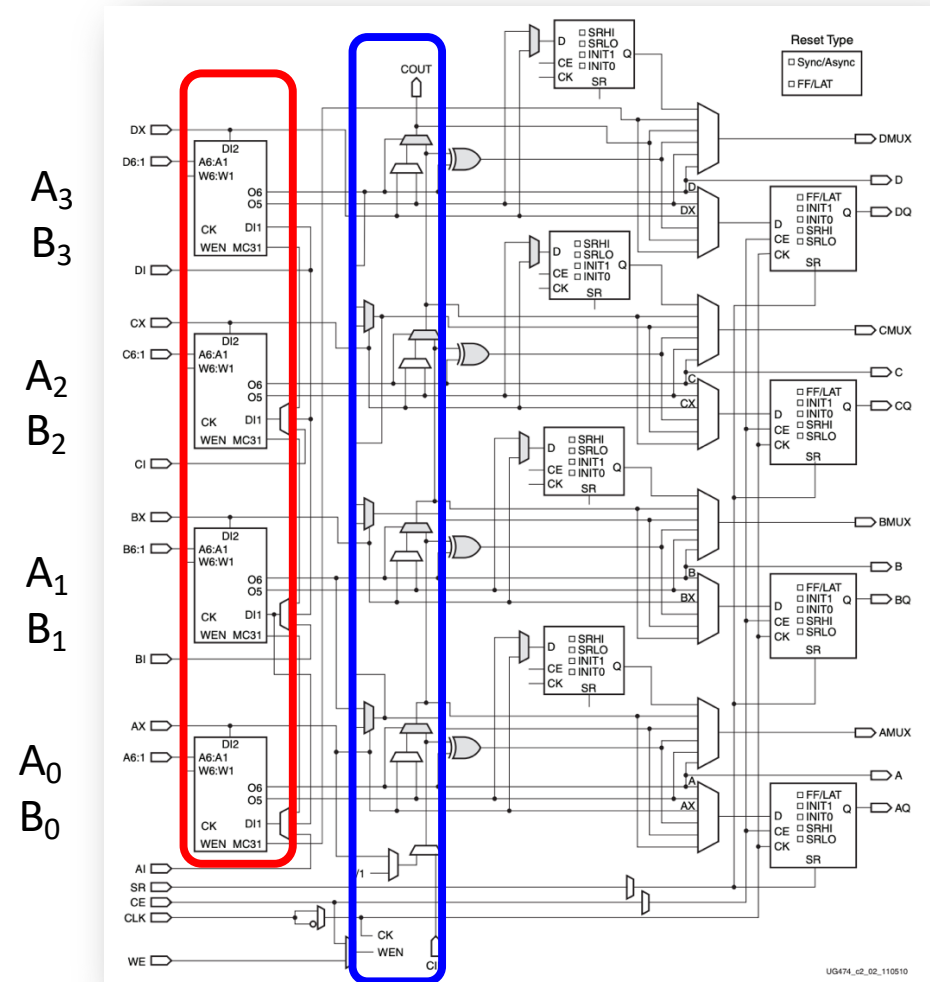


https://www.ece.uvic.ca/~fayez/courses/ceng465/lab_465/project1/adders.pdf

Logic Slices in FPGA Can Add/Subtract

- Can synthesize the addition of two 4 bit numbers with fast carry

$$\begin{array}{r} A_3A_2A_1A_0 \\ + B_3B_2B_1B_0 \\ \hline \end{array}$$



Fast Carry-Chain

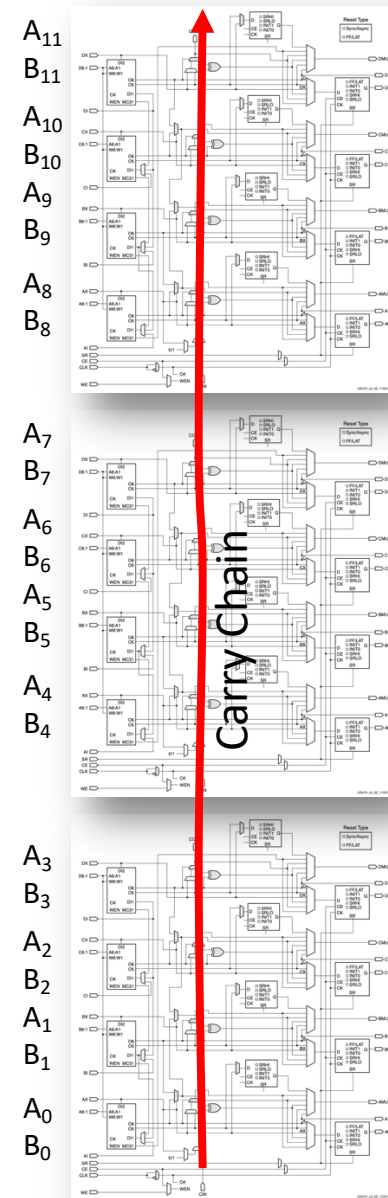
Series 7 Logic Slice

Add/Subtract on the FPGA

- + and – can be done combinationally very quickly:
 - 32 bit add can be done in a clock cycle (<10 ns) pretty easily
 - Several smaller adds ($A+B+C+D$) can be done in clock cycle as well (10 ns)
- CLBs (the generic function generators, of which we have a lot) are capable of being chained together to allow large adds.

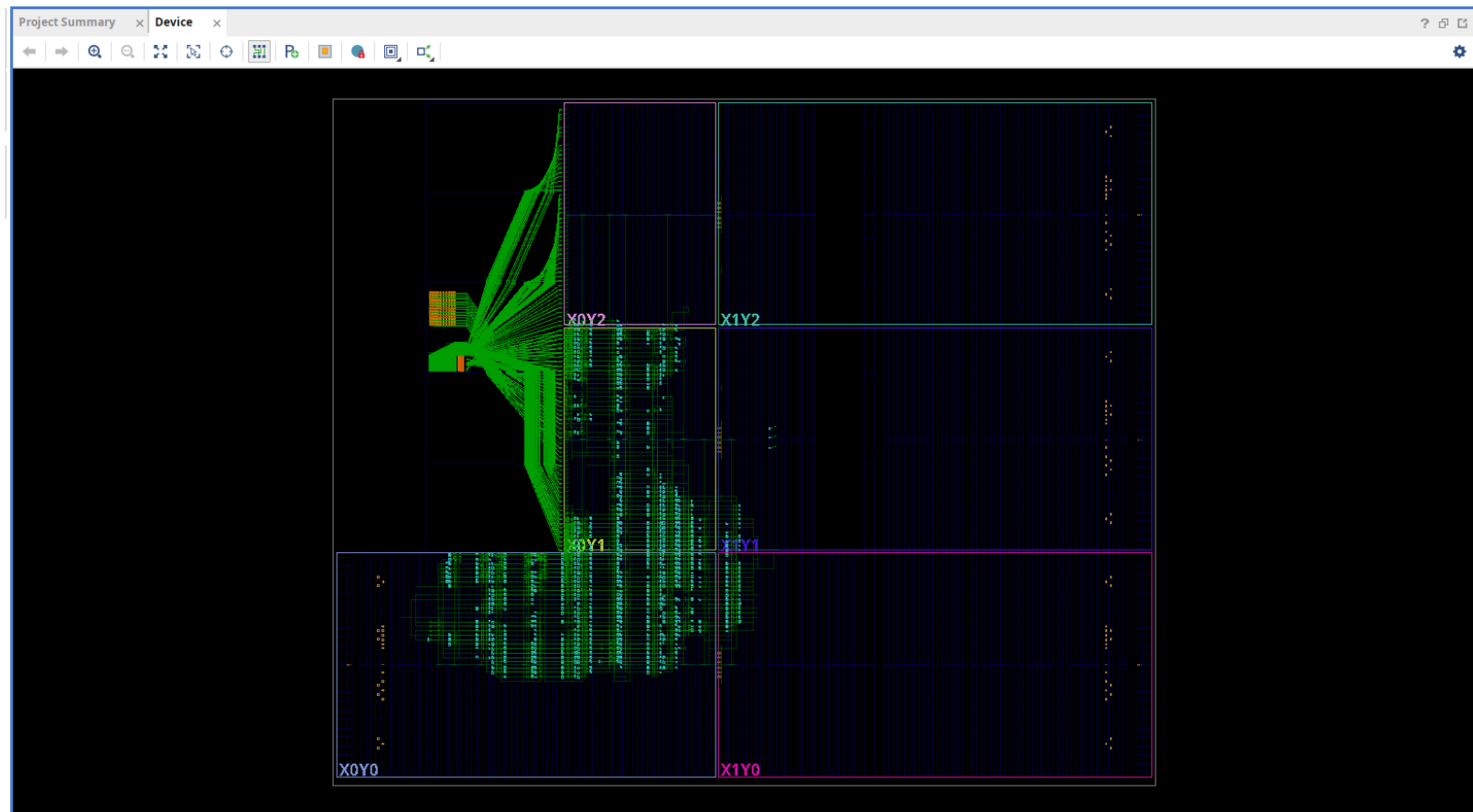
Slices can stack to give more bits

$$\begin{array}{r} A_{11}A_{10}A_9A_8A_7A_6A_5A_4A_3A_2A_1A_0 \\ + B_{11}B_{10}B_9B_8B_7B_6B_5B_4B_3B_2B_1B_0 \end{array}$$



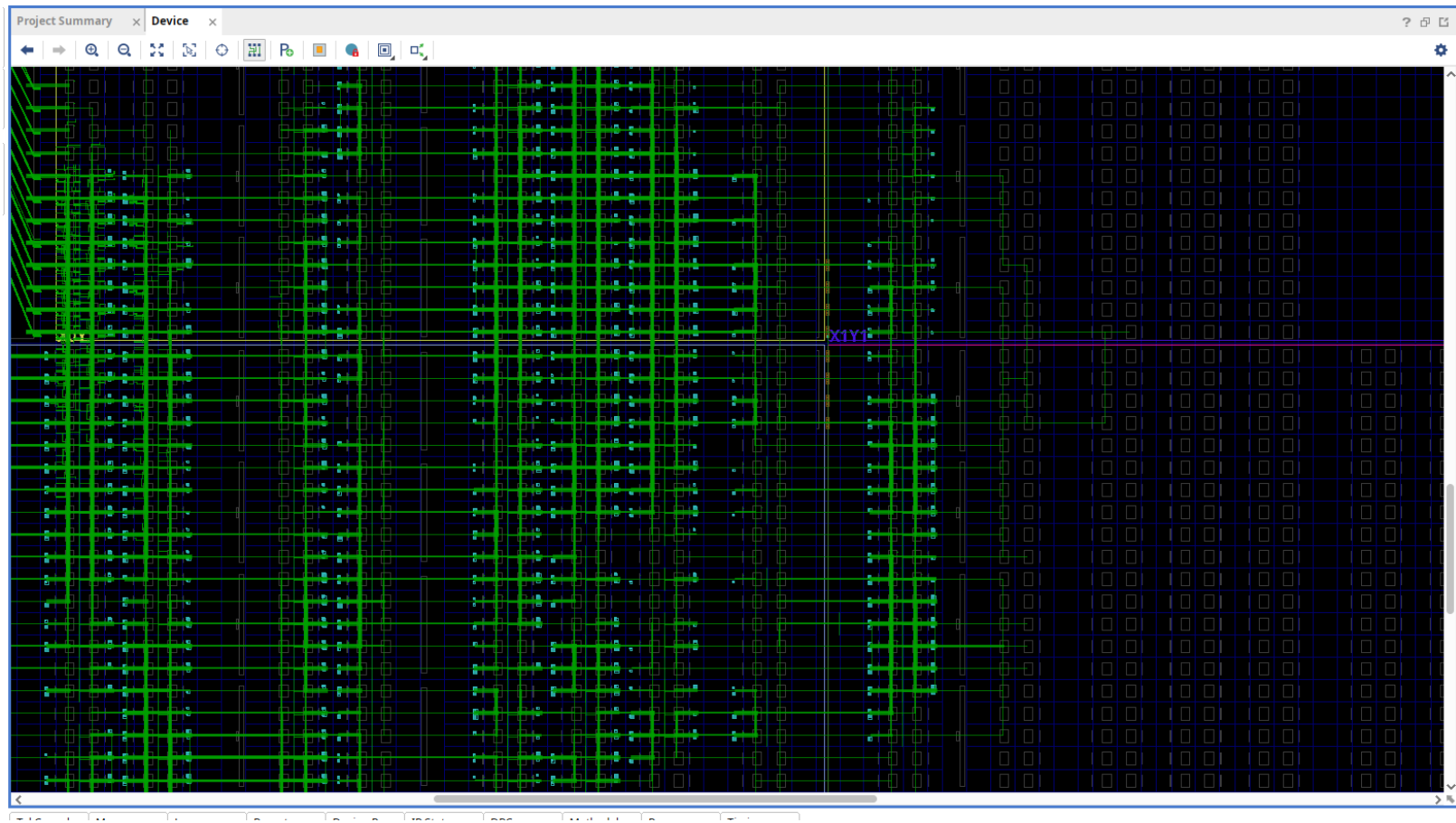
Aside in the Vivado GUI...

- Can Actually see everything that get's synthesized...



Aside in the Vivado GUI...

- Zoom in...

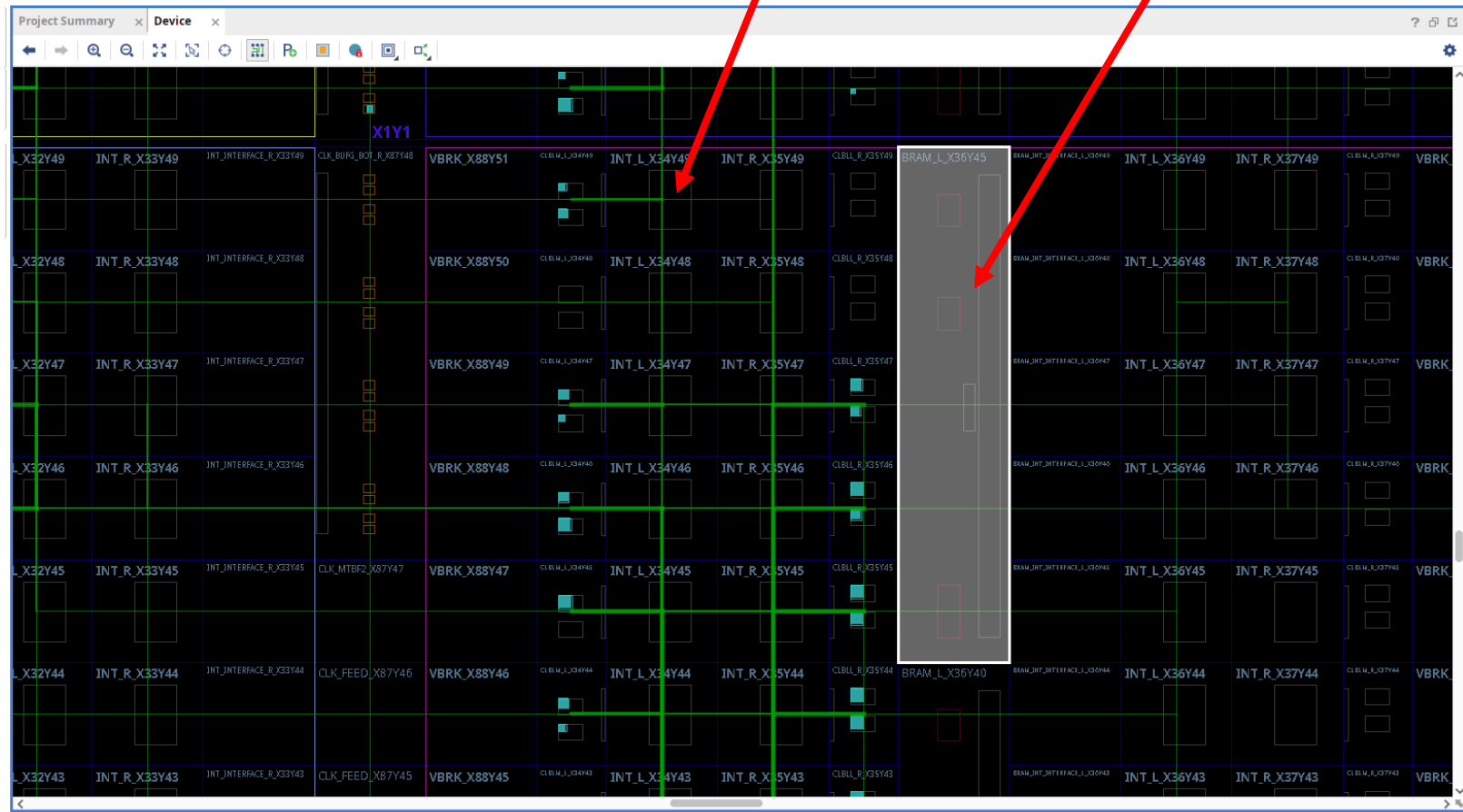


Aside in the Vivado GUI...

- Zoom in...

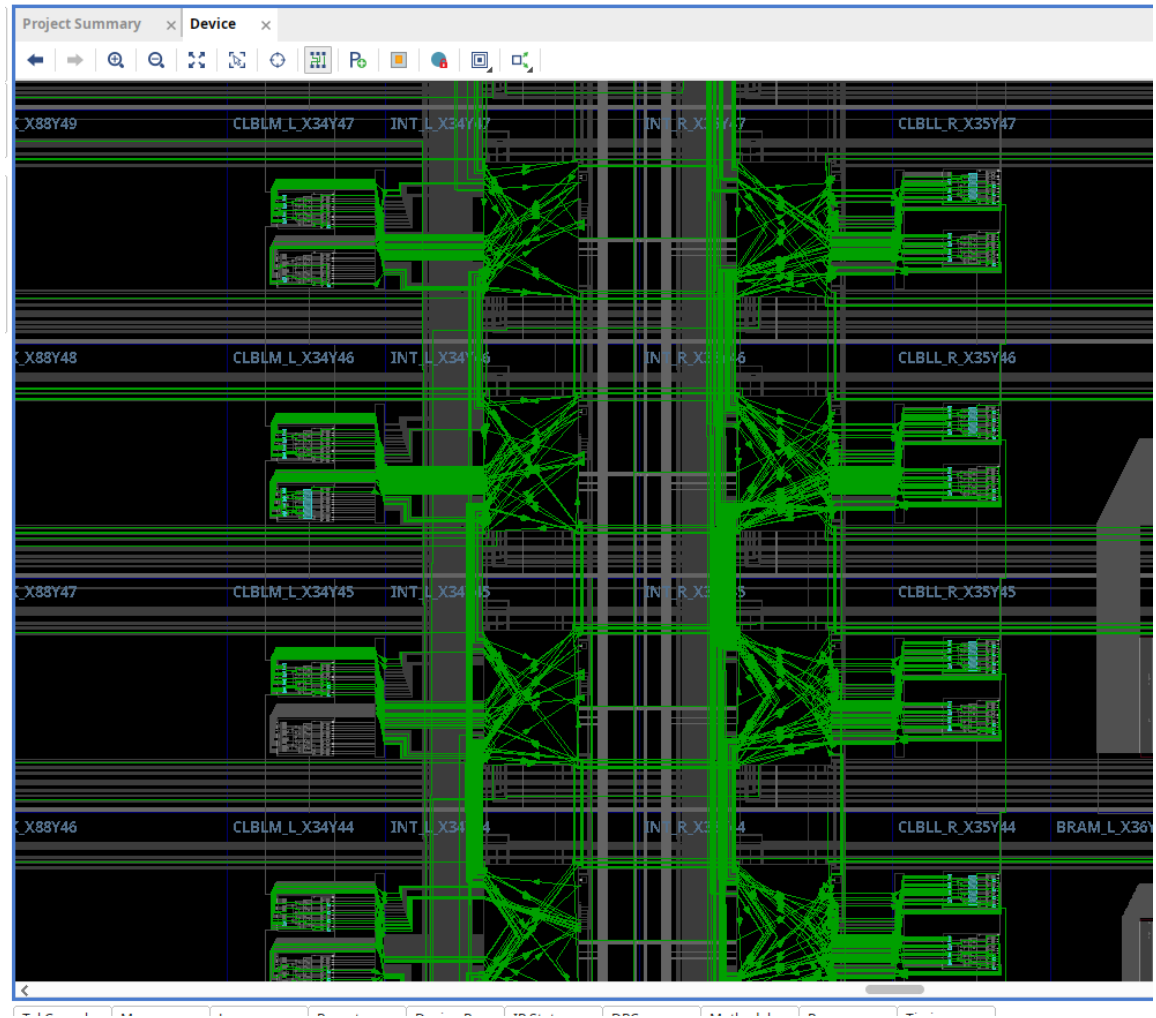
That's logic slice

That's a BRAM



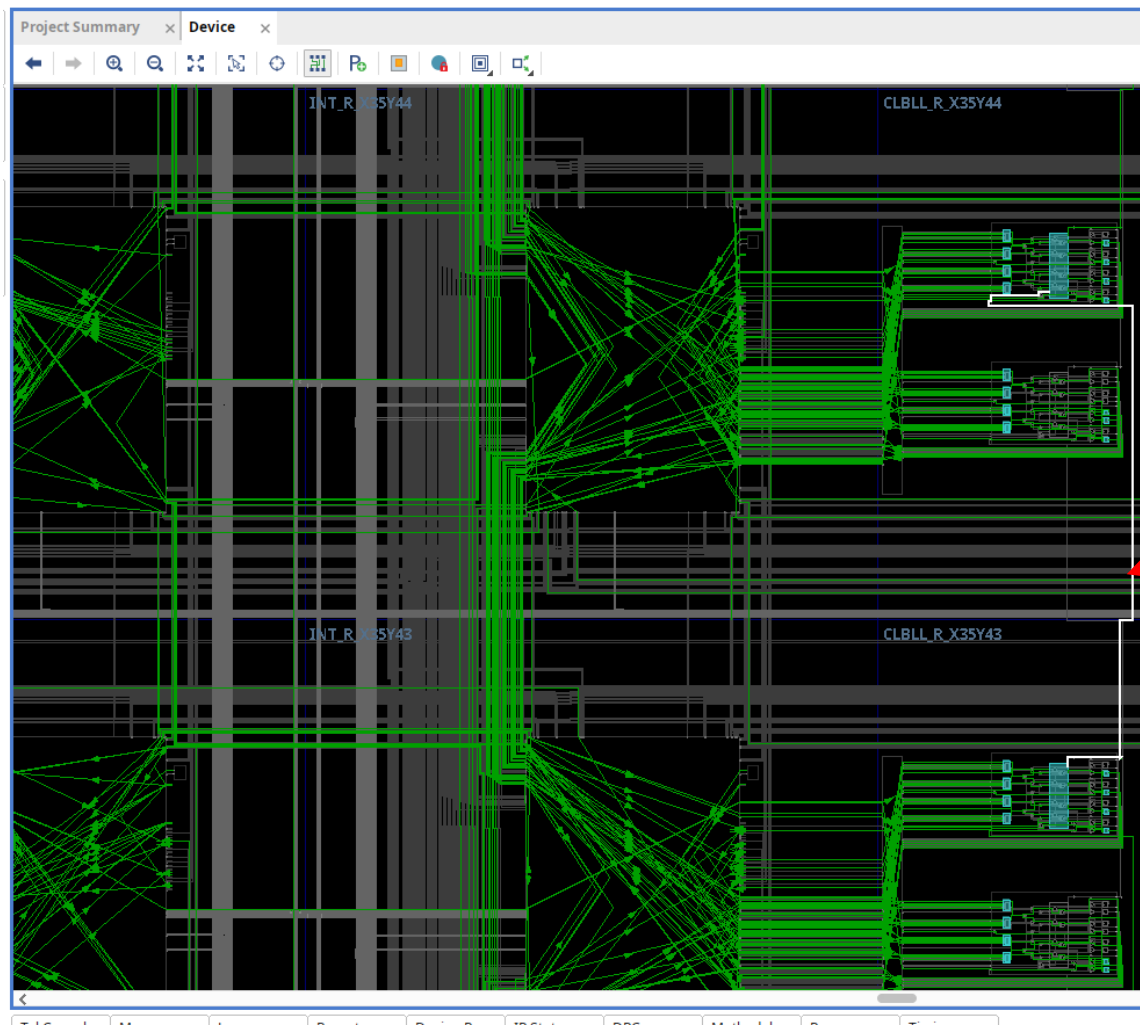
Aside in the Vivado GUI...

- Zoooooom in...



Aside in the Vivado GUI...

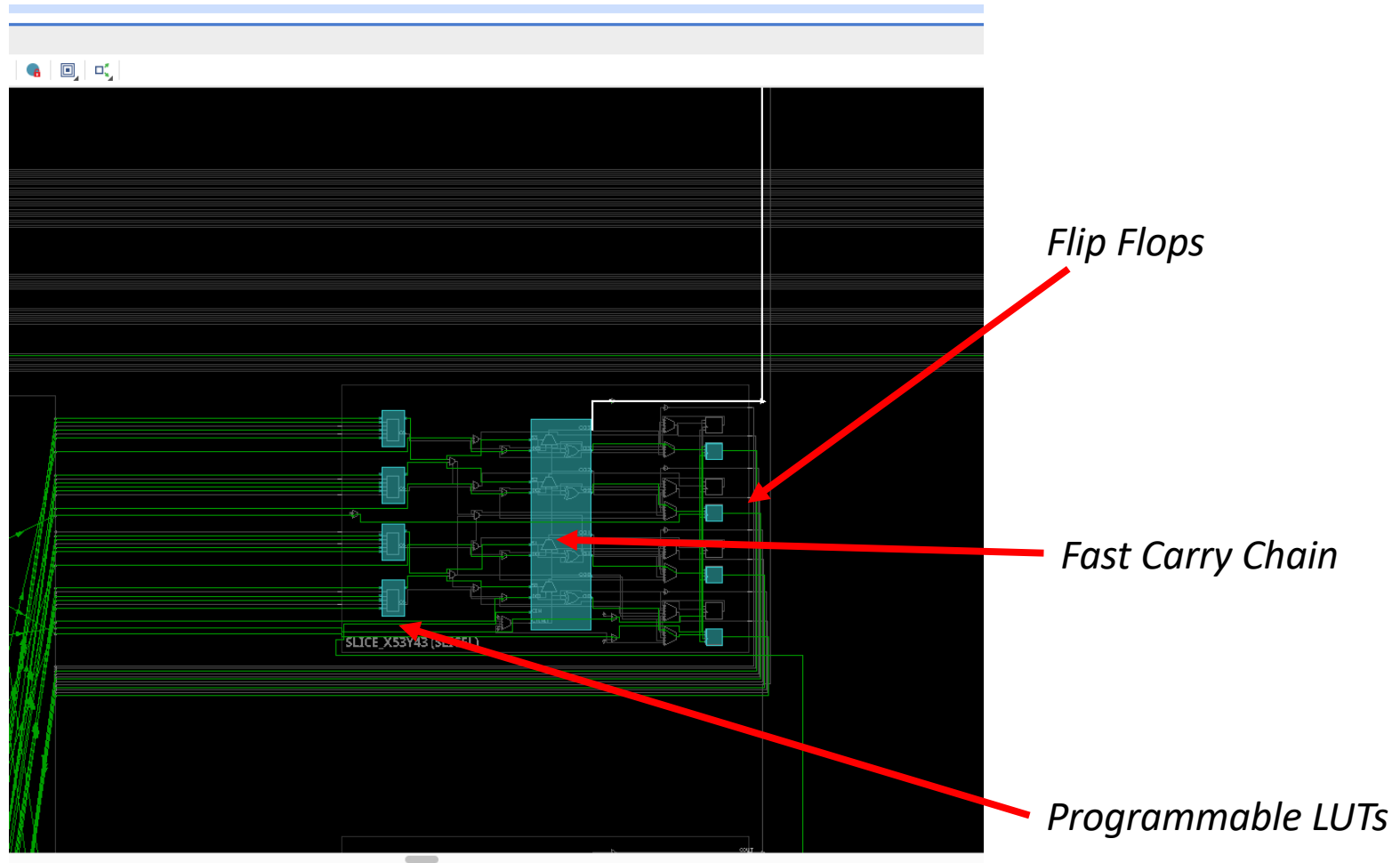
- Zoookooooom in.



*Individual wire
highlighted in white
Connecting one fast
carry chain to the next
up in the column*

Aside in the Vivado GUI...

- Zooooooooommm...in.....

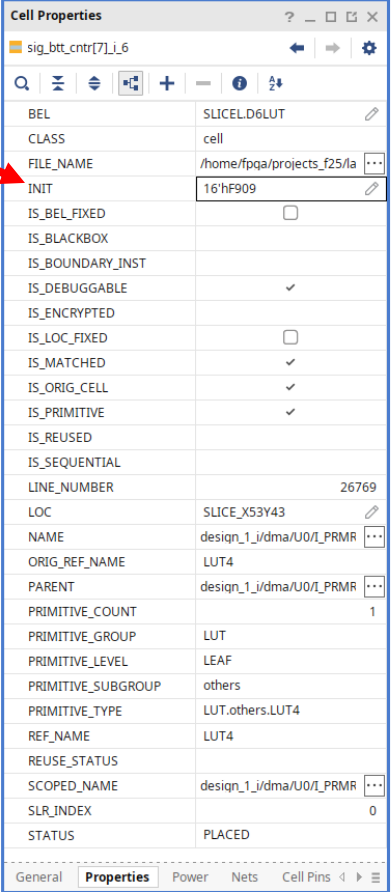


Aside in the Vivado GUI...

- Zooooooooooooooooooooommm...in.....

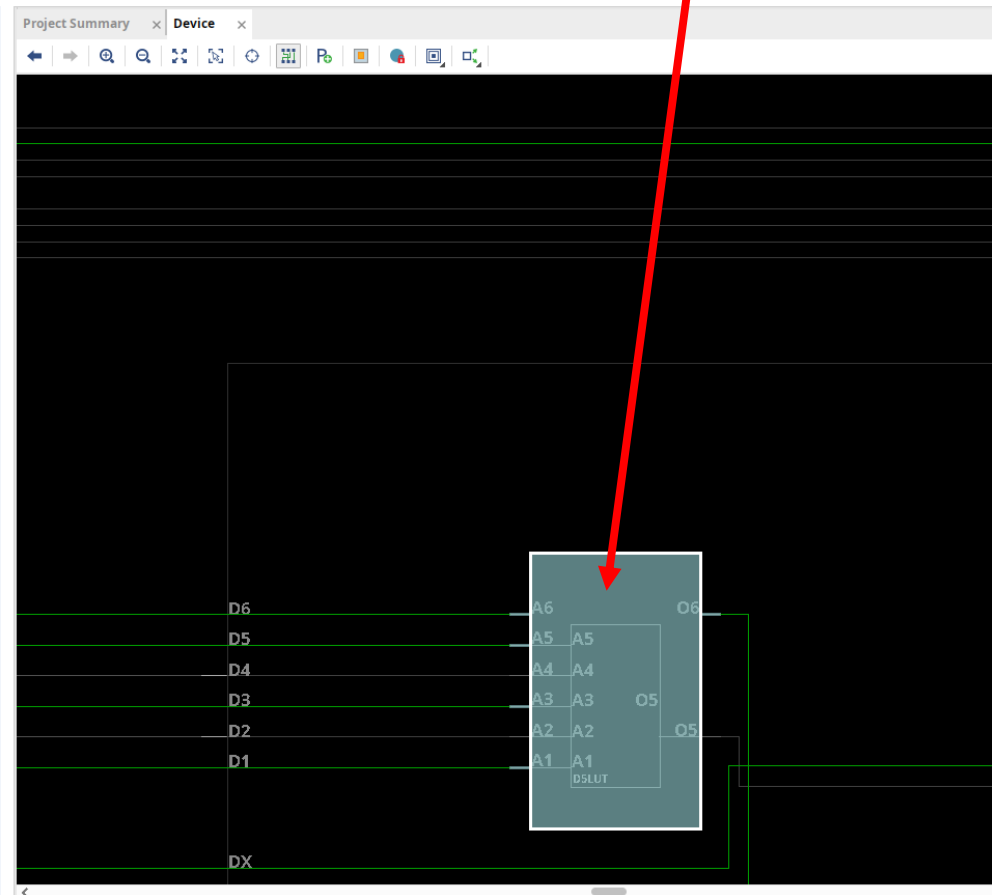
Individual LUT

64 bit
program
For that 6-
input LUT



The image shows the 'Cell Properties' window in Vivado. The title bar says 'Cell Properties' and the subtitle is 'sig_btt_ctr[7].i_6'. The window contains a table of properties for a LUT4 cell. A red arrow points from the text '64 bit program For that 6-input LUT' to the 'INIT' field.

Property	Value
BEL	SLICEL.D6LUT
CLASS	cell
FILE_NAME	/home/fpga/projects_f25/la
INIT	16'hF909
IS_BEL_FIXED	<input type="checkbox"/>
IS_BLACKBOX	
IS_BOUNDARY_INST	
IS_DEBUGGABLE	<input checked="" type="checkbox"/>
IS_ENCRYPTED	
IS_LOC_FIXED	<input type="checkbox"/>
IS_MATCHED	<input checked="" type="checkbox"/>
IS_ORIG_CELL	<input checked="" type="checkbox"/>
IS_PRIMITIVE	<input checked="" type="checkbox"/>
IS_REUSED	
IS_SEQUENTIAL	
LINE_NUMBER	26769
LOC	SLICE_X53Y43
NAME	design_1_i/dma/U0/I_PRRM
ORIG_REF_NAME	LUT4
PARENT	design_1_i/dma/U0/I_PRRM
PRIMITIVE_COUNT	1
PRIMITIVE_GROUP	LUT
PRIMITIVE_LEVEL	LEAF
PRIMITIVE_SUBGROUP	others
PRIMITIVE_TYPE	LUT.others.LUT4
REF_NAME	LUT4
REUSE_STATUS	
SCOPED_NAME	design_1_i/dma/U0/I_PRRM
SLR_INDEX	0
STATUS	PLACED



+ or - in Verilog

- Generally + or – on its own will get synthesized using logic slices unless specified
- Very large additions or subtractions may start to take *too long*!
- But doing a couple 32 bit adds in a 10 ns cycle should be possible...

**"too" is really with respect to a clock. If you're running on a 10 MHz clock, then things are different!*

But also the stuff around it matters too!

- Keep track of the stuff before and after your math.
- If you have a ton of if/else/ifs...or if you have a super-deeply nested if/if/if/ chain, all that stuff requires logic too.

Also Case Statements are Good

- If/elses and even parallel if's as shown on the previous page get encoded as priority logic

```
always_ff @(posedge clk)begin
  if (state == IDLE) begin
    q <= y;
  end else if (state == FIRST) begin
    q <= z;
  end else if (state == SECOND) begin
    q <= zz;
  end else begin
    q <= zzz;
  end
end
```

long combinational path

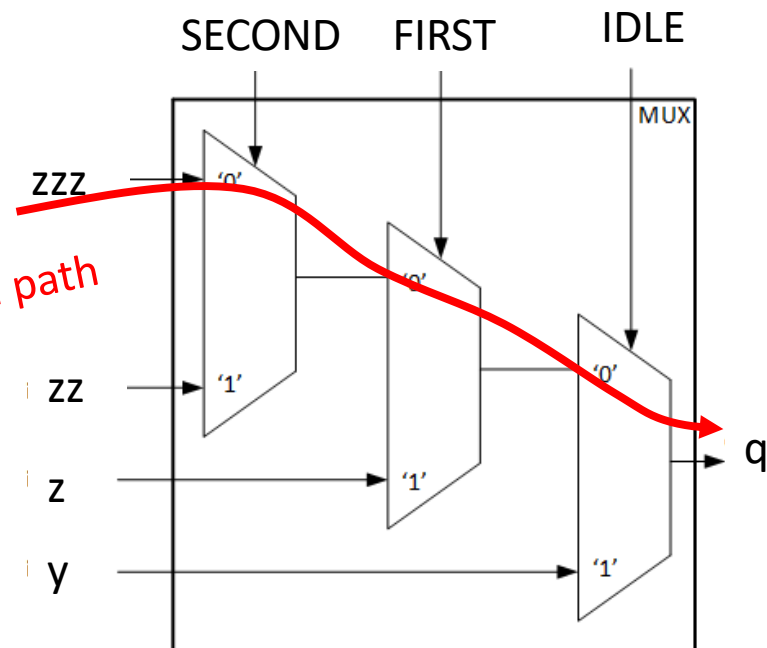


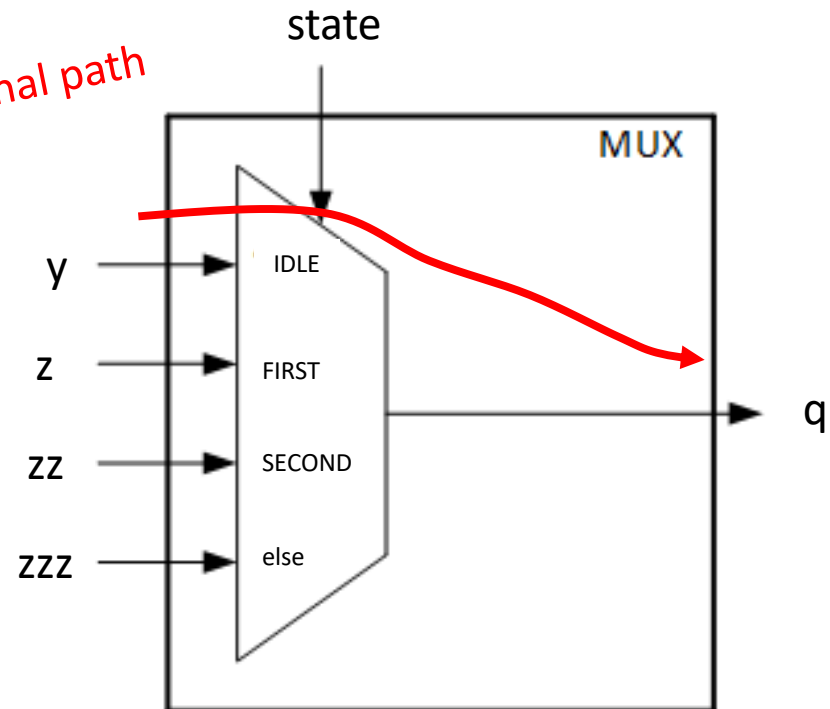
Fig 1

Also Case Statements are Good

- If logic can be structured without priority, then do it! Can yield simpler underlying logic.

```
always_ff @(posedge clk)begin
  case(state)
    IDLE: begin
      q <= y;
    end
    FIRST: begin
      q <= z;
    end
    SECOND: begin
      q <= zz;
    end
    default: begin
      q <= zzz;
    end
  endcase
end
```

shorter combinational path



https://www.kevnugent.com/2020/10/22/verilog-blogpost_002/

The stuff around it matters too!

- Keep track of the stuff before and after your math.
- If you have a ton of if/else/ifs...or if you have a super-deeply nested if/if/if/ chain, all that stuff requires logic too.
- **Also think about the stuff being used to calculate the if/else stuff.**

Example...

I potentially violate timing!

```
logic [31:0] a,b,y,z,q,s,t,r;  
always_ff @(posedge clk)begin  
    if (b > q;)begin  
        a <= y+z;  
    end  
end
```

```
always_comb begin  
    q = s + t;  
end
```

```
always_comb begin  
    t = r>98?r+100:a+11;  
end
```

Path that needs to be calculated

Multiplication on the FPGA

- Multiplication can be done on the FPGA on 2's complement numbers
- Takes more time:
 - Depending on size of operands may/may not be doable in one clock cycle
- Where possible try to get away with bit shifts and adds.

Multiplications with shifts

- $\ll 1$ is multiply by 2
- $\gg 1$ is divide by 2
- Can do a lot with this if get creative

```
1
2  logic [7:0] x;
3  logic [7:0] y; //want this to be seven times X
4  assign y = (x<<2) + (x<<1) + x;
5
```

- Vivado can be pretty good at figuring these things out for you, but largely only for constants.

Generic Digital Multiplication

$$\begin{array}{rcccc}
 & & a_3 & a_2 & a_1 & a_0 & \leftarrow \text{Multiplicand} \\
 & & b_3 & b_2 & b_1 & b_0 & \leftarrow \text{Multiplier} \\
 \hline
 & & X & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
 a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & \\
 \hline
 & & \dots & a_1b_0 + a_0b_1 & a_0b_0 & \leftarrow \text{Product}
 \end{array}$$

} *Partial products*

In base 2 multiplication these are all very simple calculations done with XOR

*Some really cool factoring can be done to make the overall propagation delay of a multiplier relatively short, though there's a lot of logic in it**

*Lecture on Multiplier architectures: <https://inst.eecs.berkeley.edu/~eecs151/sp18/files/Lecture21.pdf>

DSP Blocks

- Add-then-multiply is a common operation chain in many things, particularly Digital Signal Processing
- FPGA has dedicated hardware multiplier modules called DSP48 blocks on it
 - 150 of them on our FPGA
 - Capable of single-cycle multiplies
- Can get inferred from using `*` in your Verilog that isn't a power of 2:
 - $x*y$, for example, will likely result in DSP getting used
 - May take a full clock cycle so would need to budget timing accordingly
- Can infer multiple for larger bit multiplies

DSP48 Slice (High Level)

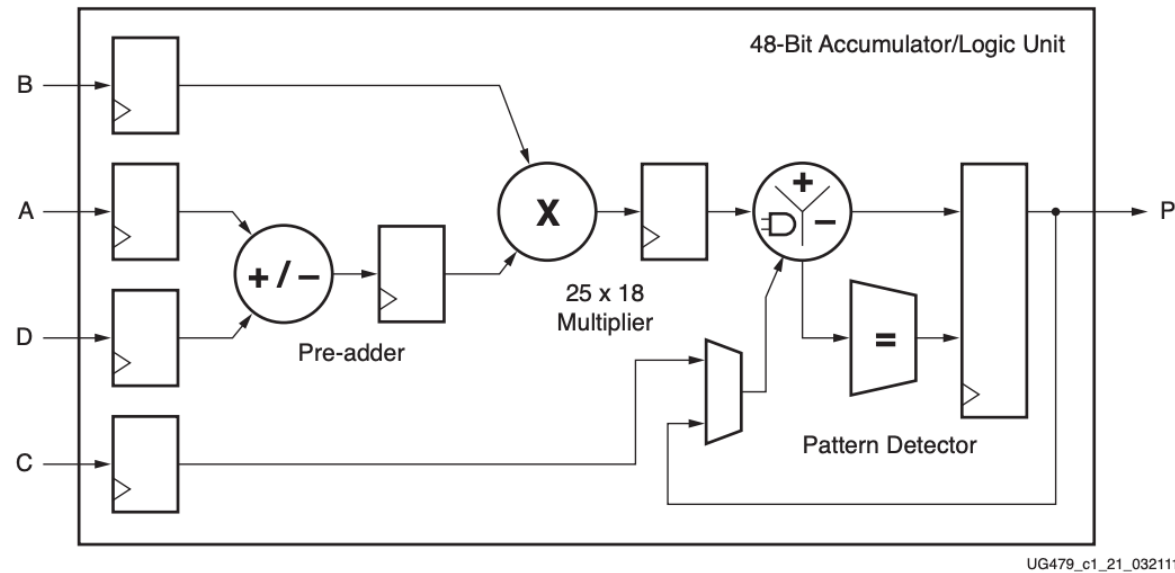


Figure 1-1: **Basic DSP48E1 Slice Functionality**

Much of the benefit/speed of this module comes from the hardwired internal routing, keeping it very fast. This device is not as generalized as a LUT/logic cell. It can only do a subset of math operations.

Located equally-spaced over the device like BRAMs

https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

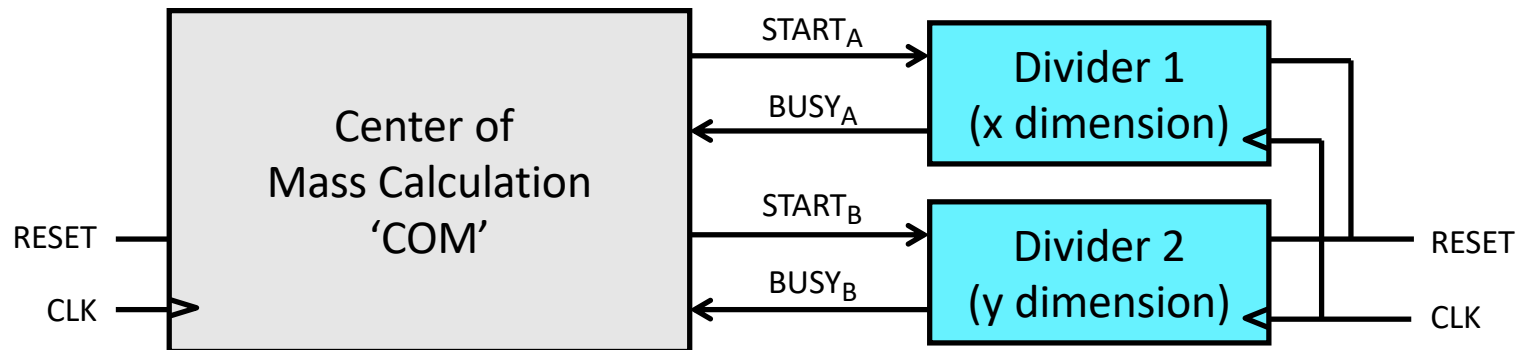
How much multiply can one do?

- At 100 MHz on these boards, I'd aim for *maybe maybe* one 32 bit multiply per clock cycle (it'll use several DSP blocks to achieve that)
- Anything more is pushing it
- If you run out of DSP blocks, it'll revert to using the generic logic...and this will become a harder problem to satisfy
- Smaller multiply-adds you can maybe get away with in one clock cycle.

Division

- The outlier in the + - * / set...
- Division is a significantly harder math operation to do compared to multiplication
- Where possible try to avoid
- Try to divide by powers of 2 (use right shift)!
- If you can't avoid we must do it. (week 05)

Lab 05 Center of Mass



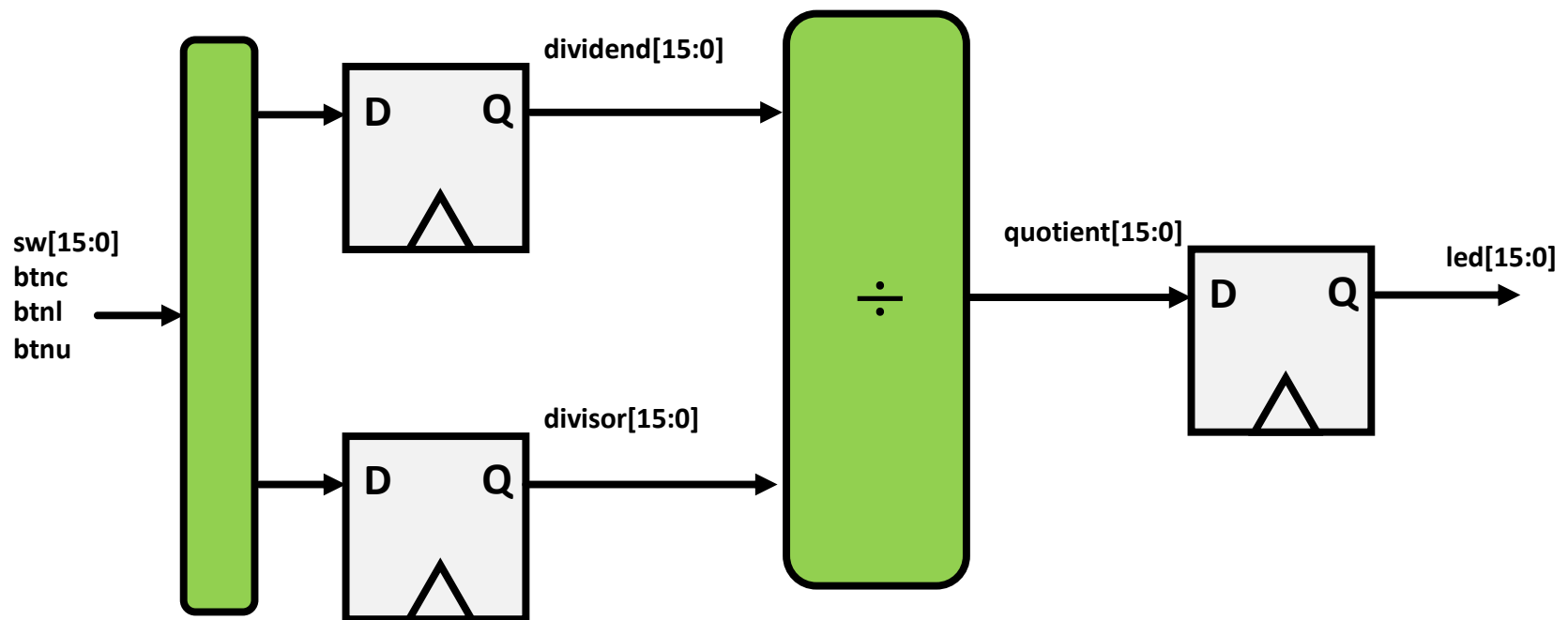
- C.O.M. will be in a “data collection state” during the active portion of a video frame
- When the frame’s active part is done, it needs to calculate the average x,y position of the “hot” pixels it has observed.
- To divide, the C.O.M. module hands the values it needs divided off to two separate dividers.
- C.O.M. waits on them monitoring their BUSY signals
- They can do division separately (in parallel)
- When done, they report back to the C.O.M with their result
- C.O.M. reports to outside world its calculation

One “Bad” Attempt at Division

- In previous lecture looked at *what* this actually builds
- We can ask Vivado to synthesize division logic for us, and it actually will do it.
- This code constrains the act of division to having to exist between two clock edges.:

```
module top_level(  
    input wire clk_100mhz, //clock @ 100 mhz  
    input wire [15:0] sw, //switches  
    input wire btnc, //btnc (used for reset)  
    input wire btnu, //btnc (used for reset)  
    input wire btnl, //btnc (used for reset)  
    output logic [15:0] led //just here for the funs  
);  
    logic old_btnl;  
    logic old_btnu;  
    logic old_btnc;  
    logic [15:0] quotient;  
    logic [15:0] dividend;  
    logic [15:0] divisor;  
    assign led = quotient;  
    always_ff @(posedge clk_100mhz)begin  
        old_btnl <= btnl;  
        old_btnu <= btnu;  
        old_btnc <= btnc;  
    end  
  
    always_ff @(posedge clk_100mhz)begin  
        if (btnc & ~old_btnc)begin  
            dividend <= sw; //divide //load dividend  
        end  
        if (btnc & ~old_btnc)begin  
            divisor <= sw; //divide //load divisor  
        end  
    end  
endmodule
```


Circuit Built:



Build the Bad Divider

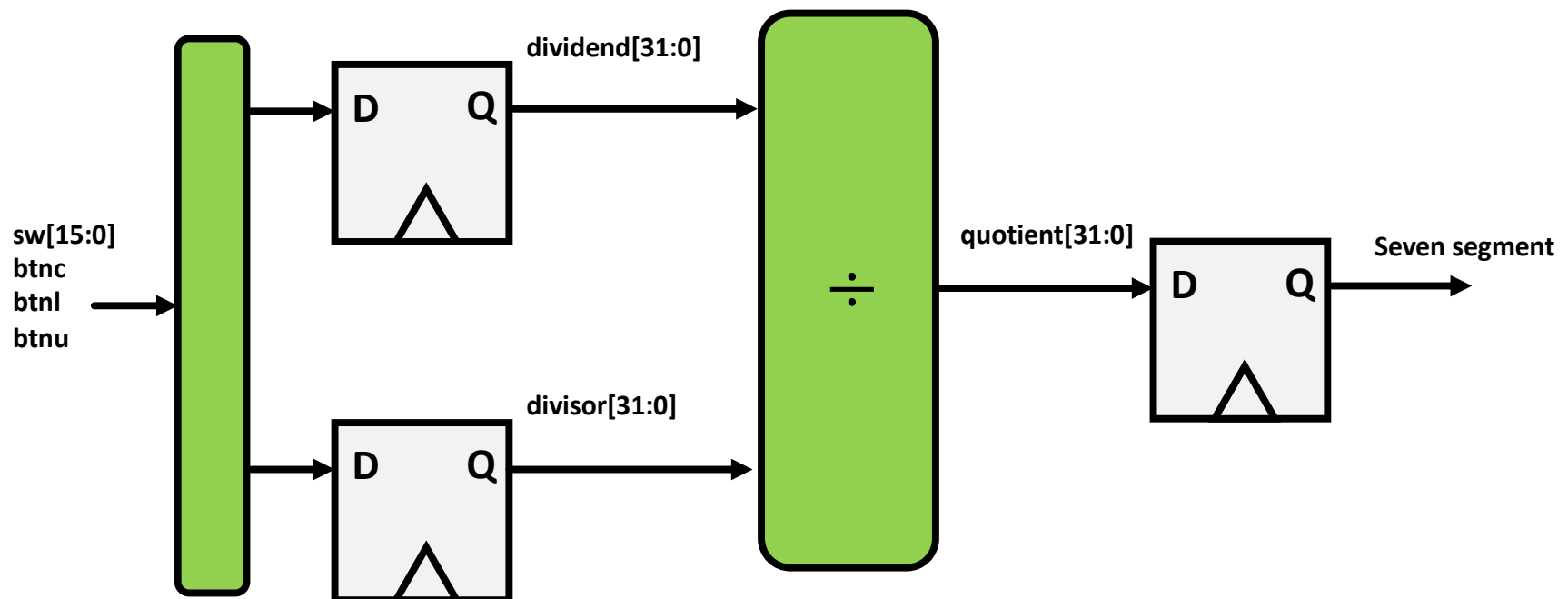
Violates timing!

Phase 22 Post Router Timing
INFO: [Route 35-20] Post Routing Timing Summary | WNS=-21.399 | TNS=-129.552 | WHS=0.090 |
THS=0.000 |

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	100	0	0	15850	0.63
SLICEL	89	0			
SLICEM	11	0			
LUT as Logic	274	0	0	63400	0.43
using 05 output only	0				
using 06 output only	274				
using 05 and 06	0				
LUT as Memory	0	0	0	19000	0.00
LUT as Distributed RAM	0	0			
LUT as Shift Register	0	0			
Slice Registers	55	0	0	126800	0.04
Register driven from within the Slice	16				
Register driven from outside the Slice	39				
LUT in front of the register is unused	26				
LUT in front of the register is used	13				
Unique Control Sets	4		0	15850	0.03

Now Do It Again With 32 bits:

```
if (pmod_pin & ~old_pmod_pin) begin  
    quotient <= dividend/divisor;  
end
```



**See lecture code for full implementation and build. (divider0)*

Build the Stupider Divider

Phase 20 Post Router Timing

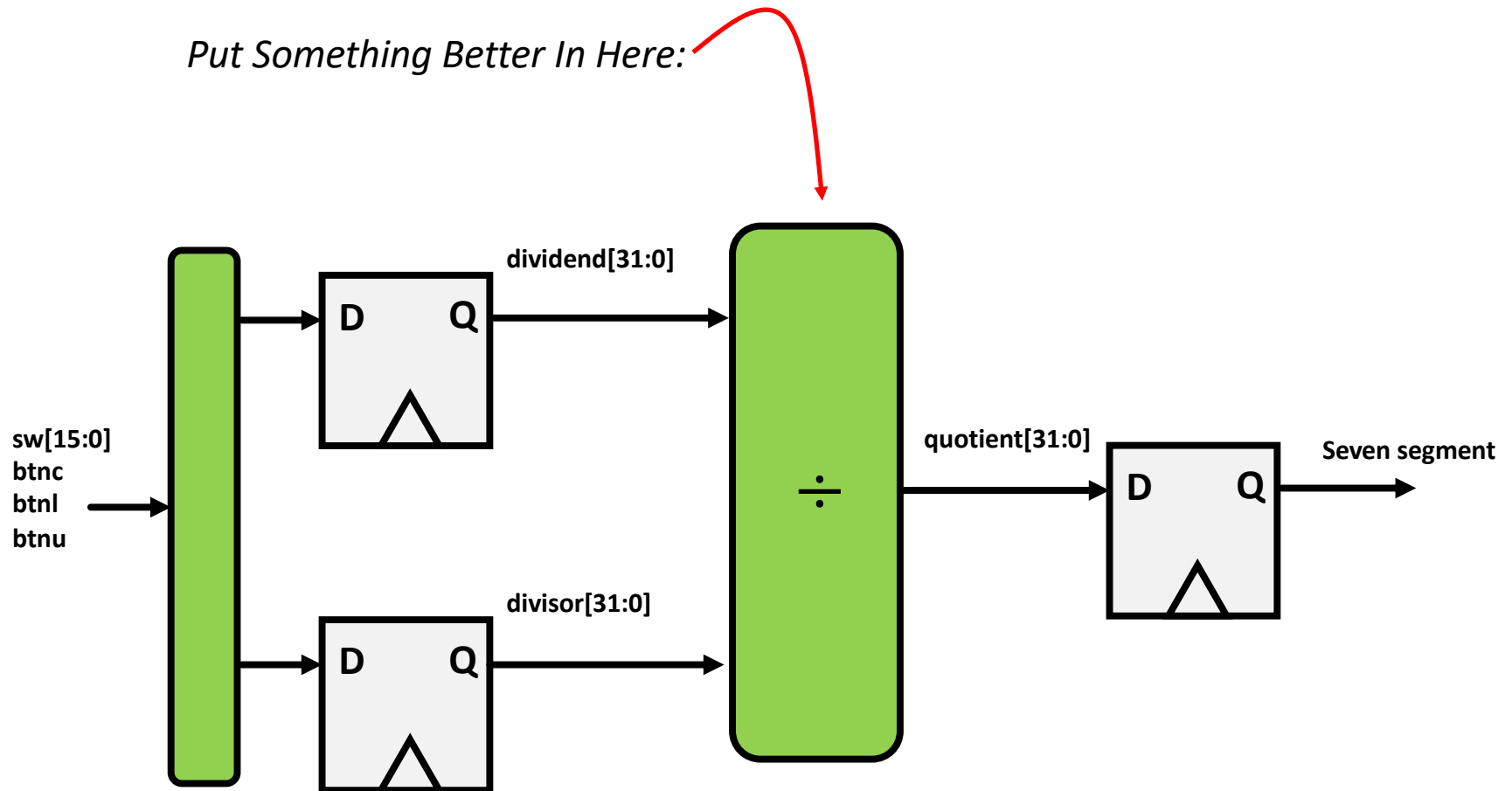
INFO: [Route 35-20] Post Routing Timing Summary | WNS=-72.004| TNS=-1004.354| WHS=0.227 | THS=0.000 |

Phase 20 Post Router Timing | Checksum: 1d10fc4d8

2. Slice Logic Distribution

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	301	0	0	8150	3.69
SLICEL	225	0			
SLICEM	76	0			
LUT as Logic	944	0	0	32600	2.90
using 05 output only	0				
using 06 output only	922				
using 05 and 06	22				
LUT as Memory	0	0	0	9600	0.00
LUT as Distributed RAM	0	0			
LUT as Shift Register	0	0			
Slice Registers	131	0	0	65200	0.20
Register driven from within the Slice	67				
Register driven from outside the Slice	64				
LUT in front of the register is unused	28				
LUT in front of the register is used	36				
Unique Control Sets	7		0	8150	0.09

A Better Divider?



**See lecture 5 code for full implementation and build. (divider0) that is an FSM.*

So conclusions

- $+$, $-$, $*$ can be done in a clock cycle with exceptions
- Watch out for flow-control logic...that can start to stack up
- $/$ will never happen in one clock cycle. Accept that.
- Similar other things like square root, cosine, etc...those need clock cycles...or if you absolutely need those in one/two cycles, you do a lookup table of pre-computed values (takes huge amounts of memory)

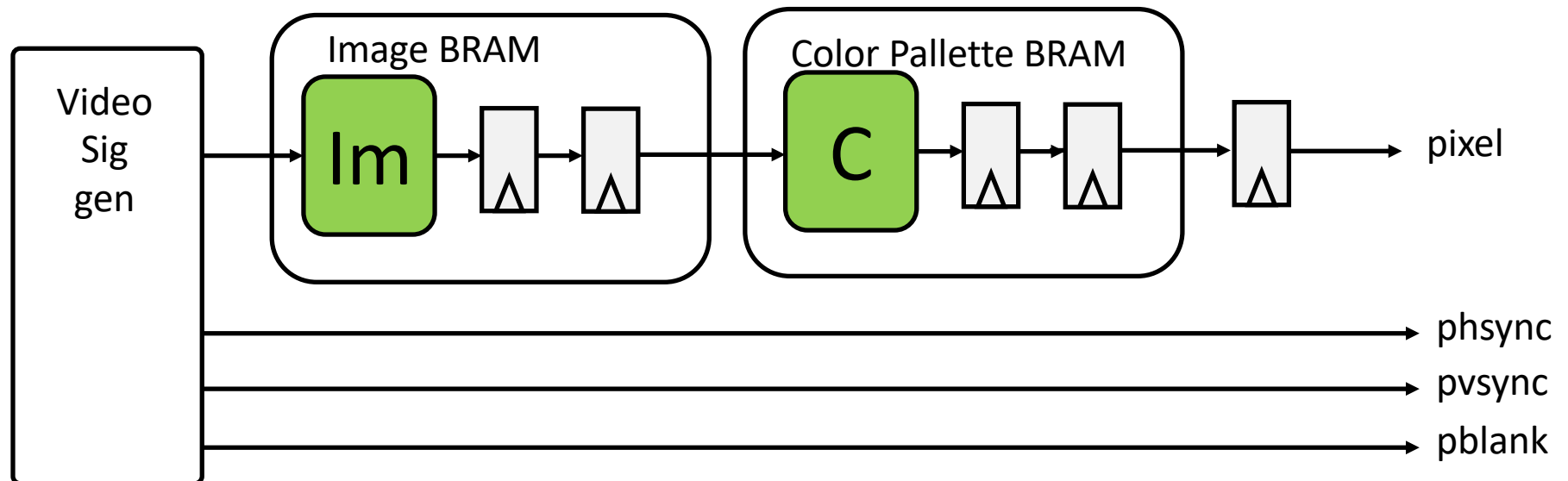
Now...Back on pipelining...Preview...

- If we have time... A white blip
- In lab 05, early on you may see an artifact on popcat



(Two registers coming from delay in memory access/read)

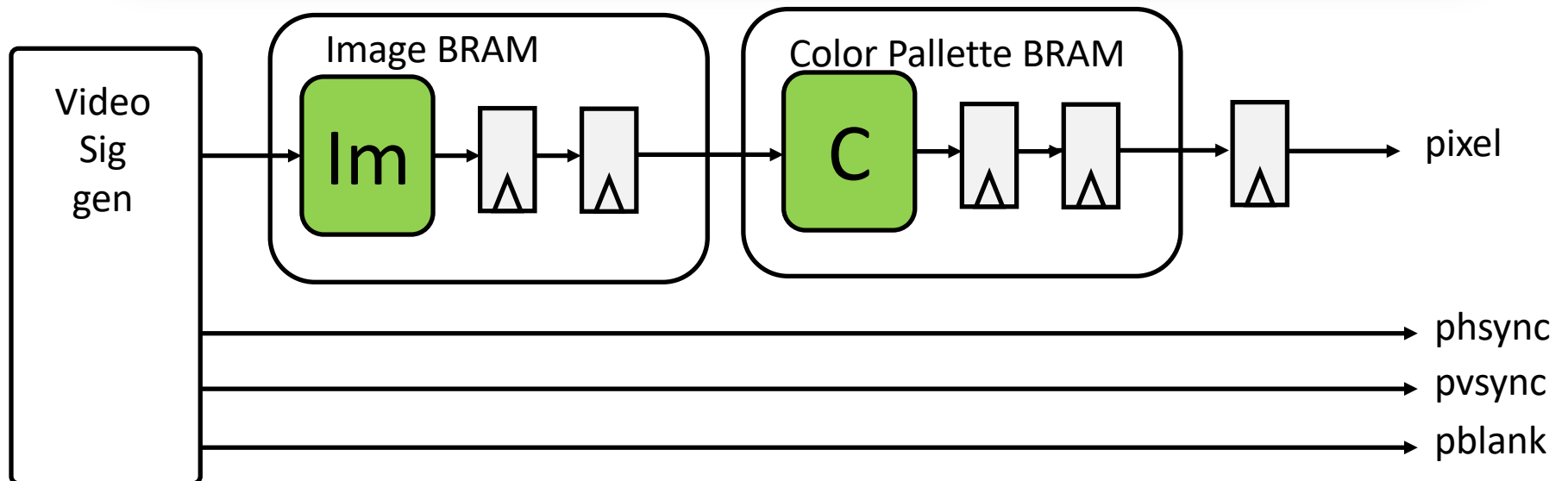
- Monitor drawing based on v_sync, h_sync, blank,
- But what image rom is giving it is 5 clock cycles behind
- At start of PopCat nothing in the “pipeline” yet




```

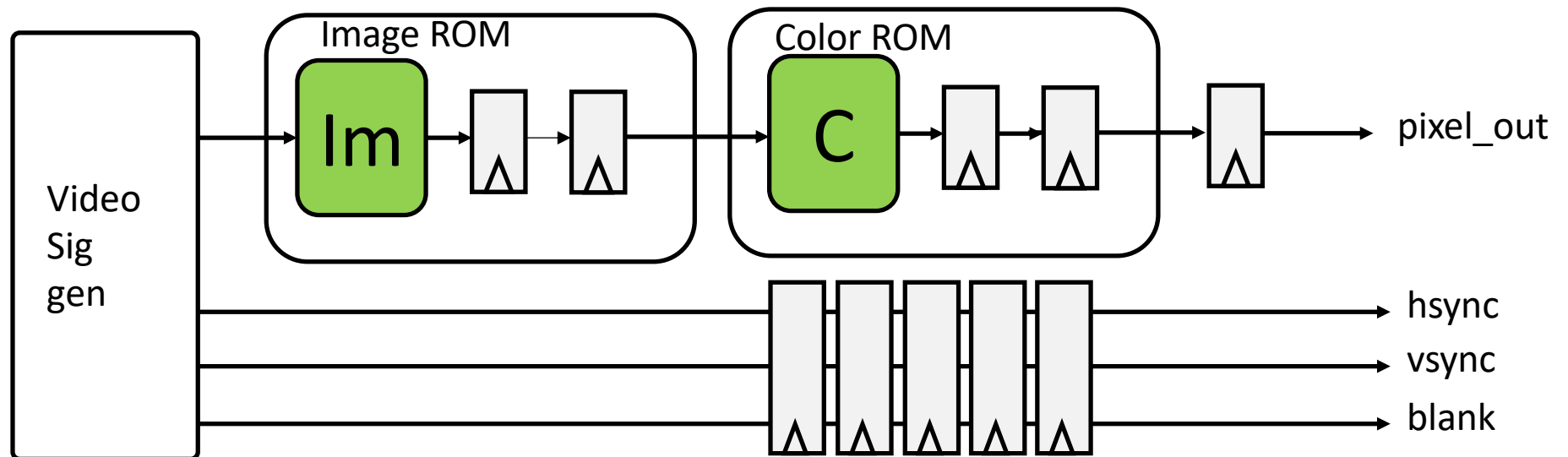
1
2 // calculate rom address and read the location
3 assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;
4 image_rom  rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits));
5
6 red_coe rcm (.clka(pixel_clk_in), .addra(image_bits), .douta(red_mapped));
7
8 always_ff @(posedge pixel_clk) begin
9     if ((hcount_in >= x && hcount_in < (x_in+WIDTH)) &&
10         (vcount_in >= y_in && vcount_in < (y_in+HEIGHT))) begin
11         pixel_out <= {red_mapped[7:4], red_mapped[7:4], red_mapped[7:4]}; // greyscale
12     end else begin
13         pixel_out <= 0;
14     end
15 end
16

```



How to Fix?

- Delay the other signals so everybody is the same



Turn the whole thing into a 5-stage pipeline!

Pipelining

- Pipeline in Verilog!
- Make sure other things are protected too!

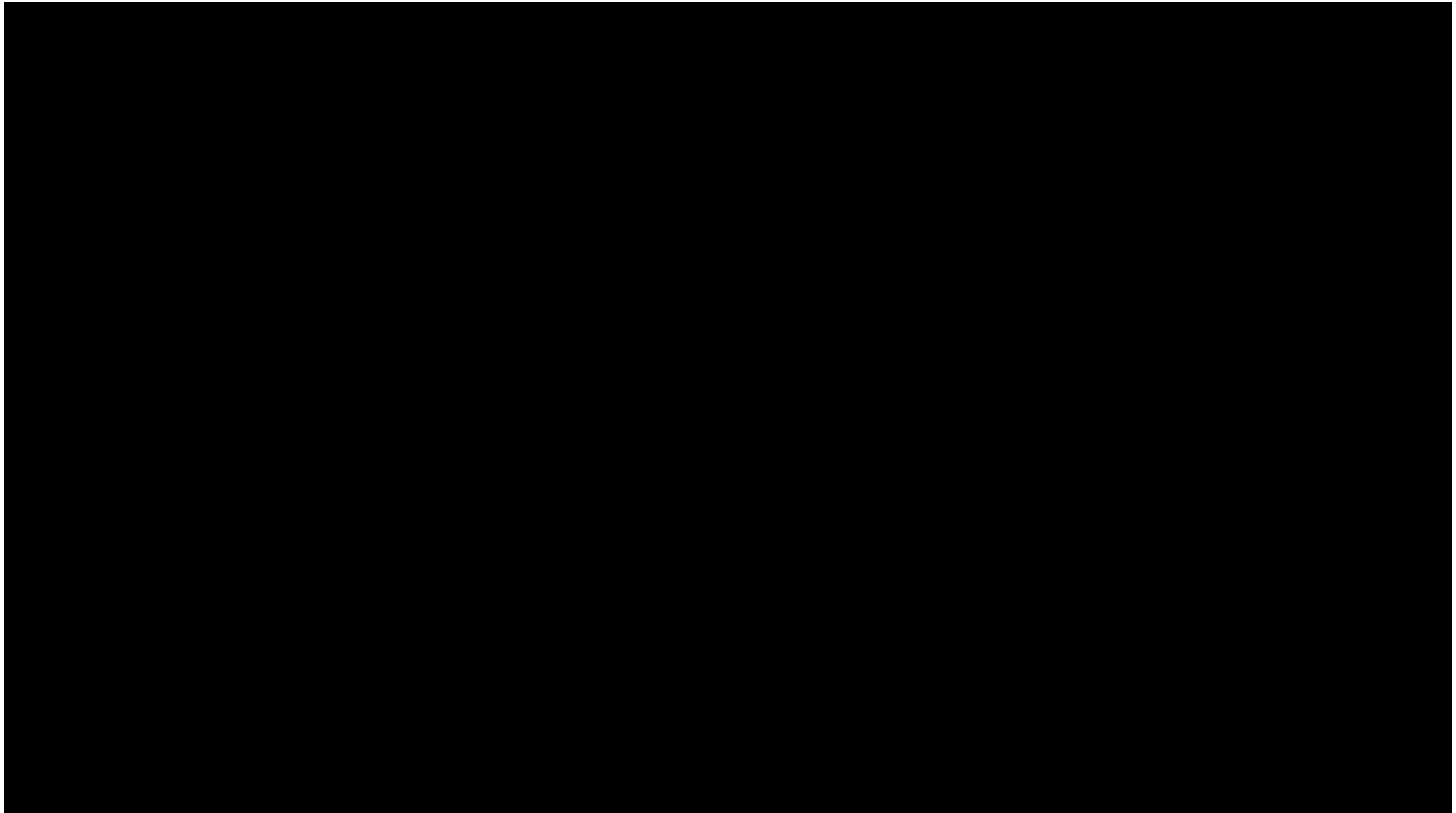
```
logic hs_pip[4:0];
logic vs_pip[4:0];
logic b_pip[4:0];

always_ff@(posedge clk_in)begin
    hs_pip[0] <= hsync_in;
    vs_pip[0] <= vsync_in;
    b_pip[0] <= blank_in;
    for (int i=1; i<5; i = i+1)begin
        hs_pip[i] <= hs_pip[i-1];
        vs_pip[i] <= vs_pip[i-1];
        b_pip[i] <= b_pip[i-1];
    end
end
assign hsync_out = hs_pip[4];
assign vsync_out = vs_pip[4];
assign blank_out = b_pip[4];
```

Final Project Ideas

- Things with video and/or related topics are very “relevant” to FPGAs
- You have to move and process very large amounts of data with demanding timing.
- This is something software often cannot do on its own.

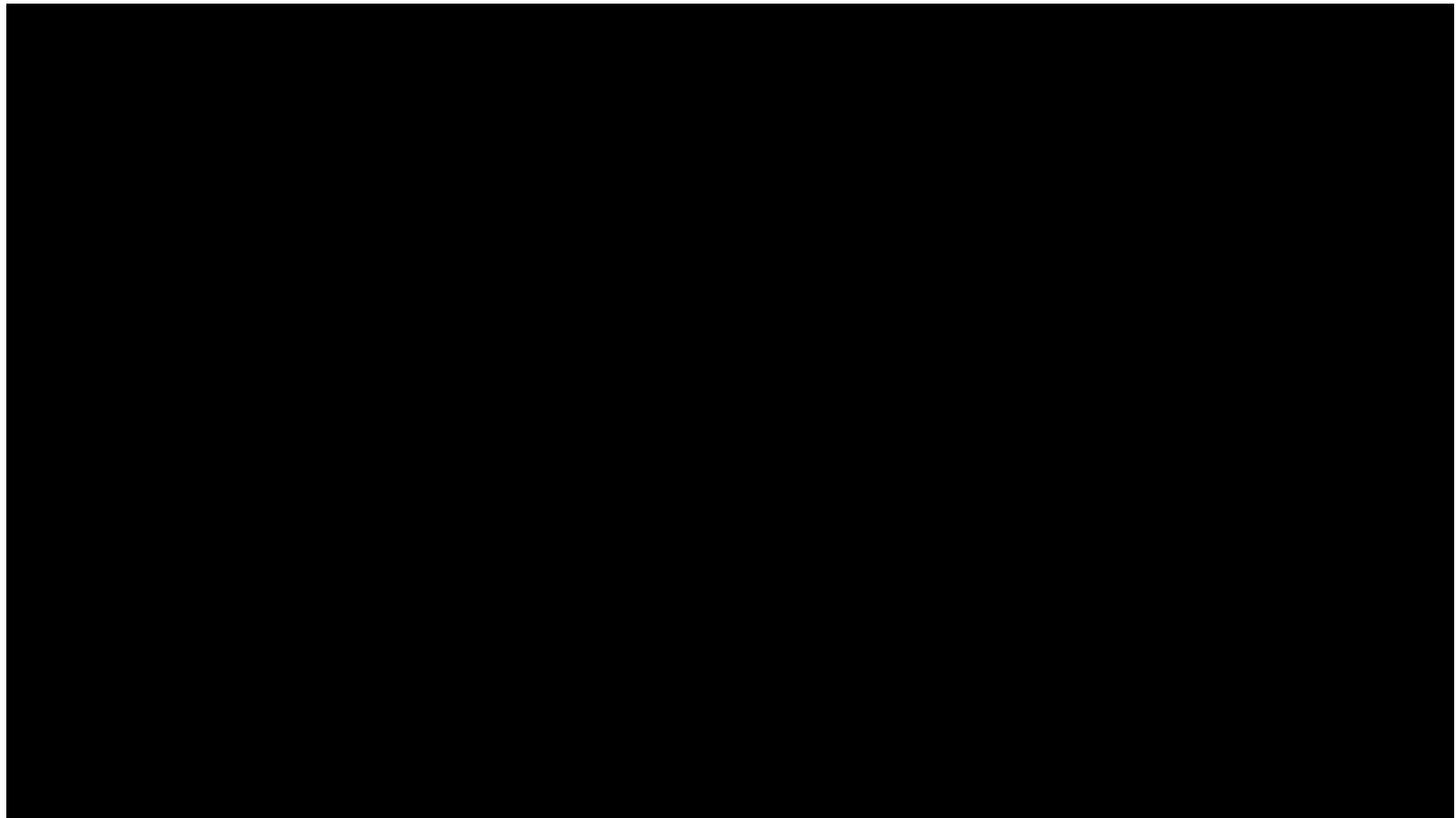
Live Pong



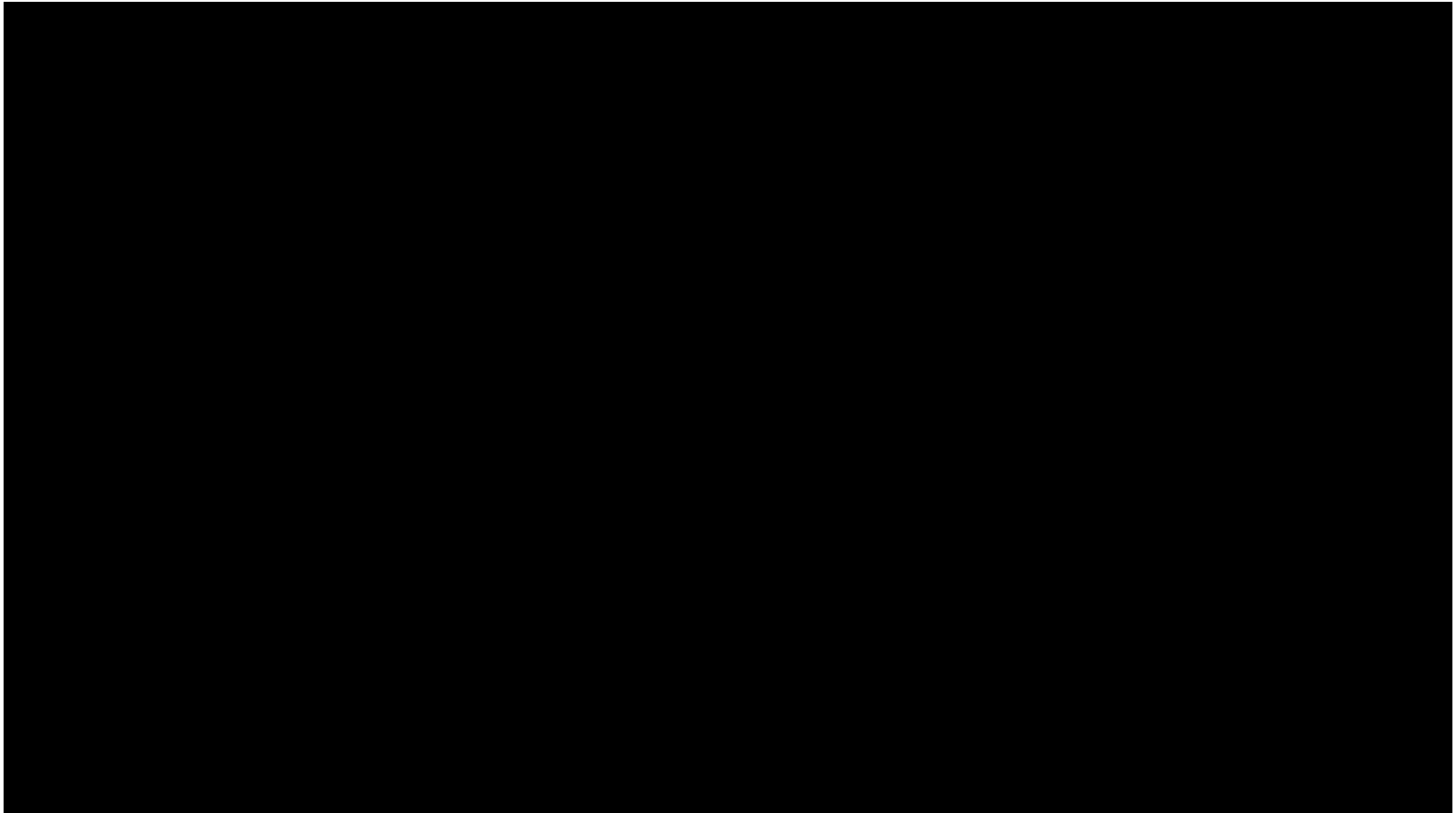
Glow Trails



DigiEyes



PacMan Extreme



Final Project Info released by tomorrow

- Start Teaming!
- Teams of 2 or 3!