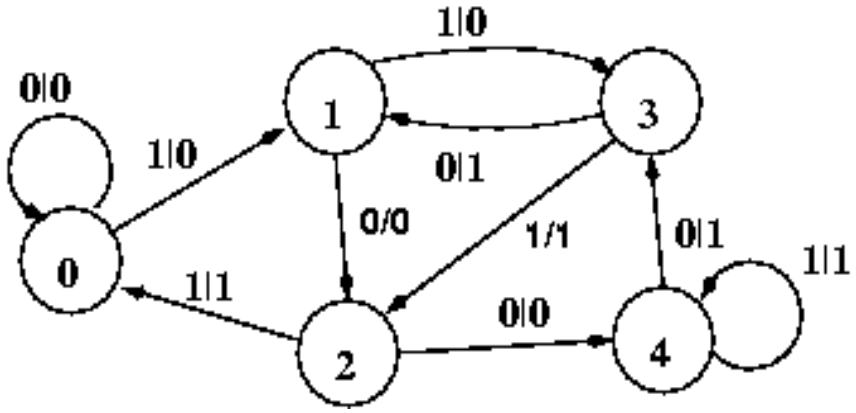
6.111 Lecture # 7

Take another look at that divide by five FSM.



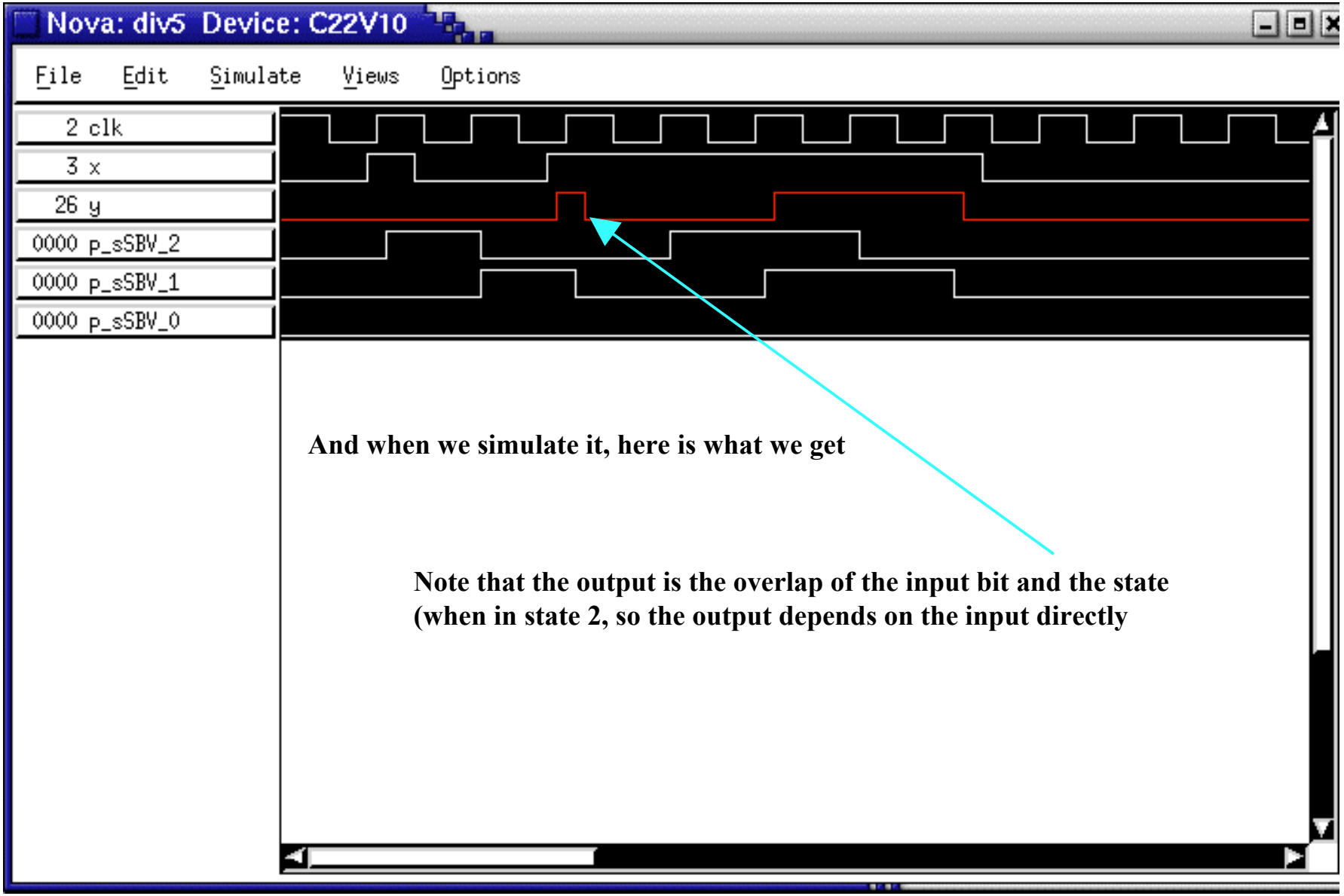
Progression Through States

x	p_s	n_s	y
1	0	1	0
0	1	2	0
1	2	0	1
1	0	1	0
1	1	3	0
1	3	2	1
1	2	0	1

Here is, roughly what we would expect from the Mealey machine model of that thing with inputs and outputs as specified

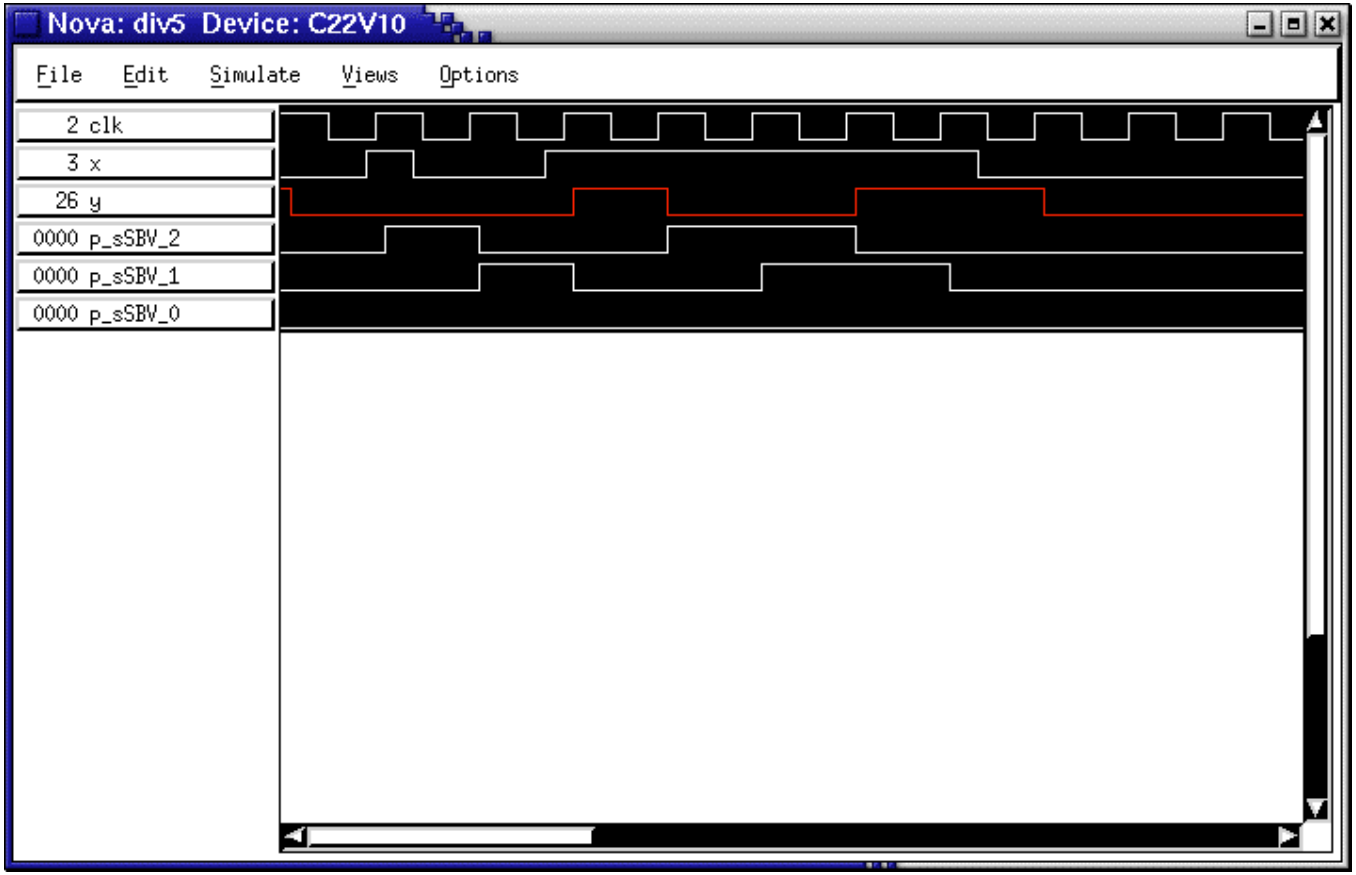
Here is the VHDL code from last time, crowded onto a single sheet:

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity divby5 is port
                (
                 x, clk : in std_logic;
                 y      : out std_logic);
end divby5;
architecture state_machine if divby5 is
        type StateType is (state0, state1, state2, state3,
state4);
        signal p_s, n_s : StateType;
begin
        fsm: process (p_s, x)
        begin
                case p_s is
                        when state0 => y <= '0';
                                if x = '1' then
                                        n_s <= state1;
                                else
                                        n_s <= state0;
                                end if;
                        when state1 => y <= '0';
                                if x = '1' then
                                        n_s <= state3;
                                else
                                        n_s <= state2;
                                end if;
                        when state2 =>
                                if x = '1' then
                                        n_s <= state0;
                                        y <= '1';
                                else
                                n_s <= state4;
                                        y <= '0';
                                end if;
                        when state3 => y <= '1';
                                if x = '1' then
                                        n_s <= state2;
                                else
                                        n_s <= state1;
                                end if;
                        when state4 => y <= '1';
                                if x = '1' then
                                        n_s <= state4;
                                else
                                        n_s <= state3;
                                end if;
                        when others => n_s <= state0;
-- avoid trap states
                end case
        end process fsm;
        state-clocked        : process (clk)
                begin
                        if rising_edge(clk) then
                                p_s <= n_s;
                        end if;
                end process state_clocked;
end architecture state_machine;
```


Nova: div5 Device: C22V10
File
Edit
Simulate
Views
Options
2 clk
3 x
26 y
0000 p_sSBV_2
0000 p_sSBV_1
0000 p_sSBV_0
And when we simulate it, here is what we get
Note that the output is the overlap of the input bit and the state
(when in state 2, so the output depends on the input directly

Modification to register the output:

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity divby5 is port
                (
                 x, clk  : in std_logic;
                 y       : out std_logic);
end divby5;

architecture state_machine of divby5 is
        type StateType is (state0, state1, state2, state3,
state4);
        signal p_s, n_s : StateType;
      signal ans : std_logic := '0'; -- new

begin
        fsm: process (p_s, x)
        begin
                case p_s is
                        when state0 => ans <= '0';
                                if x = '1' then
                                        n_s <= state1;
                                else
                                        n_s <= state0;
                                end if;
                        when state1 => ans <= '0';
                                if x = '1' then
                                        n_s <= state3;
                                else
                                        n_s <= state2;
                                end if;
                        when state2 =>
                                if x = '1' then
                                        n_s <= state0;
                                        ans <= '1';
                                else
                                        n_s <= state4;
                                        ans <= '0';
                                end if;
                        when state3 => ans <= '1';
                                if x = '1' then
                                        n_s <= state2;
                                else
                                        n_s <= state1;
                                end if;
                        when state4 => ans <= '1';
                                if x = '1' then
                                        n_s <= state4;
                                else
                                        n_s <= state3;
                                end if;
                        when others => n_s <= state0; -- avoid
trap states
                end case;
        end process fsm;
        state_clocked           : process (clk)
                begin
                        if rising_edge(clk) then
                                p_s <= n_s;
                                y <= ans;       -- register output
                        end if;
                end process state_clocked;
end architecture state_machine;
```



On simulation, we note that:

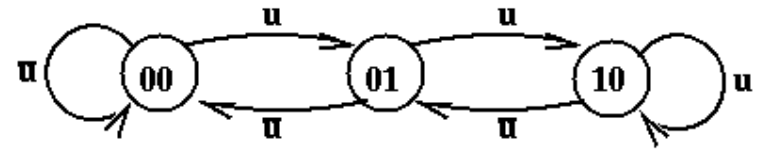
1. **Each of the output bits is one clock cycle long**
2. **But the output is delayed one clock cycle**

Topics for today

Certain issues in timing and handling pulse like signals

Lab 2

Consider a simple finite state machine:



u \ Q1 Q0	00	01	11	10
0	0	0	X	0
1	0	1	X	1

D1 = u*Q0 + u*Q1

u \ Q1 Q0	00	01	11	10
0	0	0	X	1
1	1	0	X	0

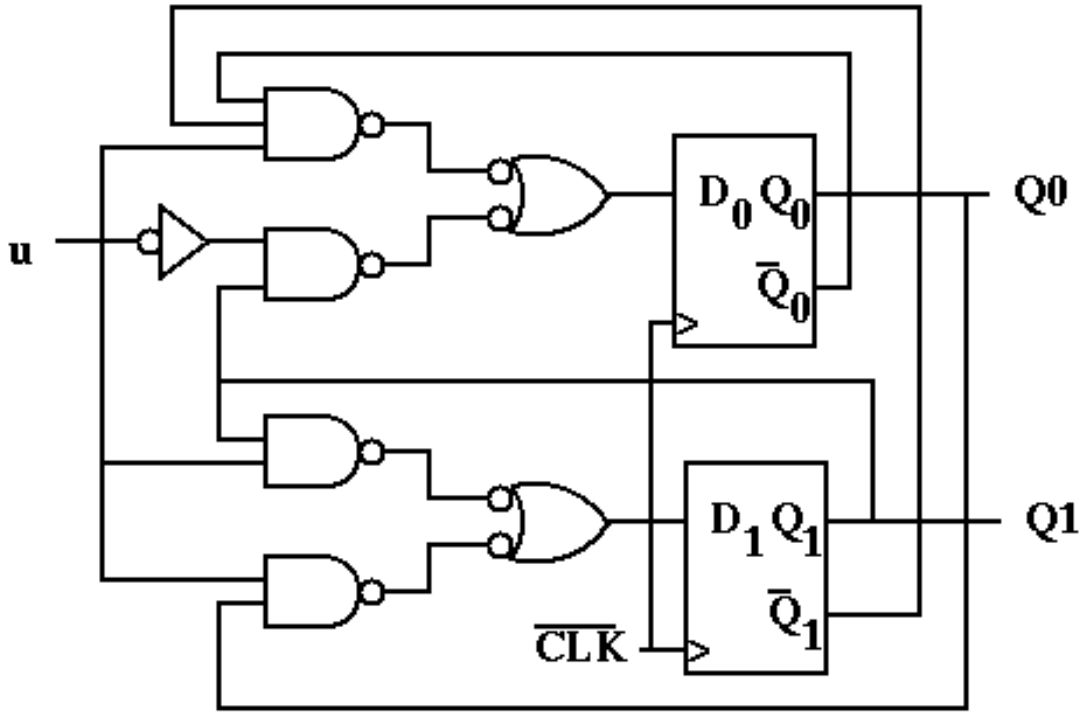
D0 = u*/Q0*/Q1 + /u*Q1

This is a very small up/down counter

The logic is straightforward to design

Note it has two flip flops but does not use all four states

Here is a simple implementation, easily implemented in TTL



But look at a possible timing issue:

IF we are in state 10

IF u = 1, we stay in state 10

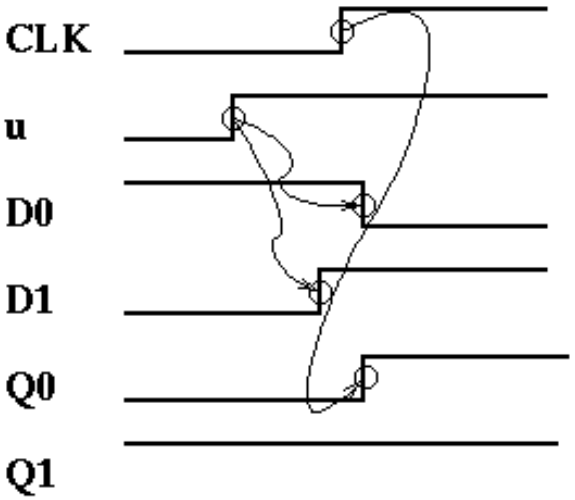
IF u=0, we go to state 01

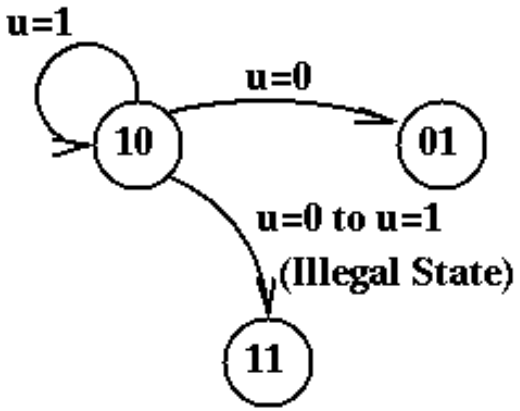
IF u=0 and then makes a transition to 1, we still want to stay in state 10

BUT if u=0 and then makes a transition to 1 too close to the clock edge,

The transition of D0 from 1 to 0 is delayed with respect to D1 (by one gate delay)

And it this happens the thing goes into a state it isn't supposed to

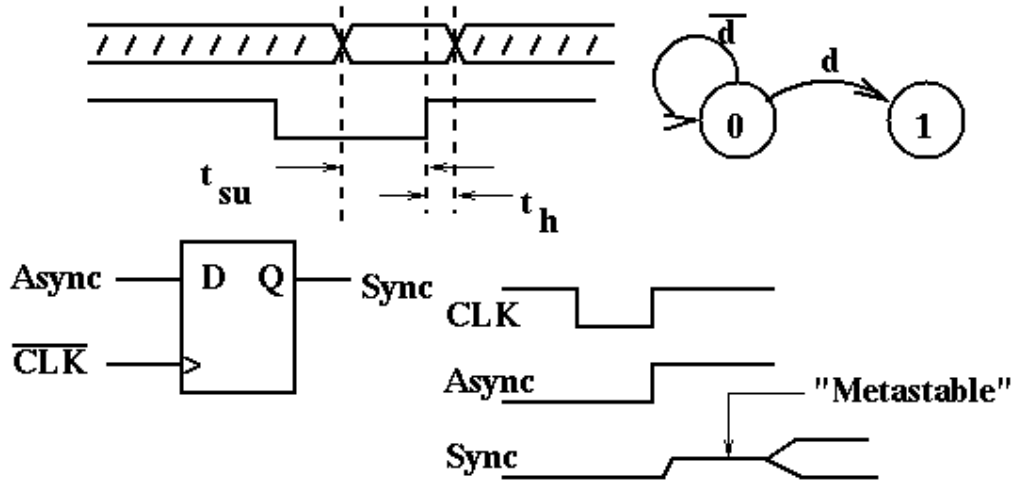




Design Rule:

1. **Synchronize ALL external signals**
2. **Any asynchrous input must affect ONLY ONE flip-flop (which is switched synchronously with all of the other flip-flops)**

"Metastable" problem: if input is too close to an edge, the next state is not well determined.

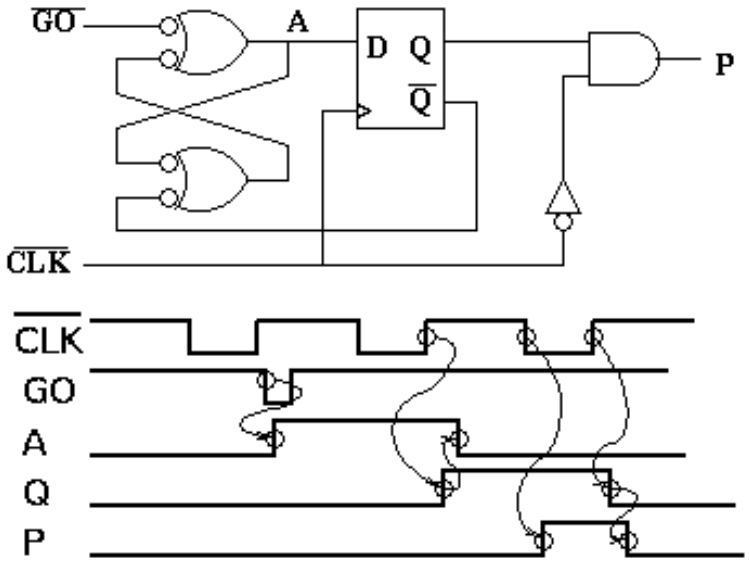


But if inputs are synchronized according to the rule, the (single) FF will make the transition on the NEXT clock edge

Timing techniques

One problem is to catch a signal that may be shorter than your clock cycle.

Catch an edge or a short pulse



Note that here we have one use for the S-R latch.

This does well at catching a very short pulse, but if /GO is low for several clock cycles, P will have several pulses.

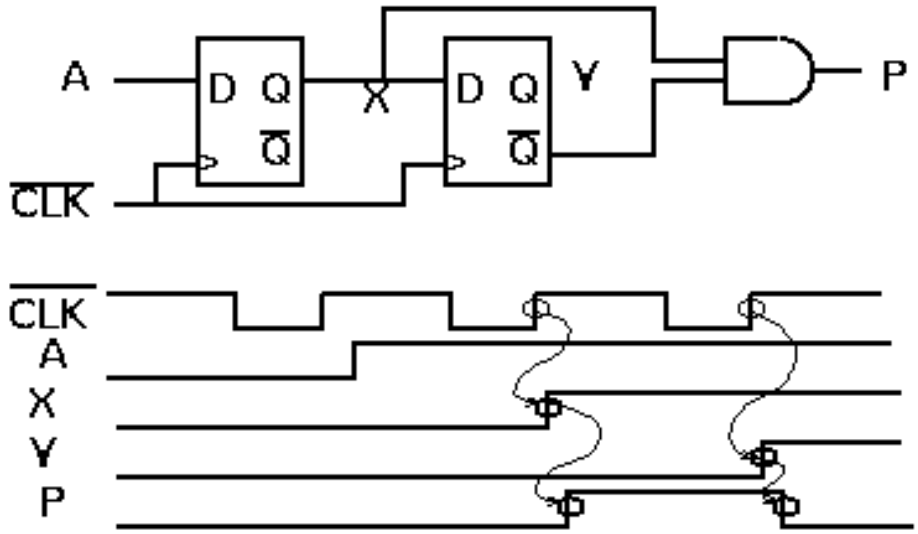
You might want to think about how to design a circuit which takes a /GO signal of arbitrary length and produces a SINGLE pulse in response.

Here is a candidate for that

It generates a single pulse (one clock cycle wide)

But note that A has to be asserted on a positive going clock edge

Or turn a level into a finite-width pulse:



VHDL Code for short pulse catcher

```
library ieee;
use ieee.std_logic_1164.all;
entity spulse is
   port(N_GO, CLK, CLKIN: in std_logic;
        P: out std_logic);
end spulse;

-- purpose: catch a short pulse
architecture behavioral of spulse is
   signal A, N_A, X, N_X, N_CLK: std_logic;
   attribute synthesis_off of A: signal is true;
   attribute synthesis_off of N_A: signal is true;
begin  -- behavioral
   A <= (not N_GO) or (not N_A);
   N_A <= (not A) or (not N_X);
   N_X <= (not X);
   P <= X and N_CLK;
   N_CLK <= (not CLKIN);
ff: process(CLK)
   begin
      if rising_edge(CLK) then
         X <= A;
      end if;
   end process ff;
end behavioral;
```

Note the rather odd looking syntax here: using the attribute synthesis_off tells the compiler to not optimize away the latch

This combinatoric part of the code describes the SR latch and the output

The process describes the d- flip-flop

VHDL Code for pulse shaper

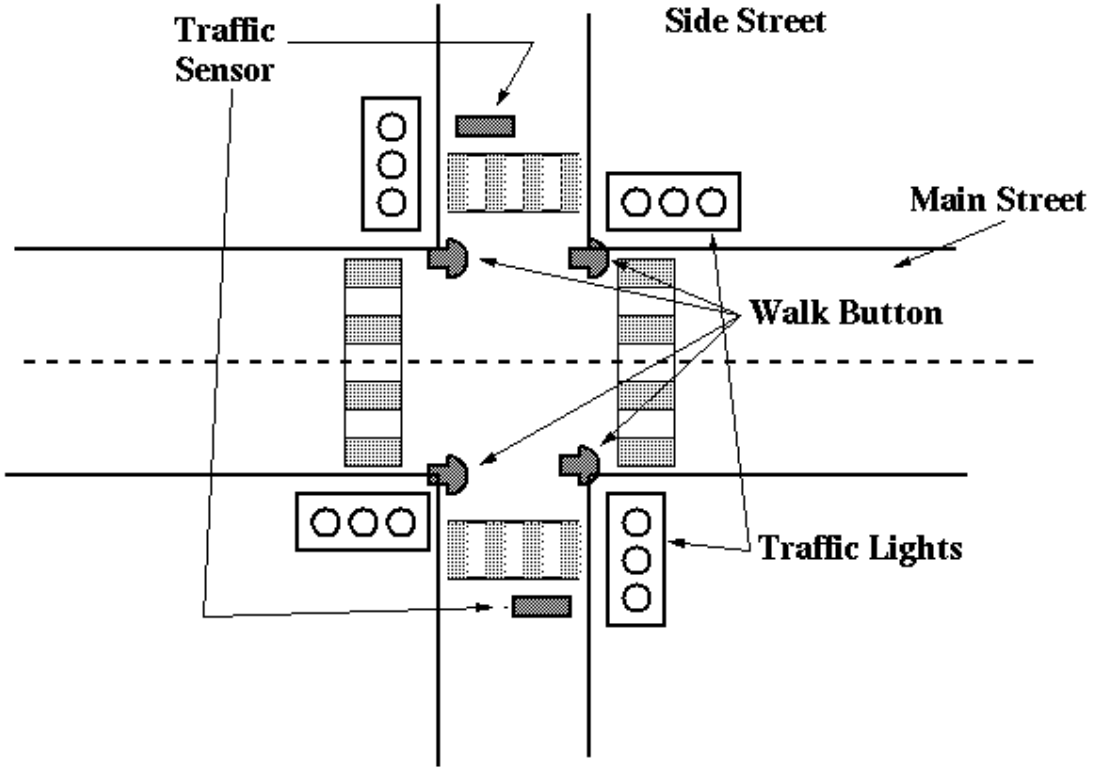
```
library ieee;
use ieee.std_logic_1164.all;
entity pform is
   port(A, CLK, CLKIN: in std_logic;
        P: out std_logic);
end pform;

-- purpose: catch a short pulse
architecture behavioral of pform is
   signal X, N_Y: std_logic;
begin  -- behavioral
ff: process(CLK)
   begin
      if rising_edge(CLK) then
         X <= A;
         N_Y <= (NOT X);
      end if;
   end process ff;
      P <= (X AND N_Y);
end behavioral;
```

Lab 2 assignment is yet another traffic light

This time you control it

It looks like a familiar situation

Main and side streets, with a walk light on demand



Main street part of cycle is longer than side street (Tbase+Text)

But side street has a traffic sensor which keeps it green a bit longer. (Text)

Traffic sensor must be synchronized.

Walk button must be latched and serviced at the right time, and unlatched after it has been serviced

Details: walk is R-Y.
Blink is Main Y, Side R, ON/OFF, equal intervals (Tblink).

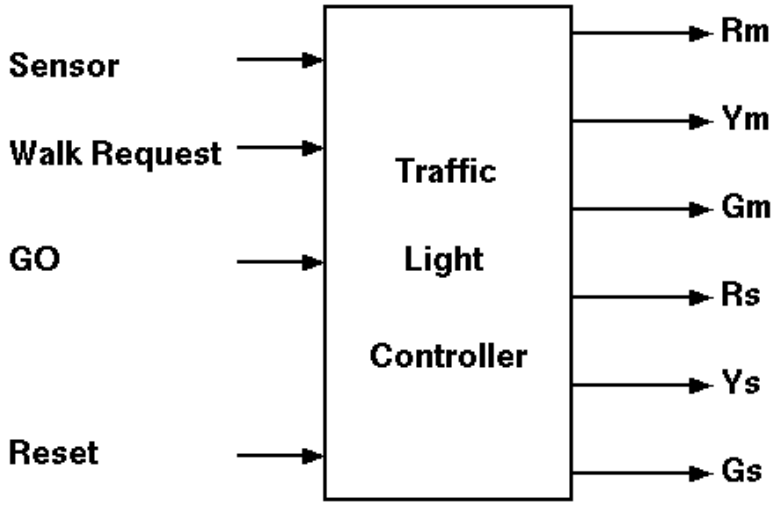
Design Procedure:

Start with a simple block diagram

Break design down into more, simpler blocks

Here is a top level block diagram for a controller

Inputs **Outputs (light signals)**

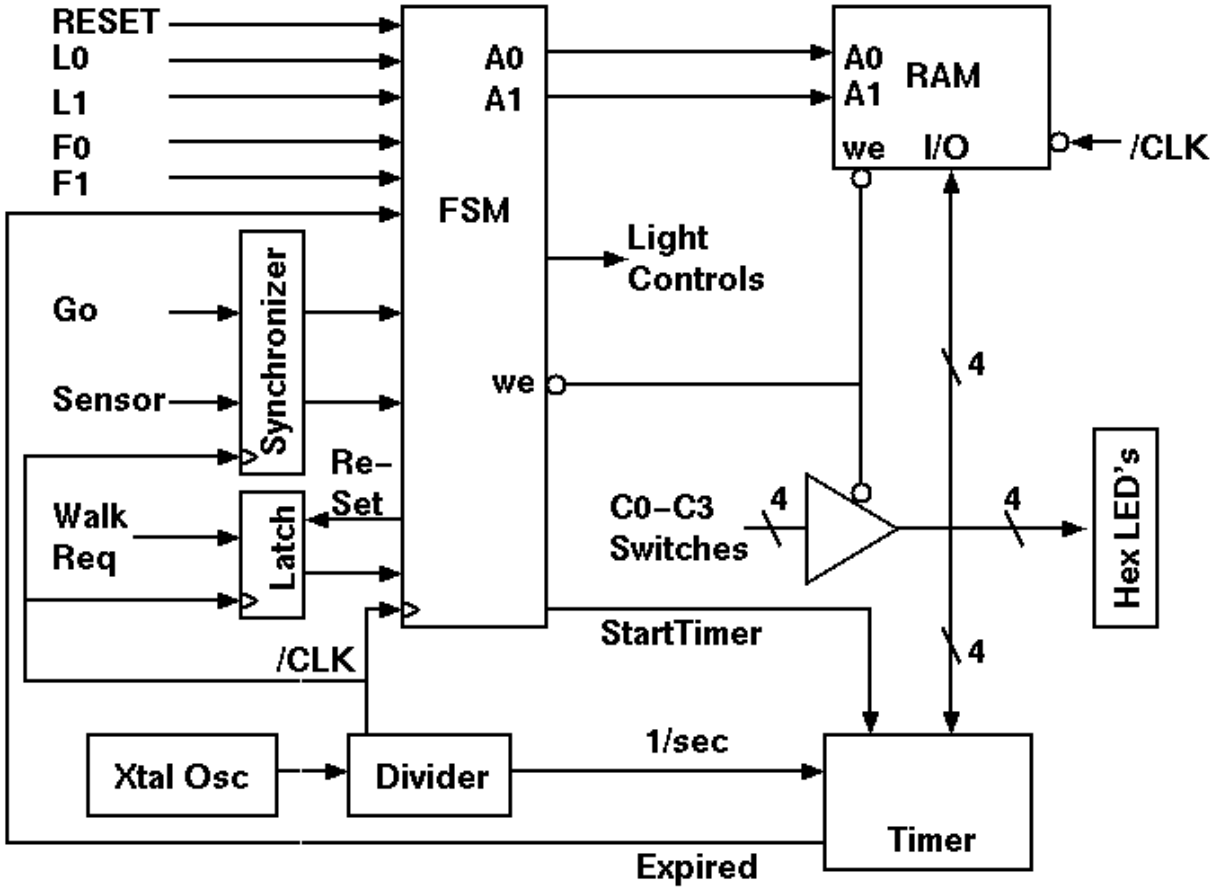


Note this GO signal is similar to what we were discussing earlier: a single pulse in response to a pushbutton (which could be any length)

This is a conceptual developed block diagram for the machine

The FSM should be implemented in a CPLD

We want you to use REAL RAM, do not include it in your CPLD



The Hex LED's are used to examine memory

It is your call if you want to implement the timer and divider in the CPLD (We expect you probably will want to do it this way

Inputs To Your FSM:

RESET	**(from a switch)**
GOSYNC	**(from Synchronizer)**
F1, F0	**Function Selection (from switches)**
L1, L0	**RAM Address**
Sensor	**Traffic Sensor (synchronized from a switch)**
WR	**Walk Request (Re-settable latch from pushbutton)**
EXPIRED	**Signal that timer has timed out**

Outputs From Your FSM

A1, A0	**SRAM Address**
WE	**SRAM Write Enable (source of bus signal)**
StartTimer	**Resets 1 second increment timer**
Gm, Ym, Rm,	
Gs, Ys, Rs	**Traffic light control signals**

Control Specifications

Here are the functions your controller must implement

F1	F0	are the function control switches
0	0	Examine Memory Location Specified by Switches
0	1	Store Value in Memory Location Specified by Switches
1	0	Run Traffic Lights
1	1	Blink

And for writing to or examining memory (functions 0 and 1) you should use these addresses:

A1	A0		
0	0	TYEL	Time for yellow light
0	1	TBASE	Base interval
1	0	TEXT	Extension interval
1	1	TBLINK	Blink Interval