Lecture 6:

High-Level Logic Design
Hardware Description Languages

Overview

Reading

W&E 1.6-1.6.2, 1.7 - Functional Representations

W&E 8.4.1 - FSM

Verilog Tutorial, Verilog According to Tom

Introduction

To deal with the large complexity of current chips, we need to work at a higher level than logic gates. Like many software projects, one starts with an imprecise description of what one wants to build, and the process of implementing it helps to define it. Many groups start the process by writing a 'C' model first, just to get a better idea of the functionality of the part. There are some natural abstractions for hardware, FSM and datapaths, that are described at the end of the lecture.

Languages have been created to help describe hardware. These languages are really a kind of parallel programming system, since hardware is just a bunch of concurrent blocks. This lecture will look at an HDL called "Verilog", and describe some of its uses.

High-Level Design Issues

Many people think that design is a straight-forward logical process

- Start with the idea of what you need to build
- And then you build it

Real design is not like that

- Think you have an idea of what you need to build
- Through the design process you figure out what you really want to build
  - Need to validate basic idea early in the process
- What you build depends on the implementation capabilities and constraints
  - Implementation issues will change the specification

Need a language that helps with the real (interactive) design process

Hardware Description Languages

Need a description level up from logic gates.

- Work at the level of functional blocks, not logic gates
  - Complexity of the functional blocks is up to the designer
  - A functional unit could be an ALU, or could be a microprocessor
- The description consists of functions blocks and their interconnections
  - Need some description for each function block (not predefined)
  - Need to support hierarchical description (function block nesting)
- To make sure the specification is correct, make it executable.
  - Run the functional specification and check what it does

Hardware Description Languages (HDLs)

There are many different systems for modeling and simulating hardware.

- **Verilog**
- **VHDL**
- L-language, M-language (Mentor)
- DECSIM (DEC)
- Aida (IBM / HaL)
- and many others

The two most standard languages are Verilog and VHDL.

- For this class (and many others at Stanford) we will be using Verilog
- Given to Stanford for classes
- Runs on many machines at Stanford (including in Sweet Hall)
- Have both a simulator and synthesis tools that work with Verilog

Verilog from 20,000 Feet

Verilog Descriptions look like programs:

C / Pascal	Verilog
Procedures/Functions	Modules
Procedure parameters	Ports
Variables	Wires / Regs

Block structure is a key principle

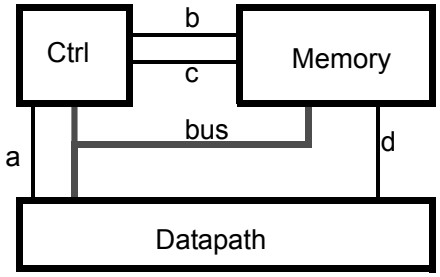
- Use hierarchy/modularity to manage complexity

But they aren't 'normal' programs

- Module evaluation is concurrent. (Every block has its own "program counter")
- Model is really communicating blocks

Verilog (or any HDL) View of the World

A design consists of a set of communicating modules



There are graphic inputs devices for Verilog, but we will not use them

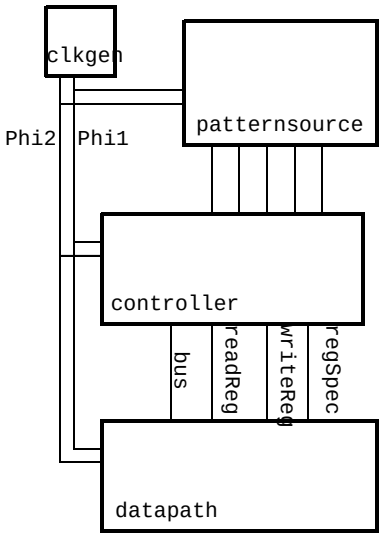
Instead we will use the text method. Label the wires, and use them as 'parameters' in the module calls.

Example Verilog

```
module system;
   wire [7:0] bus_v1, const_s1;
   wire [2:0] regSpec_s1, regSpecA_s1,
      regSpecB_s1;
   wire [1:0] opcode_s1;
   wire Phi1, Phi2, writeReg_s1,
      ReadReg_s1,nextVector_s1
   clkgen clkgen(Phi1, Phi2);
   datapath datapath(Phi1, Phi2, regSpec_s1,
      bus_v1, writeReg_s1, readReg_s1);
   controller controller1(Phi1, Phi2,
      regSpec_s1, bus_v1, const_s1,
      writeReg_s1, readReg_s1, opcode_s1,
      regSpecA_s1, regSpecB_s1,
      nextVector_s1);
   patternsource patternsource(Phi1,
      Phi2,nextVector_s1, opcode_s1,
      regSpecA_s1, regSpecB_s1, const_s1);

ModuleName InstanceName (wires);
```

In this example the instance name and the module name are the same, except for controller1.



Structural Description

This is one way to describe the function of a piece of hardware / module

- Provide the components, and wiring
- Use the structure of the unit to define its function
- Basically, what we did in the example on the previous slide.

But, if you did this all the way down to logic gates or transistors, you would have just created a text-based schematic entry system ---> Not a very good idea.

- So, need another method to specify function of a block at the bottommost level.
- Verilog provides three methods:

  Declarative

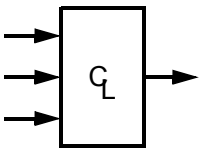
  Procedural

  Functional (we won't use in EE271, least connected to hardware)

Verilog Declarative Definitions

Provides the logical relations between inputs and outputs.

- Assign outputs to be some function of the inputs (continuously)
- Models a piece of combinational logic
- Uses a C-like expression syntax
- Denoted by keyword `assign`



Examples (all execute in parallel):

```
assign nor = ~(b | c);
assign a = x & y, o = x | y;
assign sum[4:0] = a[3:0] + b[3:0];
assign out = (Sel) ? in1: in2;
```

Outputs are wires, and can be a single bit or multiple bits

It is good practice to declare all variables even though Verilog allows undeclared single bit wires.

Verilog Procedural Descriptions

- Use the keyword `always` to provide the functionality using a tiny program that executes sequentially.
- Inside an `always` block, can use standard control flow statements[1]:
  `if  then else; case case default`
- Statements can be compound (use `begin` / `end` to form blocks)

Example:

```
always @ (stuff we still need to talk about)
   begin
       if (x==y) then
           out= in1
       else
           out = in2;
     end
```

1. You need to be a little careful when you use this type of control statement. Since the program is run sequentially, there is an implied priority in the code -- the second case entry can't happen unless the first does not match. This implied priority encoder might not be what you want, especially is cases labels are mutually exclusive. There are usually additional directives you need to add (parallel-case) to indicate this.

`Always` Block Issues

There are two questions that still need to be answered about `always` blocks. One deals with unset outputs, and the other deals with activation.

Unset outputs:

- Occur when an output of the block is not set on all the paths through the code (would happen if the `else` was missing in the example)
- In Verilog, this creates storage

  The value of the output remains unchanged.
- While this is often good, it means you need to be very careful that you don't build storage elements when you don't intend to.

Since the outputs of `always` blocks CAN act as storage elements, all left-hand sides of expressions in `always` blocks must be declared as registers (regs). Note, that does not mean the outputs will be registered, or an `always` block actually creates storage. Even if an output is set on all paths (so there is no storage), the LHS must still be declared a register.

Intentionally Creating Storage in Verilog

To make a simple latch in Verilog is easy. Just make the output of an `always` block not get set when you want to hold its value.

Example:

```
always @ (stuff we still need to talk about)
       if (Enable) then
           myout = in;
```

When `Enable` is high, the output `myout` is updated, but when `Enable` is low, `myout` will hold its last value. This is like the simple pass transistor latch in Lecture 5.

In this example, `myout` would need to be declared a register, because it is the LHS of an expression in an `always` block.

Activation List

The last tricky part about the `always` block is the activation list.

Activation List

- Tells the simulator when to run this block
- Allows the user to specify when to run the block and makes the simulator more efficient.

  If not sensitized to every input, you get a storage element
- But also enables subtle errors to enter into the design.

Two forms of activation list in Verilog:

- `@(signalName or signalName or …)`

  Evaluate this block when any of the named signals change
- `@posedge(signalName);` or `@negedge(signalName);`

  Makes an edge triggered flop. Evaluates only on one edge of a signal.

Activation Lists

Example:

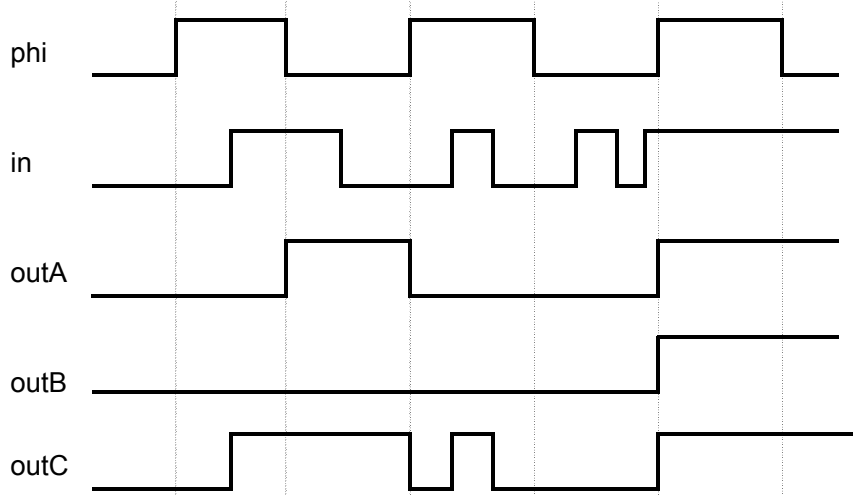
```
always @ (Enable or In)
       if (Enable) then
           out=In;
always @ (x or y or in1 or in2)
    begin
       if (x==y) then
           out= in1
       else
           out = in2;
    end
```

Vcheck will check to be sure that you make complete activation lists. The activation list should contain **everything** on the RHS of expressions in that block

Beware, if an `always` block has **no** activation list (or `# delay` statements), then the simulator goes into an infinite loop.

- Activation Errors - Examples

```
always @(phi) vs  always @(phi) vs always @(phi or in)
  outA =in;    if(phi) outB = in; if(phi) outC = in;
```



`Initial` Block

This is another type of procedural block

- Does not need an activation list
- It is run just once, when the simulation starts.

Used to do extra stuff at the very start of simulation

- Initialize simulation environment
- Initialize design

  This is usually only used in the first pass of writing a design.

  Beware, real hardware does not have `initial` blocks.
- Best to use `initial` blocks only for non-hardware statements
  (like `$display` or `$gr_waves`)

Verilog Variables

There are two types of variables in Verilog:

Wires (all outputs of `assign` statements must be wires)

Regs (all outputs of `always` blocks must be regs)

Both variables can be used as inputs anywhere

- Can use regs or wires as inputs (RHS) to `assign` statements

```
assign bus = LatchOutput + ImmediateValue
```

  `bus` must be a wire, but `LatchOutput` can be a reg
- Can use regs or wires as inputs (RHS) in `always` blocks

```
always @ (in or clk)
  if (clk) out = in
```

  (`in` can be a wire, `out` must be a reg)

The outputs of a module are a special kind of variable, that can also be either regs or wires.

+ Delays in Verilog

Verilog simulated time is in “units” or “ticks”.

- Simulated time is unrelated to the wallclock time to run the simulator.
- Simulated time is supposed to model the time in the modelled machine

  It is increased when the computer is finished modelling all the changes that were supposed to happen at the current simulated time. It then increases time until another signal is scheduled to change values.

User must specify delay values explicitly to Verilog

- `# delayAmount`

  When the simulator sees this symbol, it will stop what it is doing, and pause `delayAmount` of simulated time (# of ticks).

  Delays can be used to model the delay in functional units, but we will not use this feature. All our logic will have zero delay. Can be tricky to use properly.

+ Delay Control

Why delay is a little tricky

```
always @(phi or in)
     #10 if (phi) then out = in;
```

- This code will wait 10 ticks after either input changes, then checks to see if phi == 1, and then updates the output. If you wanted to sample the input when it changed, and then update the output later, you need to place the delay in a different place:

```
always @(phi or in)
     if (phi) then out = #10 in;
```

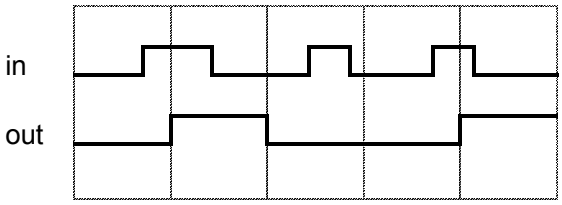
- This code runs the code every time the inputs change, and just delays the update of the output for 10 ticks.

+ Delay Control

Think about this example

```
always
        #100 out = in;
```

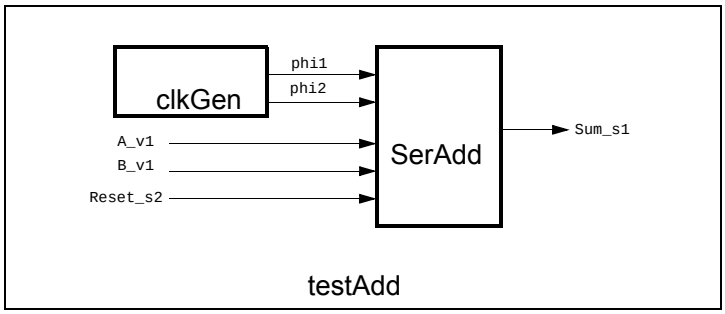
Since the `always` does not have an activation, it runs all the time. As a result every 100 time ticks the output is updated with the current version of the input.



Delay control is used mostly for clock or pattern generation; see the clock generator in the Verilog Homework.

- Simple Example

Here is a simple example of a serial Adder called `serAdd` that is called by a top-level module called `testAdd`
They are separate just to isolate the real hardware from the shell we are using for illustration.



- Code for Simple Example

```
// serAdd.v -- 2 phase serial adder module
module serAdd(Sum_s1, A_v1, B_v1, Reset_s2,
              phi1, phi2);
output Sum_s1;
input   A_v1, B_v1, phi1, phi2, Reset_s2;

reg Sum_s1;
reg A_s2, B_s2, Carry_s1, Carry_s2;

always @(phi1 or A_v1)
        if (phi1)
                A_s2 = A_v1;

always @(phi1 or B_v1)
        if (phi1)
                B_s2 = B_v1;

always @(A_s2 or B_s2 or Reset_s2 or Carry_s2 or phi2)
        if (phi2)
            if (Reset_s2) begin
               Sum_s1 = 0;
               Carry_s1 = 0;
             end
            else begin
               Sum_s1   = A_s2 + B_s2 + Carry_s2;
               Carry_s1 = A_s2 & B_s2 |
                          A_s2 & Carry_s2 |
                          B_s2 & Carry_s2;
            end

always @(Carry_s1 or phi1)
        if (phi1)
                Carry_s2 = Carry_s1;

endmodule
```

```
// testAdd.v -- serial adder test vector generator


// 2 phase clock generator

module clkGen(phi1, phi2);
output phi1,phi2;
reg phi1, phi2;

initial
        begin
                phi1 = 0;
                phi2 = 0;
        end

always
        begin
                #100
                        phi1 = 0;
                #20
                        phi2 = 1;
                #100
                        phi2 = 0;
                #20
                        phi1 = 1;
        end
endmodule

/*
The above clock generator will produce a clock with
a period of 240 units of simulation time.
*/
```



```
/* // test module for the adder
module testAdd; // top level

wire     A_v1, B_v1;
reg      Reset_s2;

serAdd serAdd(Sum_s1, A_v1, B_v1, Reset_s2, phi1,
phi2);

/*
 The serial adder takes inputs during phi1
 and produces _s1 outputs during phi2.
 The _s1 output corresponds to the addition of
 the inputs at the previous falling edge of phi1
*/

clkGen clkGen(phi1,phi2);


reg [5:0] tstVA_s1, tstVB_s1;
reg [6:0] accum_Sum;

initial
 $gr_waves("phi1",phi1,"phi2",phi2,
           "Reset_s2",Reset_s2,"A_v1",A_v1,
           "B_v1",B_v1,"Sum_s1",Sum_s1,
           "Carry_s1",serAdd.Carry_s1,
           "accum_Sum",accum_Sum);

/*
Since SerAdd is a serial adder, we put in the
operands one bit at a time, and accumulate the
output one bit at a time.
 */
assign A_v1 = tstVA_s1[0];
assign B_v1 = tstVB_s1[0];

always @(posedge phi1) begin
                #10
                release A_v1;
                release B_v1;
end

always @(posedge phi2) begin
                #10
                force A_v1 = 1'b0;
                force B_v1 = 1'b0;
end

initial begin
        Reset_s2 = 1;
        tstVA_s1 = 6'b01000;
        tstVB_s1 = 6'b11010;
        accum_Sum = 0;
        @(posedge phi1)
                #50 Reset_s2 = 0;
end

always @(negedge phi1) begin
        $display ("A_v1=%h, B_v1=%h,
                   sum_s1=%h, time=%d",
                   A_v1, B_v1, Sum_s1,$time);
        accum_Sum = accum_Sum << 1 | Sum_s1;
        $display ("tstVA=%h, tstVB=%h,
                  sum_s1=%h,accum_Sum=%h\n",

tstVA_s1,tstVB_s1,Sum_s1,accum_Sum);
end

always @(posedge phi2) begin
 #15
        if (~Reset_s2) begin
            tstVA_s1 = tstVA_s1 >> 1;
            tstVB_s1 = tstVB_s1 >> 1;
           if (tstVA_s1 == 0 && tstVB_s1 == 0) begin
                        #800 $stop;
            end
        end
end
endmodule
```

Verilog Uses

An HDL provides a means for the user to specify a design at a higher level than just gates.

So far, we have been talking mostly about form and not content

- Described how to represent combinational logic
- Have talked about how storage arises, but not how to use it

A key question to ask is, “What should my code look like?” Are there certain styles of hardware that are easier to understand / build / test? This gets back to the question of abstractions, and is really asking whether there are some hardware abstractions that work well. Luckily, the answer is yes.

While there are a few hardware blocks that are purely combinational, most use state (outputs from the previous cycle/computation). There are two powerful abstract models for machines with state: finite state machines and data flows.

Machines with State

Two very different views of state, and two different abstract models:

Data storage used for computation (**Data Flows**)

- In this abstraction, the storage is used to hold data that is being manipulated. In this model the number of bits of state can be enormous, but it does not matter. It is simply the data-set that is being manipulated.
- State is not that important, it is the flow of data that is critical.

Limited state, used for sequencing information (**Finite State Machines**)

- In this abstraction, the storage is used to hold your place in some decision making process. It indicates where you are, and using this information you decide what to do next.
- The amount of state (number of unique decision points) is finite, and usually limited. One could think about drawing out the ‘decision graph’ showing the possible transitions between states.

Machine Partitioning between Dataflow and FSMs

We often think about a chip (or system) as consisting of a number of cooperating Finite State Machines (FSMs) that control other dataflow unit(s). This model is quite general, and spans a wide range of designs.

- Dataflow

  This is called the datapath portion of the chip

  Where computation in the chip is done

  Deals with multi-bit data

  Often uses large memory arrays

- FSMs

  Mostly used to control / sequence the dataflow portion of the chip

Dataflow

Abstract model for moving data through computation units

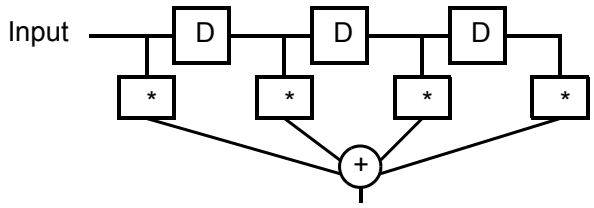
- Data is kept moving through a number of functional units.
- Movement need not be completely regular, but
  - All the bits in a data word should move (almost) the same way at the same time.
- Communication between the functional blocks is important
- Used to represent operations on multi-bit data

Represented by function blocks and lines to represent signal flow.

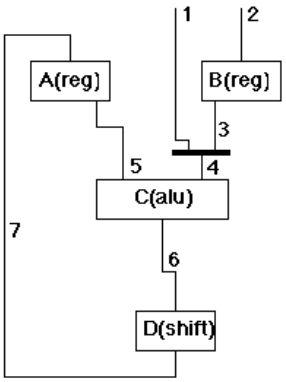
Maps nicely to an HDL.

Dataflow Examples

- Finite Impulse Response Filter



- Simple Processor



FSMs

Any machine with state is a FSM, since with finite storage there are a finite number of states. But this is not a very useful definition. Instead we will use FSM to refer to machines with relatively small number of states. Small enough that you might think about enumerating all the states.

What are FSMs with limited state good for?

- Sequencers, Controllers

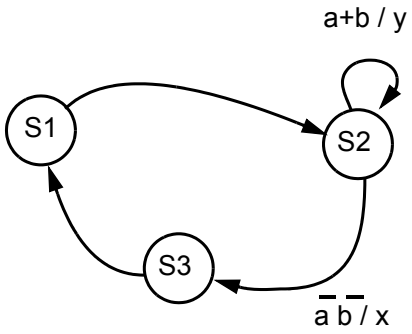
  Things that control the movement of data

There are many ways to describe a state machine

- State Transition Diagrams
- Verilog code
- State Transition Tables
- Boolean Equations

State Transition Diagram

- Natural way to think of a FSM
  We will use this method for the class
- Collection of states and transition arcs
  Arcs are labelled with `inputs/outputs`
  In this small example
    States named “S1” “S2” “S3”
    Inputs are “a” “b”
    Outputs are “x” “y”



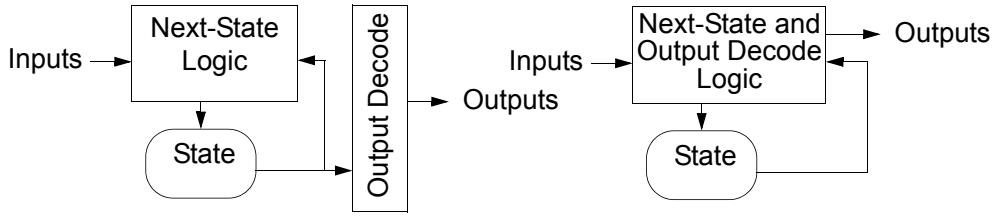
- Pretty clear and concise way to describe a machine
- Forms good documentation of the Verilog code

Mealy and Moore Machines

There are two types of finite state machines,

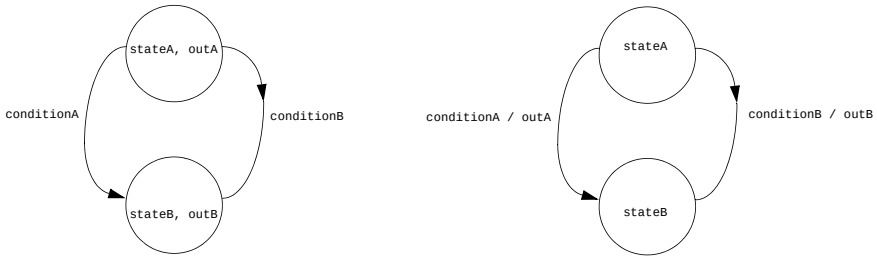
- Moore Machine:
    Outputs are only a function of the state
    Don’t need to have outputs on arcs, only on states
    Need a different state for each possible output
- Mealy Machine:
    Outputs are a function of the inputs and the state
    Outputs associated with arcs, not states

I prefer to draw Mealy Machines, since they are much smaller (because you don’t need to duplicate states that differ only in the value of a current output).



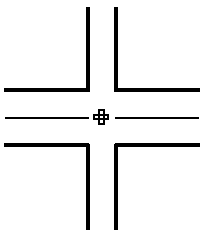
Moore FSM

Mealy FSM



A Traffic Light Example

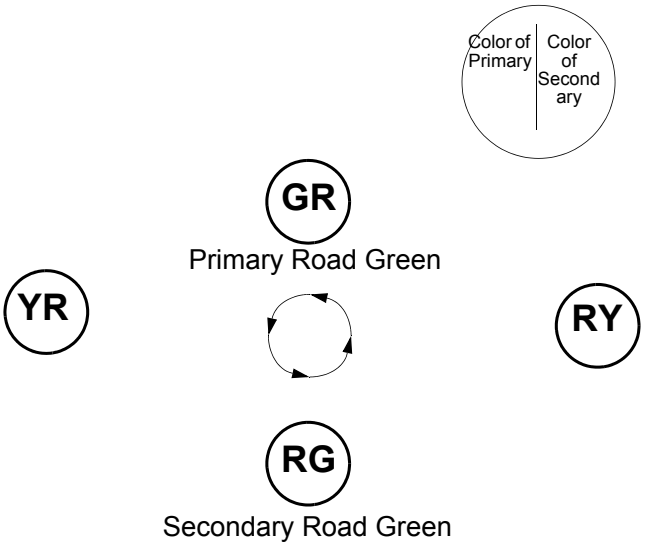
(Inspired by Mead & Conway)



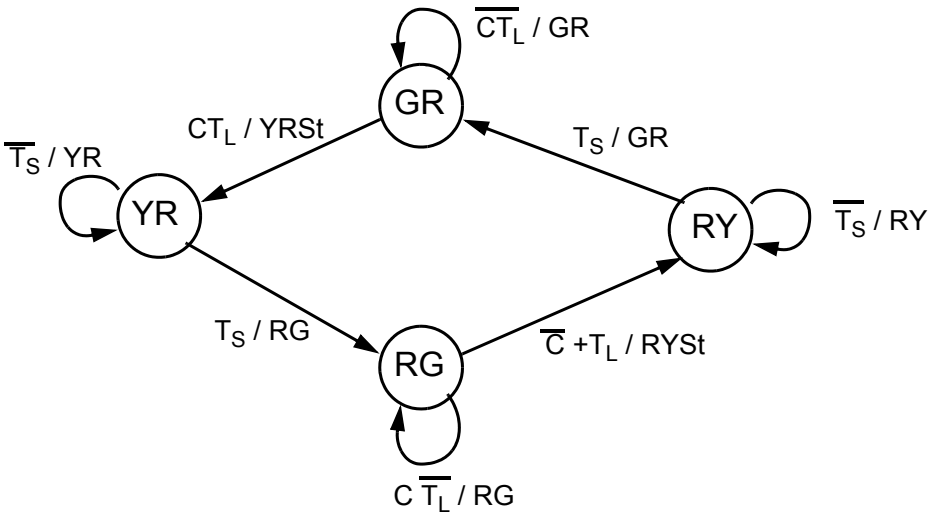
- Inputs:
  - C - Car sensed on secondary road
  - T_S - Short interval timer expired
  - T_L - Long interval timer expired
- Outputs:
  - Color of the Primary Light {R,Y,G}
  - Color of the Secondary Light {R,Y,G}
  - St - restart the timers when asserted.
- Goal:
  - 2nd road is green only when cars are waiting, max time T_L
  - Primary road is green at least T_L

Traffic Controller Pattern

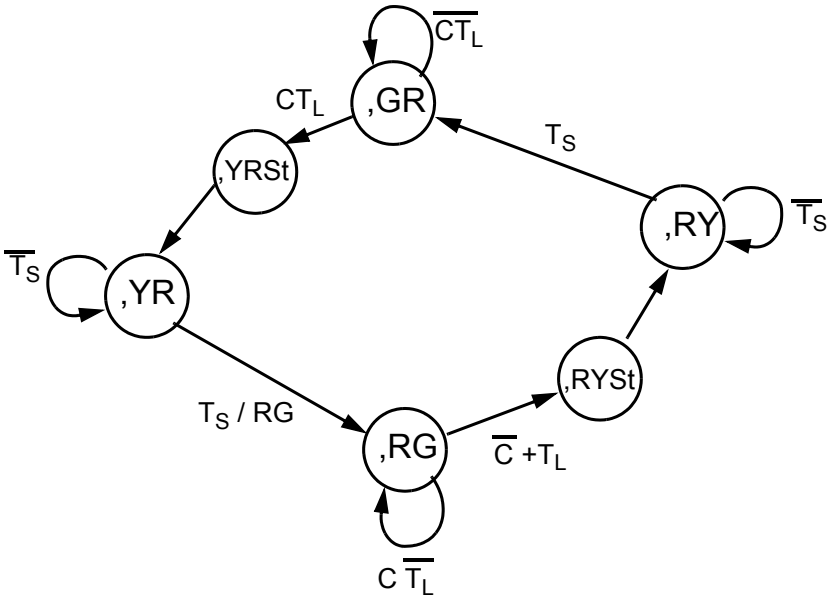
Need to have 4 states: each state



Traffic Controller: Mealy FSM



Traffic Controller: Moore FSM



Verilog Code for a State Machine

State Transition diagrams convert nicely to `always` blocks

- Use `case` statement to get into the correct state
- Use another `case`, or `if` - `then` - `else` to deal with the inputs
- At the end of every choice, set the next state, and the outputs
- Use `// synopsys parallelcase` to avoid synthesizing a priority encoder, since the states should be mutually exclusive.

Must be cautious about not creating any accidental latches.

- Often helps to make the always block be only combinational logic

  Uses *currentState* and the inputs

  Produces *nextState* and the outputs
- Then use a separate `always` block for the storage
- Easier to make sure that the “logic” block does not have any accidental latches in it (more about this in the synthesis lecture)