Introductory Comments

Perhaps the most difficult part for most students who begin writing behavioral code is to stop thinking about programming and start thinking about hardware.

In other words, for every piece of behavioral code that you write, you should have a combinational or sequential circuit in mind that it should synthesize to.

Do NOT think of an *always block* as an escape mechanism that allows you to just write code as you would in C++ to implement an algorithm.

Start by drawing the hardware system that you want to realize FIRST, and then start the coding process.

This may require some experimentation with smaller circuits.

I know what you are thinking! What's the point -- why not just write structural code directly.

You can, of course, but you will be limited in what you can do.

Introductory Comments

The synthesis tools will enable you to synthesize MUCH more complex systems, but you'll need to learn what to expect from them.

Once you've practiced with smaller circuits and understand what to expect, you will gain confidence and learn to appreciate the power of synthesis.

If you follow the coding style you learned in CMSC 201 for C, then you will certainly become frustrated and eventually give up or end up implementing a system that doesn't quite get it right.

The most important exercise that you can do as a novice is write code fragments, synthesize and look at the structural schematic that is generated.

The examples that I have provided below are a starting point.

It is impossible to provide examples for everything that you will end up doing.

So you should be prepared to use this strategy to analyze your own code and fragments therein as needed.

Combinational Logic Synthesis

As we discussed in class, the **always** statement can be used to specify combinational or sequential logic.

If you intend to use it to specify combinational logic, here are some points to remember.

As you remember from C, when you write conditional statements using, e.g., the *if stmt*, in some cases, you didn't need to include an *else* because you didn't want anything to happen to the variable under certain conditions.

For combinational logic, there are no storage elements to *remember* the value of the variable under those conditions.

Therefore, if you want the synthesis engine to generate combinational logic then you need to specify **explicitly** all alternatives in your code.

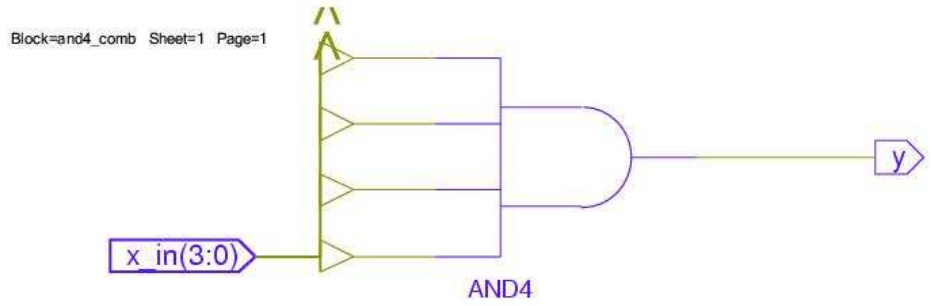
Otherwise, the synthesis engine will NOT generate what you expect.

Combinational Logic Synthesis

Here's a good example: The following code is a behavioral description of a 4-input AND gate.

```
module and4_comb(y, x_in)
   parameter word_length = 4;
   input [word_length - 1: 0] x_in;
   output y;
   reg y;
   integer k;

   always @ x_in
      begin
         y = 1;
         for (k = 0; k <= word_length -1; k = k + 1)
            if ( x_in[k] == 0 )
               y = 0;
      end
endmodule
```

Using ISE 9.2, we get

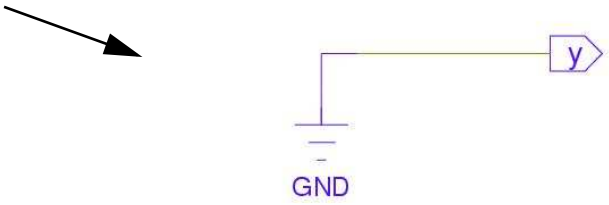


This is what we expected.

Combinational Logic Synthesis

However, failing to define both output states causes the following.

```
module and4_comb(y, x_in)
   parameter word_length = 4;
   input [word_length - 1: 0] x_in;
   output y;
   reg y;
   integer k;

   always @ x_in
      begin
         // COMMENTED OUT y = 1;
         for (k = 0; k <= word_length -1; k = k + 1)
            if ( x_in[k] == 0 )
            begin
               y = 0;
            end
      end
endmodule
```



The synthesis engine issues a warning indicating that the output *y* evaluates to a constant value 0.

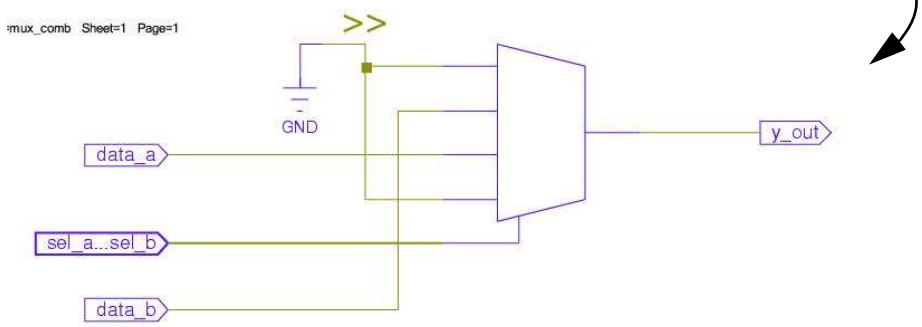
This is easy to catch stand alone, but in a larger context, it may be missed.

Combinational Logic Synthesis

The golden rule is combinational logic *must* specify the value of the output for all values of the input.

Failing to do so, in other cases, can cause the synthesis engine to *infer* a latch.

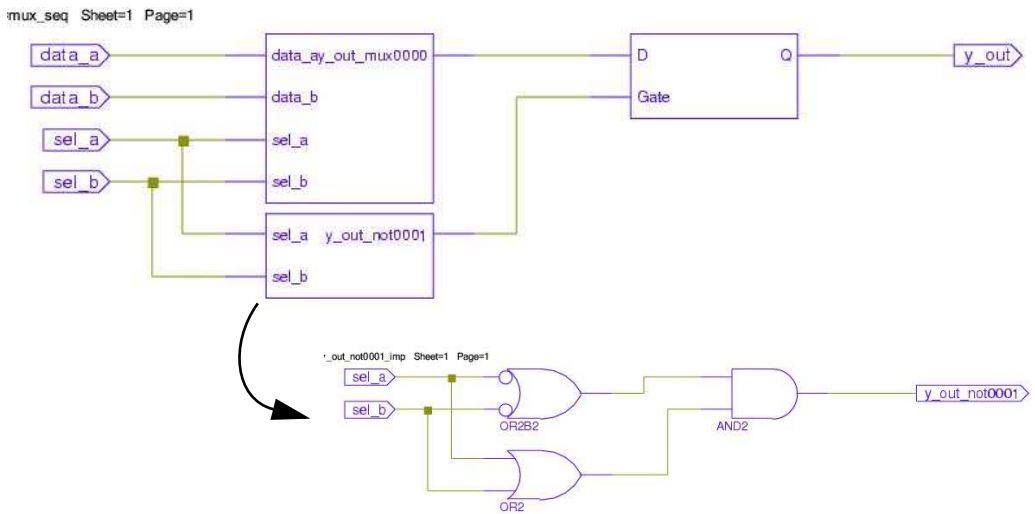
```
module mux_comb( y_out, sel_a, sel_b, data_a, data_b);
   input sel_a, sel_b, data_a, data_b;
   output y_out;
   reg y_out;

   always @(sel_a or sel_b or data_a or data_b)
      case ({sel_a, sel_b})
         2'b10: y_out = data_a;
         2'b01: y_out = data_b;
         default: y_out = 0;          // Correct way for a comb. circuit
      endcase
endmodule
```



Combinational Logic Synthesis

With the *default* statement removed, the output is undefined for input combinations *00* and *11*.

```
module mux_latch( y_out, sel_a, sel_b, data_a, data_b);
   input sel_a, sel_b, data_a, data_b;
   output y_out;
   reg y_out;

   always @(sel_a or sel_b or data_a or data_b)
      case ({sel_a, sel_b})
         2'b10: y_out = data_a;      // Incorrect, infers a latch with clk driven
         2'b01: y_out = data_b;      // by the inputs (warning is issued)
      endcase
endmodule
```

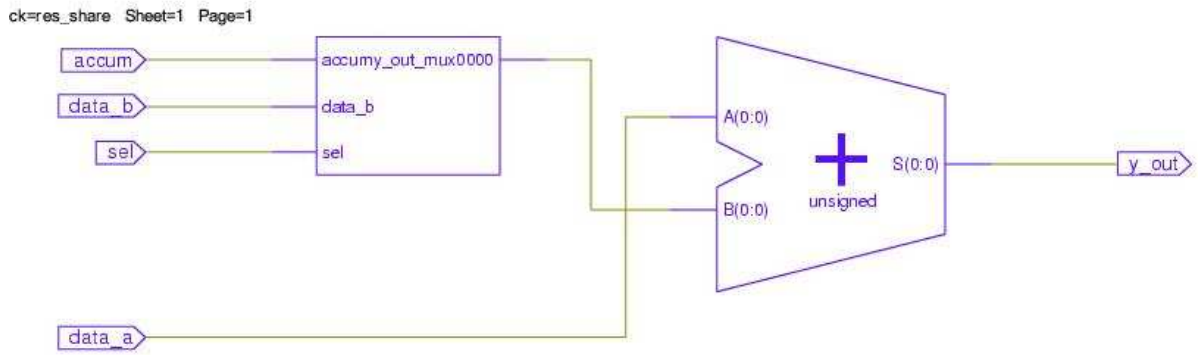


Combinational Logic Synthesis

Another important point to remember is that operands that appear on the righthand side of an assignment MUST NOT appear on the lefthand side. Combination feedback is not allowed.

For datapath operations, the synthesis tool should be able to recognize when it is possible to share resources.

```
module res_share( y_out, sel, data_a, data_b, accum);
   input sel, data_a, data_b, accum;
   output y_out;
   reg y_out;

   assign y_out = sel ? data_a + accum: data_a + data_b;
endmodule
```



Sequential Logic Synthesis

For sequential logic, there are certain hard fast rules that must always be followed in order to guarantee its synthesis.

For example, the event control expression of a cyclic behavior for a sequential circuit MUST be synchronized to a single edge (**posedge** or **negedge** but not both) of a single clock.

You are allowed to have multiple behaviors with different synchronizing signals, but all must have the same period (be apart of a single clock domain).

Latch Synthesis

Level-sensitive behavior is characterized by an output that is affected by the input only while a control signal is *asserted*.

Otherwise, the input is ignored and the output remains constant.

The synthesis tools *infers a latch* when it detects a level-sensitive behavior, i.e., NO edge constructs, in which a register variable is assigned value **in some threads** of activity but not others, e.g., an incomplete *if stmt*.

Sequential Logic Synthesis

The synthesis tool identifies a *control signal* as a signal whose value controls the branching of the activity flow, e.g., **case** and **if** stmts.

A latch is inferred if a path assigns a variable its own value, i.e., if it has ***self-feedback***, even if a register is assigned value in all activity flows.

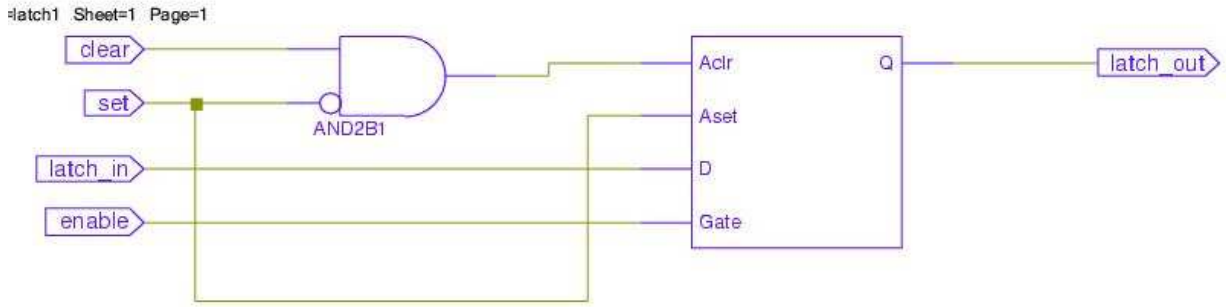
A simple way to create a latch:

```
module latch( data_out, data_in, enable);
   input data_in, enable;
   output data_out;
   reg data_out;
   assign data_out = enable ? data_in: data_out;
endmodule
```

Sequential Logic Synthesis

A latch with asynchronous set/reset latch, use:

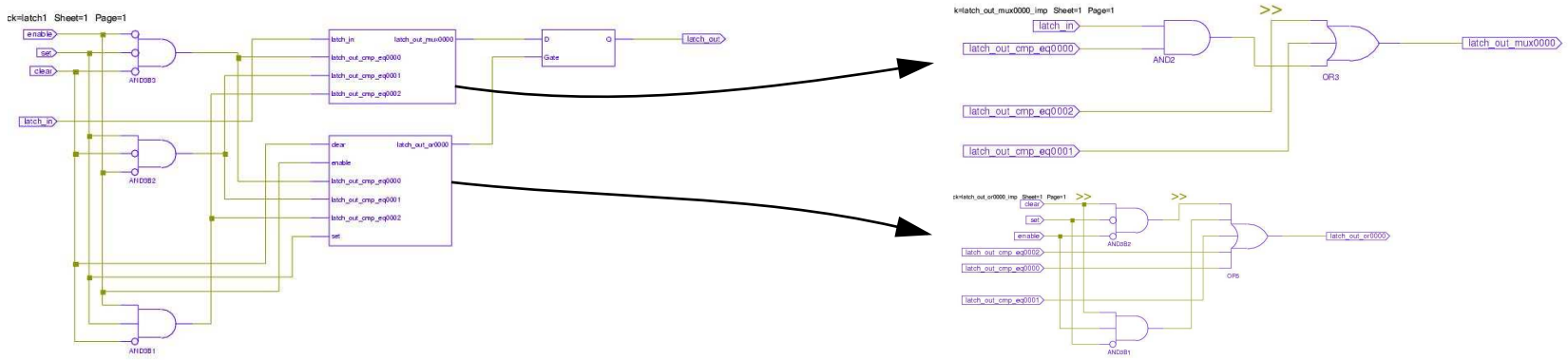
```
module latch( latch_out, latch_in, set, clear, enable);
  input latch_in, set, clear, enable;
  output latch_out;
  reg latch_out;

  always @(enable or set or clear or latch_in)
    if ( set == 1 )
        latch_out = 1'b1;
    else if ( clear == 1 )
        latch_out = 1'b0;
    else if ( enable == 0 )
        latch_out = latch_in;       // Latch is in transparent mode
    else
        latch_out = latch_out;      // Self-feedback
endmodule
```



Sequential Logic Synthesis

The synthesis engine is **not** able to recognize the same from the following:

```
module latch1(latch_out, latch_in, set, clear, enable);
   input latch_in, enable, set, clear;
   output latch_out;
   reg latch_out;
   always @(enable or set or clear or latch_in)  // latch_in in activity list.
      case ({enable, set, clear})
         3'b000: latch_out = latch_in;
         3'b110: latch_out = 1'b1;
         3'b010: latch_out = 1'b1;
         3'b101: latch_out = 1'b0;
         3'b001: latch_out = 1'b0;
         default: latch_out = latch_out;  // Explicit assignment of residual value.
      endcase
   endmodule
```



Sequential Logic Synthesis

Both of these are equivalent in terms of functionality, but the latter is much less efficient.

So, you'll need to pay attention to how you write your code if you want a *minimal* or reader-friendly realization.

Registers/Flip-Flops

Any of the following conditions will cause a register *variable* to synthesize to a FF (a memory element):

- The register variable is referenced outside the scope of the behavior
- It is referenced within a behavior before it is assigned value
- It is assigned value in **only some** branches of the activity

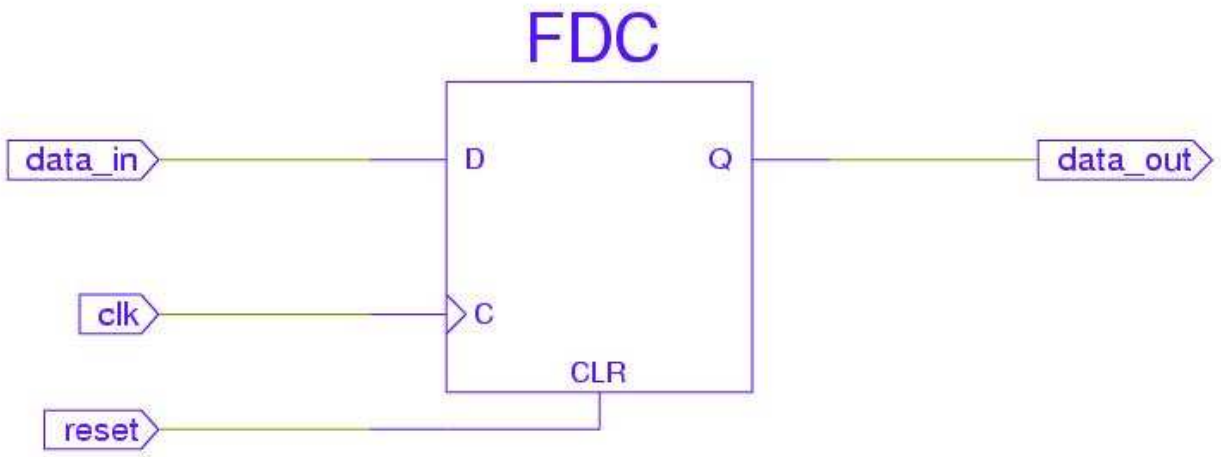
A register variable will be synthesized as the output of a FF when its value is assigned synchronously with the edge of a signal.

Decode control signals first. Last condition in, e.g. an **if** stmt, are synchronized to the rising edge of clk.

Sequential Logic Synthesis

A simple example:

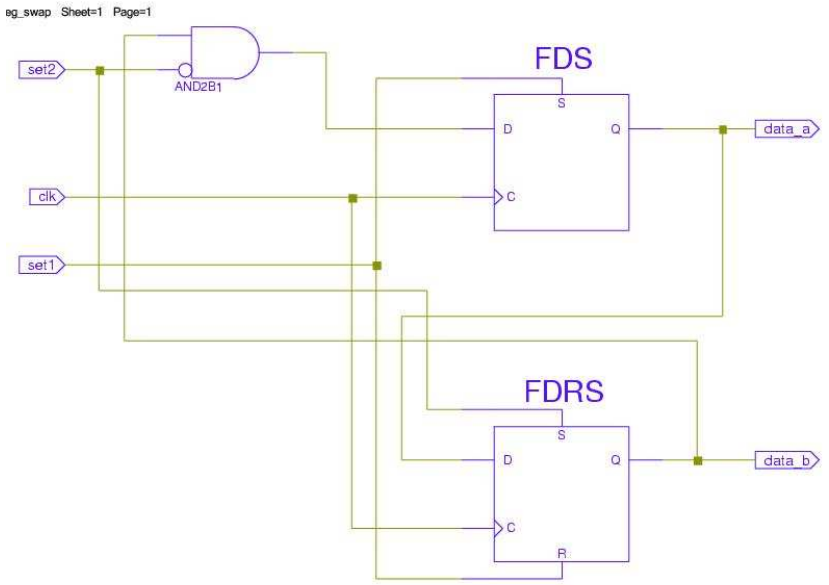
```
module reg1(data_out, data_in, clk, reset);
   input data_in, clk, reset;
   output data_out;
   reg data_out;

   always @(posedge clk or posedge reset)
      if ( reset == 1'b1 ) data_out <= 1'b0;
      else data_out <= data_in;

   endmodule
```



Sequential Logic Synthesis

A more complex example involving several control signals:

```
module reg_swap(data_a, data_b, set1, set2, clk);
   input set1, set2, clk;
   output data_a, data_b;
   reg data_a, data_b;

   always @(posedge clk)
      if (set1) begin data_a <= 1; data_b <= 0; end
      else if (set2) begin data_a <= 0; data_b <= 1; end
      else
         begin
            data_b <= data_a;
            data_a <= data_b;
         end
   endmodule
```

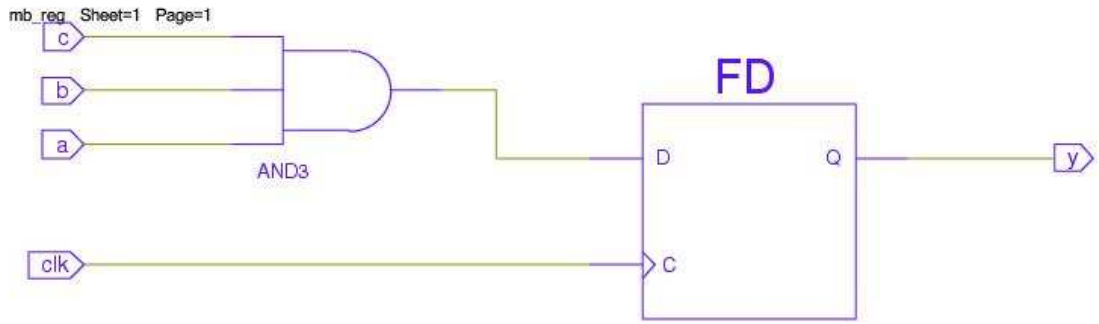


Sequential Logic Synthesis

Registering Combinational Logic:

If you want to enable the output of combinational logic to be captured in FFs, make the behavior associated with the combinational logic sensitive to *clk*.

```
module reg_and(a, b, c, clk, y);
  input a, b, c, clk;
  output y;
  reg y;

  always @(posedge clk)
    begin
      y <= a & b & c;
    end
endmodule
```

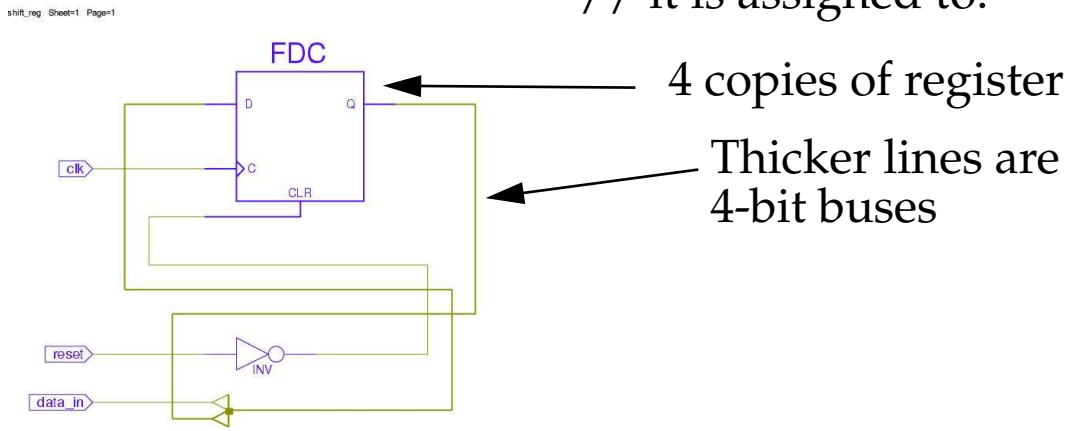


Sequential Logic Synthesis

Shift Registers and Counters:

Note values of vars on RHS are those *before* the clk edge while those on the LHS are the values *after* the clk edge.

```
module shift_reg(data_in, data_out, clk, reset);
  input data_in, clk, reset;
  output data_out;
  reg [3:0] data_reg;

  assign data_out = data_reg[0];

  always @(negedge reset or posedge clk)
    begin
      if (reset == 1'b0) data_reg <= 4'b0;
      else data_reg <= {data_in, data_reg[3:1]};    //Referenced before
    end                                             // it is assigned to.
endmodule
```

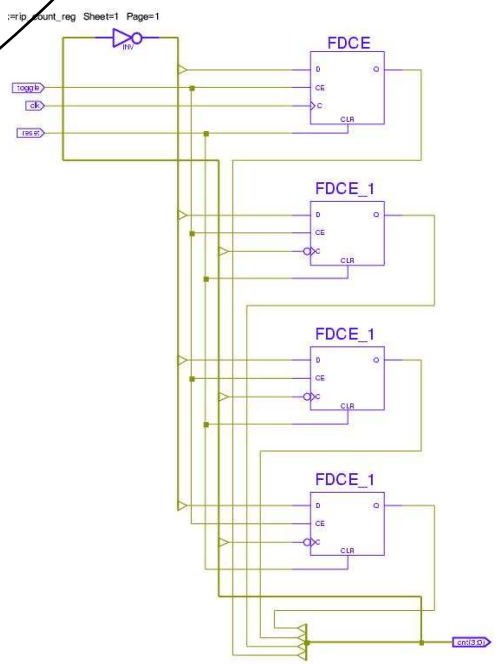


Sequential Logic Synthesis

Shift Registers and Counters:

```
module rip_cnter(clk, toggle, reset, cnt);
  input clk, toggle, reset;
  output [3:0] cnt;
  reg [3:0] cnt;                  // Synthesis tool requires a simple variable in
  wire c0, c1, c2;                // event control expression -- no bit-select.
  assign c0 = cnt[0]; assign c1 = cnt[1]; assign c2 = cnt[2];
  always @(posedge reset or posedge clk)
      if (reset == 1'b1) cnt[0] <= 1'b0;
      else if (toggle == 1'b1) cnt[0] <= ~cnt[0];
  always @(posedge reset or negedge c0)
      if (reset == 1'b1) cnt[1] <= 1'b0;
      else if (toggle == 1'b1) cnt[1] <= ~cnt[1];
   ... // see text for rest of code.
  endmodule
```

// Ripple effect to cnt[1]



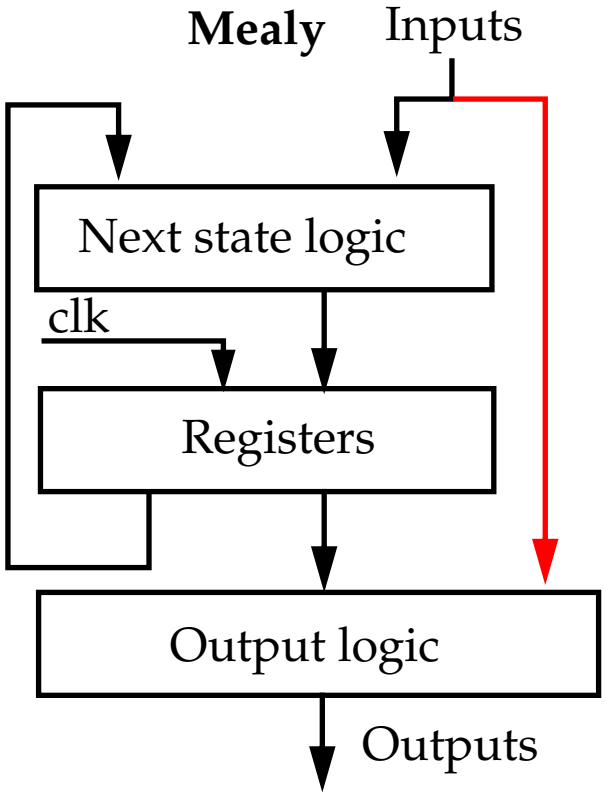
Schematic shows output of FF driving cnt[0] also drives clk input of FF driving cnt[1]

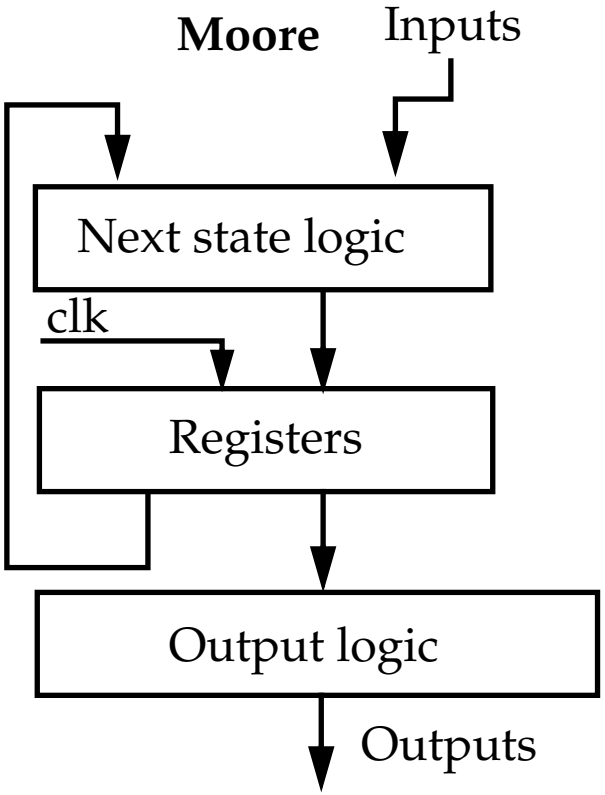
Finite State Machines

Separate behaviors are recommended for defining the *state transitions* and *next-state* logic, b/c it improves readability.

Mealy machines: output can change asynchronously with the clk (depends on the state AND the input)

Moore machines: output is synchronized with clk (and change in state).

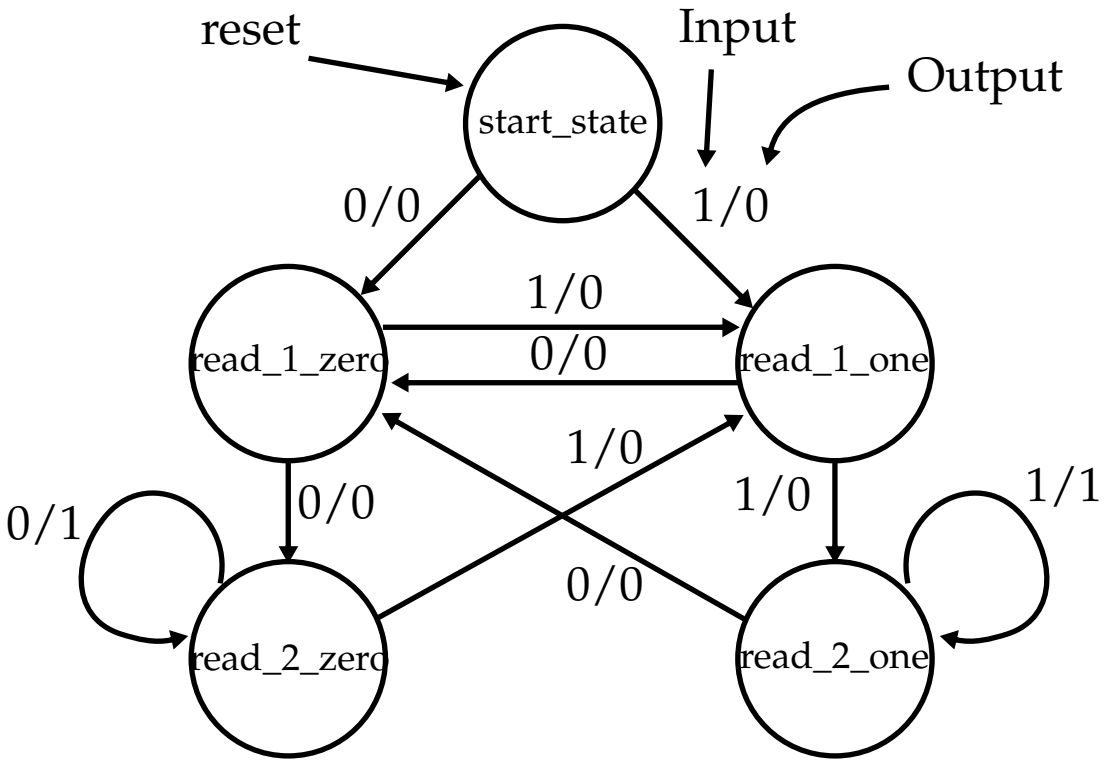




Finite State Machines

Separate behaviors are recommended for defining the *state transitions* and *next-state* logic, b/c it improves readability.

Consider a machine designed to synchronously read a serial bit stream of data and assert an output when it detects 2 successive 0s or 1s.



Finite State Machines

The machine is to be active on the rising edge of *clk* and the input data is synchronized to change on the falling edge of *clk*.

The output is to assert after two consecutive identical samples of data are detected and remain asserted as long as the condition is true.

Explicit State Machines

The verilog code enumerates the states and specifies the transitions between them.

We will implement the state transitions as a cyclic behavior synchronized to *clk*.

A separate *asynchronous* behavior will implement the combinational logic describing *next state.*

A continuous assignment will be used to generate the output combinationally from the state and input (Mealy).

Finite State Machines

```
module seq_det_mealy(clk, reset, in_bit, out_bit);
  input clk, reset, in_bit;
  output out_bit;

  reg [2:0] cur_state, next_state;

  parameter start_state = 3'b000;
  parameter read_1_zero = 3'b001;
  parameter read_1_one = 3'b010;
  parameter read_2_zero = 3'b011;
  parameter read_2_one = 3'b100;


  always @(posedge clk or posedge reset)      // State transition logic
    if (reset == 1) cur_state <= start_state;
    else cur_state <= next_state;
```

This machine incorporates an *asynchronous* reset.

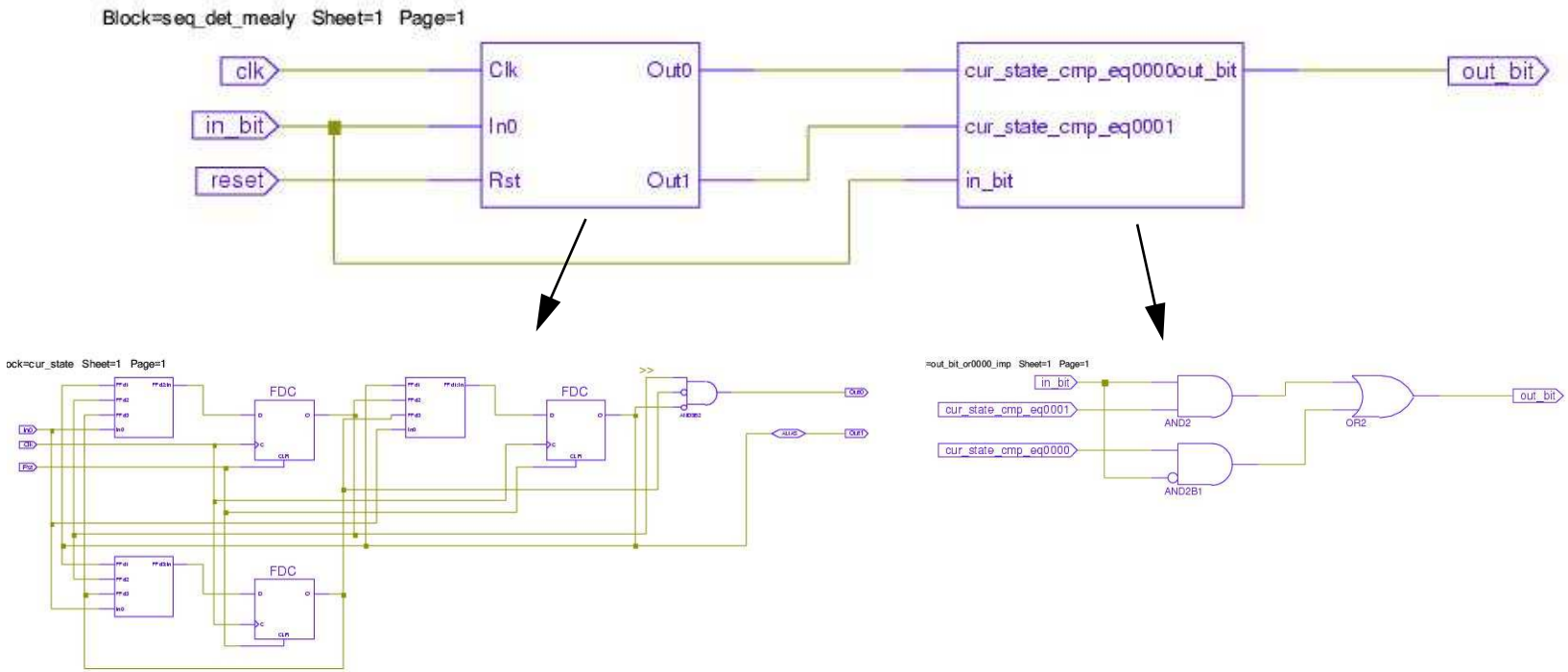
Finite State Machines

```
always @(cur_state or in_bit)                    // Asynchronous logic
  case (cur_state)
    start_state:
      if (in_bit == 0) next_state <= read_1_zero;
      else if (in_bit == 1) next_state <= read_1_one;
      else next_state <= start_state;
    read_1_zero:
      if (in_bit == 0) next_state <= read_2_zero;
      else if (in_bit == 1) next_state <= read_1_one;
      else next_state <= start_state;
    read_2_zero:
      if (in_bit == 0) next_state <= read_2_zero;
      else if (in_bit == 1) next_state <= read_1_one;
      else next_state <= start_state;
    read_1_one:
      if (in_bit == 0) next_state <= read_1_zero;
      else if (in_bit == 1) next_state <= read_2_one;
      else next_state <= start_state;
    read_2_one:
      if (in_bit == 0) next_state <= read_1_zero;
      else if (in_bit == 1) next_state <= read_2_one;
      else next_state <= start_state;
    default: next_state <= start_state;
  endcase
```

Finite State Machines

```
    assign out_bit = ((cur_state == read_2_zero && in_bit == 0) ||
       (cur_state == read_2_one && in_bit == 1)) ? 1 : 0;

endmodule
```

Synthesized schematic



Finite State Machines

Test bench:

```
module seq_det_mealy1_tb;
    reg clk;
    reg reset;
    reg in_bit;
    wire out_bit;

    seq_det_mealy uut
     (.clk(clk),.reset(reset),.in_bit(in_bit),.out_bit(out_bit));

    initial
       begin
       $monitor($time, "clk = %b reset = %b in_bit = %b out_bit = %b \
          cur_state = % next_state = %h", clk, reset, in_bit, out_bit,
          uut.cur_state[2:0], uut.next_state[2:0]);
       end

    always
       #50 clk = ~clk;

    initial begin
       clk = 0; reset = 0; in_bit = 0;

    // Wait 100 ns for global reset to finish
       #100 reset = 1; #10 reset = 0; #165 in_bit = 1;
    end
    endmodule
```

Finite State Machines

Waveforms from testbench: