



INSTITUTO
SUPERIOR
TÉCNICO

Projecto e Controlo em Lógica Digital

Refs:

Cyclone II device Handbook, Altera corp.
Quartus II Handbook, Altera corp.
DE2 documentation
Verilog HDL, S. Palnitkar, Prentice Hall

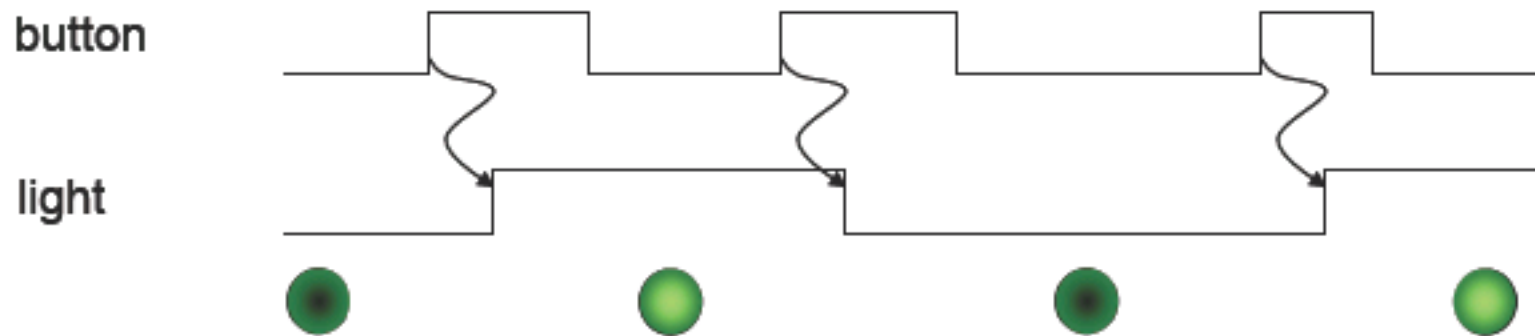
Inspired on course 6.111 from MIT
Some support material taken from 6.111 (thank you)

- Máquinas de estados
- Clocks
 - PII

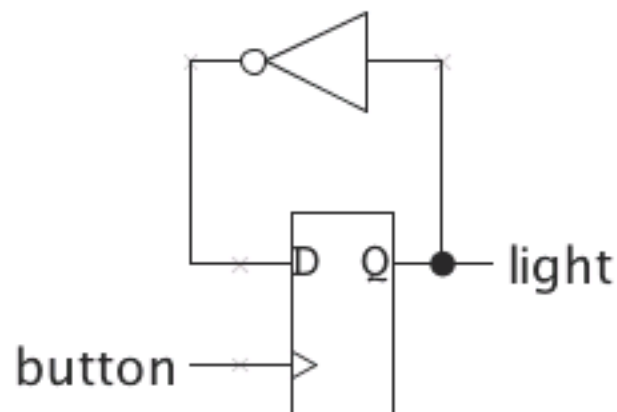
- Introdução
 - Fpgas
 - Laboratório
 - Verilog
 - Lógica combinatória
 - Lógica sequencial
- Ferramentas
 - Simulação
 - no QuartusII
 - em Verilog
 - Osciloscópio
 - Analisador lógico
 - Interno
 - Externo

- Advanced verilog (video, etc.)

Implementation for on/off button



```
module onoff(input button, output reg light);  
  always @(posedge button) light <= ~light;  
endmodule
```



Synchronous on/off button

When designing a system that accepts many inputs it would be hard to have input changes serve as the system clock (which input would we use?). So we'll use a single clock of some fixed frequency and have the inputs control what state changes happen on rising clock edges.

```
module onoff_sync(input clk, button,
                  output reg light);
    always @ (posedge clk) begin
        if (button) light <= ~light;
    end
endmodule
```



Resetting to a known state

Usually one can't rely on registers powering-on to a particular initial state*. So most designs have a RESET signal that when asserted initializes all the state to known, mutually consistent initial values.

```
module onoff_sync(input clk, reset, button,
                  output reg light);
    always @ (posedge clk) begin
        if (reset) light <= 0;
        else if (button) light <= ~light;
    end
endmodule
```

* Actually, our FPGAs will reset all registers to 0 when the device is programmed. But it's nice to be able to press a reset button to return to a known state rather than starting from scratch by reprogramming the device.



Clocks are fast, we're slow!

The circuit on the last slide toggles the light on every rising clock edge for which button is 1. But clocks are fast (27MHz!) and our fingers are slow, so how do we press the button for just one clock edge? Answer: we can't, but we can add some state that remembers what button was last clock cycle and then detect the clock cycles when button changes from 0 to 1.

```
module onoff_sync(input clk, reset, button,
                  output reg light);
    reg old_button; // state of button last clk
    always @ (posedge clk) begin
        if (reset)
            begin light <= 0; old_button <= 0; end
        else if (old_button==0 && button==1)
            // button changed from 0 to 1
            light <= ~light;
            old_button <= button;
        end
    endmodule
```



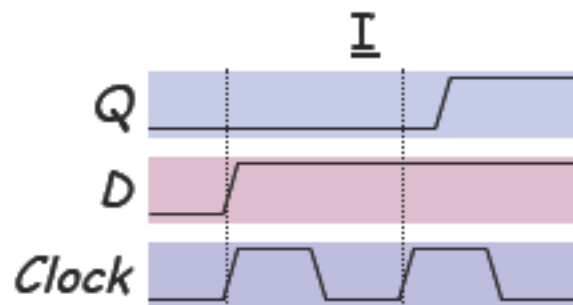
Asynchronous Inputs in Sequential Systems

What about external signals?

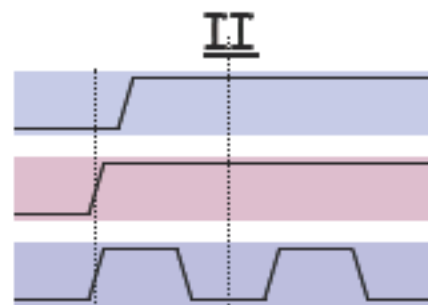


Can't guarantee setup and hold times will be met!

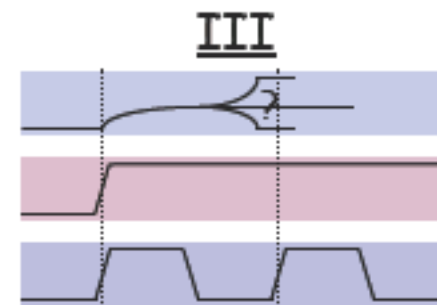
When an asynchronous signal causes a setup/hold violation...



Transition is missed on first clock cycle, but caught on next clock cycle.



Transition is caught on first clock cycle.



Output is metastable for an indeterminate amount of time.

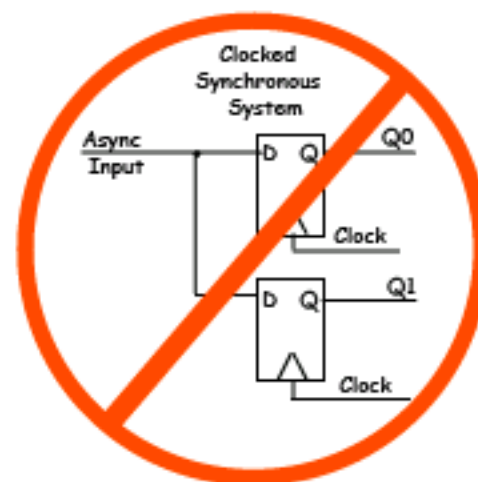
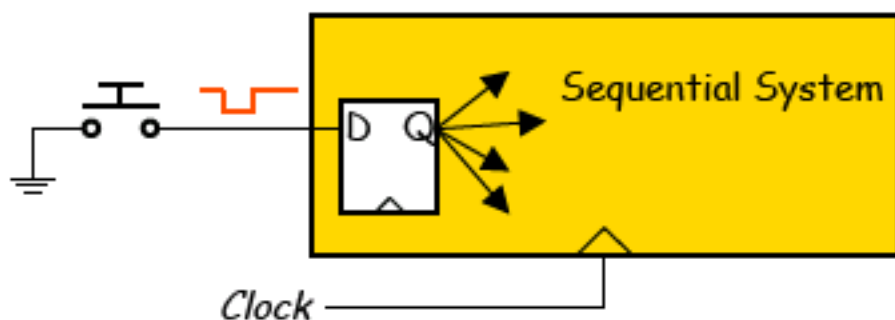
Q: Which cases are problematic?



Asynchronous Inputs in Sequential Systems

All of them can be, if more than one happens simultaneously within the same circuit.

Guideline: ensure that external signals directly feed exactly one flip-flop

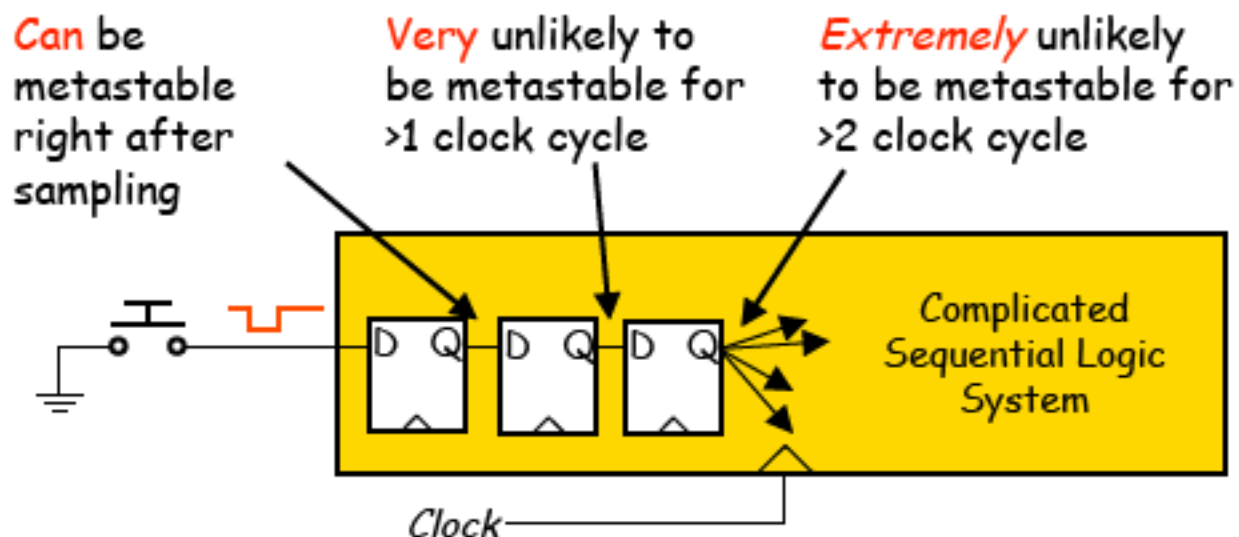


This prevents the possibility of I and II occurring in different places in the circuit, but what about metastability?



Handling Metastability

- Preventing metastability turns out to be an impossible problem
- High gain of digital devices makes it likely that metastable conditions will resolve themselves quickly
- Solution to metastability: allow time for signals to stabilize



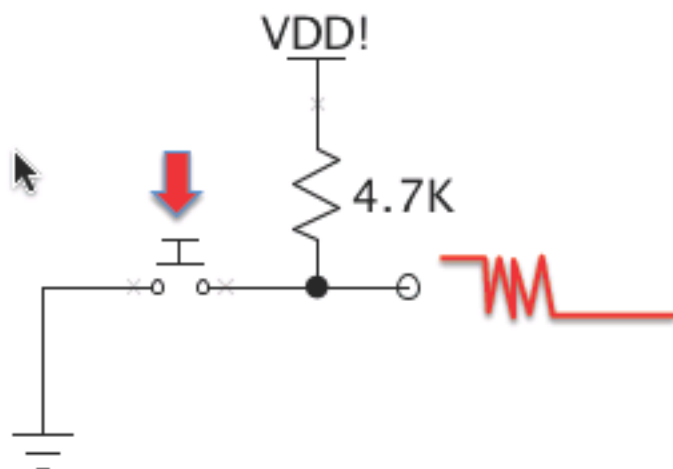
How many registers are necessary?

- Depends on many design parameters (clock speed, device speeds, ...)
- In 6.111, a pair of synchronization registers is sufficient



One last little problem...

Mechanical buttons exhibit contact "bounce" when they change position, leading to multiple output transitions before finally stabilizing in the new position:



We need a debouncing circuit!

```
// Switch Debounce Module
// use your system clock for the clock input
// to produce a synchronous, debounced output
// DELAY = .01 sec with a 27Mhz clock
module debounce #(parameter DELAY=270000)
    (input reset, clock, noisy,
     output reg clean);

    reg [18:0] count;
    reg new;

    always @(posedge clock)
        if (reset) // return to known state
            begin
                count <= 0;
                new <= noisy;
                clean <= noisy;
            end
        else if (noisy != new) // input changed
            begin
                new <= noisy;
                count <= 0;
            end
        else if (count == DELAY) // stable!
            clean <= new;
        else // waiting...
            count <= count+1;

endmodule
```

On/off button: final answer

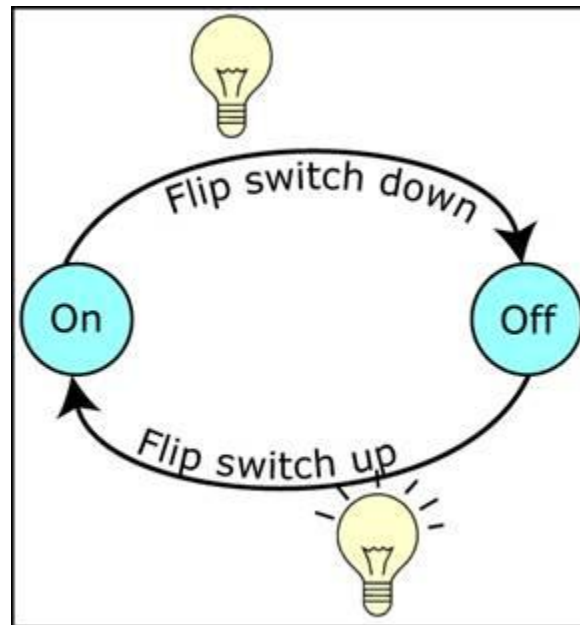
```
module onoff_sync(input clk, reset, button_in,
                  output reg light);
    // synchronizer
    reg button,btemp;
    always @(posedge clk)
        {button,btemp} <= {btemp,button_in};

    // debounce push button
    wire bpressed;
    debounce db1(.clock(clk),.reset(reset),
                .noisy(button),.clean(bpressed));

    reg old_bpressed; // state last clk cycle
    always @ (posedge clk) begin
        if (reset)
            begin light <= 0; old_bpressed <= 0; end
        else if (old_bpressed==0 && bpressed==1)
            // button changed from 0 to 1
            light <= ~light;
            old_bpressed <= bpressed;
        end
    endmodule
```

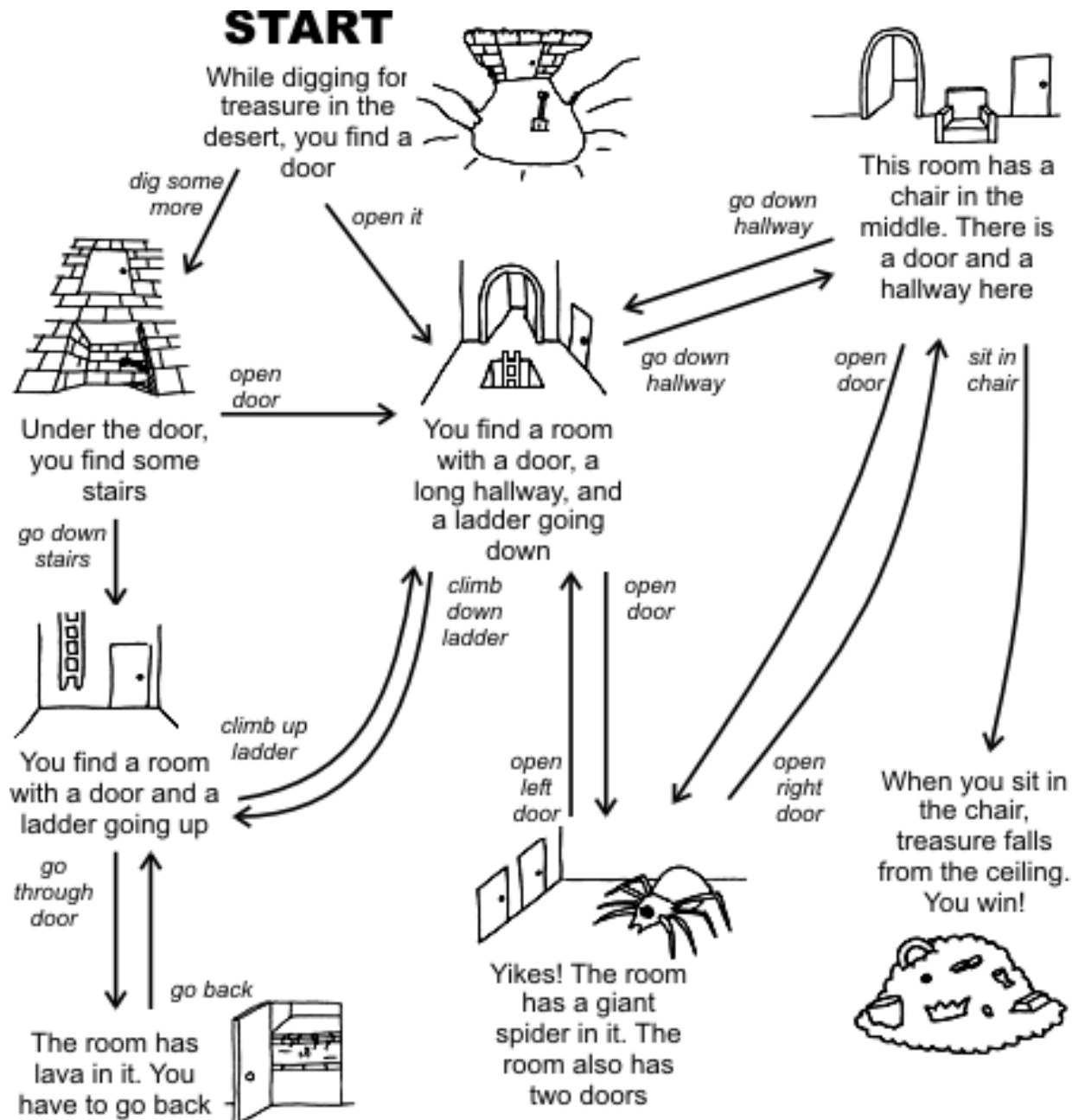
Máquinas de estado

Uma muito simples...



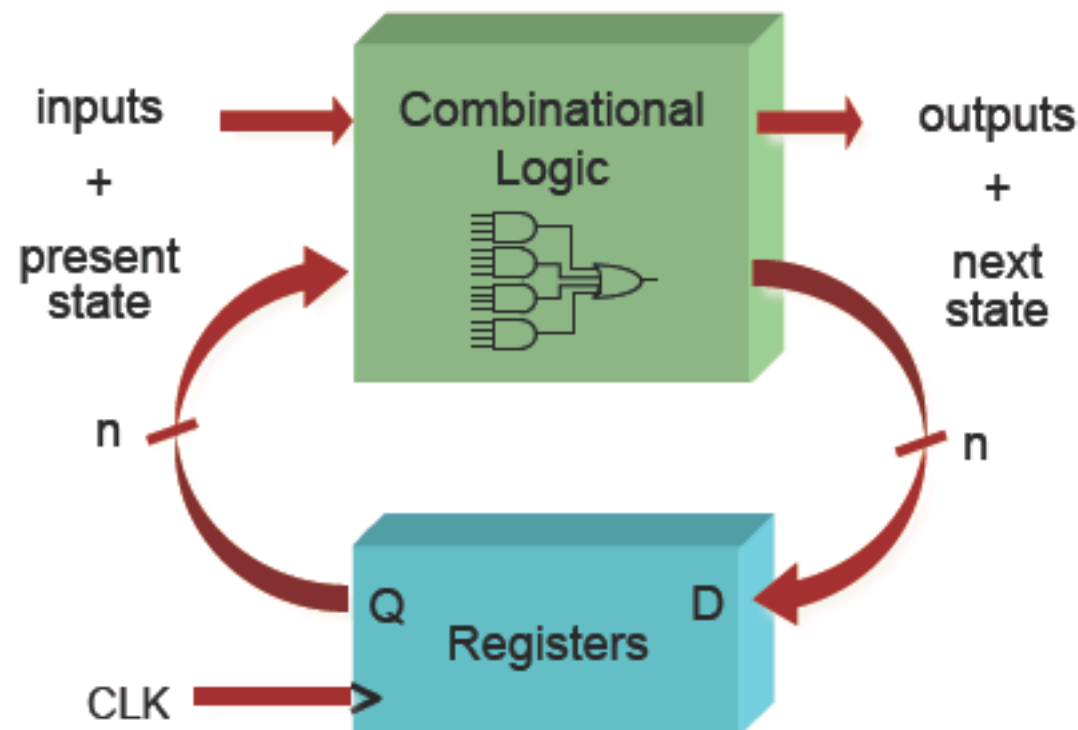
Máquinas de estado

Mas podem ficar complicadas...



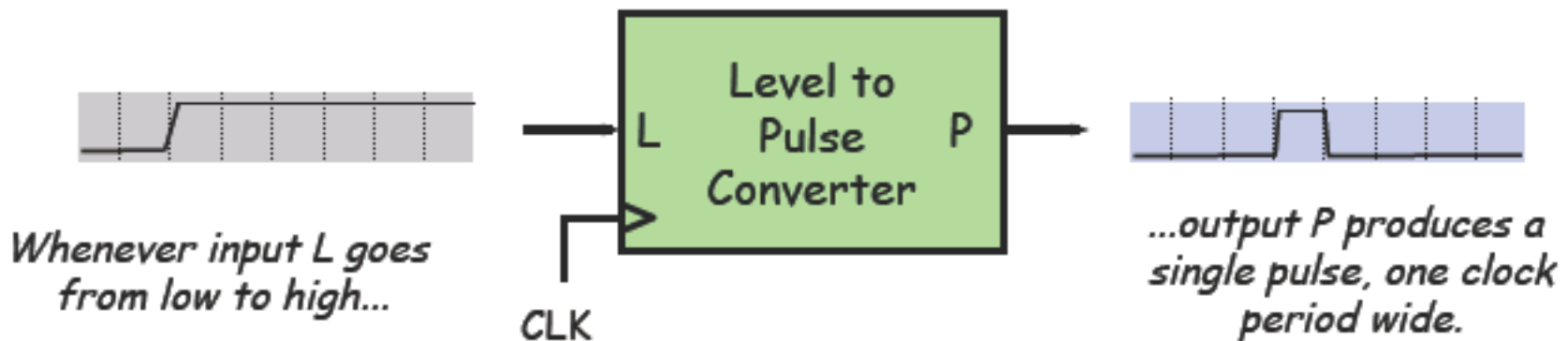
Finite State Machines

- Finite State Machines (FSMs) are a useful abstraction for *sequential circuits* with centralized "*states*" of operation
- At each clock edge, combinational logic computes *outputs* and *next state* as a function of *inputs* and *present state*



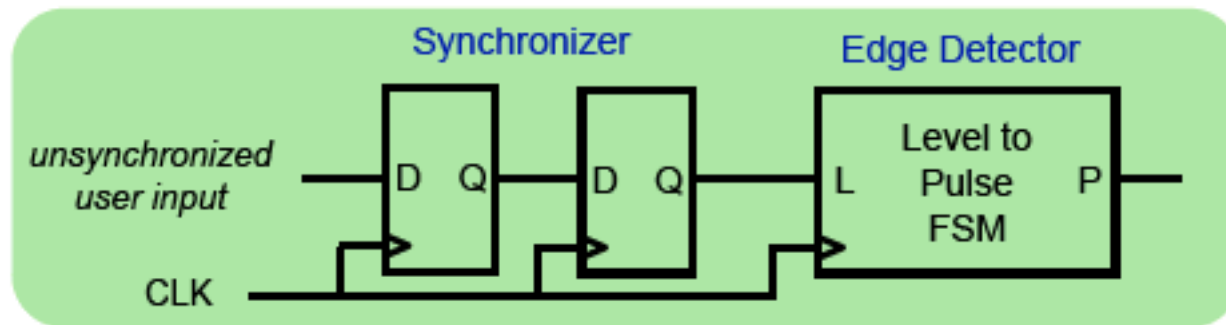
Design Example: Level-to-Pulse

- 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 counters

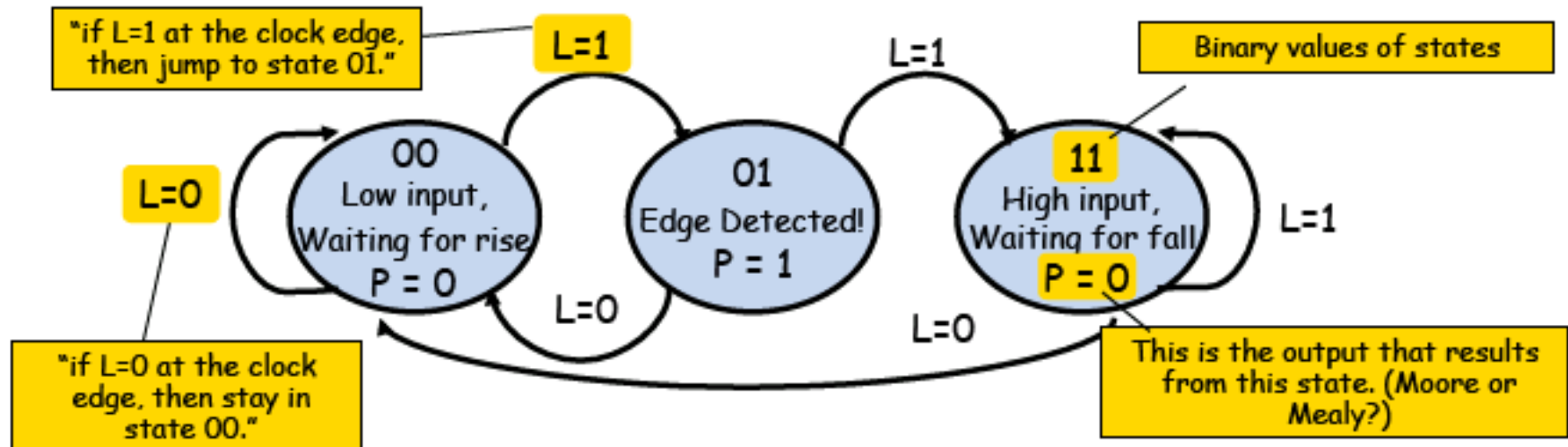


Step 1: State Transition Diagram

- Block diagram of desired system:



- State transition diagram** is a useful FSM representation and design aid:



In each state there should be one and only one arc for each input combination₁₅

Choosing State Representation

Choice #1: binary encoding

For N states, use $\text{ceil}(\log_2 N)$ bits to encode the state with each state represented by a unique combination of the bits.

Tradeoffs: most efficient use of state registers, but requires more complicated combinational logic to detect when in a particular state.

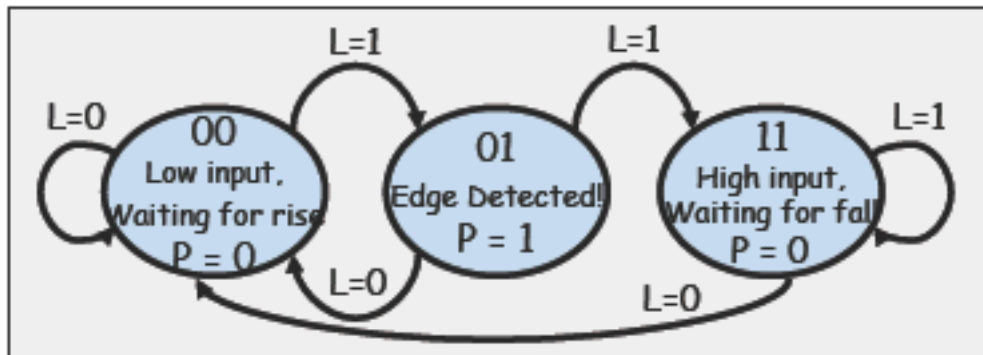
Choice #2: "one-hot" encoding

For N states, use N bits to encode the state where the bit corresponding to the current state is 1, all the others 0.

Tradeoffs: more state registers, but often much less combinational logic since state decoding is trivial.

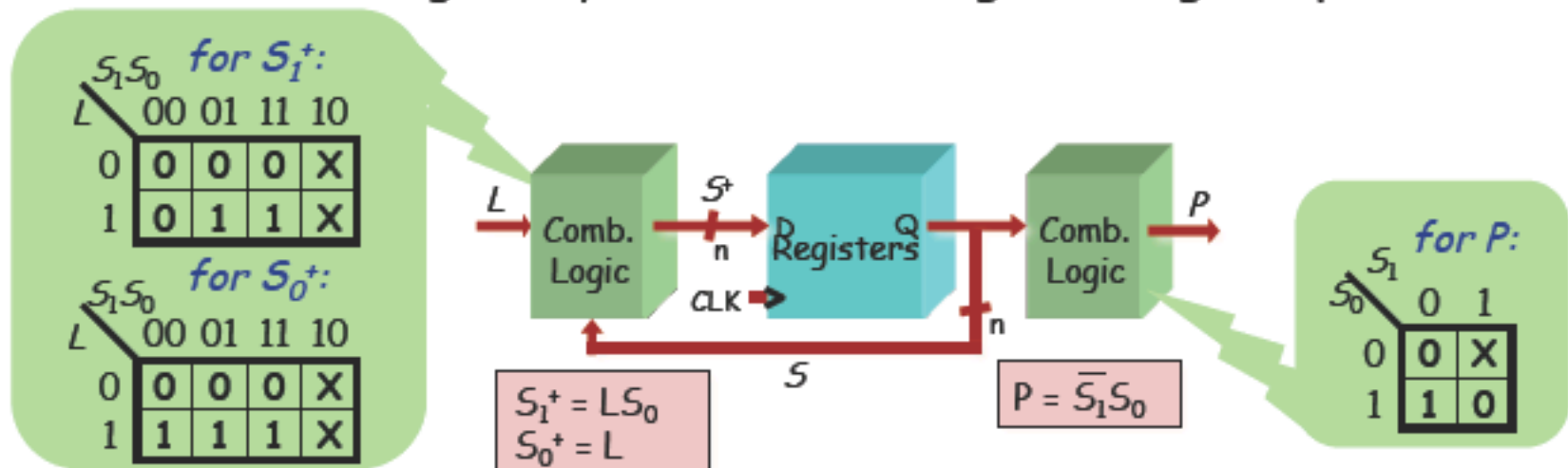
Step 2: Logic Derivation

Transition diagram is readily converted to a state transition table (just a truth table)

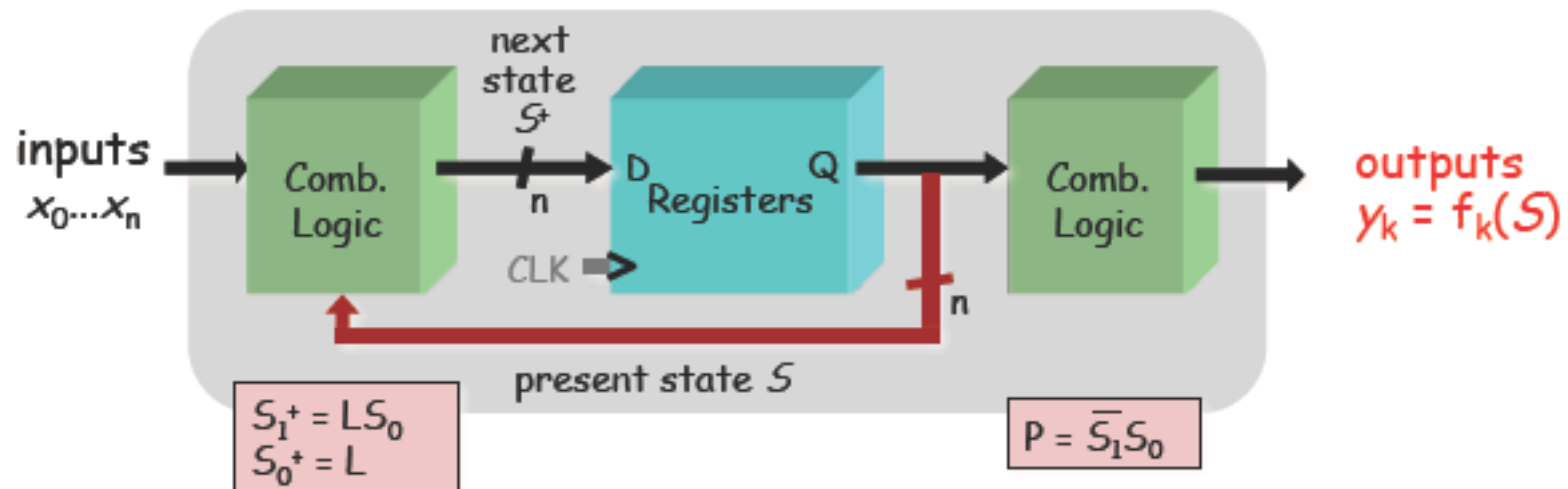


Current State		In	Next State		Out
S_1	S_0	L	S_1^+	S_0^+	P
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	1
1	1	0	0	0	0
1	1	1	1	1	0

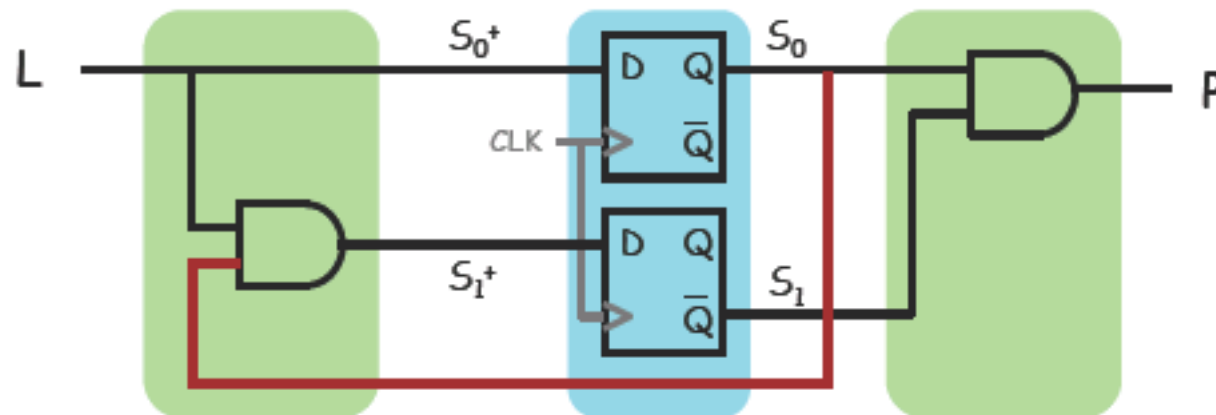
- Combinational logic may be derived using Karnaugh maps

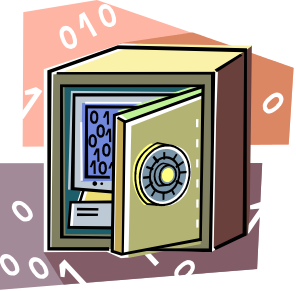


Moore Level-to-Pulse Converter



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





FSM Example

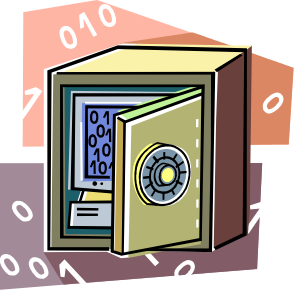
GOAL:

Build an electronic combination lock with a reset button, two number buttons (0 and 1), and an unlock output. The combination should be **01011**.

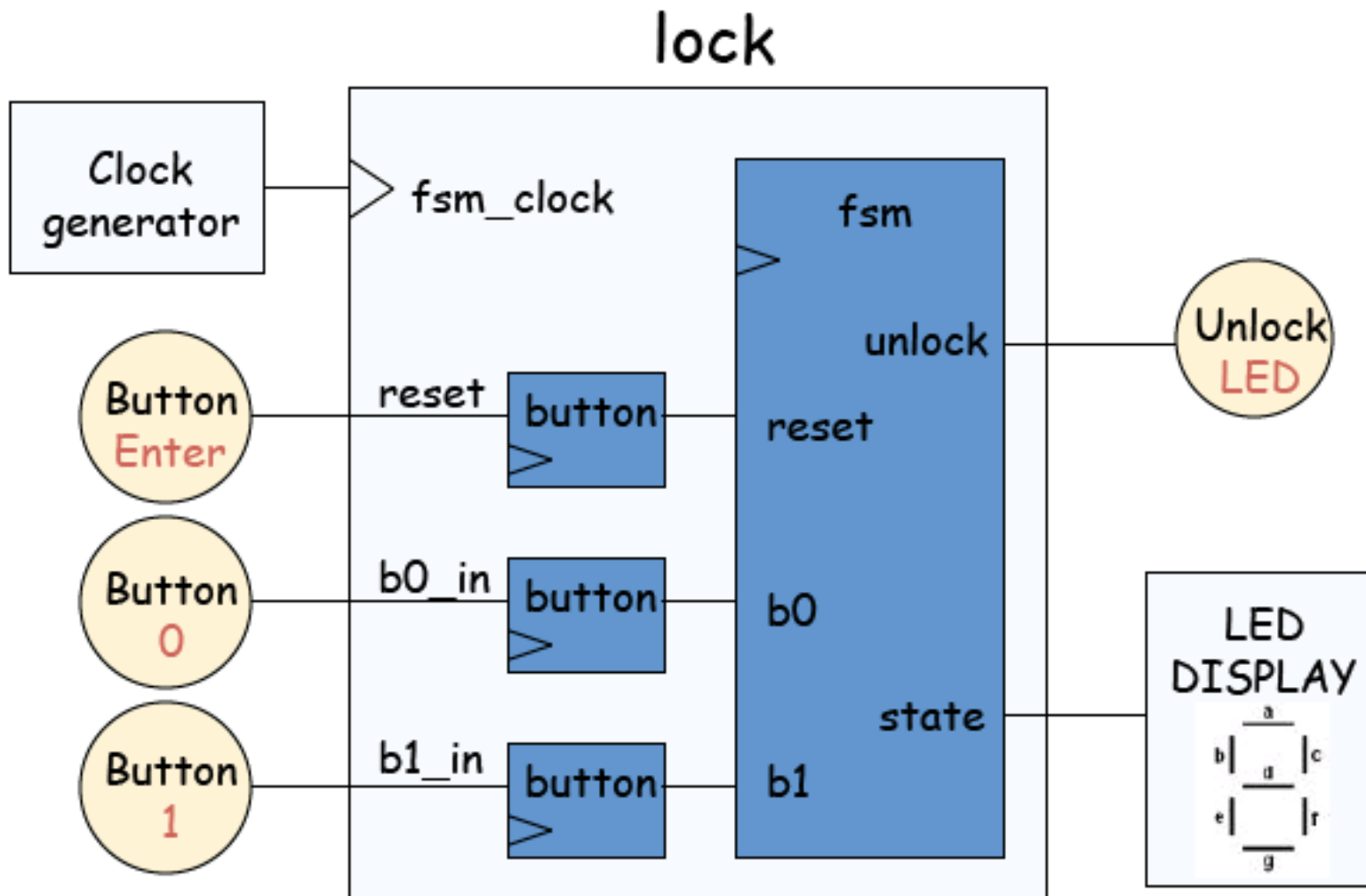


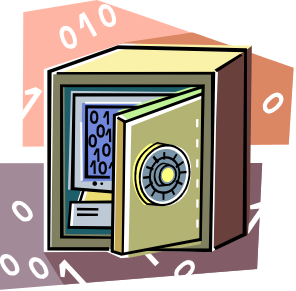
STEPS:

1. Design lock FSM (block diagram, state transitions)
2. Write Verilog module(s) for FSM

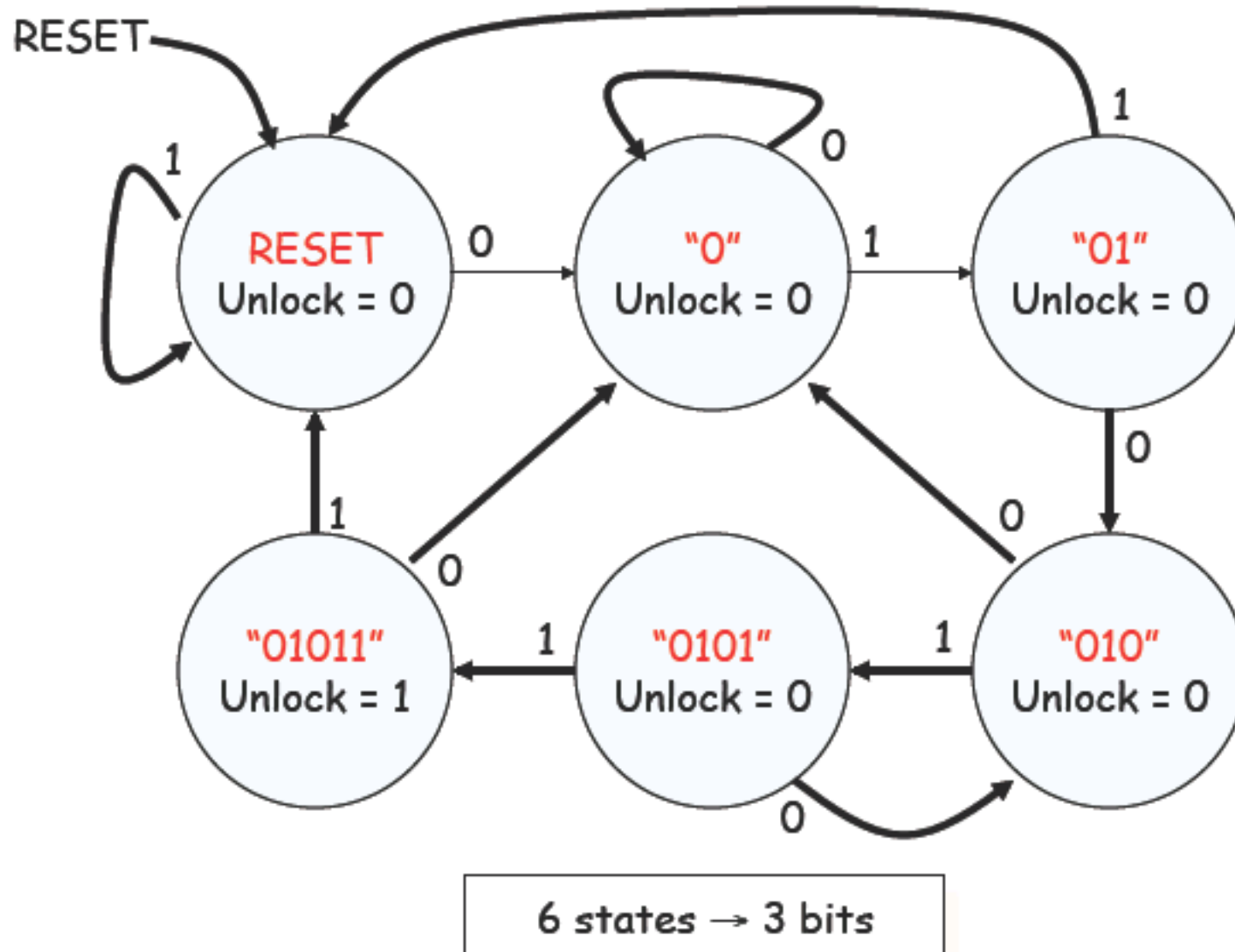


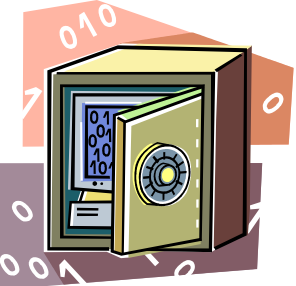
Step 1A: Block Diagram





Step 1B: State transition diagram





Step 2: Write Verilog

```
module lock(input clk,reset_in,b0_in,b1_in,  
            output out);
```

```
// synchronize push buttons, convert to pulses
```

```
// implement state transition diagram
```

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

```
always @(*) begin
```

```
    // combinational logic!
```

```
    next_state = ???;
```

```
end
```

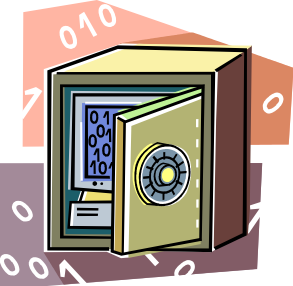
```
always @(posedge clk) state <= next_state;
```

```
// generate output
```

```
assign out = ???;
```

```
// debugging?
```

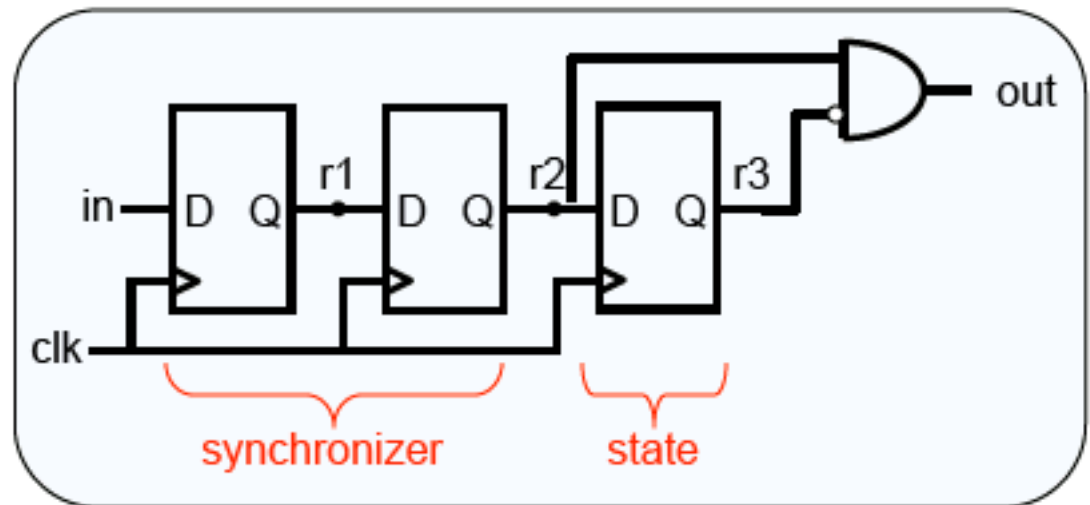
```
endmodule
```



Step 2A: Synchronize buttons

// button
// push button synchronizer and level-to-pulse converter
// OUT goes high for one cycle of CLK whenever IN makes a
// low-to-high transition.

```
module button(  
    input clk,in,  
    output out  
);  
    reg r1,r2,r3;  
    always @(posedge clk)  
    begin  
        r1 <= in;    // first reg in synchronizer  
        r2 <= r1;    // second reg in synchronizer, output is in sync!  
        r3 <= r2;    // remembers previous state of button  
    end
```



```
    // rising edge = old value is 0, new value is 1  
    assign out = ~r3 & r2;  
endmodule
```

Step 2B: state transition diagram

```
parameter S_RESET = 0; // state assignments
parameter S_0 = 1;
parameter S_01 = 2;
parameter S_010 = 3;
parameter S_0101 = 4;
parameter S_01011 = 5;
```

```
reg [2:0] state, next_state;
always @(*) begin
```

```
    // implement state transition diagram
```

```
    if (reset) next_state = S_RESET;
```

```
    else case (state)
```

```
        S_RESET: next_state = b0 ? S_0 : (b1 ? S_RESET : state);
```

```
        S_0: next_state = b0 ? S_0 : (b1 ? S_01 : state);
```

```
        S_01: next_state = b0 ? S_010 : (b1 ? S_RESET : state);
```

```
        S_010: next_state = b0 ? S_0 : (b1 ? S_0101 : state);
```

```
        S_0101: next_state = b0 ? S_010 : (b1 ? S_01011 : state);
```

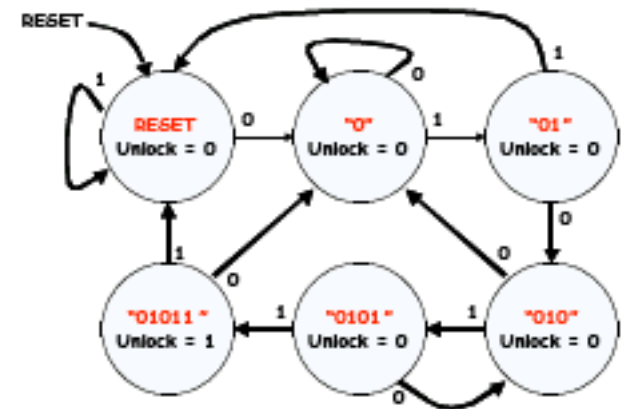
```
        S_01011: next_state = b0 ? S_0 : (b1 ? S_RESET : state);
```

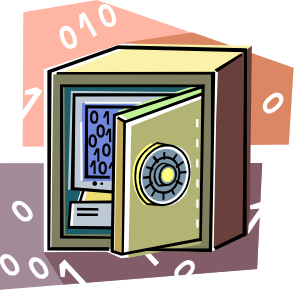
```
        default: next_state = S_RESET; // handle unused states
```

```
    endcase
```

```
end
```

```
always @(posedge clk) state <= next_state;
```





Step 2C: generate output

```
// it's a Moore machine! Output only depends on current state  
assign out = (state == S_01011);
```

Step 2D: debugging?

```
// hmmm. What would be useful to know? Current state?  
assign hex_display = {1'b0, state[2:0]};
```



Step 2: final Verilog implementation

```
module lock(input clk,reset_in,b0_in,b1_in,
            output out, output [3:0] hex_display);

    wire reset, b0, b1; // synchronize push buttons, convert to pulses
    button b_reset(clk,reset_in,reset);
    button b_0(clk,b0_in,b0);
    button b_1(clk,b1_in,b1);

    parameter S_RESET = 0; parameter S_0 = 1; // state assignments
    parameter S_01 = 2; parameter S_010 = 3;
    parameter S_0101 = 4; parameter S_01011 = 5;

    reg [2:0] state,next_state;
    always @(*) begin // implement state transition diagram
        if (reset) next_state = S_RESET;
        else case (state)
            S_RESET: next_state = b0 ? S_0 : (b1 ? S_RESET : state);
            S_0:      next_state = b0 ? S_0 : (b1 ? S_01 : state);
            S_01:     next_state = b0 ? S_010 : (b1 ? S_RESET : state);
            S_010:    next_state = b0 ? S_0 : (b1 ? S_0101 : state);
            S_0101:   next_state = b0 ? S_010 : (b1 ? S_01011 : state);
            S_01011:  next_state = b0 ? S_0 : (b1 ? S_RESET : state);
            default:  next_state = S_RESET; // handle unused states
        endcase
    end
    always @(posedge clk) state <= next_state;

    assign out = (state == S_01011); // assign output: Moore machine
    assign hex_display = {1'b0,state}; // debugging
endmodule
```

Intervalo

Partitioning state machines

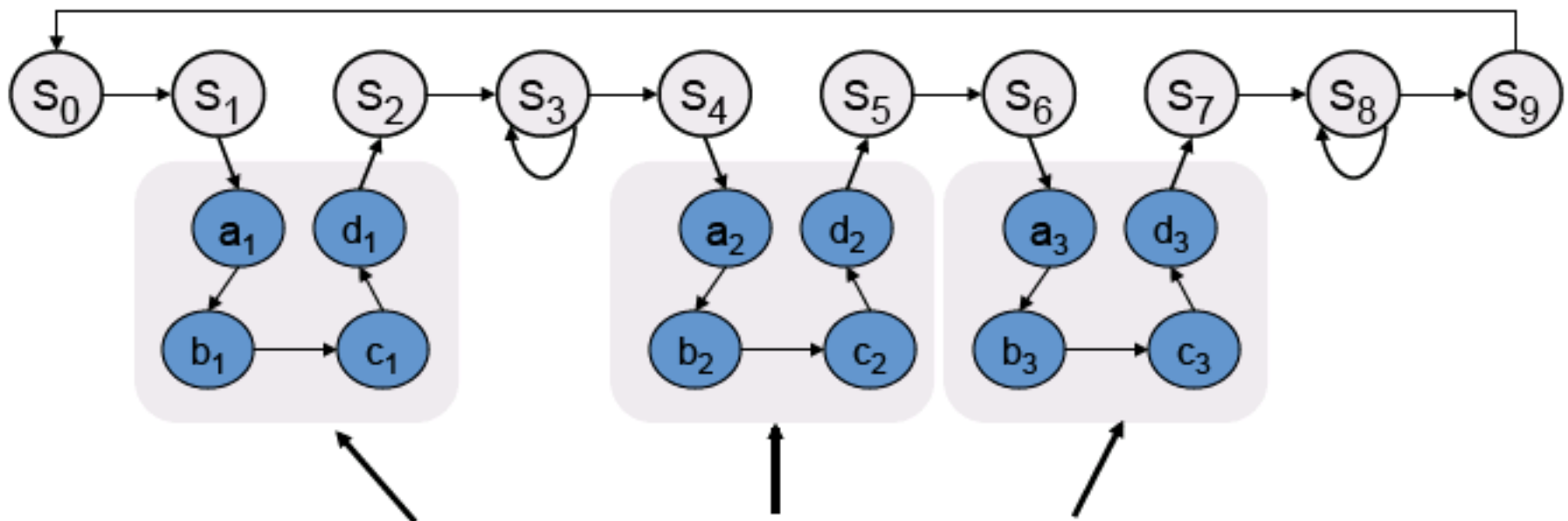
Basically the trick is that you need to put the top one on hold

Partitioning state machines

Basically the trick is that you need to put the top one on hold

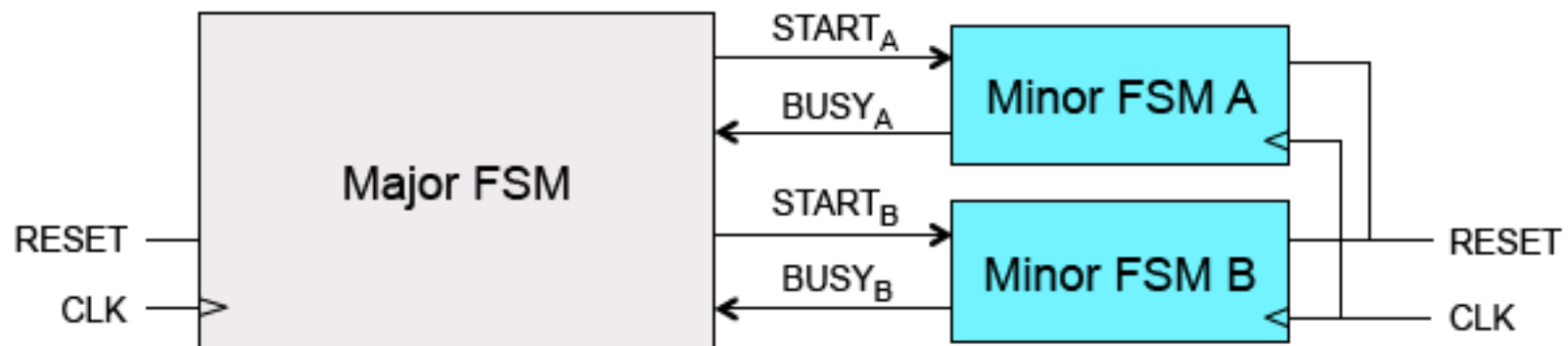
Toward FSM Modularity

- Consider the following abstract FSM:



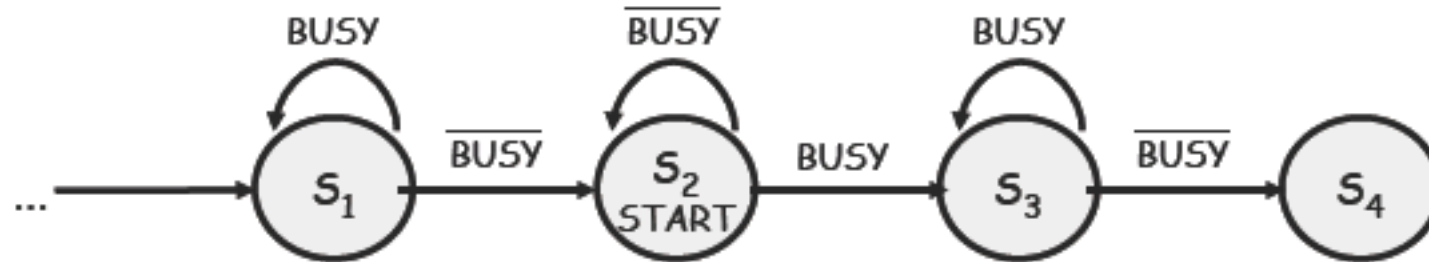
- Suppose that each set of states $a_x \dots d_x$ is a "sub-FSM" that produces exactly the same outputs.
- Can we simplify the FSM by removing equivalent states?
No! The outputs may be the same, but the next-state transitions are not.
- This situation closely resembles a **procedure call** or **function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

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

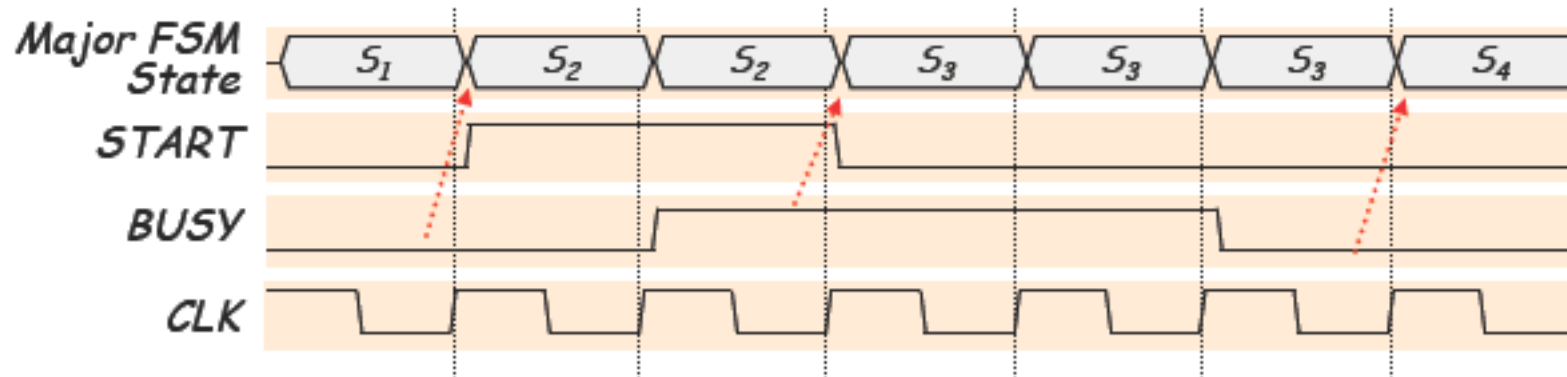
Inside the Major FSM



1. Wait until the minor FSM is ready

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

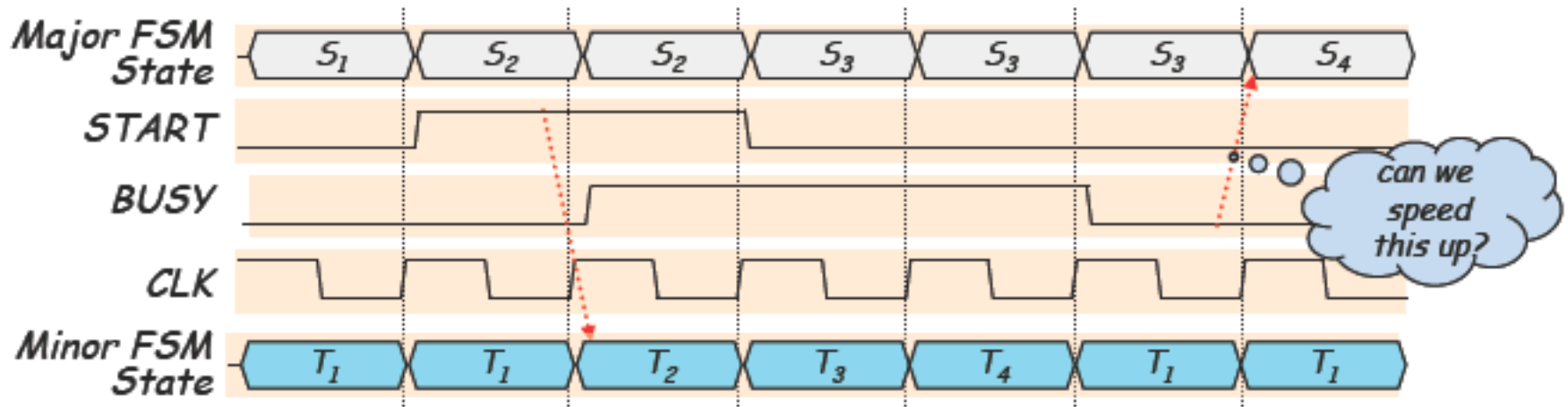
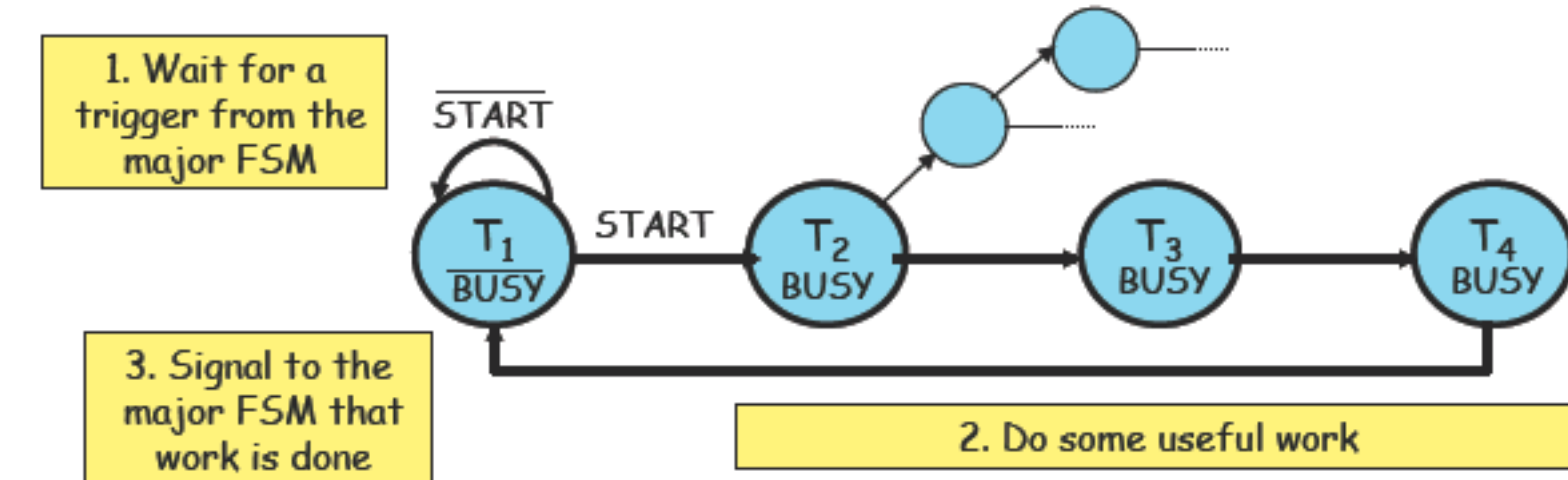
3. Wait until the minor FSM is done



Variations:

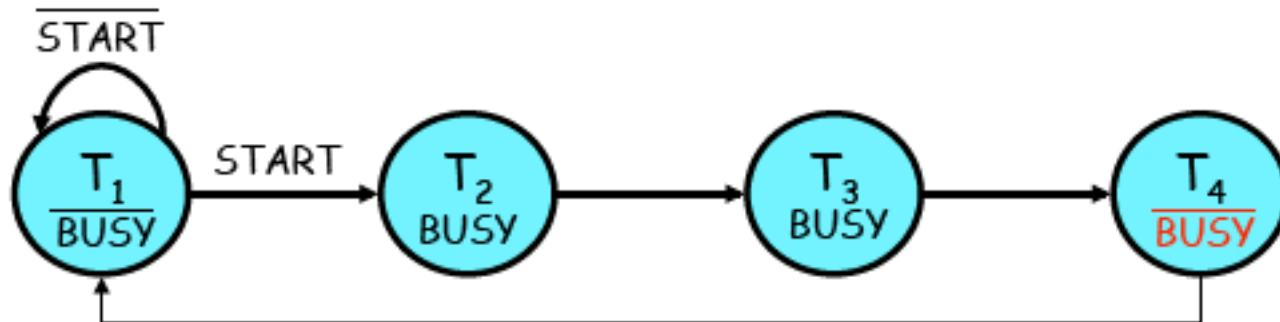
- Usually don't need both Step 1 and Step 3
- One cycle "done" signal instead of multi-cycle "busy"

Inside the Minor FSM



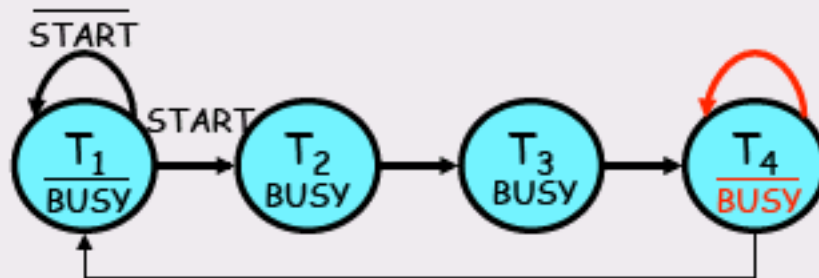
Optimizing the Minor FSM

Good idea: de-assert BUSY one cycle early



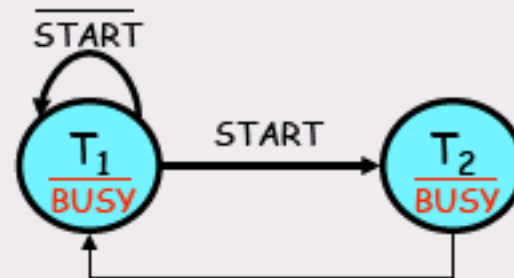
Bad idea #1:

T₄ may not immediately return to T₁

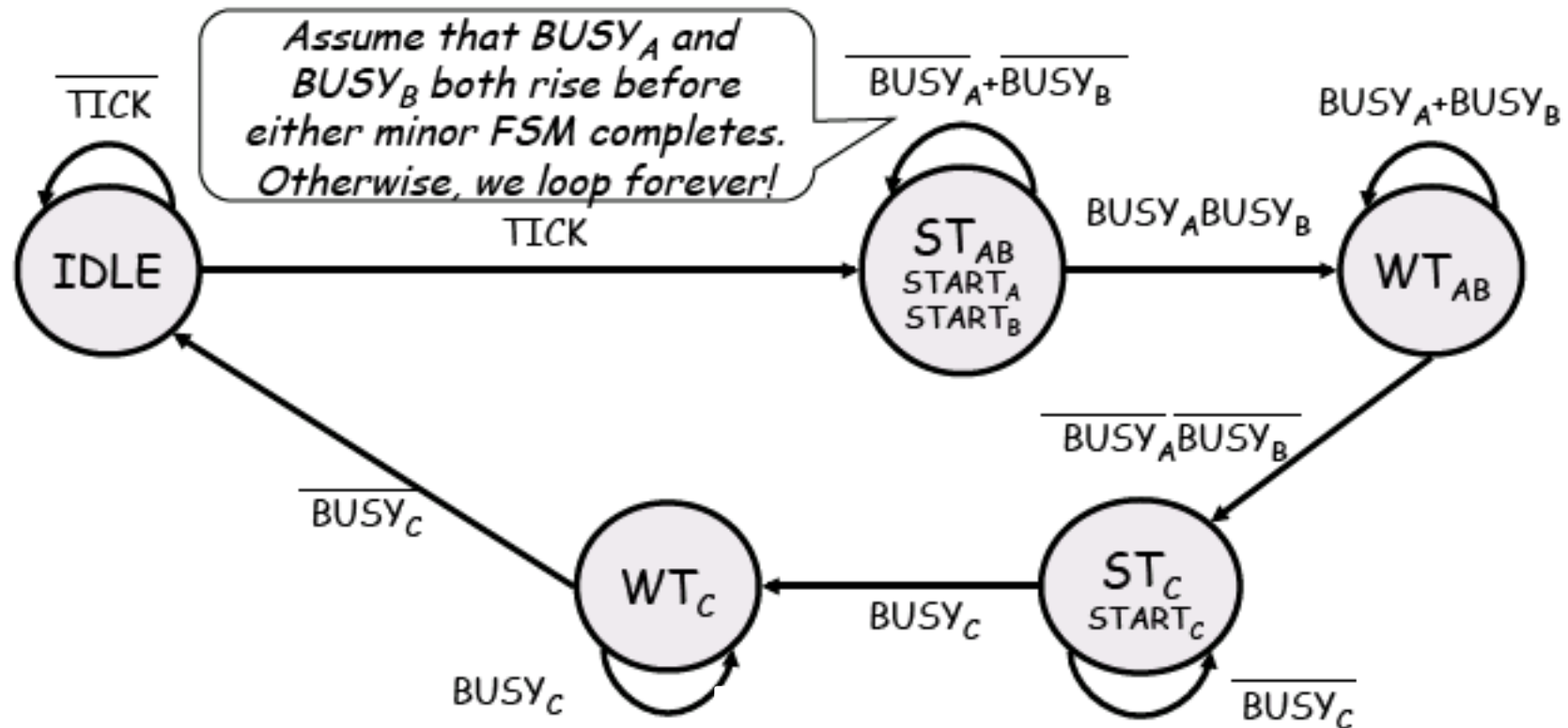
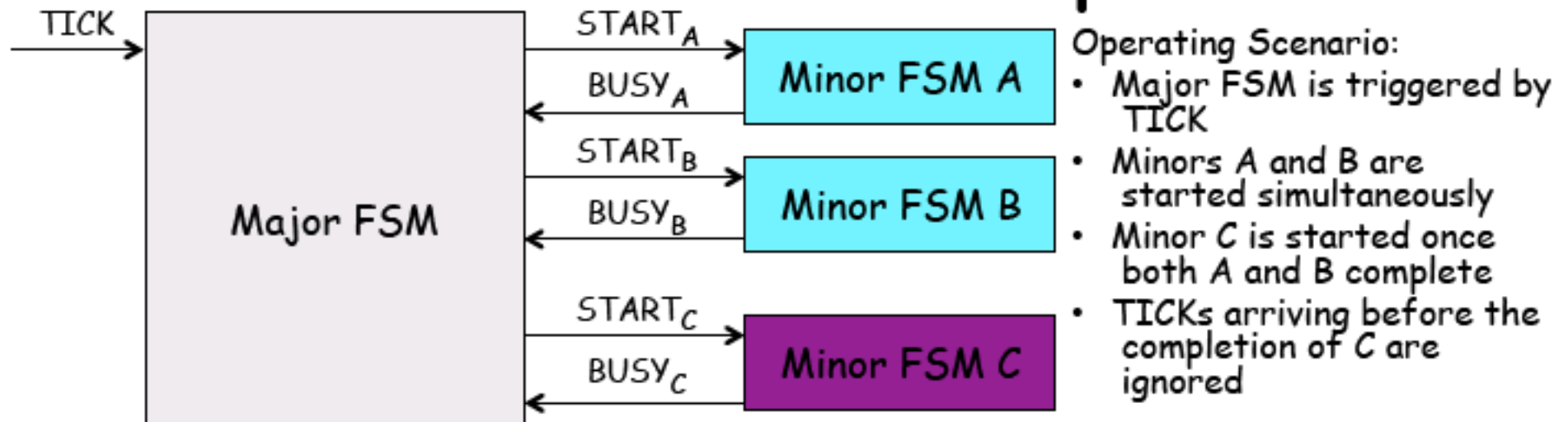


Bad idea #2:

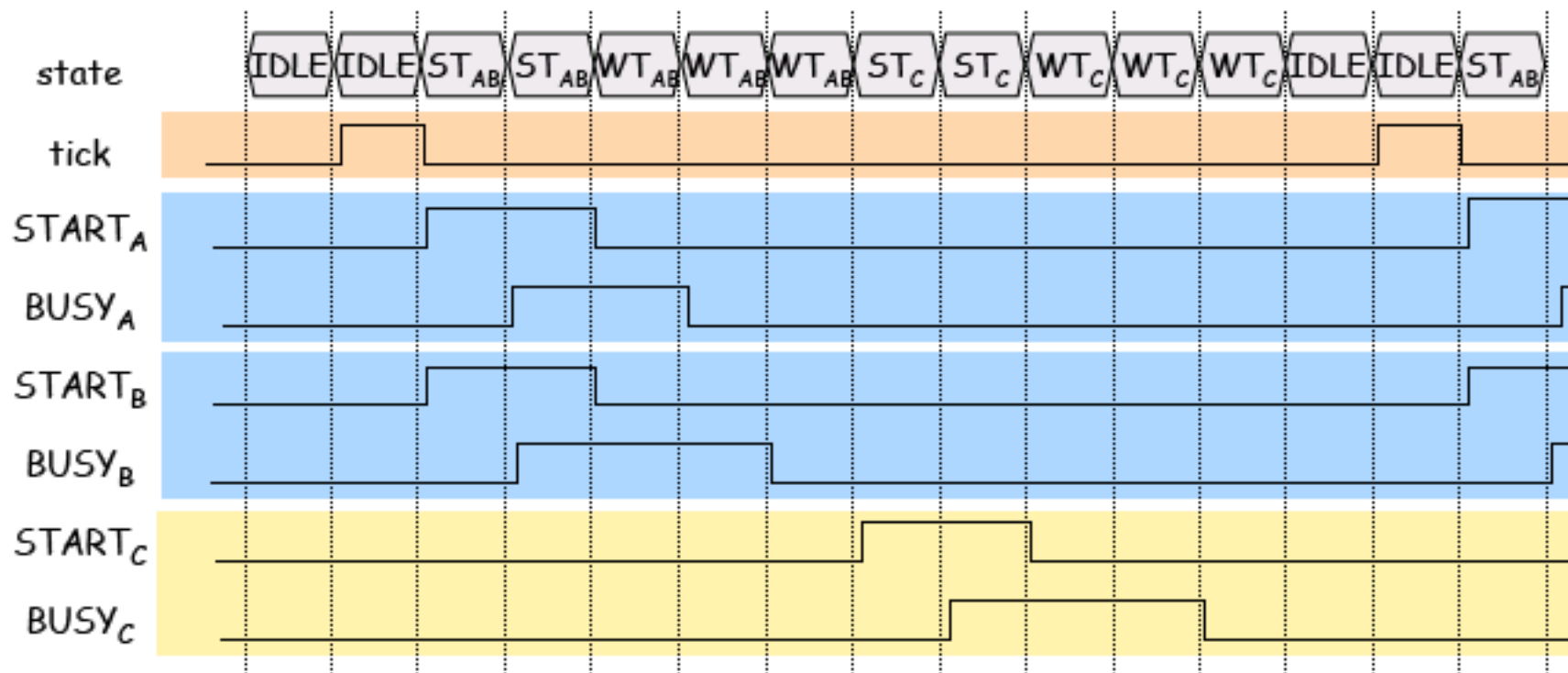
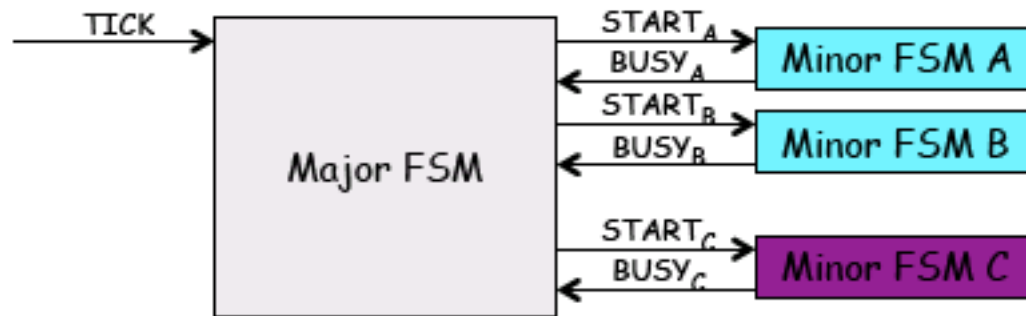
BUSY never asserts!



A Four-FSM Example



Four-FSM Sample Waveform



A vending machine

- All selections are \$0.30.
- The machine makes change. (Dimes and nickels only.)
- Inputs: limit 1 per clock
 - Q - quarter inserted
 - D - dime inserted
 - N - nickel inserted
- Outputs: limit 1 per clock
 - DC - dispense can
 - DD - dispense dime
 - DN - dispense nickel



What states?



- A starting (idle) state:

idle

- A state for each possible amount of money captured:

got5c

got10c

got15c

...

- What's the maximum amount of money captured before purchase?

25 cents (just shy of a purchase) + one quarter (largest coin)

...

got35c

got40c

got45c

got50c

- States to dispense change (one per coin dispensed):

got45c

Dispense
Dime

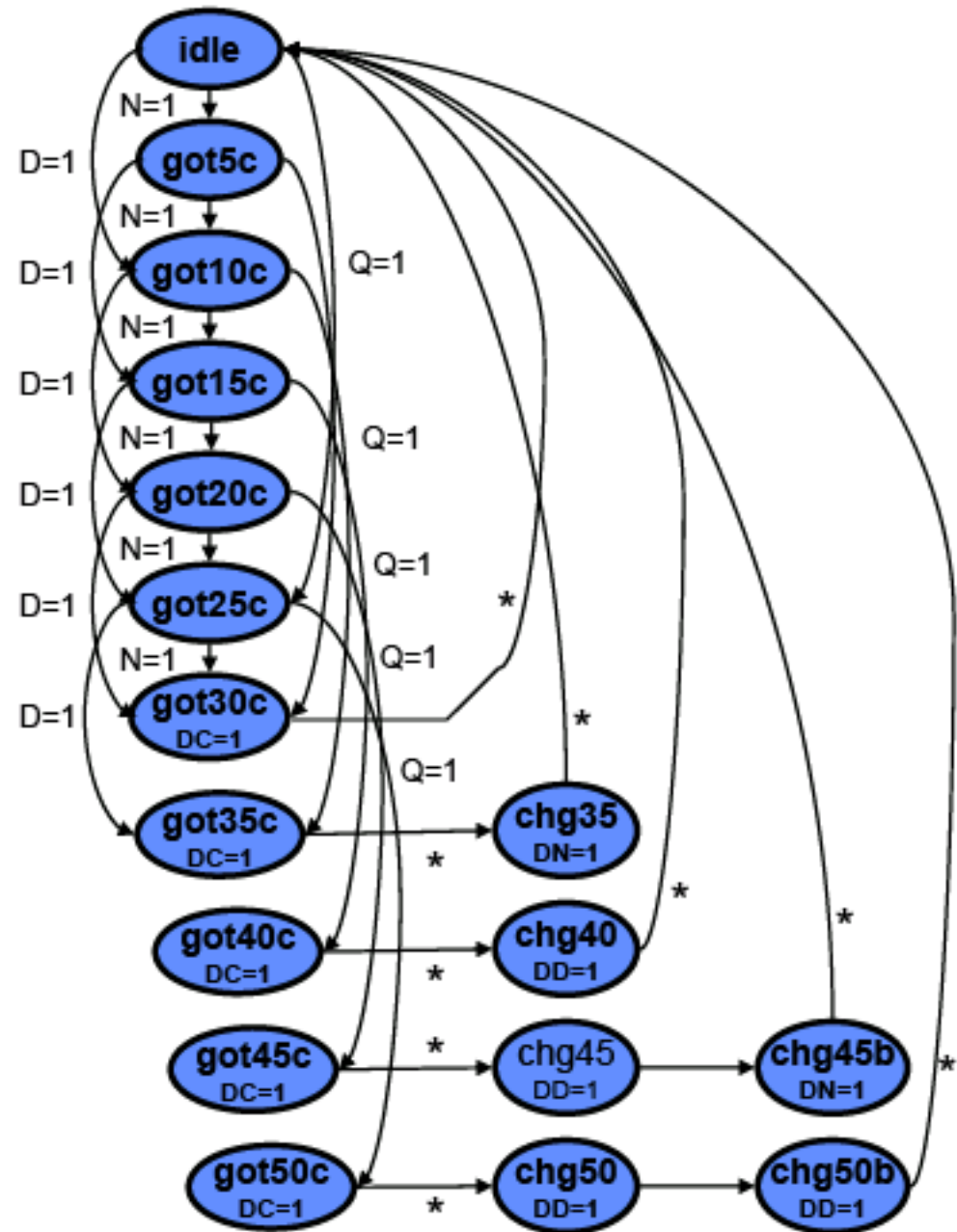
Dispense
Nickel

A Moore vender



Here's a first cut at the state transition diagram.

See a better way?
So do we.
Don't go away...

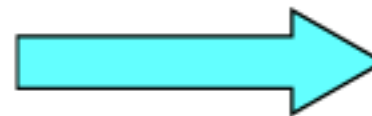
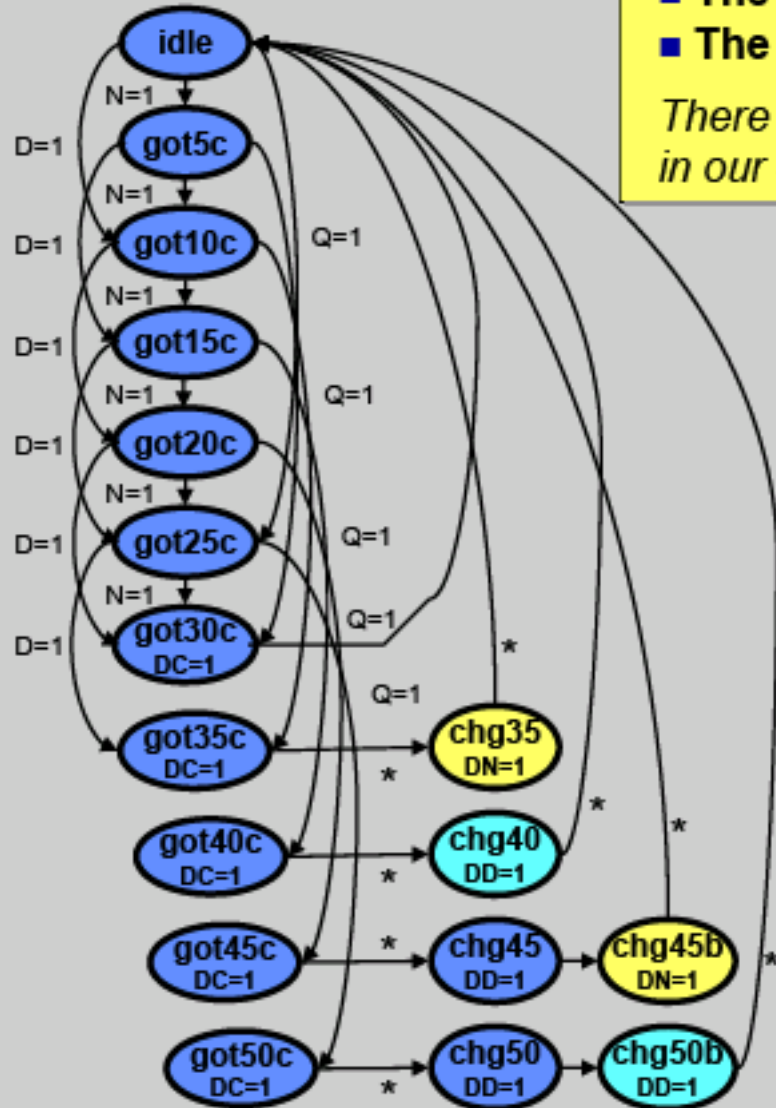


State reduction

Duplicate states have:

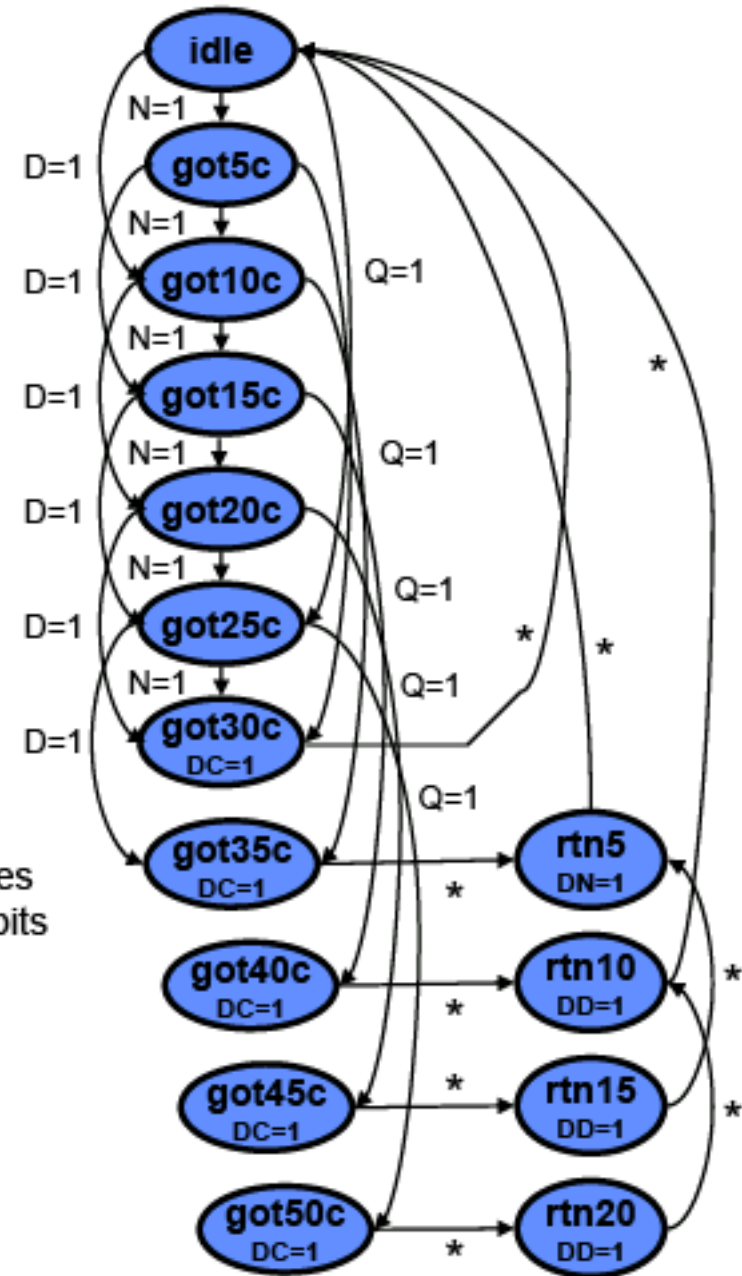
- The same outputs, and
- The same transitions

There are two duplicates in our original diagram.

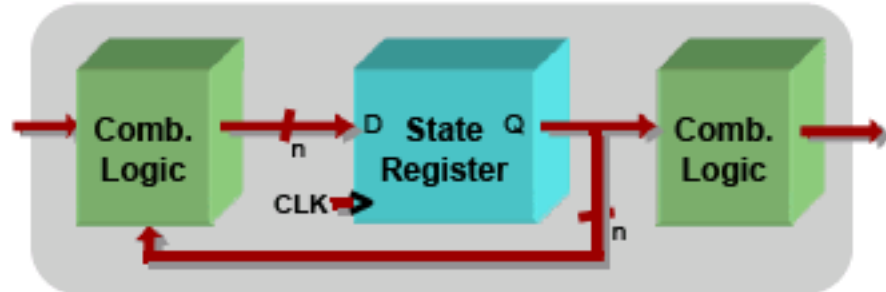


17 states
5 state bits

15 states
4 state bits



The verilog code



***FSMs are easy in Verilog.
Simply write one of each:***

- **State register**
(sequential always block)
- **Next-state combinational logic**
(comb. always block with case)
- **Output combinational logic block**
(comb. always block or assign statements)

```
module mooreVender (N, D, Q, DC, DN, DD,  
                    clk, reset, state);  
    input N, D, Q, clk, reset;  
    output DC, DN, DD;  
    output [3:0] state;  
    reg [3:0] state, next;
```

States defined with **parameter keyword**

```
    parameter IDLE = 0;  
    parameter GOT_5c = 1;  
    parameter GOT_10c = 2;  
    parameter GOT_15c = 3;  
    parameter GOT_20c = 4;  
    parameter GOT_25c = 5;  
    parameter GOT_30c = 6;  
    parameter GOT_35c = 7;  
    parameter GOT_40c = 8;  
    parameter GOT_45c = 9;  
    parameter GOT_50c = 10;  
    parameter RETURN_20c = 11;  
    parameter RETURN_15c = 12;  
    parameter RETURN_10c = 13;  
    parameter RETURN_5c = 14;
```

State register defined with sequential always block

```
    always @ (posedge clk or negedge reset)  
        if (!reset)    state <= IDLE;  
        else           state <= next;
```




The Verilog code

Next-state logic within a combinational **always** block

```
always @ (state or N or D or Q) begin

    case (state)
        IDLE:      if (Q) next = GOT_25c;
                   else if (D) next = GOT_10c;
                   else if (N) next = GOT_5c;
                   else next = IDLE;

        GOT_5c:    if (Q) next = GOT_30c;
                   else if (D) next = GOT_15c;
                   else if (N) next = GOT_10c;
                   else next = GOT_5c;

        GOT_10c:   if (Q) next = GOT_35c;
                   else if (D) next = GOT_20c;
                   else if (N) next = GOT_15c;
                   else next = GOT_10c;

        GOT_15c:   if (Q) next = GOT_40c;
                   else if (D) next = GOT_25c;
                   else if (N) next = GOT_20c;
                   else next = GOT_15c;

        GOT_20c:   if (Q) next = GOT_45c;
                   else if (D) next = GOT_30c;
                   else if (N) next = GOT_25c;
                   else next = GOT_20c;
```

```
GOT_25c:  if (Q) next = GOT_50c;
           else if (D) next = GOT_35c;
           else if (N) next = GOT_30c;
           else next = GOT_25c;
```

```
GOT_30c:  next = IDLE;
GOT_35c:  next = RETURN_5c;
GOT_40c:  next = RETURN_10c;
GOT_45c:  next = RETURN_15c;
GOT_50c:  next = RETURN_20c;
```

```
RETURN_20c: next = RETURN_10c;
RETURN_15c: next = RETURN_5c;
RETURN_10c: next = IDLE;
RETURN_5c:  next = IDLE;
```

```
    default: next = IDLE;
endcase
```

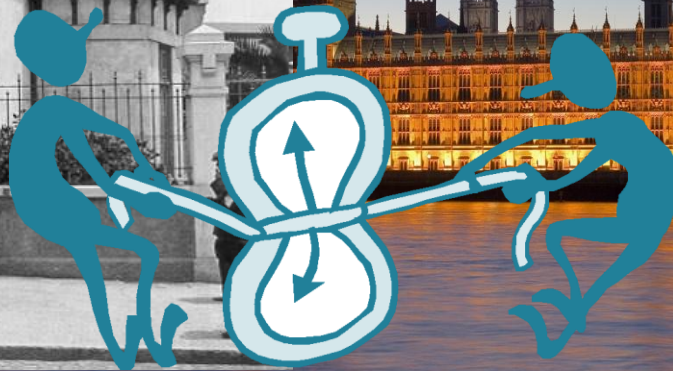
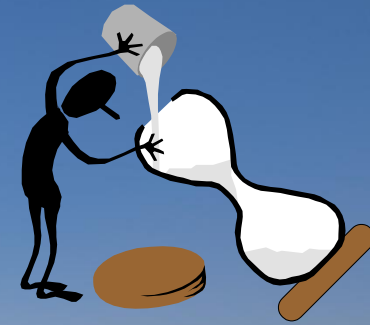
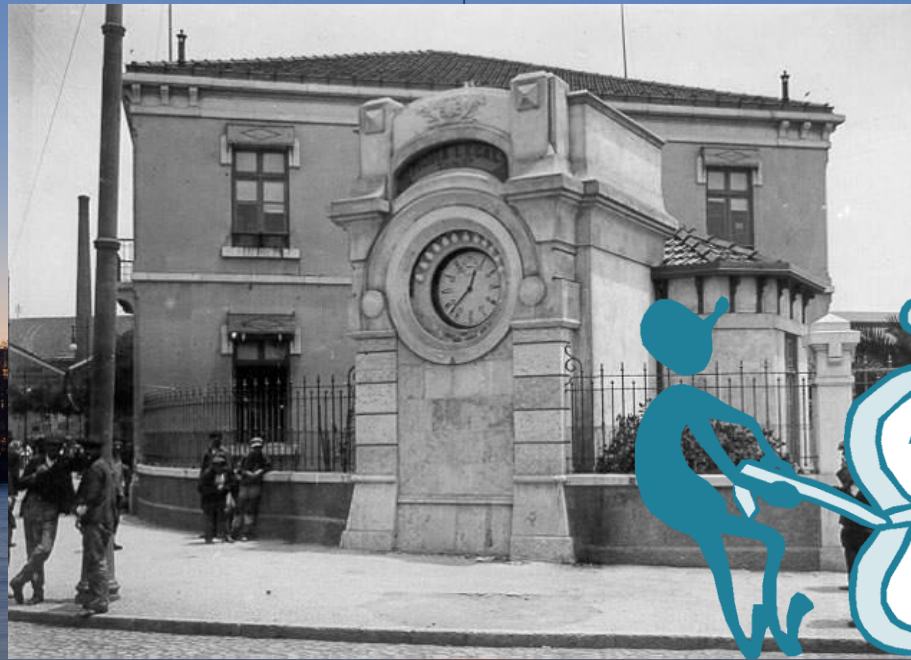
```
end
```

Combinational output assignment

```
assign DC = (state == GOT_30c || state == GOT_35c ||
             state == GOT_40c || state == GOT_45c ||
             state == GOT_50c);
assign DN = (state == RETURN_5c);
assign DD = (state == RETURN_20c || state == RETURN_15c ||
             state == RETURN_10c);
```

```
endmodule
```

Clocking

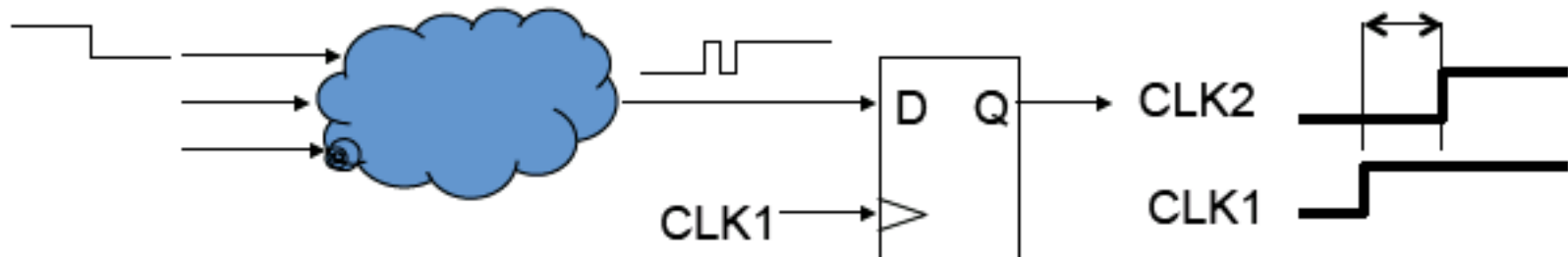


Where should CLK come from?

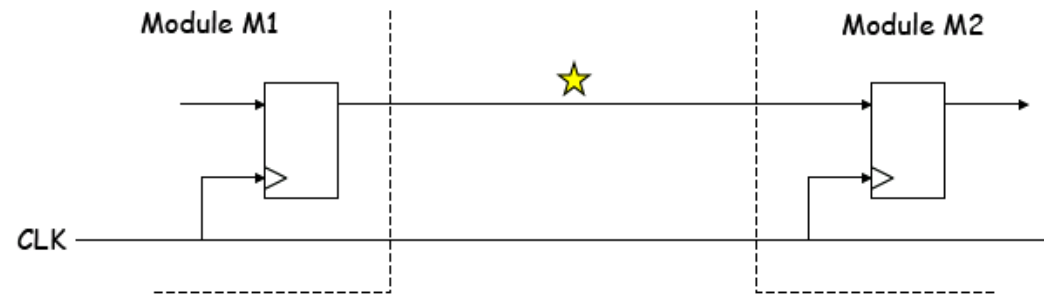
- Option 1: external crystal
 - Stable, known frequency, typically 50% duty cycle
- Option 2: internal signals
 - Option 2A: output of combinational logic



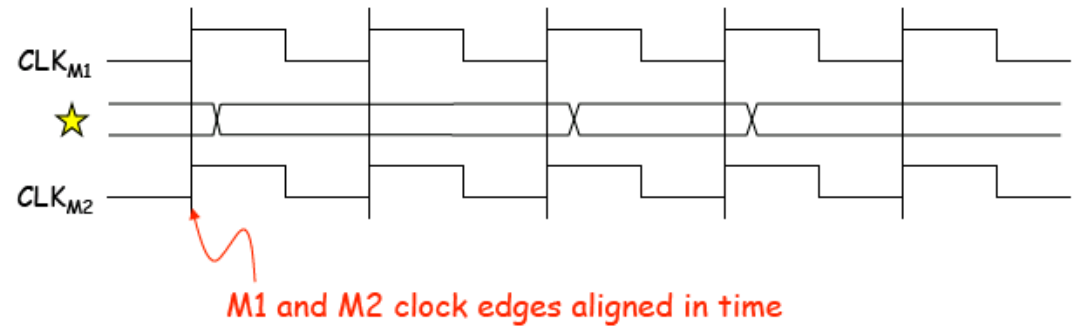
- No! If inputs to logic change, output may make several transitions before settling to final value → several rising edges, not just one! Hard to design away output glitches...
- Option 2B: output of a register
 - Okay, but timing of CLK2 won't line up with CLK1



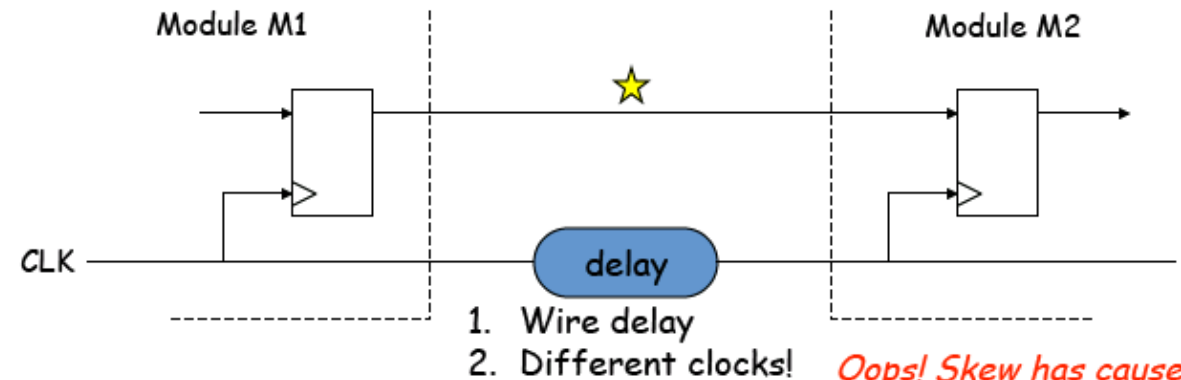
Clocking in and ideal world



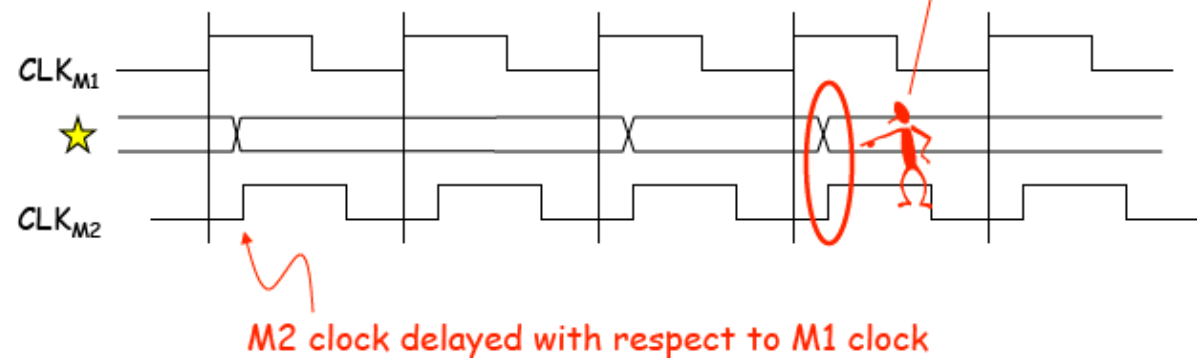
Ideal world:



Clocking in a not so perfect reality



Real world has clock skew:



FPGAs contain global
clock lines that have
low skew. Number is
limited

Clocks with periods multiple of two

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

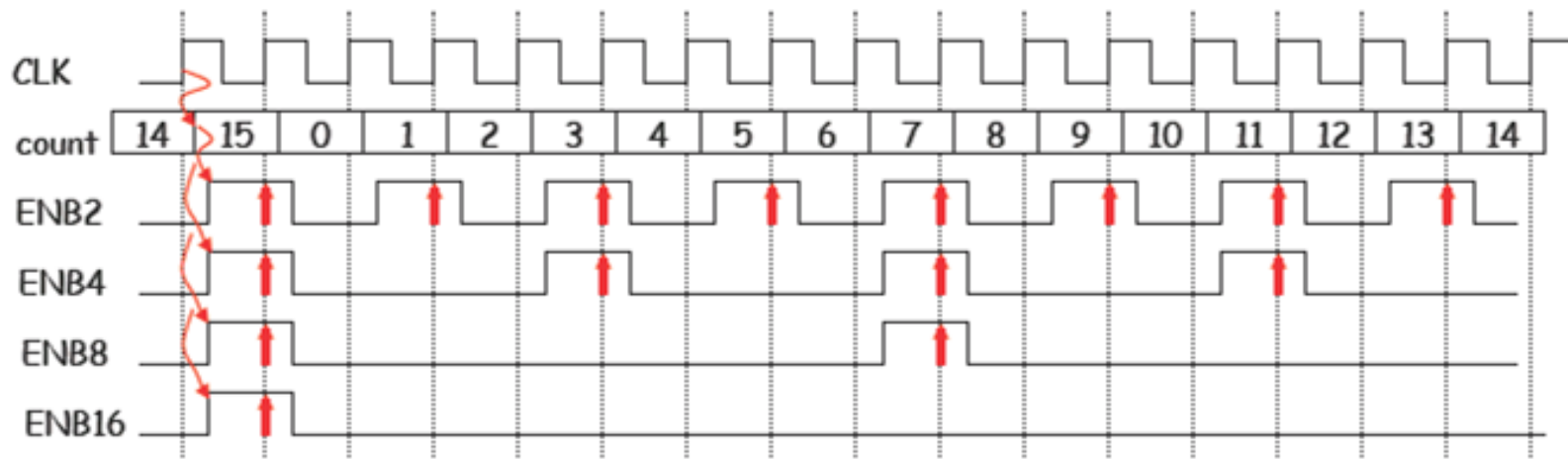


Solution: 1 clock, many enables

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

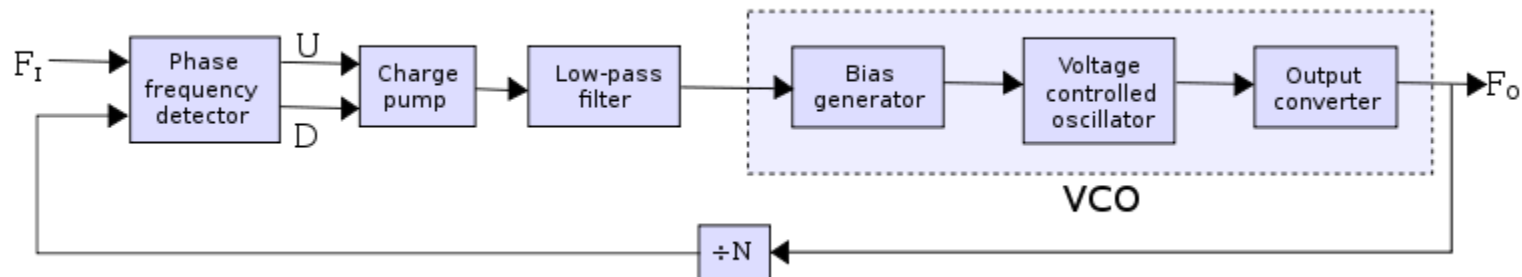
```
reg [3:0] count;  
always @(posedge clk) count <= count + 1; // counts 0..15  
wire enb2 = (count[0] == 1'b1);  
wire enb4 = (count[1:0] == 2'b11);  
wire enb8 = (count[2:0] == 3'b111);  
wire enb16 = (count[3:0] == 4'b1111);
```

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



↑ = clock edge selected by enable signal

The Phase Locked Loop (PLL)



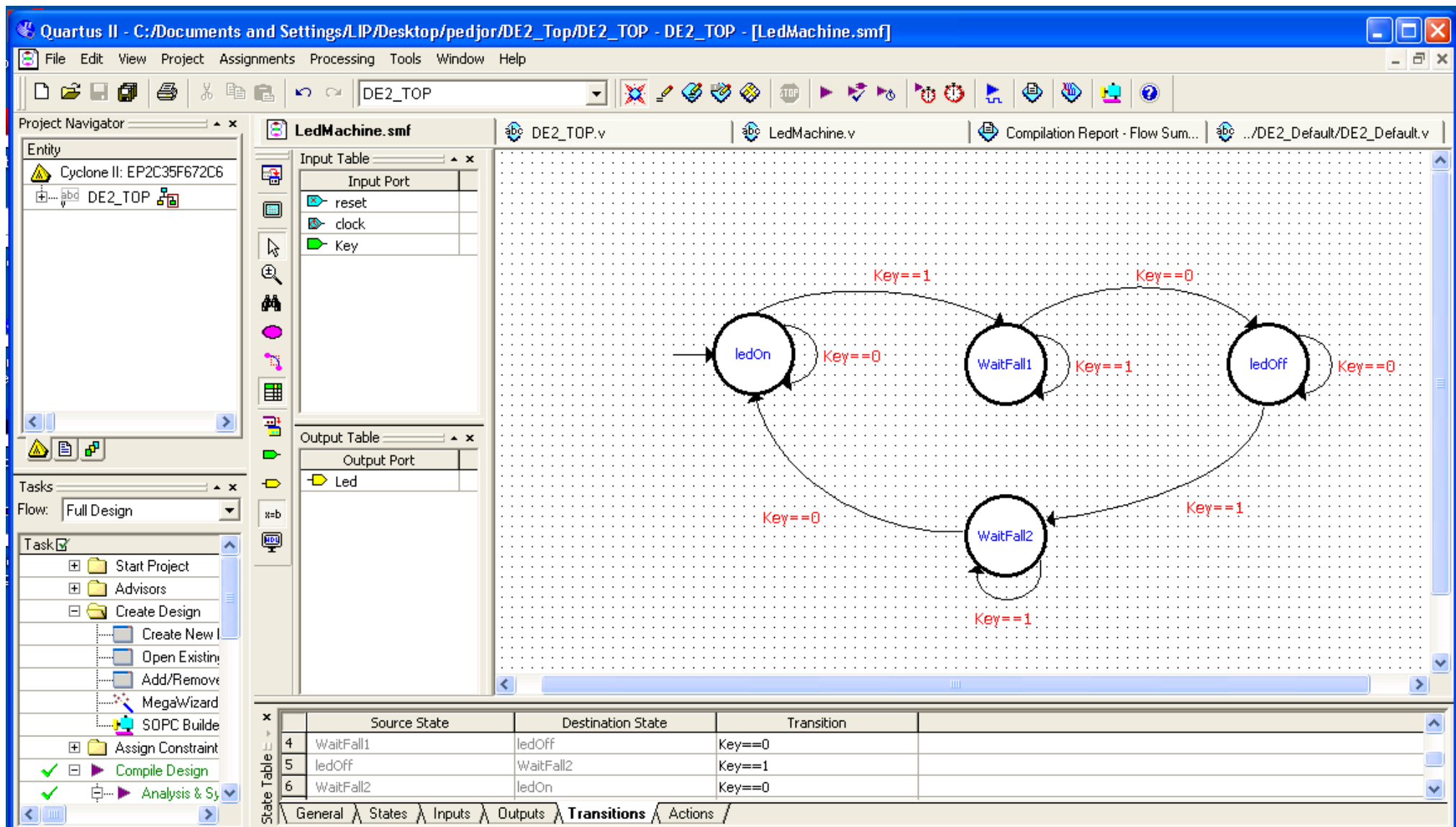
A feature available in Cyclone II devices that employs a phase-locked loop (PLL). The Cyclone II PLL provides advanced multiplication, programmable duty cycle, phase shifting, programmable bandwidth, manual clock switchover, clock outputs driving all networks, and a source synchronous mode.

You can take advantage of the Cyclone II PLL with the [altpll megafunction](#).

Phase-Locked Loops (ALTPLL)

Megafunction User Guide

File → new → statemachine



For more information about creating and editing state machine diagrams, refer to the Quartus II Help.

Define the states output

The screenshot shows the Quartus II IDE with the following components:

- Project Navigator:** Shows the project hierarchy with 'DE2_TOP' selected.
- Input Table:** Lists input ports: 'reset', 'clock', and 'Key'.
- Output Table:** Lists output ports: 'Led'.
- State Properties Dialog:** A modal window with tabs for 'General', 'Incoming Transitions', 'Outgoing Transitions', 'Actions', and 'Format'. The 'General' tab is active, showing a table with two rows: 'Led' and '<< new >>'. A blue circle highlights the 'Led' row.
- State Table:** A table at the bottom of the IDE showing state transitions. It has columns for 'Source State', 'Destination State', and 'Transition'. The table contains three rows: 'ledOn' to 'ledOn' (Key==0), 'WaitFall1' to 'WaitFall1' (Key==1), and 'ledOn' to 'WaitFall1' (Key==1). A red circle highlights the 'ledOn' state in the first row.
- Message Window:** Displays the message 'Info: Device 1 contains JTAG ID code 0x020B40DD'.

The State Table data is as follows:

Source State	Destination State	Transition
ledOn	ledOn	Key==0
WaitFall1	WaitFall1	Key==1
ledOn	WaitFall1	Key==1

Quartus II - C:/Documents and Settings/LIP/Desktop/pedjor/DE2_Top/DE2_TOP - DE2_TOP - [LedMachine.smf]

File Edit View Project Assignments Processing Tools Window Help

DE2_TOP

Project Navigator

Entity

- Cyclone II: EP2C35F672C6
- DE2_TOP

Tasks

Flow: Full Design

Task

- Start Project
- Advisors
- Create Design
 - Create New I
 - Open Existing
 - Add/Remove
 - MegaWizard
 - SOPC Builder
- Assign Constraint
- Compile Design
- Analysis & Syn

LedMachine.smf

Input Table

Input Port
reset
clock
Key

Output Table

Output Port
Led

DE2_TOP.v

LedMachine.v

Compilation Report - Flow Sum...

.../DE2_Default/DE2_Default.v

State Table

	Source State	Destination State	Equation
1	ledOn	ledOn	Key==0
2	WaitFall1	WaitFall1	Key==1
3	ledOn	WaitFall1	Key==1

General States Inputs Outputs Transitions Actions

Transition Properties

Source state: ledOn

Destination state: WaitFall1

Equation: Key==1

☐ Use VHDL 'OTHERS' transition.

Note: The VHDL 'OTHERS' transition is used when none of the outgoing transitions from the state are true.

OK Cancel

Messages

Type Message

- Info: Device 1 contains JTAG ID code 0x020B40DD
- Info: Configuration succeeded -- 1 device(s) configured
- Info: Successfully performed operation(s)
- Info: Ended Programmer operation at Tue Mar 17 14:16:59 2009

System (20) Processing (195) Extra Info Info (51) Warning (144) Critical Warning Error Sup

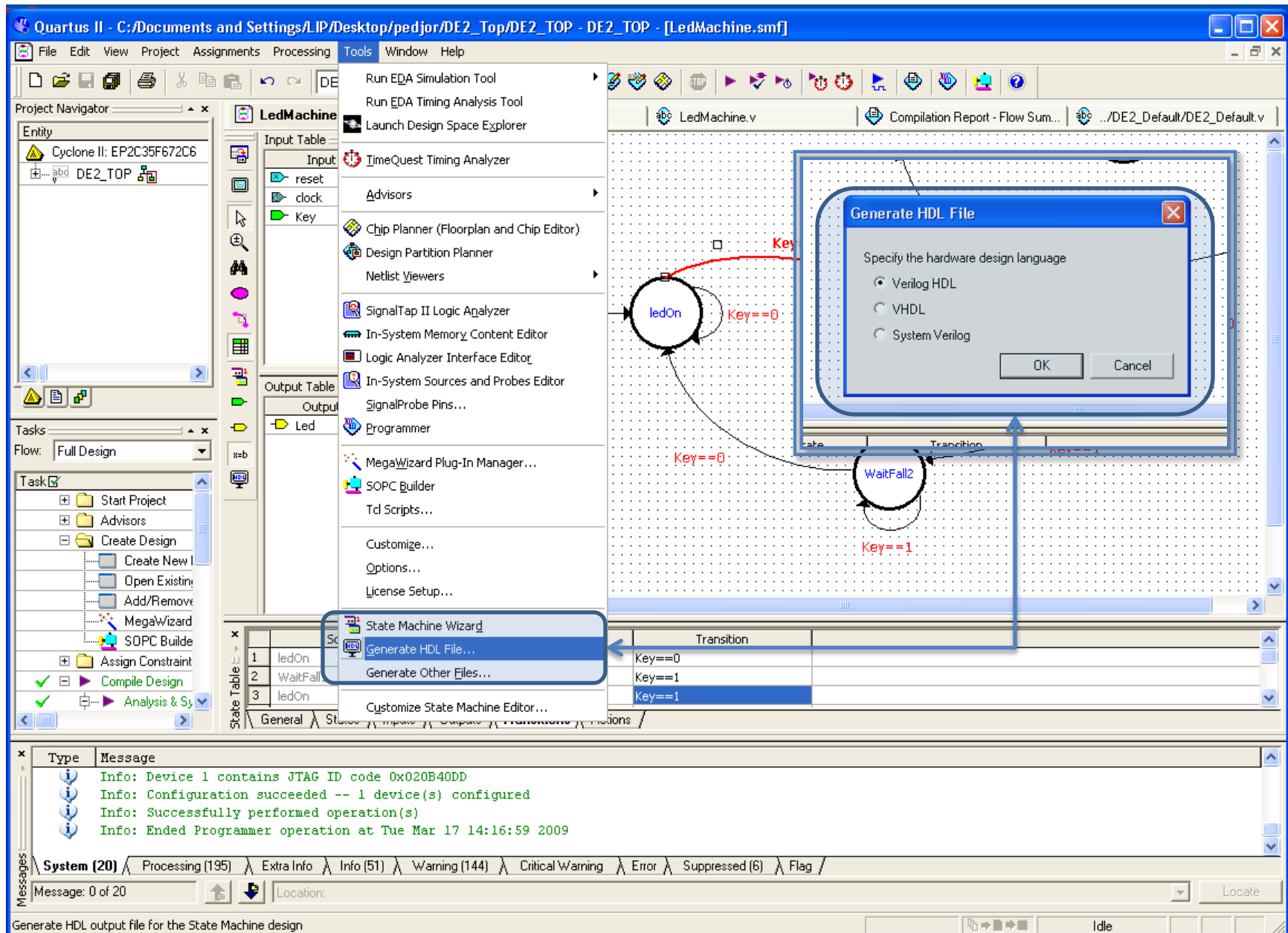
Message: 0 of 20

Location:

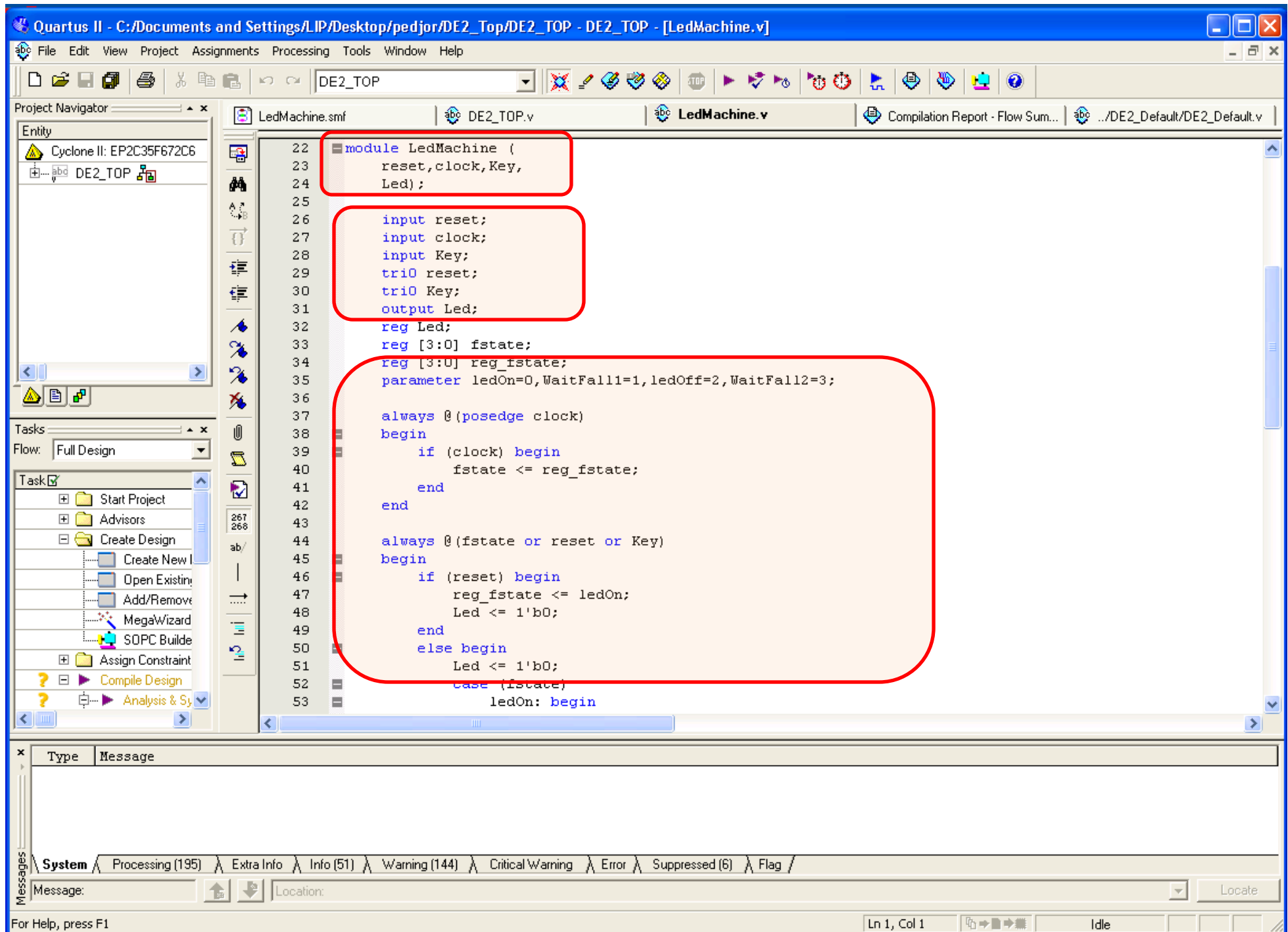
For Help, press F1

298, 62

Idle



The “LedMachine” module



The top design

Quartus II - C:/Documents and Settings/LIP/Desktop/pedjor/DE2_Top/DE2_TOP - DE2_TOP - [DE2_TOP.v]

File Edit View Project Assignments Processing Tools Window Help

DE2_TOP

Project Navigator

Entity

Cyclone II: EP2C35F672C6

DE2_TOP

Tasks

Flow: Full Design

Task

Start Project

Advisors

Create Design

Create New I

Open Existing

Add/Remove

MegaWizard

SDPC Build

Assign Constraint

Compile Design

Analysis & Sy

LedMachine.smf

DE2_TOP.v

LedMachine.v

Compilation Report - Flow Sum...

../DE2_Default/DE2_Default.v

```
310 assign LEDR[17:3] = 15'h3FFFF;
311 assign LCD_ON = 1'b1;
312 assign LCD_BLON = 1'b1;
313
314 // All inout port turn to tri-state
315 assign DRAM_DQ = 16'hzzzz;
316 assign FL_DQ = 8'hzz;
317 assign SRAM_DQ = 16'hzzzz;
318 assign OTG_DATA = 16'hzzzz;
319 assign LCD_DATA = 8'hzz;
320 assign SD_DAT = 1'bz;
321 assign I2C_SDAT = 1'bz;
322 assign ENET_DATA = 16'hzzzz;
323 assign AUD_ADCLRCK = 1'bz;
324 assign AUD_DACLCK = 1'bz;
325 assign AUD_BCLK = 1'bz;
326 assign GPIO_0 = 36'hzzzzzzzz;
327 assign GPIO_1 = 36'hzzzzzzzz;
328
329 //LedMachine Maq(KEY[0],CLOCK_50,KEY[1],LEDG[0]);
330
331 assign LEDR[0]=KEY[0];
332 assign LEDR[1]=KEY[1];
333 assign LEDR[2]=KEY[2];
334
335 LedMachine u1 (
336     .reset(!KEY[0]),
337     .clock(!KEY[2]),
338     .Key(!KEY[1]),
339     .Led(LEDG[0]) );
340
341 endmodule
```

Why is the clock the KEY[2]?

Ln 333, Col 16

Idle

The 2nd lab

The Alarm

- Activate the alarm;
- Deactivate;
- Waiting times.
- Code input and verification

Remember to split the program in blocks!!
(you introduce the code to activate and deactivate in the same way)



Multiplication in Verilog

You can use the "*" operator to multiply two numbers:

```
wire [9:0] a,b;  
wire [19:0] result = a*b;    // unsigned multiplication!
```

If you want Verilog to treat your operands as signed two's complement numbers, add the keyword `signed` to your `wire` or `reg` declaration:

```
wire signed [9:0] a,b;  
wire signed [19:0] result = a*b;    // signed multiplication!
```

Remember: unlike addition and subtraction, you need different circuitry if your multiplication operands are signed vs. unsigned. Same is true of the >>> (arithmetic right shift) operator. To get signed operations all operands must be signed.

To make a signed constant: 10'sh37C