Synthesis of Sequential Logic

A Verilog description of sequential logic can be synthesized only if certain conditions are met.

In general, the event control expression of a cyclic behavior *must* be **synchronized** to a single edge (**posedge** or **negedge** but not both) of a single clk.

Multiple behaviors need not have the same synchronizing signal, nor the same edge of the same signal, but all clks should have the same period.

This yields a single clock domain for optimization.

Commonly synthesized sequential logic:

- Data register, latch, shift register
- Accumulator, parallel/serial converter, binary counter, BCD counter
- FSM, synchronizer, pulse generator, timing generator, clk generator
- Event counter, memory address counter, FIFO memory pointer
- Sequencer, controller, edge detector

Synthesis of Latches

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.

Synthesis tools infer a latch when they

- Detect a level-sensitive behavior (no edge constructs)
- A register variable is assigned value in some threads of activity but not others, e.g., an incomplete *if stmt.*

The synthesis tool must identify the datapaths and their control signals.

The control signal is the signal whose value controls the branching of the activity flow to the stmts that assign/don't assign value to a reg variable.

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

Otherwise, it's combinational.

Synthesis of Latches

General rules:

- Latches implement incompletely-specified assignments in **case** and **if** stmts.
- If a **case** stmt has a default assignment in which the variable is explicitly assigned to itself, synthesis will choose a MUX structure with feedback.
- If an **if** stmt assigns a variable to itself, a MUX with feedback is used.
- If the behavior is *edge-sensitive*, incomplete **case** and **if** stmts synthesize to FFs.
- If feedback is present, the FF output is fed back using MUX.
- A latch is inferred when the conditional operator, **? ...** :, is implemented with feedback. Actual implementation depends on the context.

    If conditional operator is used in a *continuous assignment*, the result is a MUX with feedback.

    If used in an *edge-sensitive* cyclic behavior, the result is a register with a gated datapath in a feedback configuration.

    If used in a *level-sensitive* cyclic behavior, the result is a hardware latch.

Synthesis of Latches

Be aware that explicit *gate-level* latches, e.g., cross-coupled **nand** gates, and other combinational feedback, will not be synthesized into hardware latches.

The synthesis tools will detect this and issue a warning.

Here, the **case** list is incomplete (only 5 of 8)

```
module latch_case_assign(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)
   case ({enable, set, clear})
      3'b000: assign latch_out = latch_in;  //Transparent mode
      3'b110: assign latch_out = 1'b1;  // Set
      3'b010: assign latch_out = 1'b1;  // Set
      3'b101: assign latch_out = 1'b0;  // Clear
      3'b001: assign latch_out = 1'b0;  // Clear
      default: deassign latch_out;  // Hold residual value
   endcase
endmodule
```

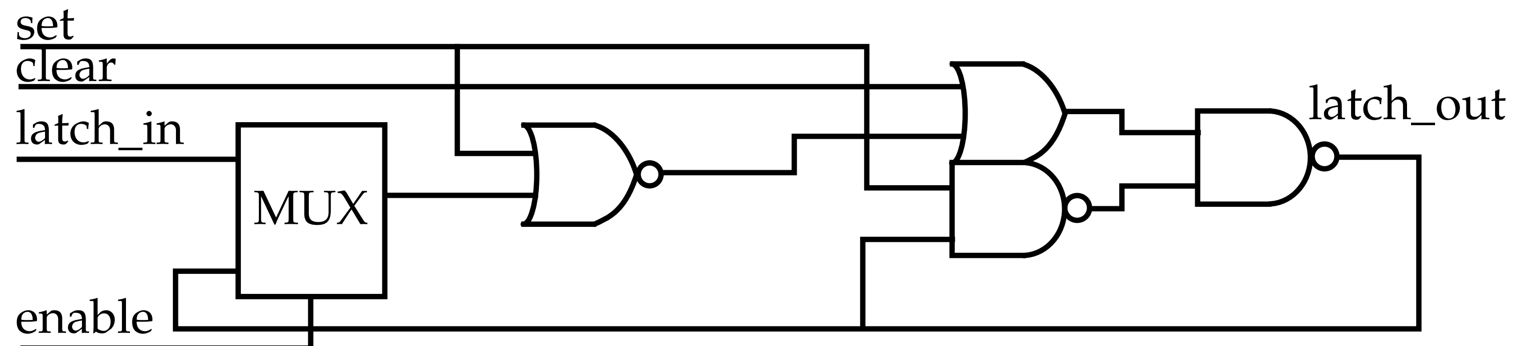
Simulates efficiently but does not synthesize by all tools.

Synthesis of Latches

This one is synthesizable by all tools but yields *feedback* logic.

```
module latch_case1(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: assign latch_out = latch_in;
      3'b110: assign latch_out = 1'b1;
      3'b010: assign latch_out = 1'b1;
      3'b101: assign latch_out = 1'b0;
      3'b001: assign latch_out = 1'b0;
      default: latch_out = latch_out;  // Explicit assignment of residual value
   endcase
endmodule
```

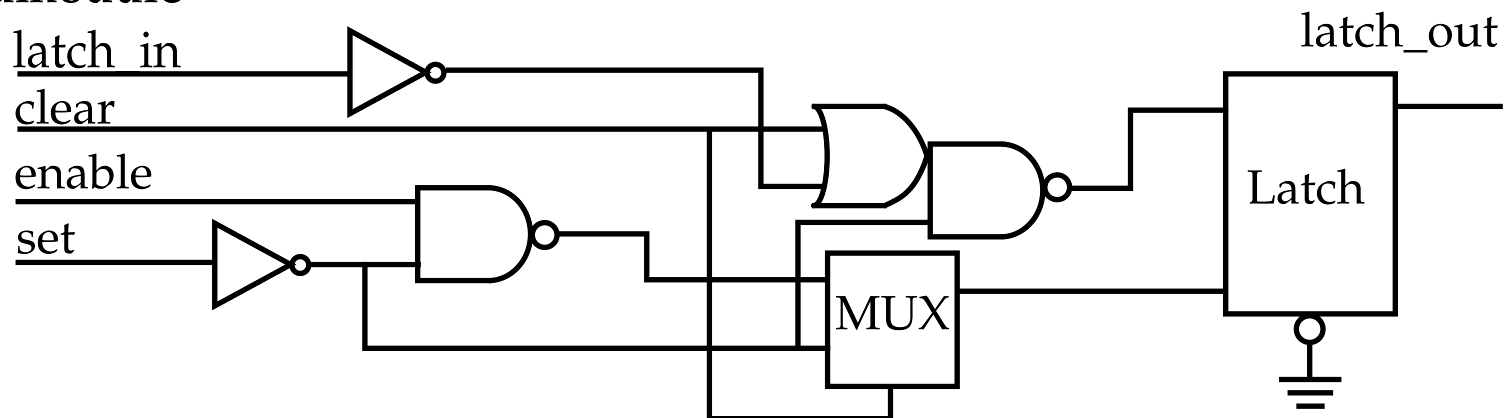


Synthesis of Latches

This one is synthesizable by all tools and generates a latch.

```
module latch_case2(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: assign latch_out = latch_in;
      3'b110: assign latch_out = 1'b1;
      3'b010: assign latch_out = 1'b1;
      3'b101: assign latch_out = 1'b0;
      3'b001: assign latch_out = 1'b0;   // Incomplete specification
   endcase
endmodule
```



Synthesis of Latches

As another example:

```
module latch_if1(data_out, data_in, latch_enable);
   input [3:0] data_in;
   output [3:0] data_out;
   input latch_enable;
   reg [3:0] data_out;                                  // MUX with feedback

   always @(latch_enable or data_in)
      if (latch_enable) data_out = data_in;
      else data_out = data_out;
endmodule

assign data_out[3:0] = latch_enable ? data_in[3:0] : data_out[3:0];


module latch_if2(data_out, data_in, latch_enable);
   input [3:0] data_in;
   output [3:0] data_out;
   input latch_enable;                                  // Incompletely specified
   reg [3:0] data_out;                                  // yields an array of latches.

   always @(latch_enable or data_in)
      if (latch_enable) data_out = data_in;
endmodule
```

Synthesis of FFs

A register variable will be synthesized into a FF (a memory element) if

- It 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 will be synthesized as the output of a FF when its value is assigned synchronously with the edge of a signal.

By decoding signals *immediately after* the event control expression allows the synthesis tool to determine:

- Which of the edge-sensitive signals are control signals
- Which is the synchronizing signal.

If the event control expression is sensitive to the edge of *more than one* signal, an **if** stmt *must* be the first stmt in the behavior.

The control signals *must* be decoded explicitly in the branches of the **if** stmt.

Decode the *reset* condition first.

Synthesis of FFs

The synchronizing signal is **not** tested explicitly in the body of the **if** stmt, but by default, the last branch must describe the synchronous activity.

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

  always @(posedge clk)
    begin
      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;          //These assignments implicitly
          data_a <= data_b;          // synchronized with rising edge
        end                          // of clk
    end
endmodule
```

See text for synthesized circuit, which includes two FFs and logic to explicitly decode *set1* and *set2*.

The outputs of the FFs are feedback to implement **else**.

Synthesis of FFs

As we have seen, not every register variable synthesizes to a hardware storage device, e.g., the *nand* gate example covered previously.

Here *temp* is used only within the behavior and is not referenced before it is assigned value.

In this example, the synthesis tool is able to correctly infer the need for a resettable FF.

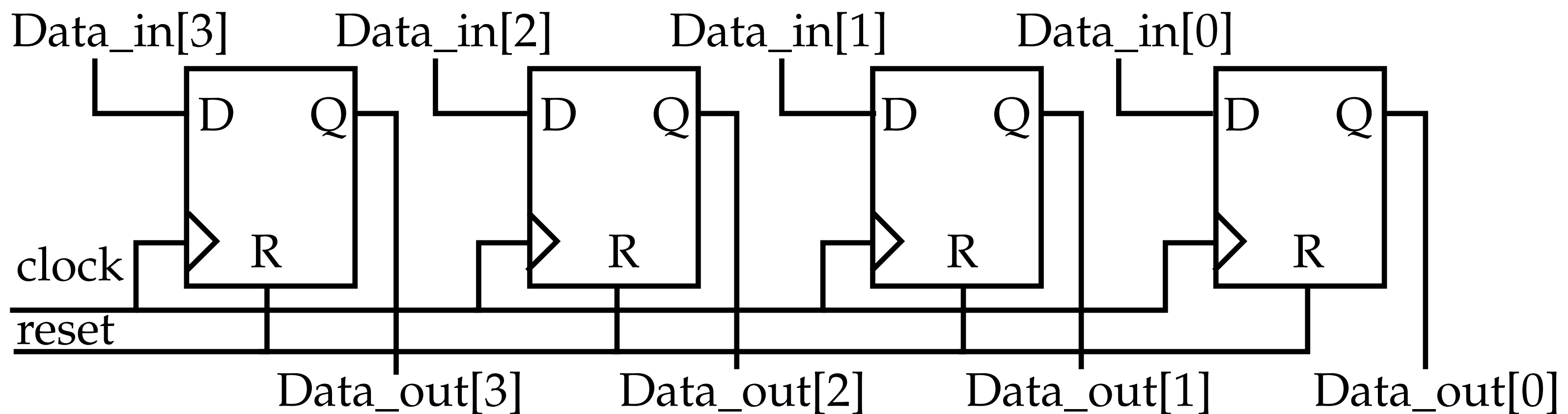
```
module D_reg4_a(Data_in, clock, reset, Data_out);
   input [3:0] Data_in;
   input clock, reset;
   output [3:0] Data_out;
   reg [3:0] Data_out;

   always @(posedge clock or posedge reset)
      begin
         if (reset == 1'b1) Data_out <= 4'b0;
         else Data_out <= Data_in;
      end
endmodule
```

Synthesis of FFs

An alternative, that may not be synthesizable:

```
module D_reg4_b(Data_in, clock, reset, Data_out);
   input [3:0] Data_in;
   input clock, reset;
   output [3:0] Data_out;
   reg [3:0] Data_out;
   always @(posedge clock)
      begin
         Data_out <= Data_in;
      end
   always @(reset)
      if (reset) assign Data_out <= 4'b0; else deassign Data_out;
endmodule
```

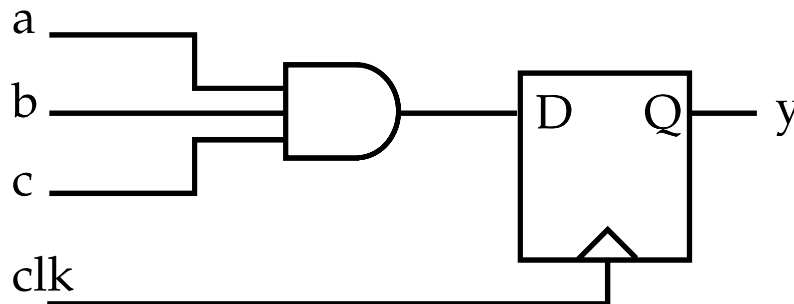


Registered Combinational Logic

When a cyclic behavior that implements combinational logic is changed to be sensitive *only* to the clk signal, the combinational logic output is registered.

```
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
```

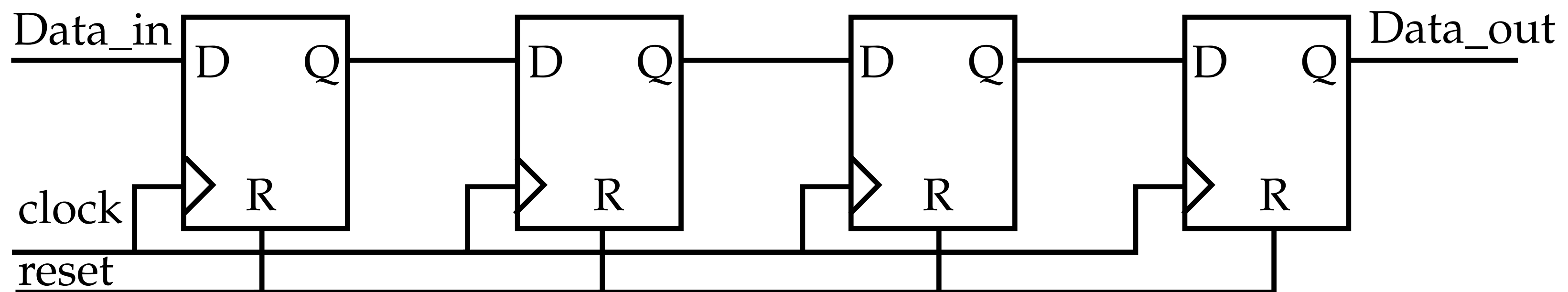


Shift Registers and Counters

```
module Shift_reg4(Data_in, Data_out, clock, reset);
   input Data_in, clock, reset;
   output Data_out;
   reg [3:0] Data_reg;

   assign Data_out = Data_reg[0];

   always @(negedge reset or posedge clock)
      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
```



Shift Registers and Counters

```
module ripple_counter(clock, toggle, reset, count);
  input clock, toggle, reset;
  output [3:0] count;
  reg [3:0] count;
  wire c0, c1, c2;

  assign c0 = count[0];  // Synthesis tool requires a simple variable in
  assign c1 = count[1];  // event control expression -- no bit-select.
  assign c2 = count[2];

  always @(posedge reset or posedge clock)
    begin
      if (reset == 1'b1) count[0] <= 1'b0;
      else if (toggle == 1'b1) count[0] <= ~count[0];
    end

  always @(posedge reset or negedge c0)  // Ripple effect with c0
    begin
      if (reset == 1'b1) count[1] <= 1'b0;
      else if (toggle == 1'b1) count[1] <= ~count[1];
    end
   ...

  endmodule
```

Implemented with FFs and XOR gates -- see text.

Synthesis

Output variables that are assigned values within a synchronized behavior will be synthesized as FFs or latches (depending on edge or level stmt).

A signal that is assigned value

- outside a behavior
- or within a behavior that *does not* include a synchronizing signal in its event control expression

will be synthesized as combinational logic, provided that it does not have an incomplete **if** or **case** operator.

Synthesis of FSMs

Verilog offers several options for modeling the combinational part, as we have seen in previous examples.

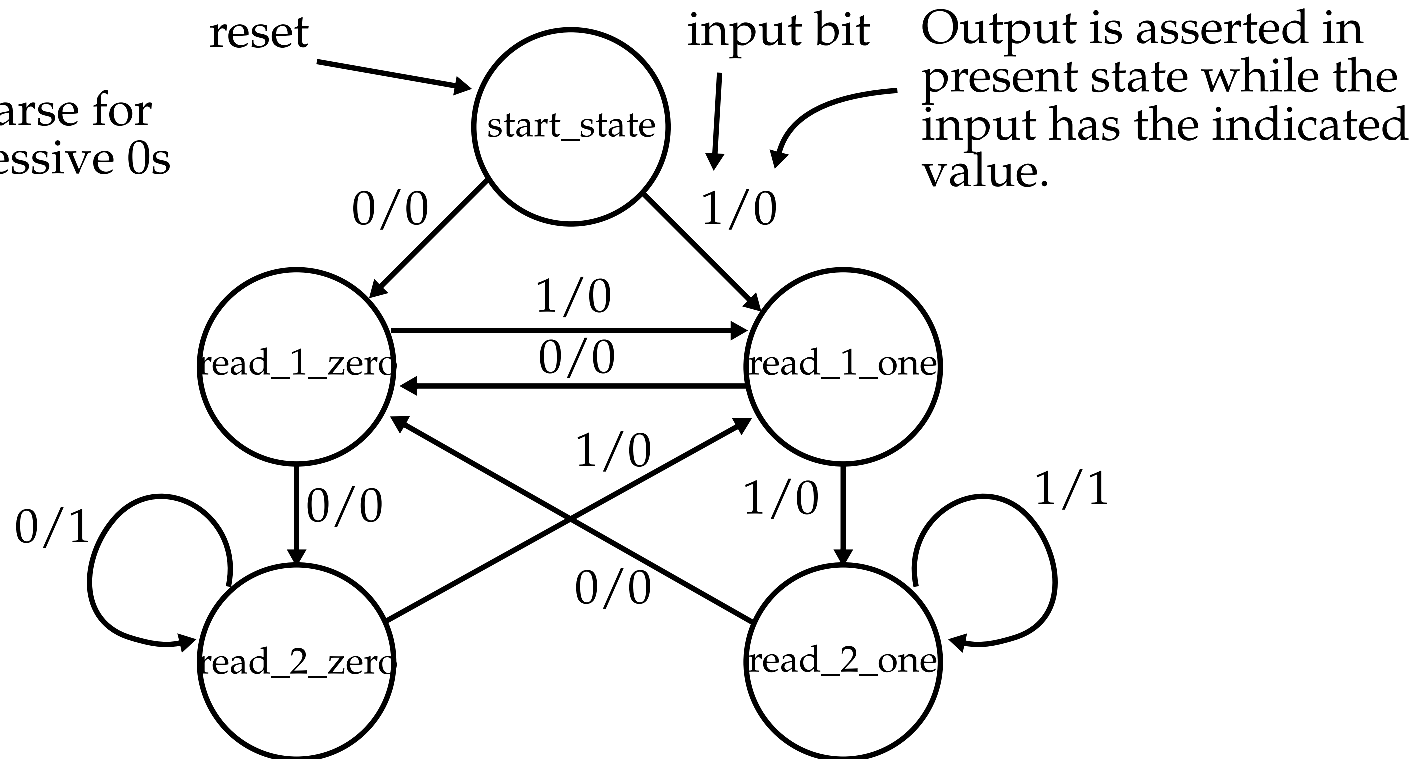
- Continuous assignment stmts
- A behavior whose event control expression consists of an 'event-or' of the state and inputs.

Synthesis of FSMs

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

Also, *non-blocking* assignments should be used, and you cannot mix non-blocking and blocking assignments to the same variable.

FSM to parse for two successive 0s or 1s.



FSM to Parse two 0s or 1s: Explicit Mealy

Since the machine is active on the rising edge of clk, the input is synchronized to change on the falling edge of the clk.

```
module seq_det_mealy_1exp(clk, reset, in_bit, out_bit);
   input clk, reset, in_bit;
   output out_bit;
   reg [2:0] 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)
      if (reset == 1) state <= start_state;
      else state <= next_state;

   always @(state or in_bit)  // Asynchronous logic
      case (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;
```

FSM to Parse two 0s or 1s: Explicit Mealy

```
            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

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

      endmodule
```

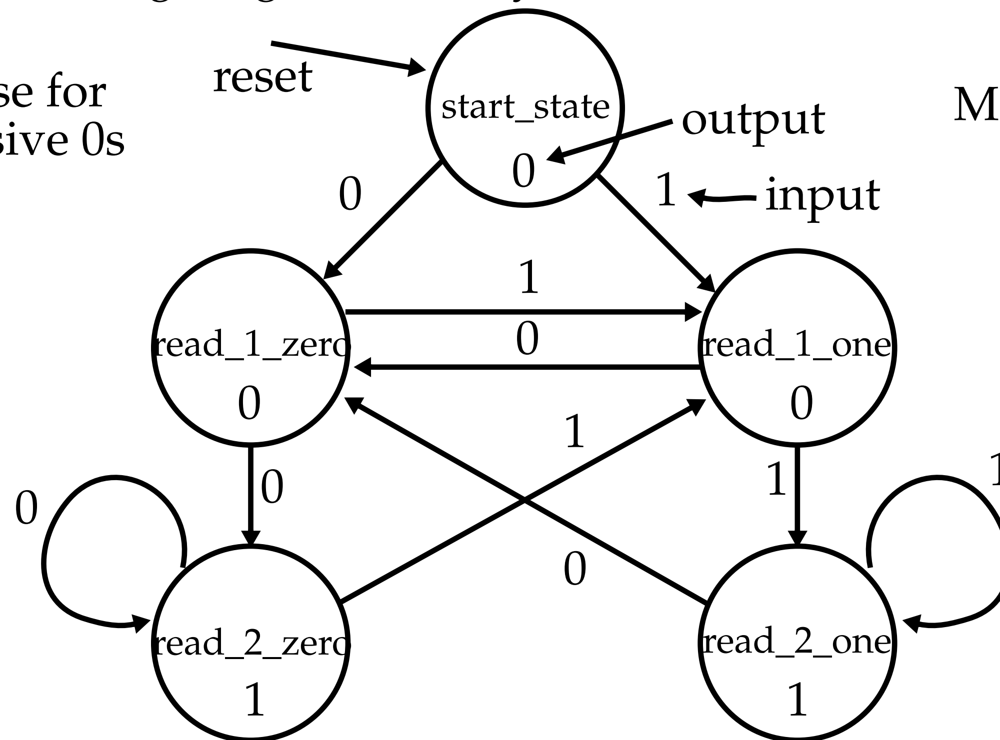
FSM to Parse two 0s or 1s: Explicit Mealy

next_state is set asynchronously (actually on the falling edge of clk when the *in_bit* value changes) since the event control expr includes *in_bit*.

out_bit can go high only on the rising edge of clk, b/c the continuous assignment includes *state*, but can go low asynchronously (thus Mealy).

See text for timing diagrams and synthesis result.

FSM to parse for two successive 0s or 1s.



Moore machine

FSM to Parse two 0s or 1s: Explicit Moore

Here, *out_bit* can only change at the state boundaries (rising edge of clk).

```
module seq_det_moore_1exp(clk, reset, in_bit, out_bit);
   input clk, reset, in_bit;
   output out_bit;
   reg [2:0] 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)
      if (reset == 1) state <= start_state;
      else state <= next_state;

   always @(state or in_bit)  // Asynchronous logic
      case (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;
```

FSM to Parse two 0s or 1s: Explicit Moore

```
        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

     assign out_bit = ((state == read_2_zero) ||
        (state == read_2_one)) ? 1 : 0;

  endmodule
```

Synthesis of Explicit State Machines

The state register of an explicit state machine *must* be assigned as an aggregate -- bit-select and part-select assignments are not allowed.

Asynchronous control signals, e.g., *set* and *reset must* be scalars in the event control expression.

The value assigned to a state register *must* be a constant or a variable that evaluates to a constant after static evaluation.

The state transition diagram must specify a **fixed** relationship.

state_reg == data; // NOT allowed

state_reg == state_reg + 1; // allowed

The synchronous activity of an explicit state machine may contain only one clk-synchronized event control expression.