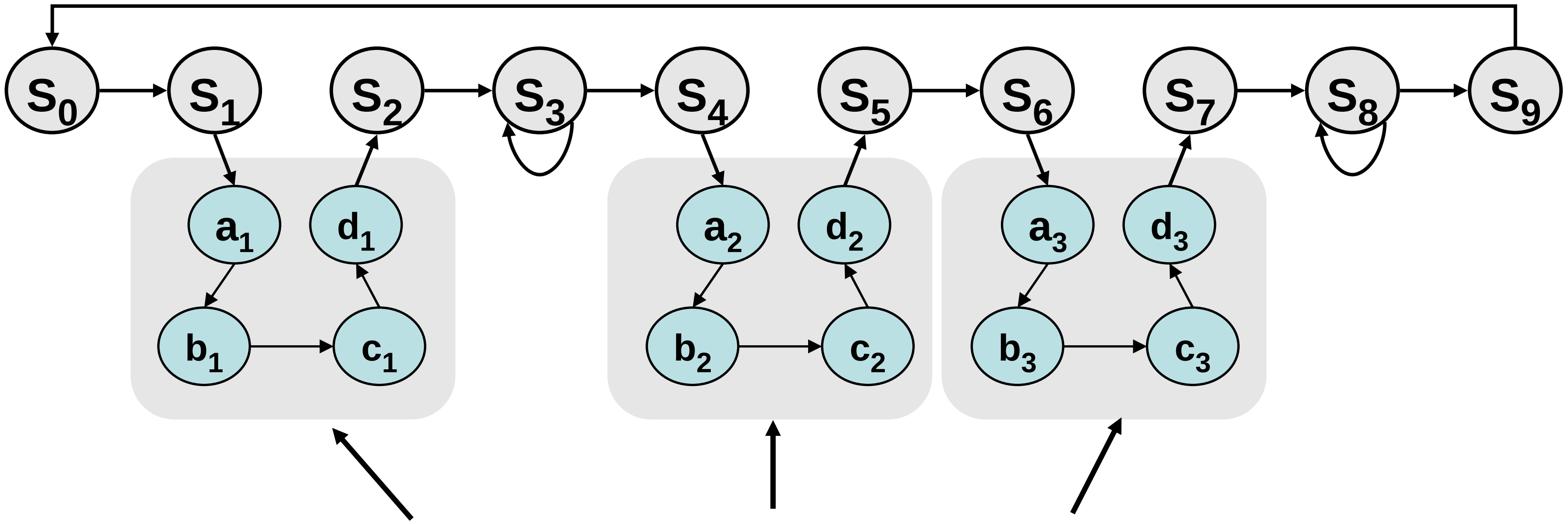
Toward FSM Modularity

- Consider the following abstract FSM:

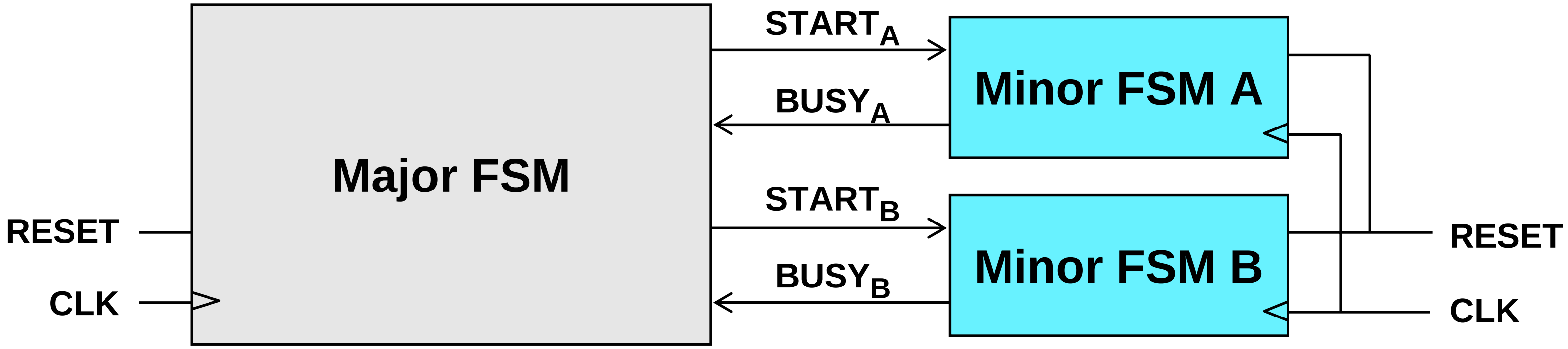


- Suppose that each set of states $a_x \ldots d_x$ is a "sub-FSM" that produces exactly the same outputs.
- Can we simplify the FSM by removing equivalent states?

  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?

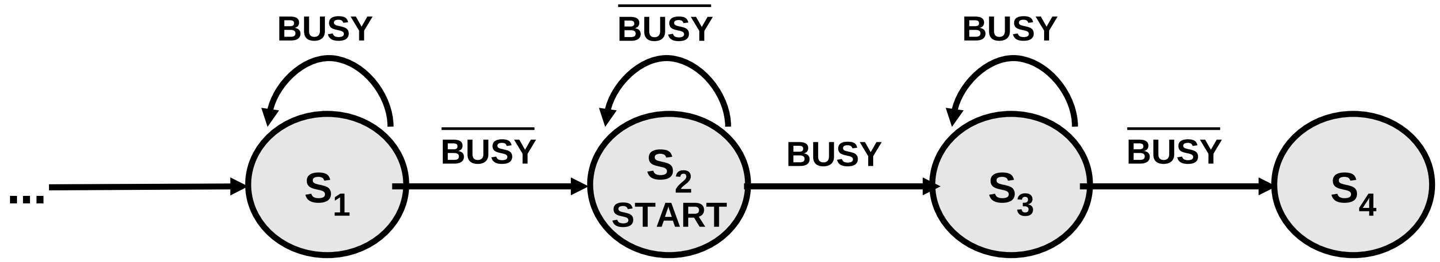
Acknowledgements: Rex Min

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

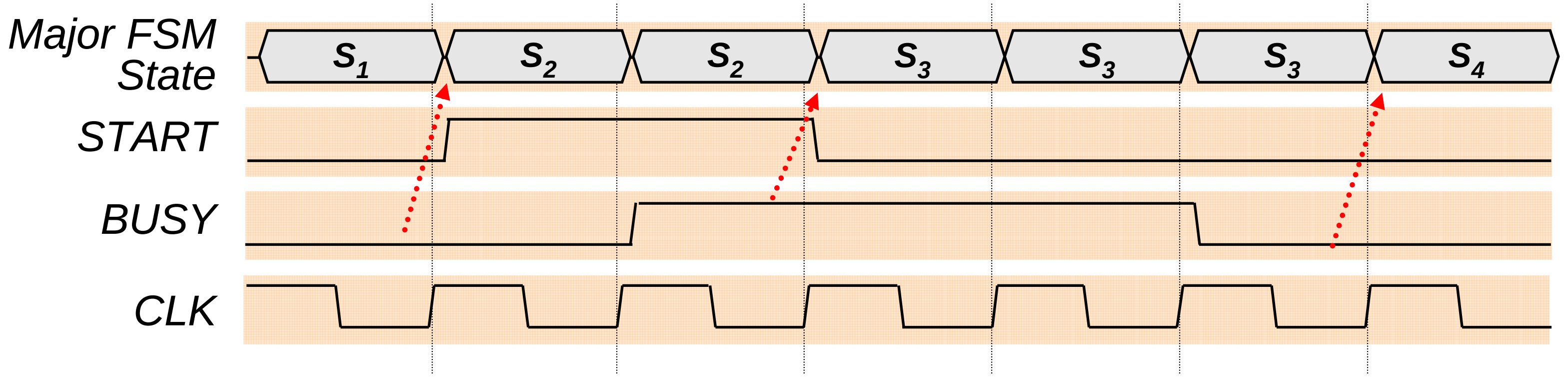
Inside the Major FSM



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

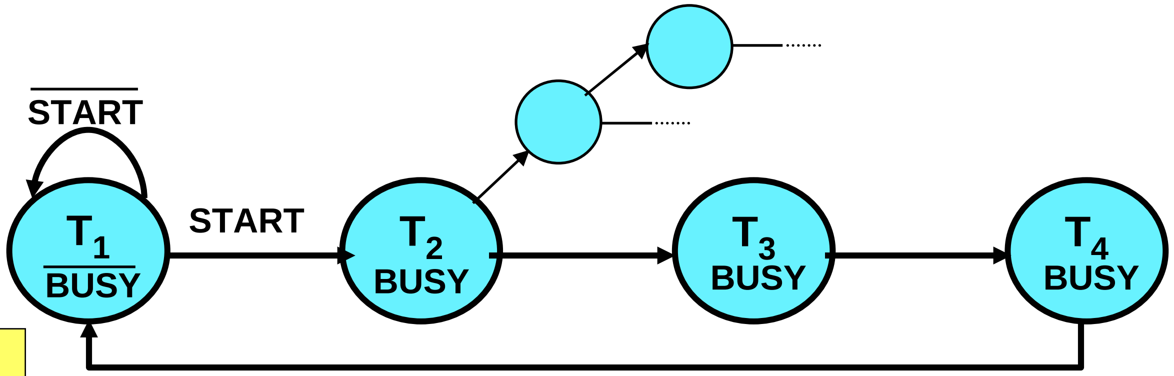


Variations:

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

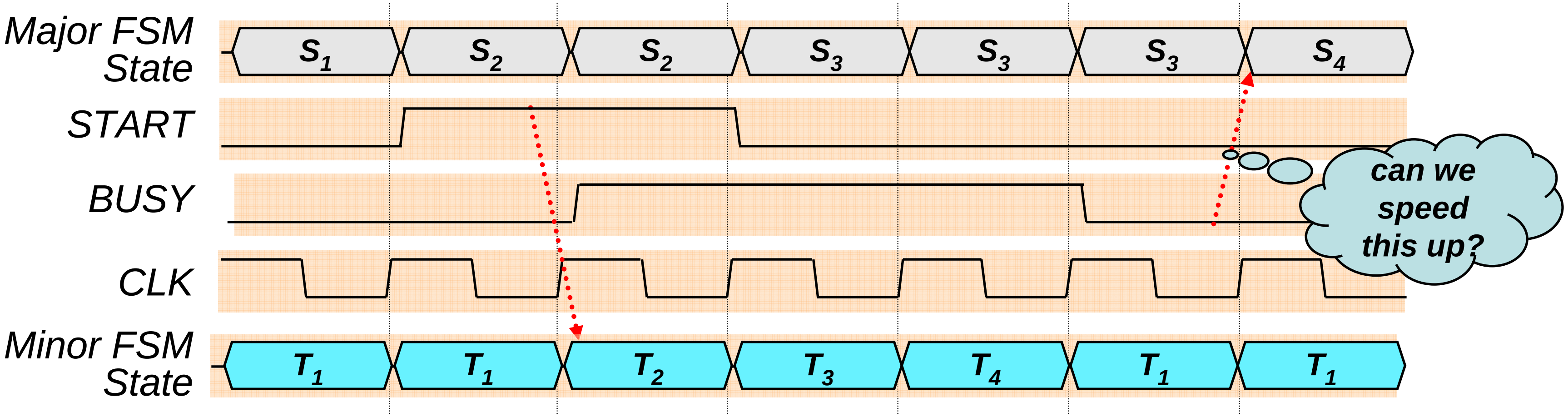
Inside the Minor FSM

1. Wait for a trigger from the major FSM



3. Signal to the major FSM that work is done

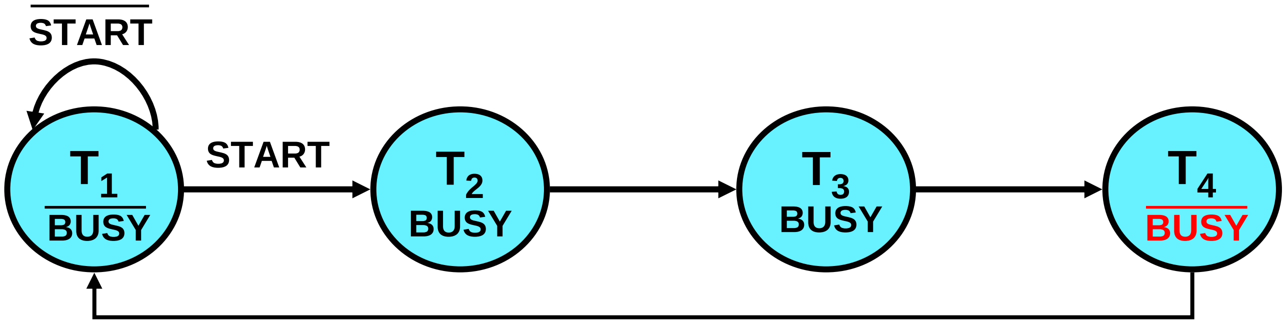
2. Do some useful work



can we speed this up?

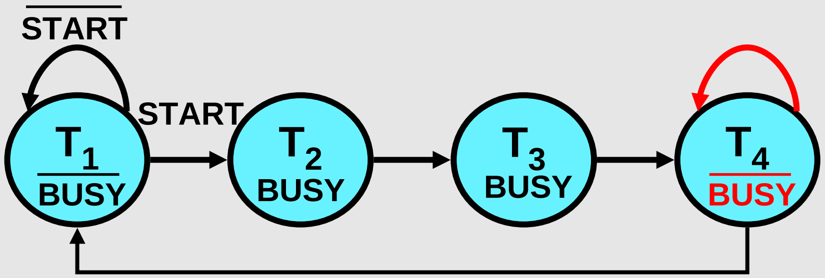
Optimizing the Minor FSM

Good idea: de-assert BUSY one cycle early



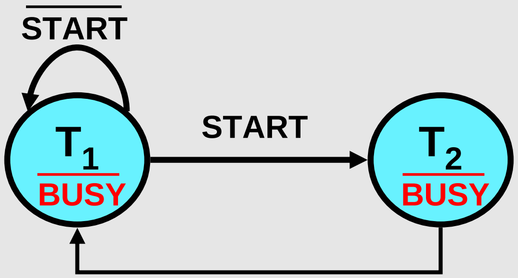
Bad idea #1:

T_4 may not immediately return to T_1

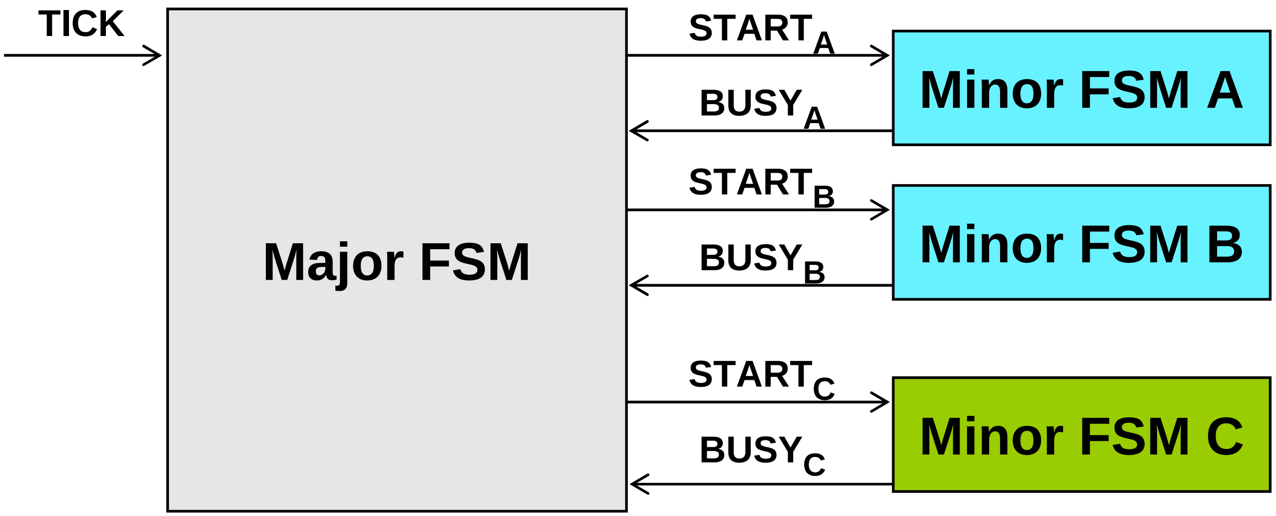


Bad idea #2:

BUSY never asserts!



A Four-FSM Example



TICK

Major FSM

$START_A$

$BUSY_A$

$START_B$

$BUSY_B$

$START_C$

$BUSY_C$

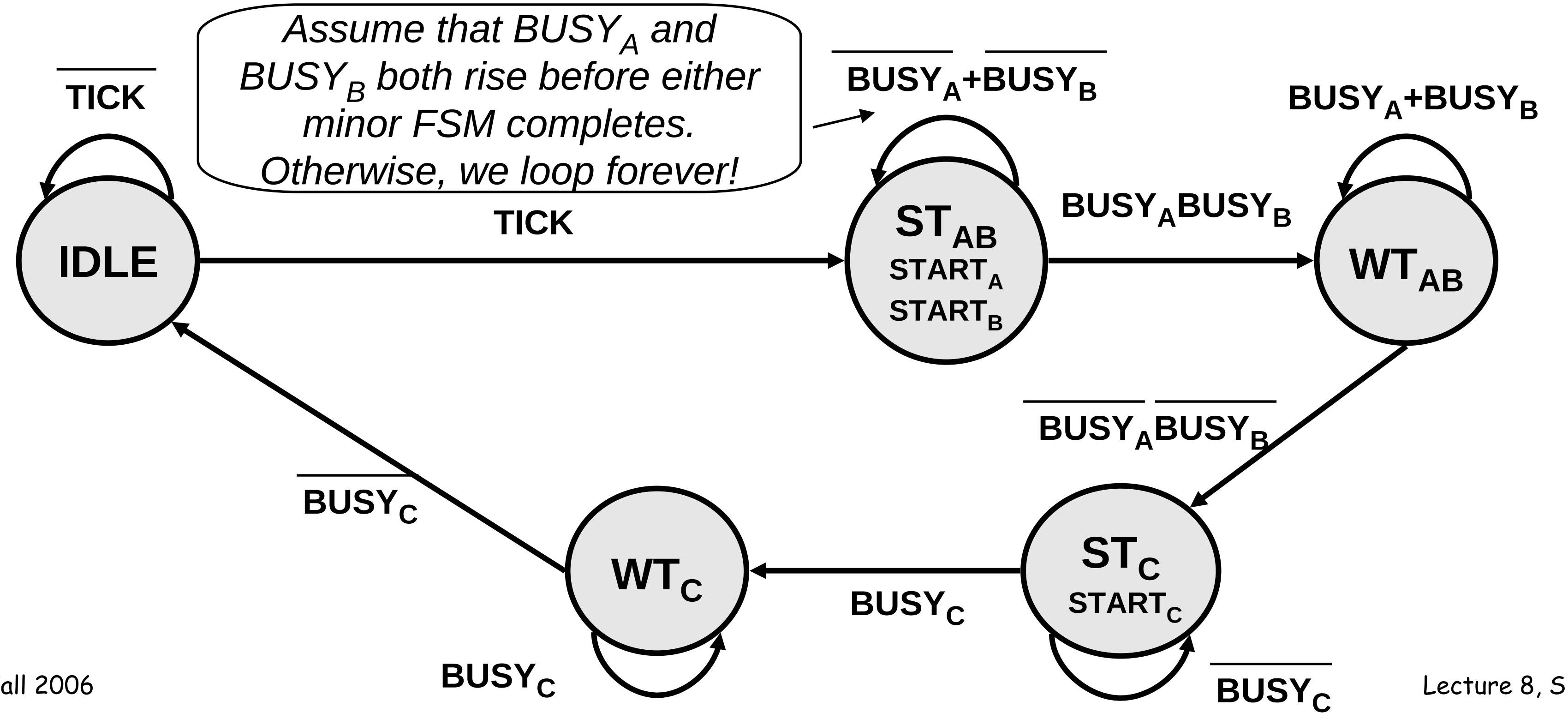
Minor FSM A

Minor FSM B

Minor FSM C

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



Assume that $BUSY_A$ and $BUSY_B$ both rise before either minor FSM completes. Otherwise, we loop forever!

$\overline{TICK}$

IDLE

TICK

$\overline{BUSY_A}+\overline{BUSY_B}$

ST_{AB}

$START_A$

$START_B$

$BUSY_A BUSY_B$

$BUSY_A+BUSY_B$

WT_{AB}

$\overline{BUSY_A}\,\overline{BUSY_B}$

ST_C

$START_C$

$\overline{BUSY_C}$

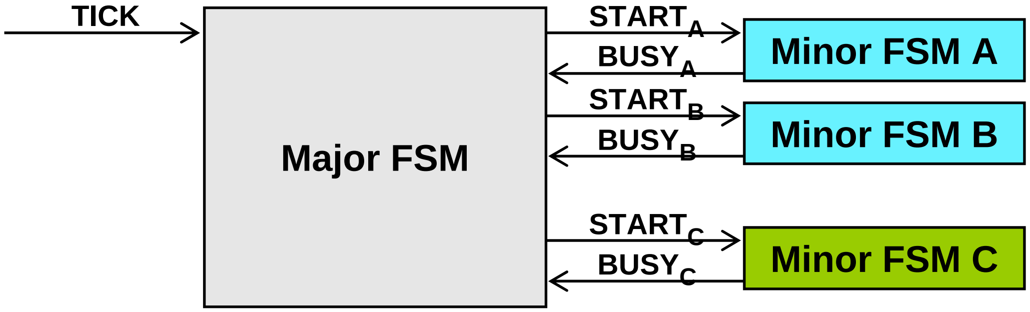
$BUSY_C$

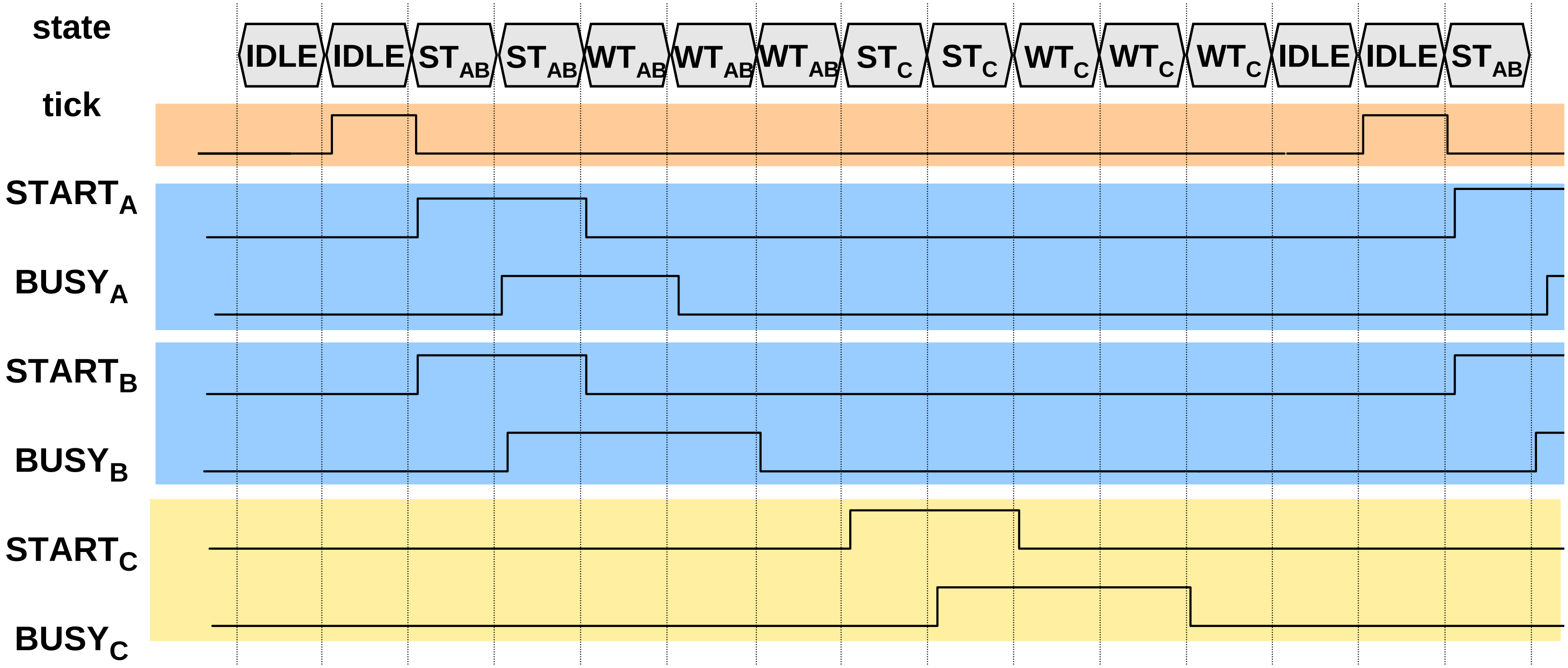
WT_C

$BUSY_C$

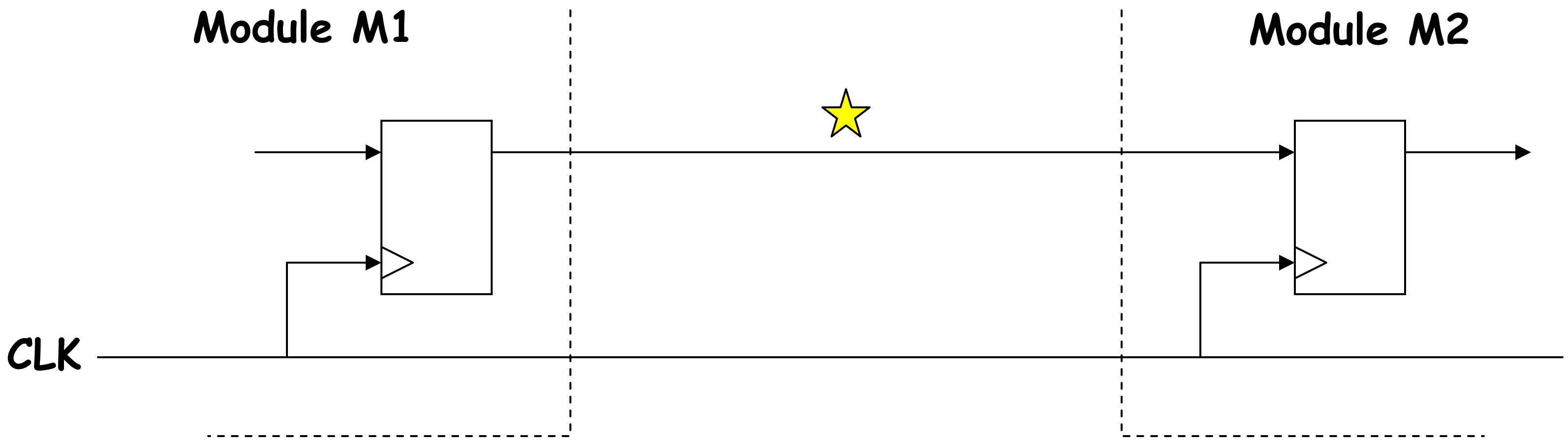
$\overline{BUSY_C}$

Four-FSM Sample Waveform

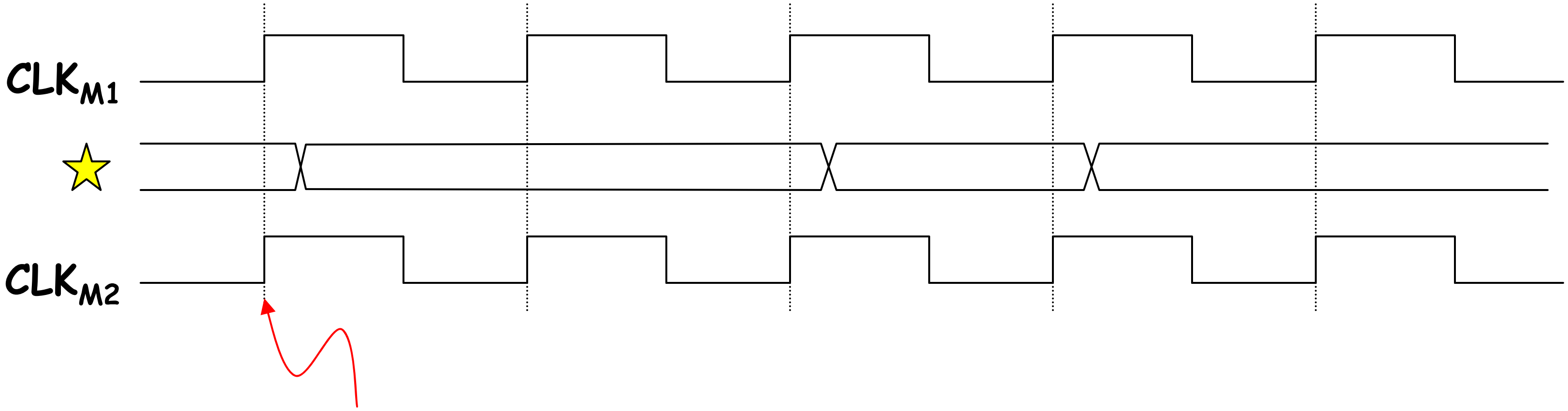




Clocking and Synchronous Communication

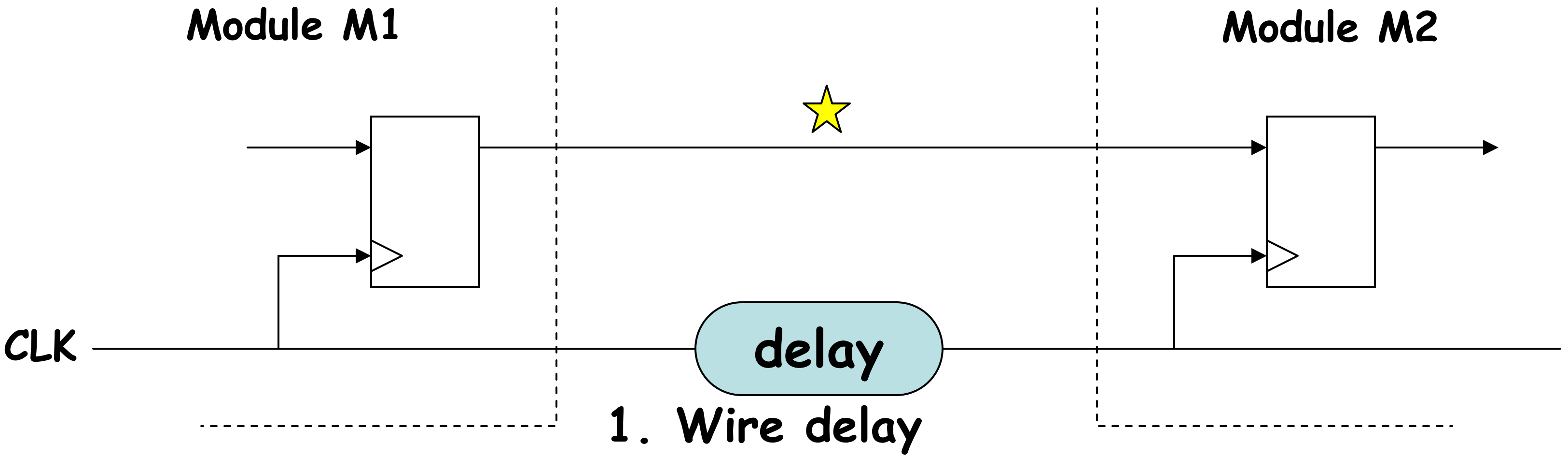


Ideal world:



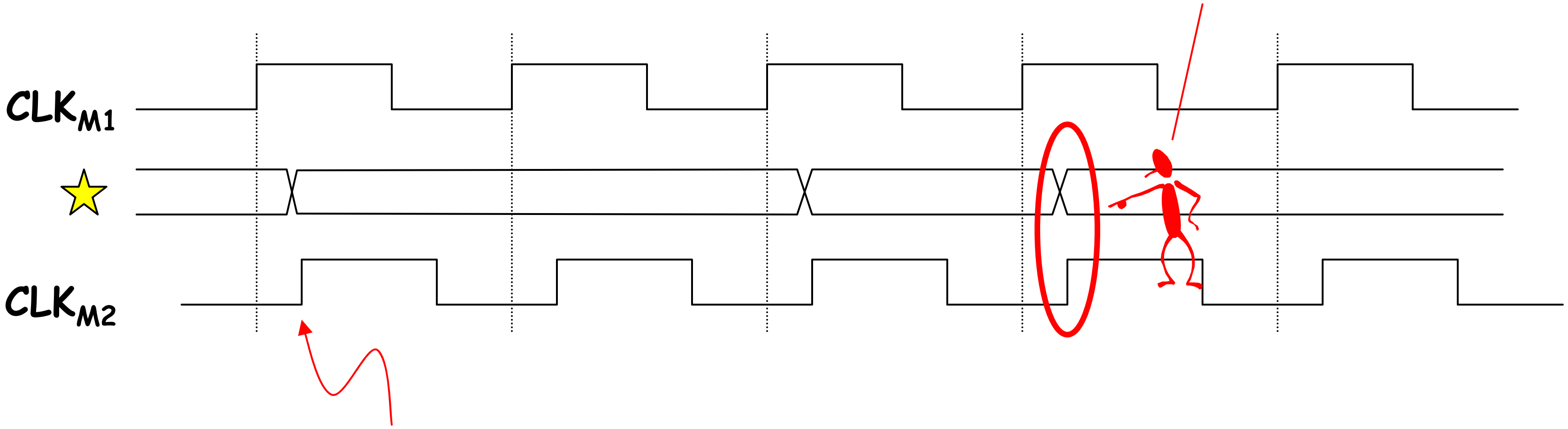
M1 and M2 clock edges aligned in time

Clock Skew



1. Wire delay
2. Different clocks!

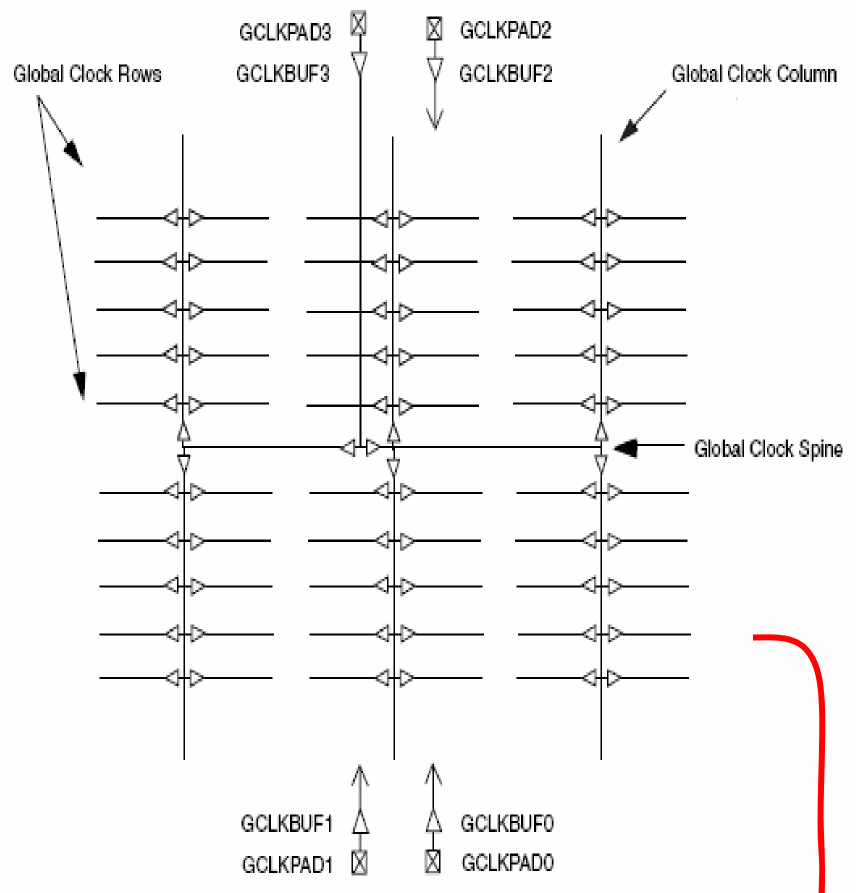
Real world has clock skew:



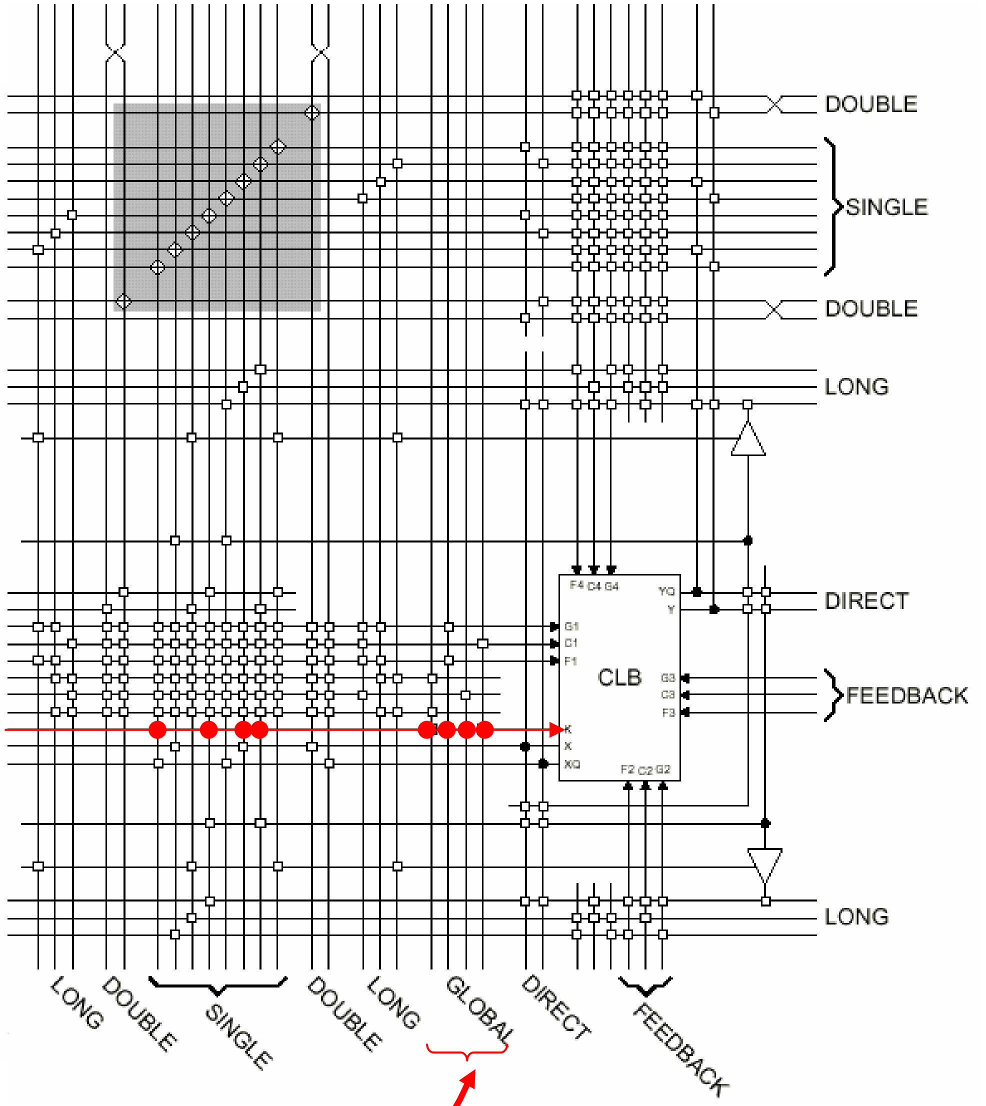
Oops! Skew has caused a hold time problem!

M2 clock delayed with respect to M1 clock

Low-skew Clocking in FPGAs



Global Clock Distribution Network



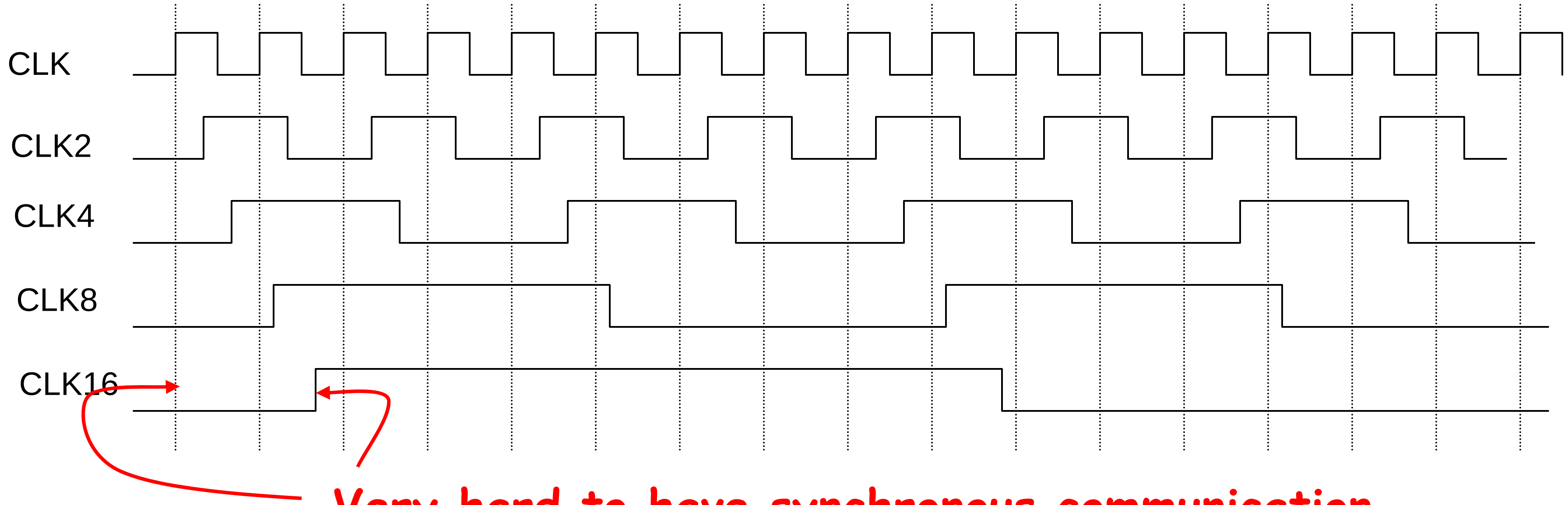
Figures from Xilinx App Notes

Goal: use as few clock domains as possible

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

```
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! don't do it this way



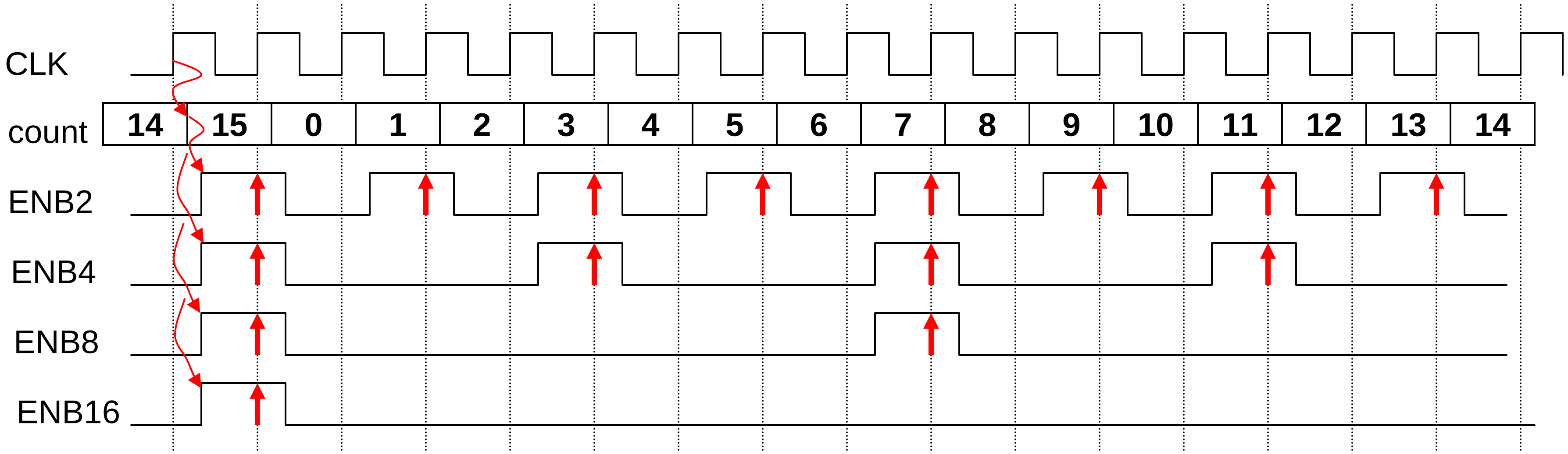
Very hard to have synchronous communication between clk and clk16 domains

Solution: 1 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

```
reg [3:0] count;
always @ (posedge clk) count <= count + 1;   // counts 0..15
wire enb2 = (clock[0] == 1’b1);
wire enb4 = (clock[1:0] == 2’b11);
wire enb8 = (clock[2:0] == 3’b111);
wire enb16 = (clock[3:0] == 4’b1111);
```

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



↑ = clock edge selected by enable signal

Using External Clocks

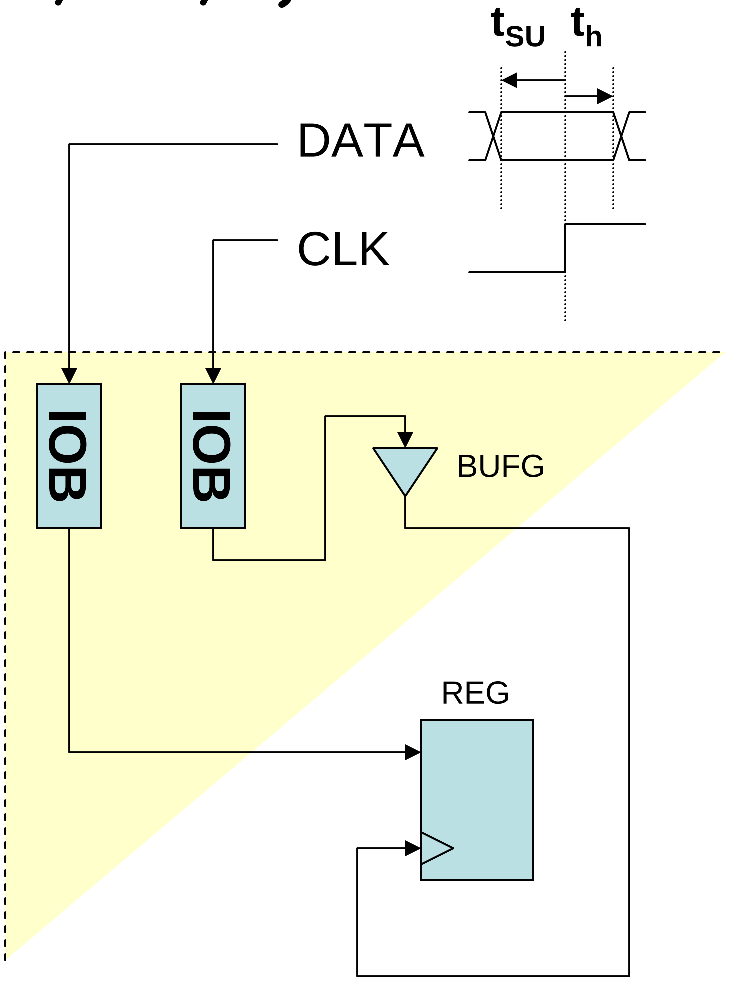
Sometimes you need to communicate synchronously with circuitry outside of the FPGA (memories, I/O, …)

Problem: different delays along internal paths for DATA and CLK change timing relationship

Solutions:

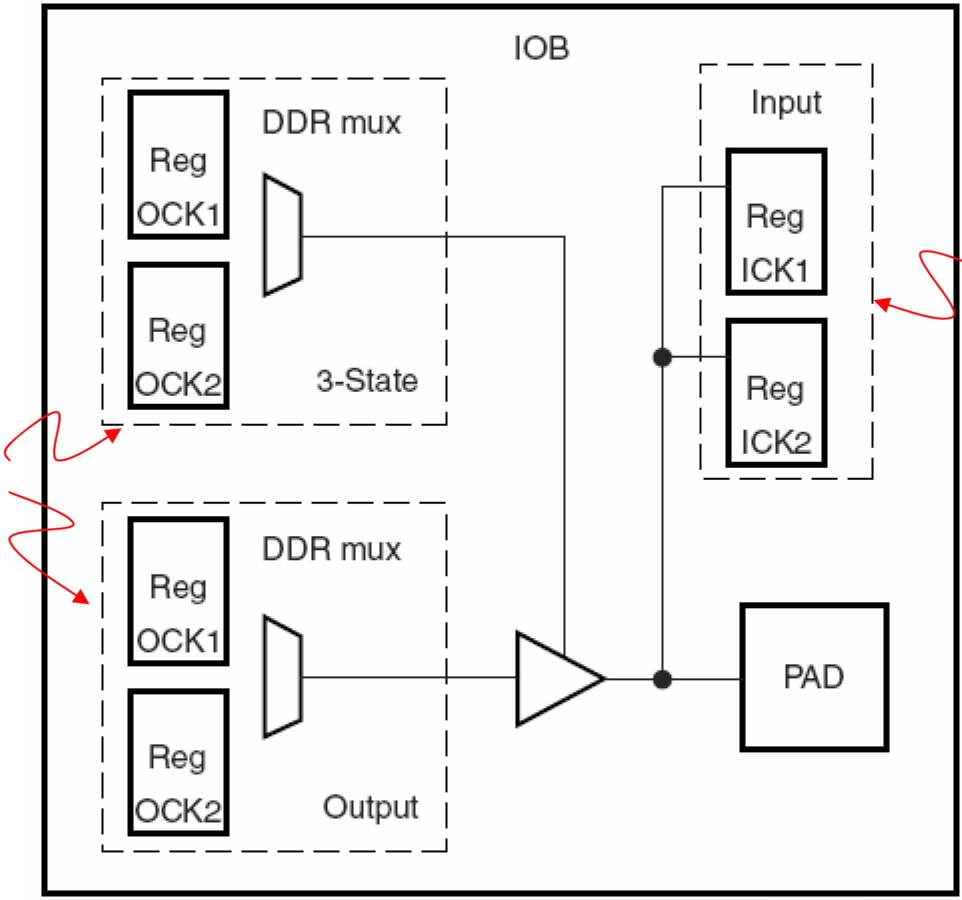
1) Bound internal delay from pin to internal reg; add that delay to setup time (t_{SU}) specification

2) Make internal clock edge aligned with external clock edge (but what about delay of pad and clock driver)



1) Bound Internal Data Delay

Solution: use registers built into the IOB pin interface:

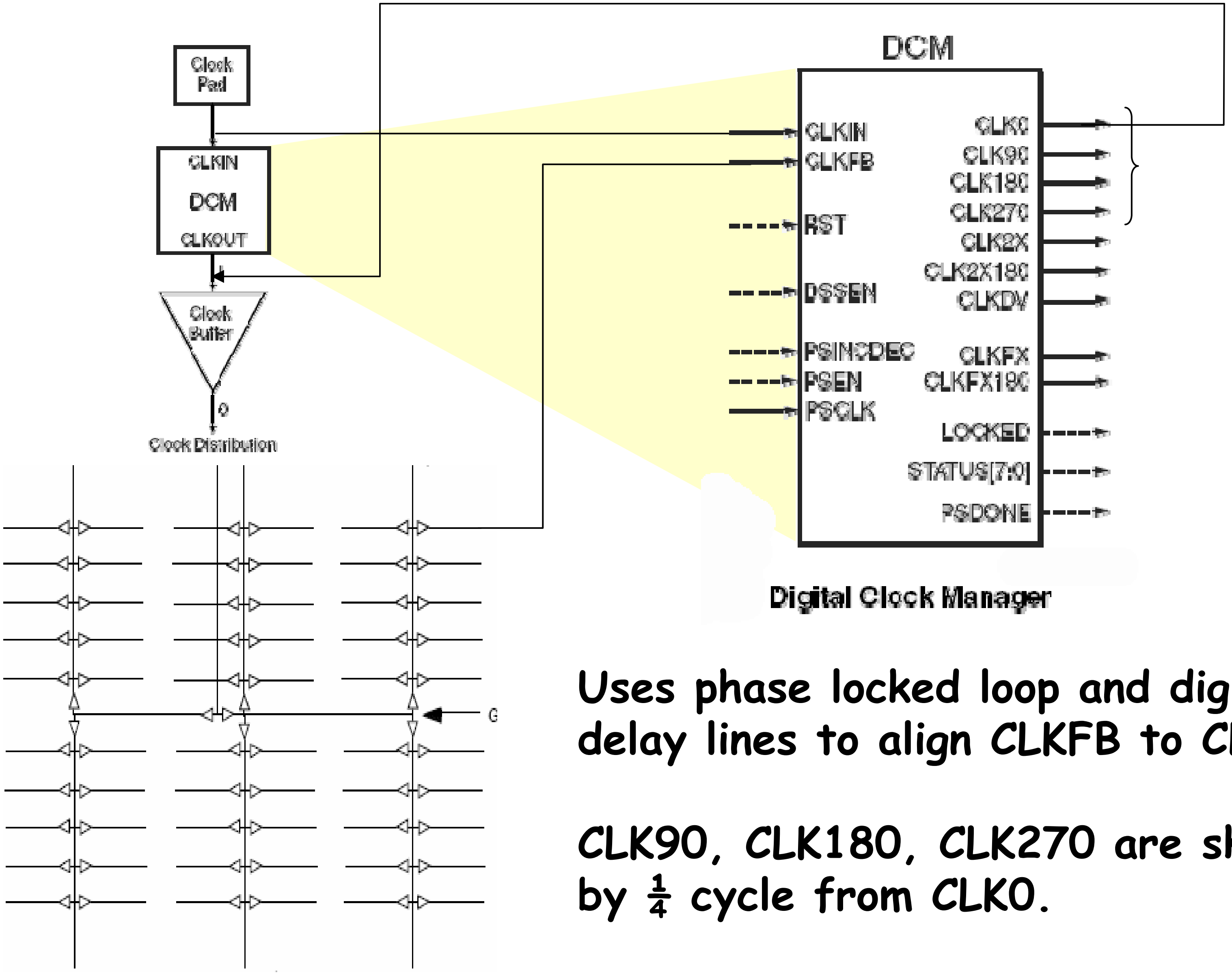


Low-delay tristate outputs

Low-delay inputs

Virtex-II IOB Block

2) Align external and internal clocks

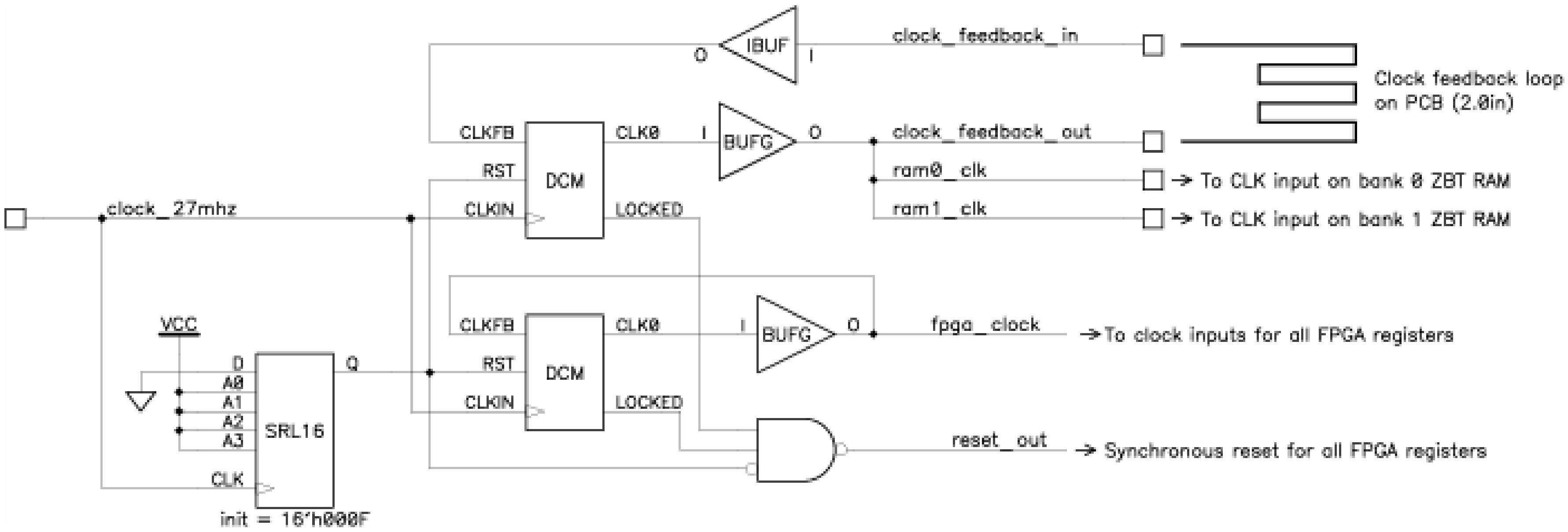


Digital Clock Manager

Uses phase locked loop and digital delay lines to align CLKFB to CLKIN.

CLK90, CLK180, CLK270 are shifted by ¼ cycle from CLK0.

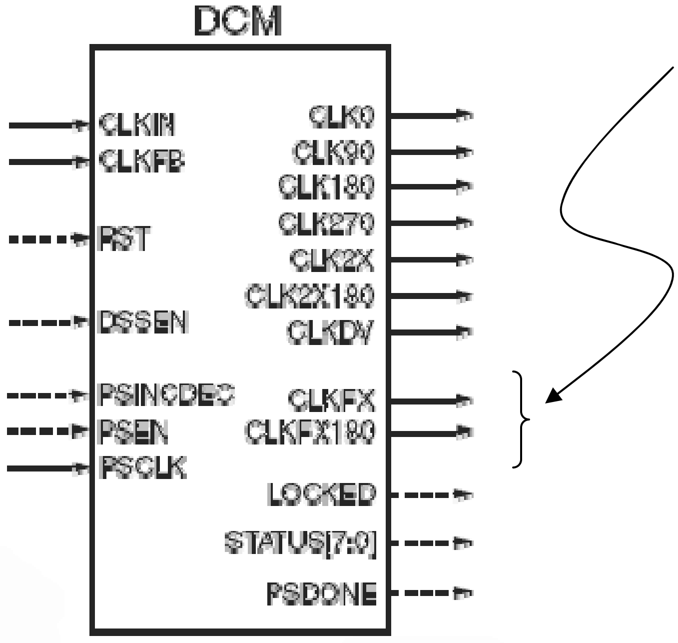
Example: Labkit ZBT interface



In the circuitry above, the lower DCM is used to ensure that the fpga_clock signal, which clocks all of the FPGA flip-flops, is in phase with the refence clock (clock_27mhz, in this example). The upper DCM is used to generate the de-skewed clock for the external ZBT memories. The feedback loop for this DCM includes a 2.0 inch long trace on the labkit PCB. Since all of the PCB traces from the FPGA to the ZBT memories are also 2.0 inches long, the propagation delay from the output of the upper DCM back to its CLKFB input should be almost exactly the same as the propagation delay from the DCM output to the ZBT memories.

Generating Other Clock Frequencies

The labkit has a 27MHz crystal (37ns period). But what if we need a different frequency, e.g., 65MHz to generate 1024x768 VGA video?



Digital Clock Manager

The DCM can also synthesize certain multiples of the CLKIN frequency (eg, multiples of 27MHz):

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

Where M = 2..32 and D = 2..32 with a output frequency of range of 24MHz to 210MHz.

Verilog to generate 65MHz clock

```
// use FPGA's digital clock manager to produce a
// 65MHz clock (actually 64.8MHz)
wire clock_65mhz_unbuf,clock_65mhz;
DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
// synthesis attribute CLKFX_DIVIDE of vclk1 is 10
// synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
// synthesis attribute CLKIN_PERIOD of vclk1 is 37
BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));
```

$$f_{CLKFX} = \left(\frac{24}{10}\right)(27MHz) = 64.8MHz$$

RESETing to a known state

Just after configuration, all the registers/memories are in a known state (eg, default value for regs is 0). But you may need to include a RESET signal to set the initial state to what you want. *Note the Verilog* `initial` *block only works in simulation and has no effect when synthesizing hardware.*

Solution: have your logic take a RESET signal which can be asserted on start up and by an external push button:

```
// power-on reset generation
wire power_on_reset;    // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(clock_27mhz), .Q(power_on_reset),
                .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

// ENTER button is user reset
wire reset,user_reset;
debounce db1(power_on_reset, clock_27mhz, ~button_enter, user_reset);
assign reset = user_reset | power_on_reset;
```

Debugging: making the state visible

To figure out what your circuit is doing it can be very useful to include logic that makes various pieces of state visible to the outside world. Some suggestions:

- turn the leds on and off to signal events, entry into particular pieces of code, etc.

- use the 16-character flourescent display to show more complex state information

- drive useful data onto the USER pins and use the adapters to hook them up to the logic analyzer. Include your master clock signal and the configure the logic analyzer to sample the data on the non-active edge of the clock (to avoid setup and hold problems introduced by I/O pad delays). The logic analyzer can capture thousands of cycles of data and display the results in useful ways.