# Module Review Solutions

1. For what conditions does logic synthesis infer a latch in logic you intend to be purely combinational?
   - If you fail to specify an output value for at least one combination of input values then logic synthesis will insert a latch.
2. Explain why a synthesis tool may or may not infer a register for a blocking assignment to a variable.
   - Synthesis infers a register for a blocking assignment to a variable that occurs after the variable is already read in the execution of a sequential procedure. If the variable is written before it is read then it is only a temporary variable.
3. For what Verilog construct does synthesis infer a three-state gate?
   - Synthesis infers a three-state gate if you conditionally assign the high-impedance value to a variable.
4. In what most significant way does synthesis restrict a **for** loop?
   - You can synthesize a for loop if the number of iterations is a compile-time constant. The synthesis tool unrolls the loop so needs to know the exact number of iterations.

cadence

*This page does not contain notes.*

# Module Exercise

Fix these procedures to make them legal for synthesis:

```
// 1
always @(posedge clk)
  if (clk)
    q <= d;
  else
    if (rst)
      q <= 0;
```

```
// 2
always @(posedge clk
      or rst)
  if (rst)
    q <= 0;
  else
    q <= d;
```

```
// 3
always @(posedge clk
      or negedge rst)
  if (rst)
    q <= 0;
  else
    q <= d;
```

```
// 4
always @(posedge clk
      || posedge rst)
  if (rst)
    q <= 0;
  else
    q <= d;
```

```
// 5
always @(posedge clk)
  if (!rst)
    q <= d;

always @(posedge rst)
  q <= 0;
```

```
// 6
always @(posedge clk
      or posedge rst)
  begin
    if (clk)
      q <= d;
    else
      q <= 0;
    if (rst)
      q <= 0;
    else
      q <= d;
  end
```

*Refer to the synthesis templates*

cadence

*This page does not contain notes.*

# Module Exercise Solution

Fix these procedures to make them legal for synthesis:

```
// 1
always @(posedge clk)
  if (/*clk*/rst)
    q <= /*d*/0;
  else
 // if (rst)
      q <= /*0*/d;
```

```
// 2
always @(posedge clk
      or posedge rst)
  if (rst)
    q <= 0;
  else
    q <= d;
```

```
// 3
always @(posedge clk
      or negedge rst)
  if (~rst)
    q <= 0;
  else
    q <= d;
```

```
// 4
always @(posedge clk
/*||*/or posedge rst)
  if (rst)
    q <= 0;
  else
    q <= d;
```

```
// 5
always @(posedge clk)
  if (/*!*/rst)
    q <= /*d*/0;
  else q <= d;
//always @(posedge rst)
// q <= 0;
```

```
// 6
always @(posedge clk
      or posedge rst)
  begin
 // if (clk)
 //   q <= d;
 // else
 //   q <= 0;
    if (rst)
      q <= 0;
    else
      q <= d;
  end
```

cadence

*This page does not contain notes.*

# Lab

Lab 13-1 Using a Component Library

- For this Lab, you demonstrate mastery of basic coding styles.
- And you code and synthesize some representative models.

cadence

Your objective for this lab is to describe design behavior for logic synthesis.

For this lab, you code and synthesize several simple models and observe the results of different coding techniques.

**Module 14**

# Designing Finite State Machines

cadence

This module more specifically presents the various ways to code a state machine for synthesis.

## Module Objective

In this module, you:

- Code state machines for synthesis

**Topics**

- FSM introduction
- Defining the FSM states
- Example read-write synchronizer FSM
- Coding FSMs in various styles
- Various ways to optimize FSMs

cadence

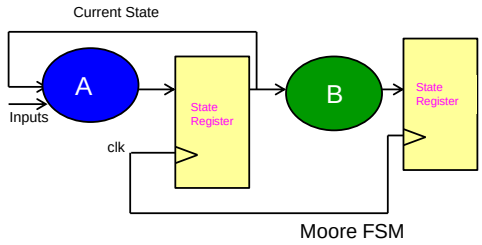
Your objective is to code state machines for synthesis.

To help you do that, this module discusses what an FSM is, how to define FSM states, various ways to code FSMs for synthesis, and various ways to optimize FSMs.

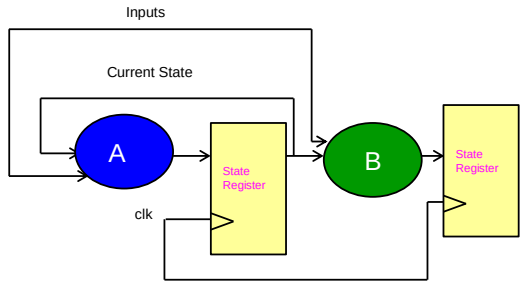
## Finite State Machine (FSM) Review

### FSM Structure

- State Register
  - Stores current state
- Next State decode logic
  - Decides next state based on current state and inputs
- Output Logic
  - Decodes state (or states and inputs) to produce outputs
- Outputs from the FSM can be a function of:
  - Current state only – **Moore**
  - Current state and the current inputs – **Mealy**




Moore FSM




Mealy FSM

266 © Cadence Design Systems, Inc. All rights reserved.

cadence

An FSM consists of state-encoding combinational logic, state-storing sequential elements, and output-decoding combinational logic.

- A Moore machine has no combinational path from inputs to outputs. The outputs are a function solely of the current state vector.
- A Mealy machine has at least one combinational from an input to an output. The outputs are a function of the current state and at least one current input. Mealy outputs can be available up to one clock earlier path than Moore outputs, but may complicate synthesis timing constraint definition.

If the current state of your design depends at least partially upon the previous state, then your design is an FSM.

---

Perhaps one way to remember which is the Mealy machine and which is the Moore machine is the phrase “Mealy is more and Moore is less”. This refers to the outputs, which in the Mealy machine can include unregistered inputs.

## Defining the FSM States

You will likely see FSM descriptions that define state vector values with text replacement macros. This training does not do that and you should neither.

**Macros**

```
`define IDLE  2'd0
`define READ  2'd1
`define WRITE 2'd2
`define DONE  2'd3
reg [1:0] state, nstate;

always @*
 case ( state )
  `IDLE: nstate = do_write ? `WRITE
                           : `READ;
 ...
```

- Scope is global – across files and modules from `define to `undef
- Accepted by synthesis tools but not for FSM optimization

**Parameters**

```
localparam IDLE  = 2'd0,
           READ  = 2'd1,
           WRITE = 2'd2,
           DONE  = 2'd3;
reg [1:0] state, nstate;

always @*
 case ( state )
  IDLE: nstate = do_write ? WRITE
                          : READ;
 ...
```

- Scope is local to declaring block
- Required by synthesis tools that perform FSM optimizations



The scope of any compiler directive is from the point of the direction to the point of its redirection or removal, potentially across multiple files and multiple modules. To inadvertently use a macro defined elsewhere or to define a macro inadvertently used elsewhere is very easy. To help prevent such erroneous use, designers typically place text macro definitions in a separate file that any compilation unit that uses the definitions includes during compilation. This encapsulates those definitions in a single easily-modified place.

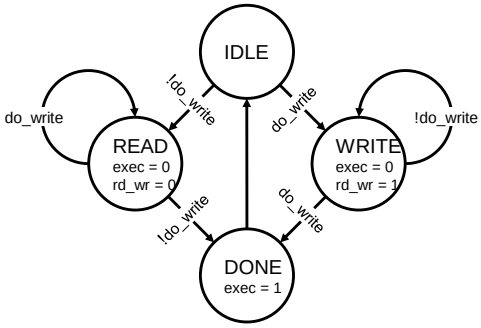
The scope of a local parameter is the declaring block. No code can change its initialized value.

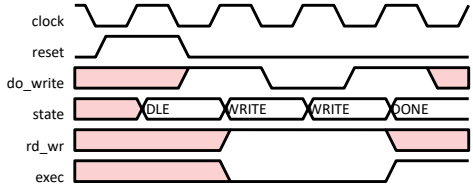
Synthesis tools that perform state machine optimizations require you to define the states with parameters if you want the tool to perform the optimizations.

## Example Read-Write Synchronizer FSM

The following examples refer to this FSM specification:

- If do_write is true, transition to WRITE and set exec = 0 and rd_wr = 1.
  - When do_write is again true, transition to DONE and set exec = 1.
- If do_write is false, transition to READ and set exec = 0 and rd_wr = 0.
  - When do_write is again false, transition to DONE and set exec = 1.



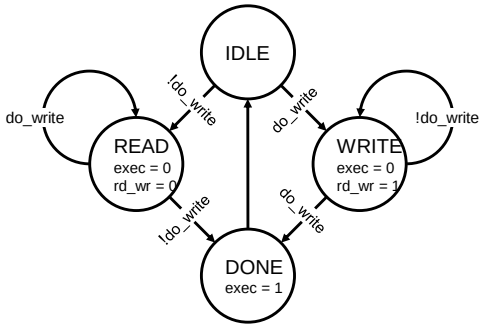

Outputs(exec and rd_wr) are considered unknown at reset for simplicity, in reality it may be either 1 or 0.

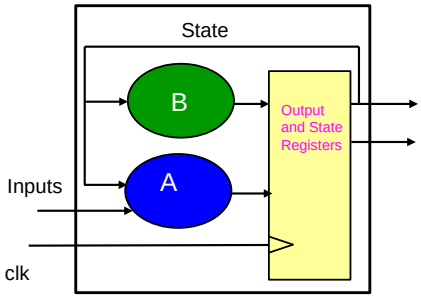


The following examples refer to this state machine specification:

- If do_write is true, it transitions to the WRITE state and sets exec to 0 and rd_wr to 1. When do_write is again true, it transitions to the DONE state and sets exec to 1.
- If do_write is false, it transitions to the READ state and sets exec to 0 and rd_wr to 0. When do_write is again false, it transitions to the DONE state and sets exec to 1.

## Coding the FSM in One Block: Sequential Outputs



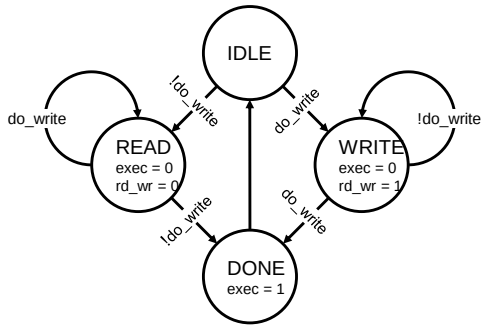

```
localparam IDLE  = 2'd0, READ = 2'd1,
           WRITE = 2'd2, DONE = 2'd3;
reg [1:0] state;
reg exec, rd_wr;

always @(posedge clock)
 if ( reset )
  state <= IDLE;
 else
  case ( state )
   IDLE: begin
           exec  <= 0;
           rd_wr <= do_write;
           state <= do_write ? WRITE
                             : READ;
         end
   READ:  if ( !do_write )
            {state,exec}<={DONE,1'b1};
   WRITE: if (  do_write )
            {state,exec}<={DONE,1'b1};
   DONE:  state <= IDLE;
  endcase
```

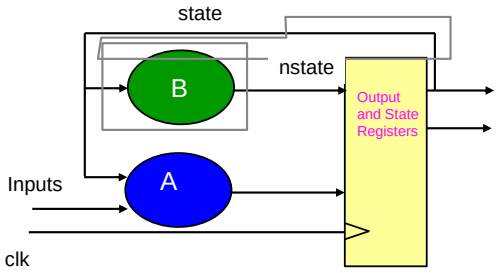
269 

cadence

The one-block coding style generates the next-state, state, and outputs all in one sequential block.

## Coding the FSM in Two Blocks: Sequential Outputs



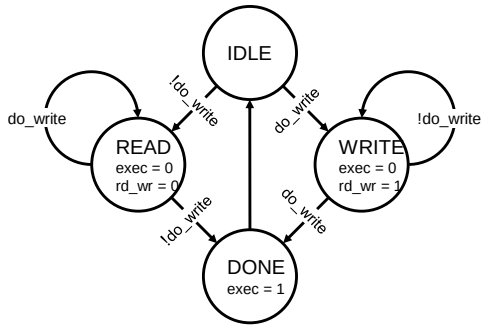

```
localparam IDLE  = 2'd0, READ = 2'd1,
           WRITE = 2'd2, DONE = 2'd3;
reg [1:0] state, nstate;
reg exec, rd_wr;

always @*
 case ( state )
  IDLE:  nstate=  do_write? WRITE: READ;
  READ:  nstate= !do_write? DONE : READ;
  WRITE: nstate=  do_write? DONE : WRITE;
  DONE:  nstate= IDLE;
 endcase

always @(posedge clock)
 if ( reset )
  state <= IDLE;
 else
  begin
   state <= nstate;
   case ( nstate )
    READ:  {rd_wr,exec} <= 2'b00;
    WRITE: {rd_wr,exec} <= 2'b10;
    DONE:  exec <= 1;
   endcase
  end
```

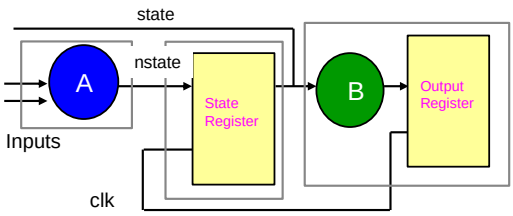
270 

cadence

The two-block coding style encodes the next state in a separate combinational block. You must place asynchronous reset in the sequential block. You can place synchronous reset in either block, but synthesis tools might more accurately identify the component's reset pin if you place the reset in the sequential block.

# Coding the FSM in Three Blocks: Sequential Outputs



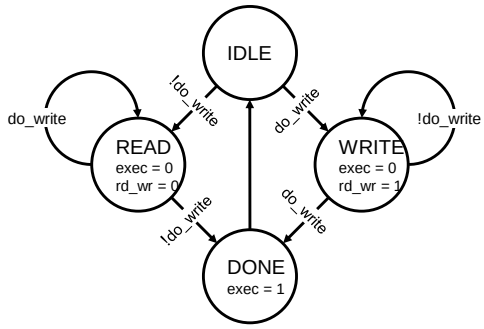



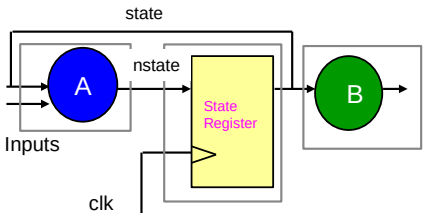

```
localparam IDLE  = 2'd0, READ = 2'd1,
           WRITE = 2'd2, DONE = 2'd3;
reg [1:0] state, nstate;
reg exec, rd_wr;

always @*
 case ( state )
  IDLE:  nstate=  do_write? WRITE: READ;
  READ:  nstate= !do_write? DONE : READ;
  WRITE: nstate=  do_write? DONE : WRITE;
  DONE:  nstate= IDLE;
 endcase

always @(posedge clock)
 if ( reset )
  state <= IDLE;
 else
   state <= nstate;

always @(posedge clock)
 if ( !reset )
  case ( nstate )
    READ:  {rd_wr,exec} <= 2'b00;
    WRITE: {rd_wr,exec} <= 2'b10;
    DONE:  exec <= 1;
   endcase
```



The three-block coding style decodes the outputs in a third block. Here, that third block is sequential. Note that the outputs do not transition while the reset is active.

## Coding in Three Blocks: Combinational Outputs




```
localparam IDLE  = 2'd0, READ = 2'd1,
           WRITE = 2'd2, DONE = 2'd3;
reg [1:0] state, nstate;
reg exec, rd_wr;

always @*
 case ( state )
  IDLE:  nstate=  do_write? WRITE: READ;
  READ:  nstate= !do_write? DONE : READ;
  WRITE: nstate=  do_write? DONE : WRITE;
  DONE:  nstate= IDLE;
 endcase

always @(posedge clock)
 if ( reset )
  state <= IDLE;
 else
  state <= nstate;

always @*
 case ( state )
  IDLE:  {rd_wr,exec} = 2'bxx;
  READ:  {rd_wr,exec} = 2'b00;
  WRITE: {rd_wr,exec} = 2'b10;
  DONE:  {rd_wr,exec} = 2'bx1;
 endcase
```



The three-block coding style decodes the outputs in a third block. Here, that third block is combinational. The combinational block takes advantage of the specification that for some states we do not care what the output value is.

# Optimizing Register Count

- Registering outputs is preferable for synthesis as it simplifies timing constraints.
- Examine the mapping between state and output.
- Can output registers replace state bits? If so then you can reduce your overall register count.

| State | Encoding | rd_wr | exec |
|---|---|---|---|
| IDLE | 01 | ? | ? |
| READ | 00 | 0 | 0 |
| WRITE | 10 | 1 | 0 |
| DONE | 11 | ? | 1 |
| match: | | state[1] | state[0] |

```
localparam READ  = 2'd0, IDLE = 2'd1,
           WRITE = 2'd2, DONE = 2'd3;
reg [1:0] state, nstate;
reg exec, rd_wr;

always @*
 case ( state )
  IDLE:  nstate=  do_write? WRITE: READ;
  READ:  nstate= !do_write? DONE : READ;
  WRITE: nstate=  do_write? DONE : WRITE;
  DONE:  nstate= IDLE;
 endcase

always @(posedge clock)
 if ( reset )
  state <= IDLE;
 else
  state <= nstate;

always @*
 begin
  rd_wr = state[1];
  exec  = state[0];
 end
```

cadence

*This page does not contain notes.*

## Optimizing Power and Noise: Gray Encoding

Where vectors regularly traverse a range of values, opportunities exist to encode values in consideration of power efficiency and noise reduction. Perhaps the most classic of examples is Gray-encoded counting.

| Binary | Transitions |
|---|---|
| **00** | 2 |
| **01** | 1 |
| **10** | 2 |
| **11** | 1 |
| mean: | 1.5 |

| Gray | Transitions |
|---|---|
| **00** | 1 |
| **01** | 1 |
| **11** | 1 |
| **10** | 1 |
| mean: | 1 |

Gray code effective for predictable state patterns with a specific number of state transitions.

| State | Transitions |
|---|---|
| **(00)IDLE** | 2 |
| **(01)READ** | 1 |
| **(11)DONE** | 1 |
| mean: | 1.3 |

| State | Transitions |
|---|---|
| **(00)IDLE** | 2 |
| **(10)WRITE** | 1 |
| **(11)DONE** | 1 |
| mean: | 1.3 |

cadence

Bell Labs researcher Frank Gray applied for patent 2,632,058 “Pulse Code Communication” in 1947, defining a “reflected binary code” for lack of a better term. Coincidentally with the patent award, the codes became popularly known as Gray codes. A Gray code is “a binary numeral system where two successive values differ in only one digit” – http://en.wikipedia.org/wiki/Gray_code

The tabulated sequence is the original “binary reflected Gray code” (that you can generate from binary using g=b^(b>>1))but other Gray codes also exist.

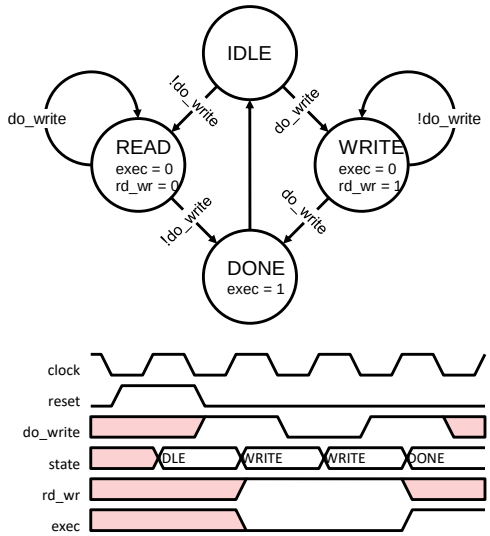
While initially used to work around switch-bounce, as the new vector value can be used immediately without further waiting for the value to “settle”, gray-encoding is indispensable for address counters that increment in one clock domain and are read in another, for example in asynchronous FIFOs.

Gray-encoding has more recently become valuable in the quest for reduced switching activity. The simple counting sequence displayed here averages 1.5 transitions per clock for binary encoding and 1 transition per clock for Gray encoding.

Paths through the example machine have a length of only three states, so present little opportunity to reduce switching activity.

# Optimizing Performance: One-Hot Encoding

Reducing state encoding/decoding shortens timing paths to increase performance.




```
localparam IDLE  = 4'd1, READ = 4'd2,
           WRITE = 4'd4, DONE = 4'd8;
reg [3:0] state, nstate;
reg exec, rd_wr;

always @*
 case ( state )
  IDLE:  nstate=  do_write? WRITE: READ;
  READ:  nstate= !do_write? DONE : READ;
  WRITE: nstate=  do_write? DONE : WRITE;
  DONE:  nstate= IDLE;
 endcase

always @(posedge clock)
 if ( reset )
  state <= IDLE;
 else
  state <= nstate;

always @*
 begin
  rd_wr = state == WRITE;
  exec  = state == DONE;
 end
```

One-Hot is effective for any number of arbitrary transition patterns.

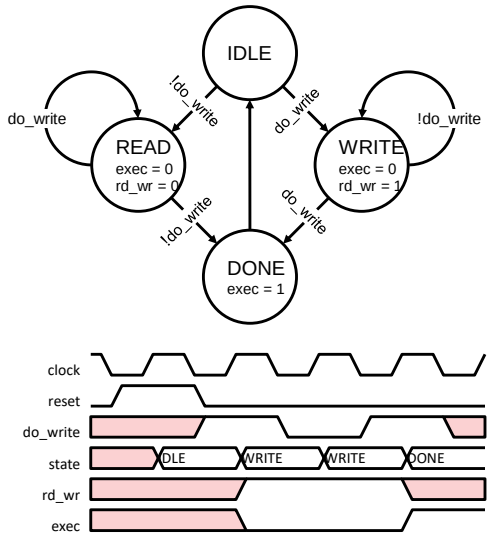


cadence

One-hot state encoding provides a unique state bit for each state. This reduces the combinational logic required to encode and decode the state, thus shortens the timing path to permit increased clock speed.

# One-Hot Encoding by Indexing State Bits

Indexing state bits more clearly reveals the reduced encoding.




```
localparam IDLE  = 0, READ = 1,
           WRITE = 2, DONE = 3;
reg [3:0] state, nstate;
reg exec, rd_wr;

always @*
 begin
  nstate[READ]= state[IDLE] && !do_write
              || state[READ] &&  do_write;
  nstate[WRITE]=state[IDLE] &&  do_write
              || state[WRITE] && !do_write;
  nstate[DONE]=state[READ]  && !do_write
              || state[WRITE] &&  do_write;
  nstate[IDLE] = state[DONE];
 end

always @(posedge clock)
 begin
  state[IDLE] <=  reset || nstate[IDLE];
  state[READ] <= !reset && nstate[READ];
  state[WRITE]<= !reset && nstate[WRITE];
  state[DONE] <= !reset && nstate[DONE];
 end

always @*
 begin
  rd_wr = state[WRITE];
  exec  = state[DONE];
 end
```

cadence

You can code the one-hot machine in a style that individually references each state bit. This style may more clearly reveal the reduced state encoding. Here, the longest path to any state bit is a sum of two products of three terms each.

## Module Summary

You should now be able to code state machines for synthesis.

This module discussed:

- What is a FSM
  - State encoding, storage, and decoding
- How to define FSM states
  - By use of local constants
- Various ways to code FSMs for synthesis
  - 1-block, 2-block, 3-block
- Various ways to optimize FSMs
  - For area, power, noise, performance

cadence

You should now be able to code state machines for synthesis.

This module discussed what an FSM is, how to define FSM states, various ways to code FSMs for synthesis, and various ways to optimize FSMs.

## Module Review

1. What is the difference between a Moore machine and a Mealy machine?
2. Why should you define state values using parameters instead of macros?
3. In how many blocks should you define the FSM to generate the best quality netlist?

cadence

*This page does not contain notes.*