PURDUE UNIVERSITY DEPARTMENT OF PHYSICS

Physics 53600

Electronics Techniques for Research

Now in PowerPoint!

Spring 2020 Semester

Prof. Matthew Jones

The usual ANNOUNCEMENT

- Obvious changes to the course:
  - No in-person lectures: you'll have to read the lecture notes yourself
  - No more labs: don't worry about it – your grade will be based on work done so far
  - Remaining assignments will try to cover topics that would have been explored in the lab
  - Second mid-term: simplest to cancel it
  - Final exam: I think it will be a 24 hour exam with written responses that can be easily sent by e-mail.
- Changes to grading scheme:
  - Old scheme: Assignments (30%) exams (40%) lab (30%)
  - New scheme: Assignments (50%) exams (25%) lab (25%)

The usual ANNOUNCEMENT

- Because there won't be any in-person lectures, you will have to read the lecture notes yourself.
- To demonstrate that you have read them, you will be required to answer *one or two simple questions* before the next lecture is posted.
- The question will probably be at the beginning and you just have to e-mail me the answer

mjones@physics.purdue.edu

- To make this easy, please make your subject look like this:

"PHYS53600 Lecture xx questions Your Name"

- These will be part of your assignment grade, maybe contributing 10% of your total grade.

More ANNOUNCEMENTS

- Feel free to send me questions about the lecture material if there is anything you don't understand. I'm happy to give more explanation (and I'm soooo bored.)
- Send me e-mail if you think it would be useful to arrange a time as a class to have a time where you can ask questions by video.
  - So far a couple of people have said it would be
  - Maybe something like Wednesday, April 30th at 10:30 am EDT?

LECTURE 25 QUESTIONS

1. *Dream up some other type of problem that can be described using a finite state machine. What are the states? What are the inputs?*

Digital Design Principles

- In the previous lecture we discussed programmable logic devices
- It is much more likely that you will implement complex logic using an FPGA rather than build it out of discrete integrated circuits
- But, how does one design a digital system that accomplishes some specific task?
- There are some well-defined design methodologies that can help…

Digital Design Principles

- So far we have discussed lots of digital logic elements out of which a digital system can be built
- What we need next is a model for thinking about problems that can be easily translated into digital logic
- A very important design methodology is the use of "finite state machines"

Finite State Machines

- Many (but not all) systems can exist in one of several discrete "states"
- An "event" will cause a transition to occur between the states
- This model can be very easily translated into digital logic
- Let's start with a simple example…

Traffic Light Problem

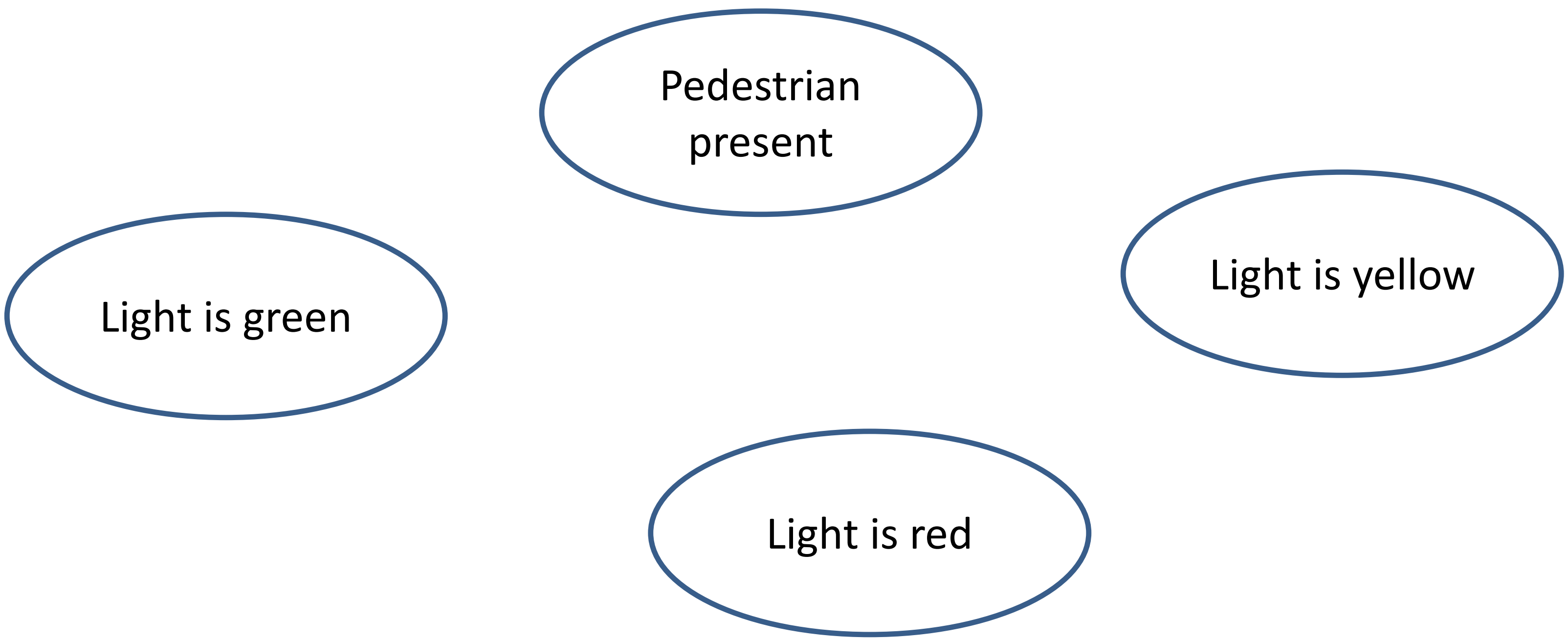
- Suppose we need to solve the problem of controlling a set of traffic lights.
- Start with one pedestrian controlled light, like the one on Northwestern Avenue
- There are obviously at least two states:
  - Traffic light is green (cars keep driving)
  - Traffic light is red (cars must stop)
- There are a couple other states that make it even better:
  - A pedestrian has pushed the “walk” button
  - Traffic light is yellow (prepare to stop/floor it!)

Traffic Light Problem

- The transitions between the states can happen on a fixed period of maybe a few seconds
- Whether there is a transition or not depends on external and internal information
- These are some possible events:
  - A pedestrian pushes the “walk” button
  - A timer has expired

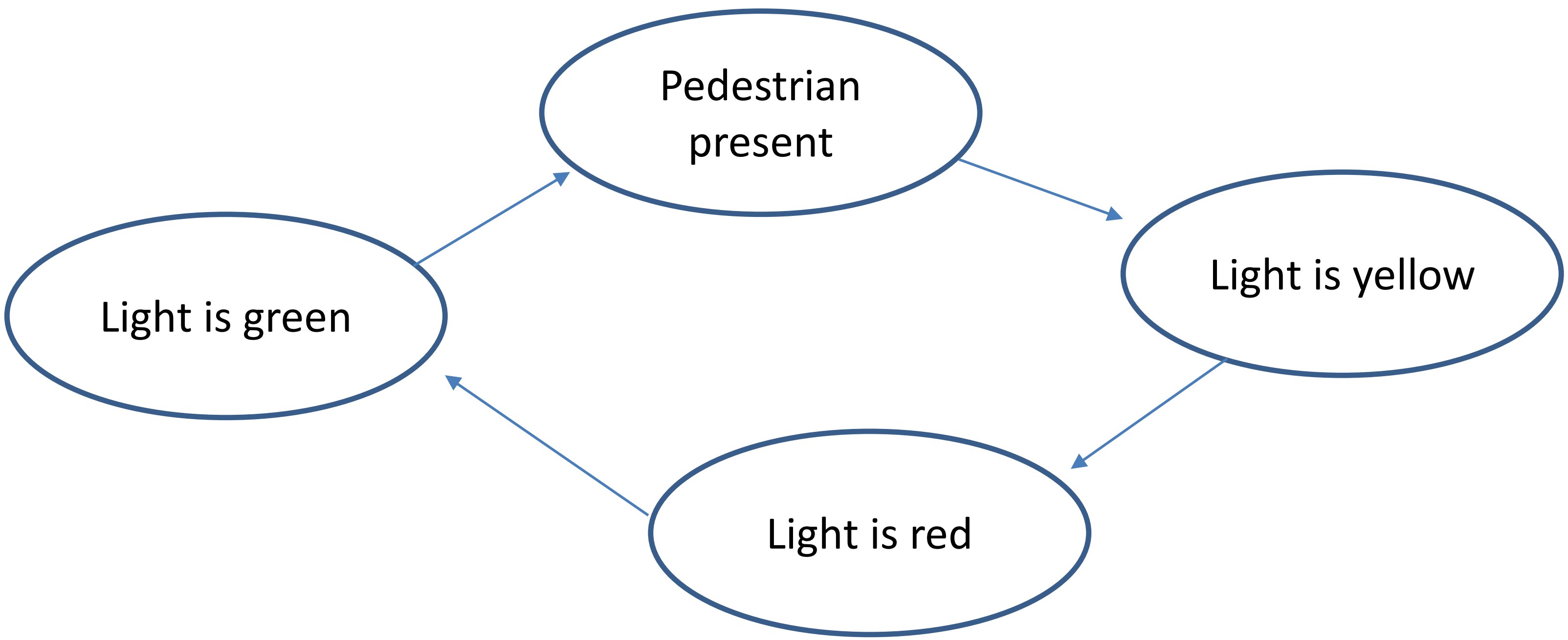
Traffic Light Problem

- We represent this problem as a graph where the nodes are the states:



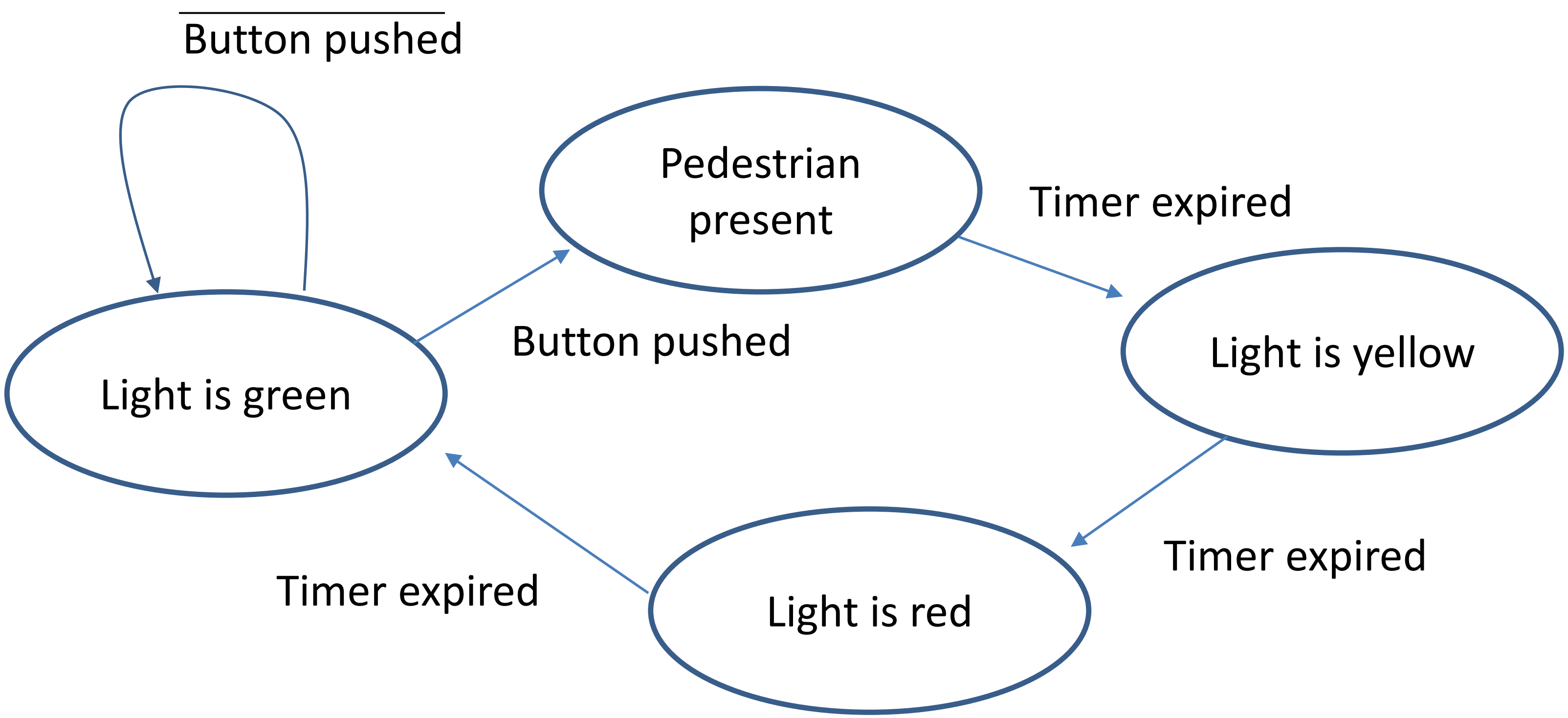
Traffic Light Problem

- The edges on the graph represent the allowed state transitions



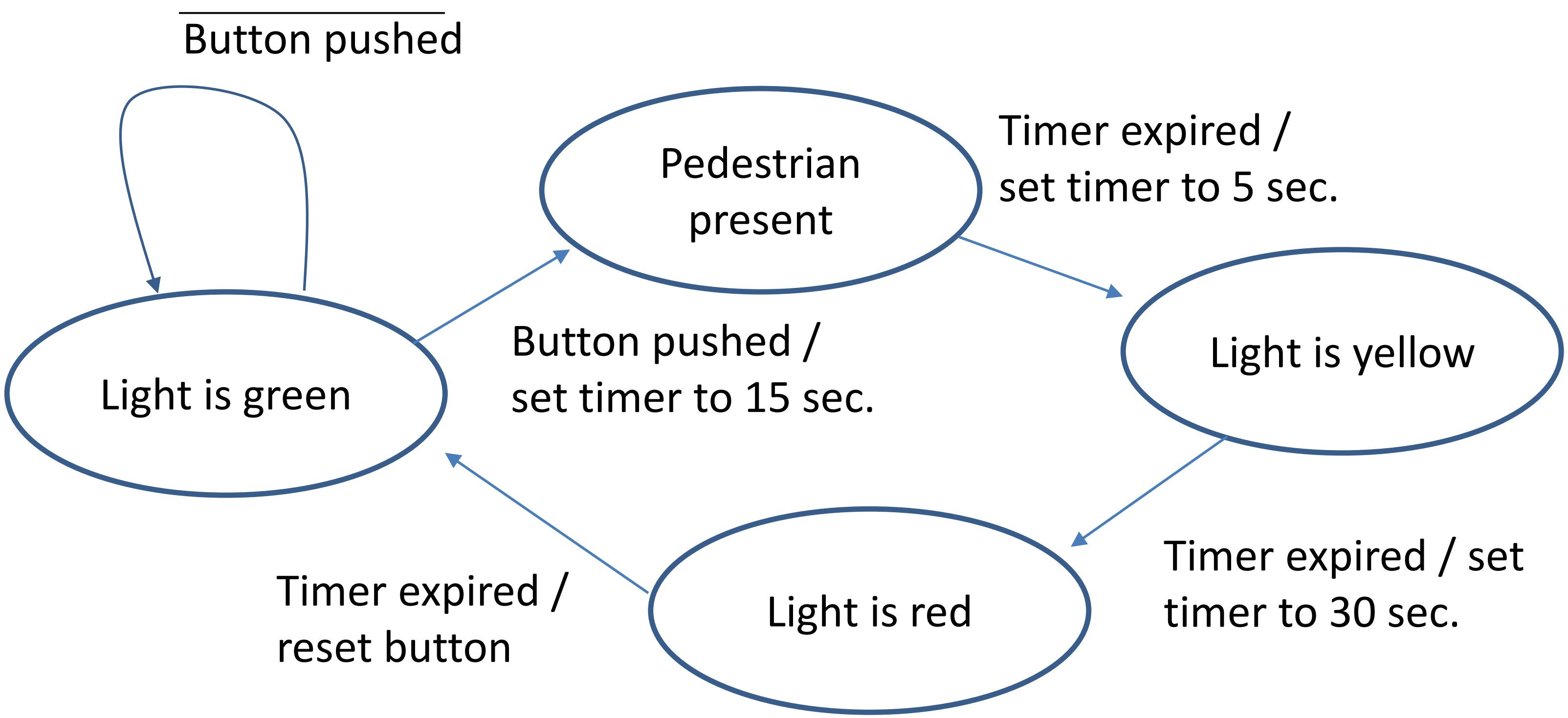
Traffic Light Problem

- Next, the edges are labeled events that are sampled on each clock cycle.



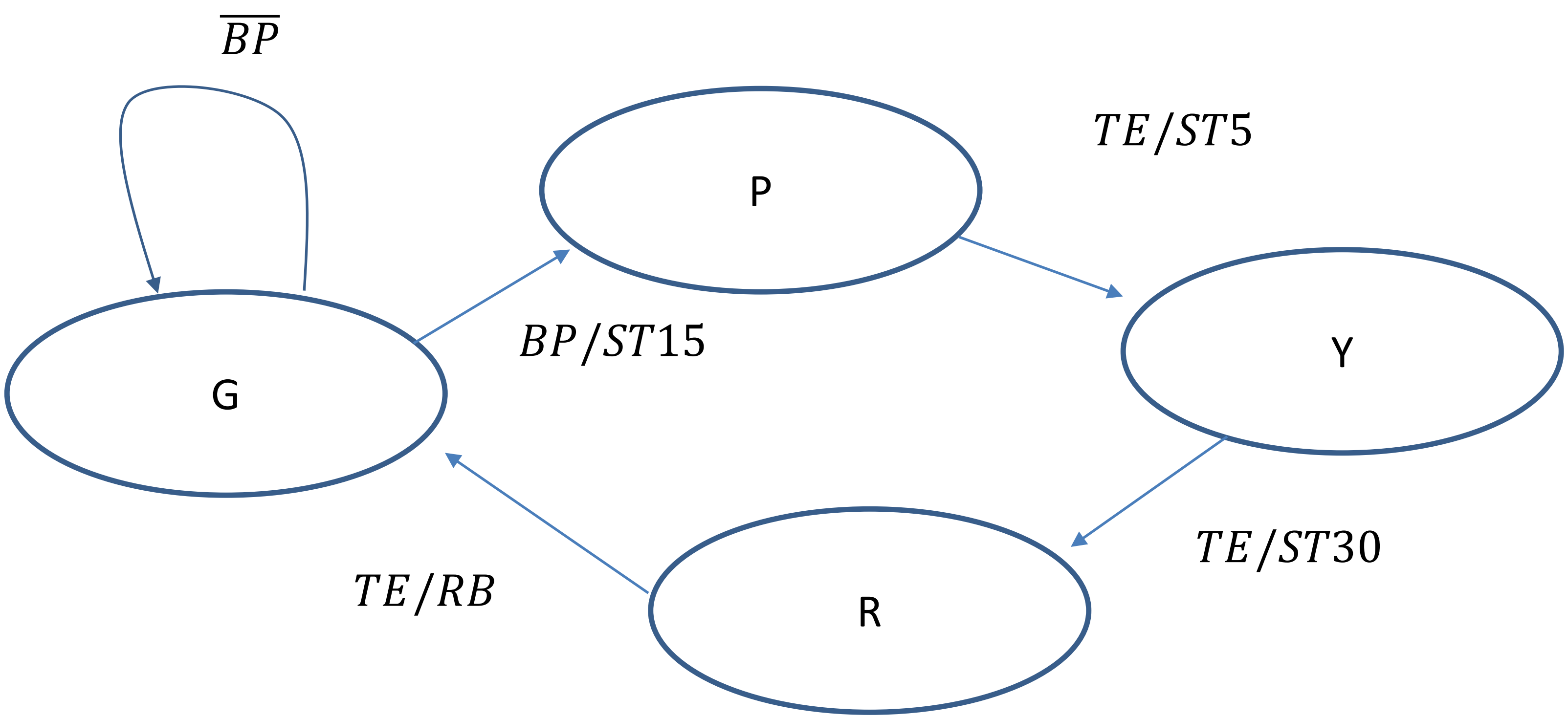
Traffic Light Problem

- Certain actions happen when transitions between states occur



Traffic Light Problem

- Let's re-label these to make it easier to describe the inevitable Boolean algebra:



Finite State Machine

- The states can be represented in various ways by the contents of a set of flip-flops.
- The states are encoded in the contents of a set of D flip-flops
- The outputs and the next states are determined by the current state and the external/internal inputs

Finite State Machine

- One way to encode the states is using a binary number.
- In this case we have four states, so we need two bits, D_0 and D_1:

$$R = 00$$
$$P = 01$$
$$Y = 10$$
$$G = 11$$

D_1 D_0

Finite State Machines

Now we work out the "next state" logic:

D_1	D_0	BP	TE	D_1'	D_0'
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	0	1
0	1	0	1	1	0
0	1	1	0	0	1
0	1	1	1	1	0
1	0	0	0	1	0
1	0	0	1	1	1
1	0	1	0	1	0
1	0	1	1	1	1
1	1	0	0	1	1
1	1	0	1	0	0
1	1	1	0	1	1
1	1	1	1	0	0

Finite State Machines

- One way to proceed is to reduce the next state logic to a bunch of Boolean algebra:

$$D_0' = \overline{D_0} \cdot \overline{D_1} \cdot BP + D_0 \cdot \overline{D_1} \cdot \overline{TE} + \overline{D_0} \cdot D_1 \cdot TE + D_0 \cdot D_1 \cdot \overline{TE}$$
$$D_1' = D_0 \cdot \overline{D_1} \cdot TE + \overline{D_0} \cdot D_1 + D_0 \cdot D_1 \cdot \overline{TE}$$

- Another way (especially for complex logic) would be to store the transitions in a lookup table
  - in this case a 16x2 bit read-only-memory

Finite State Machines

- Output logic:

D_1	D_0	BP	TE	ST	RB
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	1	0
0	0	1	1	1	0
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	1	0
1	0	0	0	0	0
1	0	0	1	1	0
1	0	1	0	0	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	1	0	1
1	1	1	0	0	0
1	1	1	1	0	1

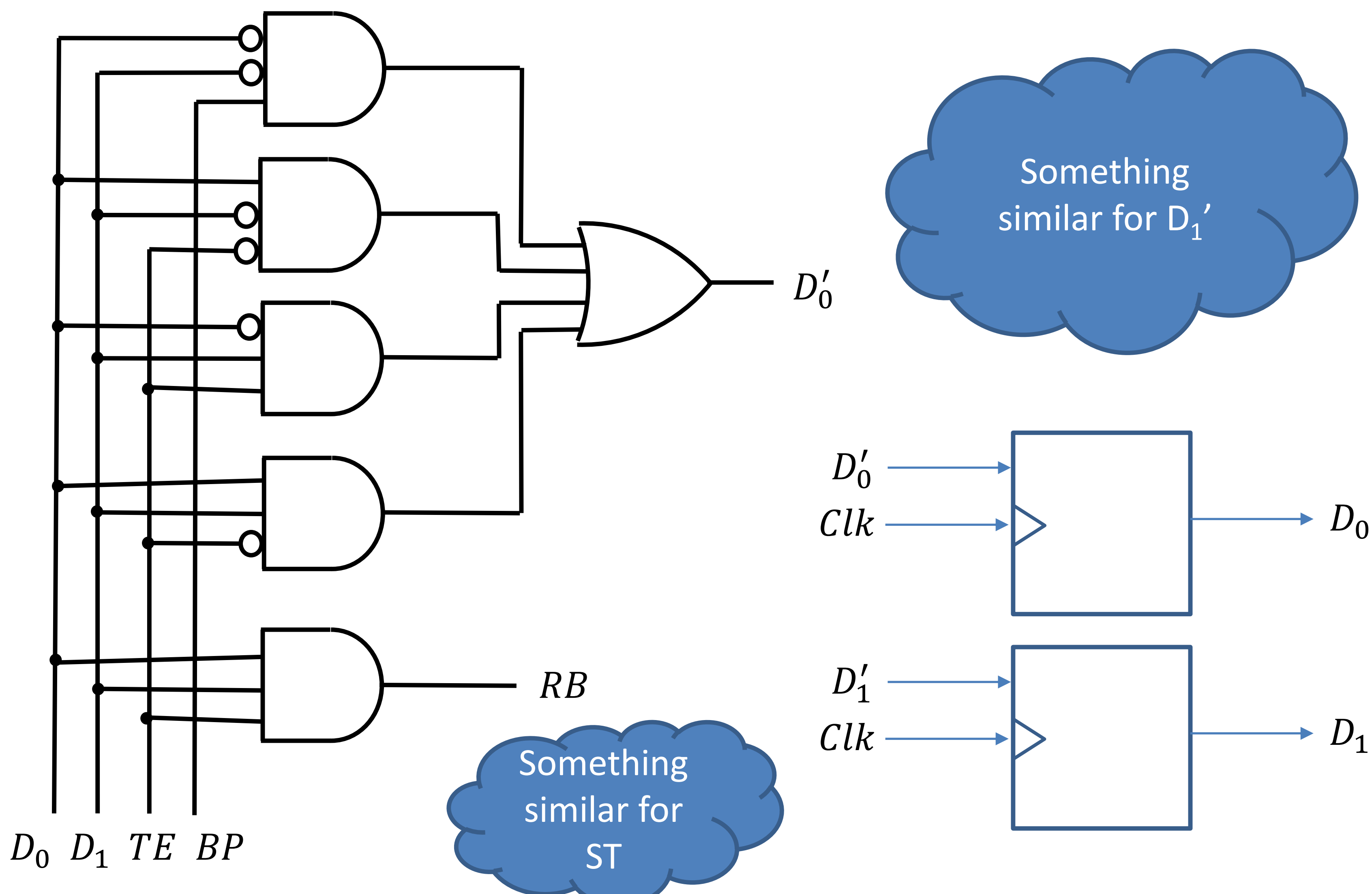
Finite State Machines

- Again, the output logic could be implemented using Boolean algebra:

$$ST = \overline{D_0} \cdot \overline{D_1} \cdot BP + D_0 \cdot \overline{D_1} \cdot TE + \overline{D_0} \cdot D_1 \cdot TE$$
$$RB = D_0 \cdot D_1 \cdot TE$$

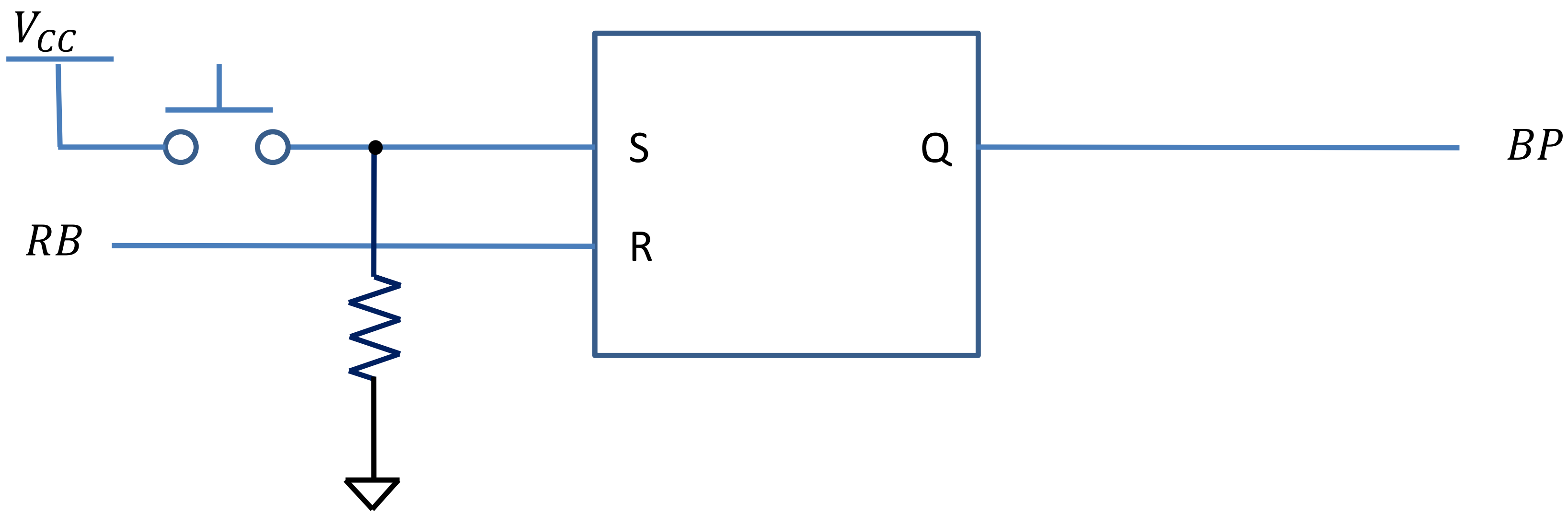
- Again, this could also be implemented using a look-up table / read-only-memory

Hardware Implementation



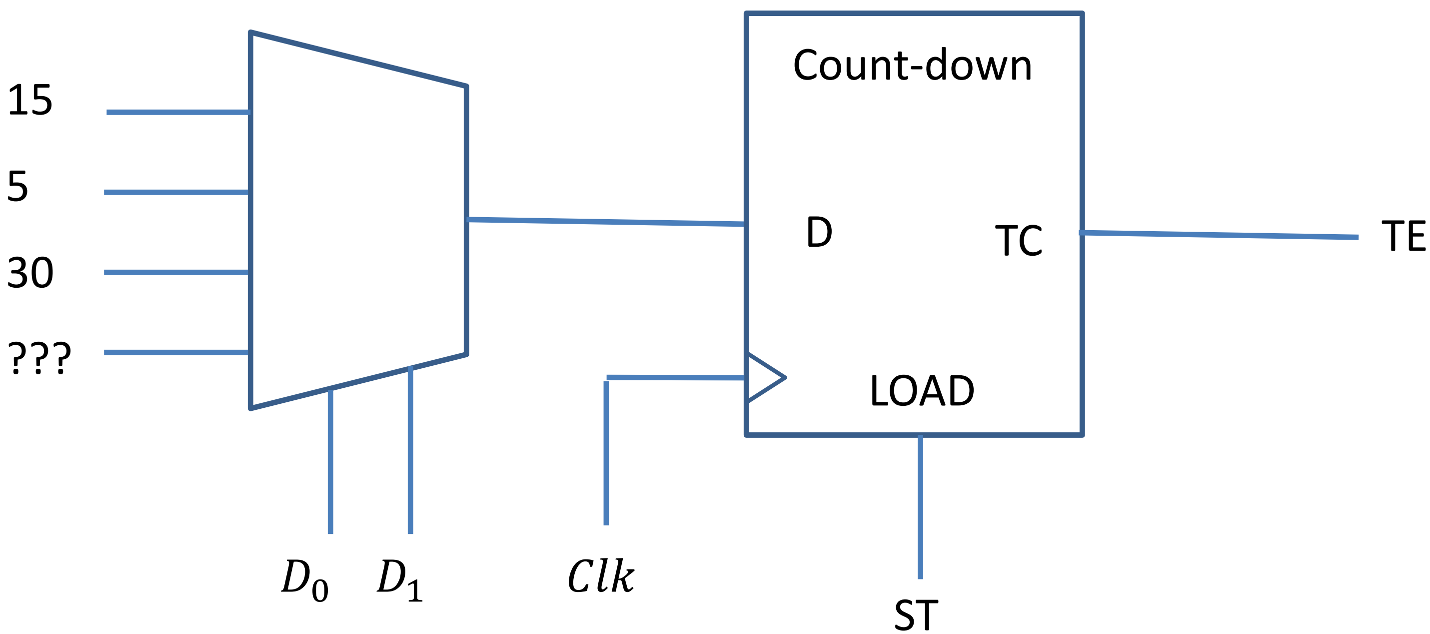
Final Details

- This implements all the state transition logic but there a few things left over…
- The button can be implemented as an S-R flip-flop. It doesn't need to be edge sensitive.



Final Details

- The counter can load different values selected using a multiplexer:



Final Details

- The power to the lights can be controlled using high power MOSFET's that are driven by logic based on the individual states.

- This same design process could be applied to more complicated traffic light configurations
  - Multiple lanes of traffic
  - Sensors to determine when cars are present
  - Multiple pedestrian crossing inputs

- Obviously the whole process gets more complicated and tedious…

Description using VHDL

- The previous discussion was intended to show how digital design can be carried out in principle
- The important point to recognize is that it can be rather prescriptive
- Translating the finite state machine diagram into logic might be tedious, but it doesn’t require much creativity
- These steps are well suited to be carried out by a computer

Description using VHDL

- This is NOT a course in VHDL
- However, it is intended to teach you about the advantages of using hardware definition languages for complex digital design
- The following examples illustrate how the components of the traffic light problem can be *described* in a way that will be correctly interpreted by the design tools.

VHDL Description of a Multiplexer

- A multiplexer is an example of combinatorial logic.
  - It can be synthesized using only logic gates
  - There is no need for latches or memory
- Synthesizable VHDL description:

```
  signal q : std_logic_vector(5 downto 0);
  signal d : std_logic_vector(1 downto 0);
...

  q <= "001111" when d = "00" else     -- value = 15
       "000101" when d = "01" else     -- value =  5
       "010100" when d = "10" else     -- value = 30
       "000000";
```

VHDL Description of a Counter

- A counter is an example of sequential logic.
- First, we can describe what the interface will look like:

```
entity counter is
  port (
    clk : in std_logic;
    d : in std_logic_vector(5 downto 0);
    load : in std_logic;
    tc : out std_logic
  );
end counter;
```

VHDL Description of a Counter

- Then we can describe how it works:

```
architecture Behavioral of counter is

  signal value : std_logic_vector(5 downto 0) := "000000";
  signal done : std_logic;

  begin
    process ( clk ) begin
      if ( clk'event and clk = '1' ) then
        if ( load = '1' ) then
          value <= d;
        elsif ( done = '0' ) then
          value <= std_logic_vector(unsigned(value)-1);
        end if;
      end if;
    end process;

    done <= '1' when value = "000000" else '0';
    tc <= done;

  end;

end Behavioral;
```

VHDL Description of a Finite State Machine

```
architecture Behavioral of traffic_light is

  component counter
  port (
    clk : in std_logic;
    d : in std_logic_vector(5 downto 0);
    load : in std_logic;
    tc : out std_logic
  );
  end component;

  type state_t is ( Green, Pedestrian, Yellow, Red );
  signal state : state_t := Green;
  signal load : std_logic := '0';
  signal tc : std_logic;
```

- The interface to the counter component must be described before it can be used.
- All the signals used in the design need to be declared
- The state is defined using an enumerated type with four states.

VHDL Description of a Finite State Machine

```
begin

  counter_imp : counter
  port map (
    clk => clk,
    d => delay,
    load => load,
    tc => tc
  );

  process ( clk ) begin
    if ( clk'event and clk = '1' ) then
      case state is
      when Green =>
        if ( bp = '1' ) then
          state <= Pedestrian;
          load <= '1';
          rb <= '0';
        else
          state <= Green;
          load <= '0';
          rb <= '0';
        end if;
      when Pedestrian =>
        if ( tc = '1' ) then
          state <= Yellow;
          load <= '1';
          rb <= '0';
        else
          state <= Pedestrian;
          load <= '0';
          rb <= '0';
        end if;
      when Yellow =>
        if ( tc = '1' ) then
          state <= Red;
          load <= '1';
          rb <= '0';
        else
          state <= Yellow;
          load <= '0';
          rb <= '0';
        end if;
```

```
      when Red =>
        if ( tc = '1' ) then
          state <= Green;
          load <= '0';
          rb <= '1';
        else
          state <= Red;
          load <= '0';
          rb <= '0';
        end if;
      end case;
    end if;
  end process;

  delay <= "001111" when d = "00" else     -- value = 15
           "000101" when d = "01" else     -- value =  5
           "010100" when d = "10" else     -- value = 30
           "000000";

end Behavioral;
```

Summary

- Again, this is not a course in VHDL.
- However, the point I'm trying to make is that describing the functionality using a hardware definition language can be much easier than implementing the finite state machine by hand.
- Describing a problem in terms of a finite state machine is a very useful way to build complex digital designs.