

FSM-Design

There are at least seven different Finite State Machine (FSM) design techniques. This [FSM-Design \(https://github.com/Quy0109/FSM-Design\)](https://github.com/Quy0109/FSM-Design) repository will describe four of the FSM design techniques:

- 1 Always Block Style with registered outputs,
- 2 Always Block Style with combinatorial outputs,
- 3 Always Block Style with registered outputs, and
- 4 Always Block Style with registered outputs

1. Introduction

This repository covers four of the seven commonly taught FSM (Finite State Machine) design techniques. The selected techniques are highlighted below:

- ☒ **2 Always Block Coding Style** with combinatorial outputs
- ☒ **1 Always Block Coding Style** with registered outputs
- ☒ **3 Always Block Coding Style** with registered outputs
- ☒ **4 Always Block Coding Style** with registered outputs (new style, not previously shown)
- ☐ Indexed One-Hot Coding Style with registered outputs
- ☐ Parameter One-Hot Coding Style with registered outputs
- ☐ Output Encoded Coding Style with registered outputs

Registering module outputs is typically recommended by synthesis tool vendors, as it helps achieve timing goals more consistently without requiring numerous input and output timing constraints. Registered outputs are also glitch-free, ensuring stable and reliable operation.

2. Important design goals

2.1. Reduce debugging time by following RTL coding guidelines

Industry trend studies conducted by a leading research group have shown consistent design and verification trends for ASIC and FPGA projects since 2010. Seminars held worldwide in 2010 highlighted that the primary factor causing projects to fall behind schedule was the time spent on debugging.

Research studies from 2010, 2012, 2014, 2016, and 2018 have consistently confirmed that debugging consumes the most significant amount of verification engineering time. Since 2014, debugging has required approximately 95% more effort on average than any other verification task.

These findings make it clear that any coding practices that can reduce debug time are valuable to adopt. The RTL coding guidelines presented in this repository aim to help reduce debug time and increase project efficiency.

2.2. MORE LINES OF CODE = MORE BUGS!!!

It's often suggested that more lines of code lead to more bugs. For this reason, I believe concise coding styles that adhere to defensive coding principles—either to avoid bugs or to enable their early detection

—are highly valuable, particularly in RTL design and FSM design. In this repository, I highlight where concise coding styles are used and how certain practices help make bugs easier to identify.





3. Evaluation Criteria

To determine what makes a good coding style, we have selected the following goals to evaluate different FSM (Finite State Machine) coding styles:

1. The FSM coding style should be easily modifiable, allowing for changes in state encodings and FSM structures.
2. The coding style should be concise.
3. The coding style should be straightforward to write and understand.
4. The coding style should facilitate debugging.
5. The coding style should yield efficient synthesis results.
6. The coding style should support easy modifications to the FSM, including adjustments to the number of inputs and outputs.

Each coding style is assessed based on these criteria, with the results summarized in the table below:

Goals	2 Always, comb outputs	1 Always, reg outputs	3 Always, reg outputs	4 Always, reg outputs
Easily modifiable state encodings	✓	✓	✓	✓
Concise	✗	✓	✓	✓
Easy to understand	✗	✓	✓	✓
Facilitates debugging	✓	✓	✓	✓
Efficient synthesis	✓	✗	✓	✓
Easily modifiable for FSM changes	✗	✓	✓	✓

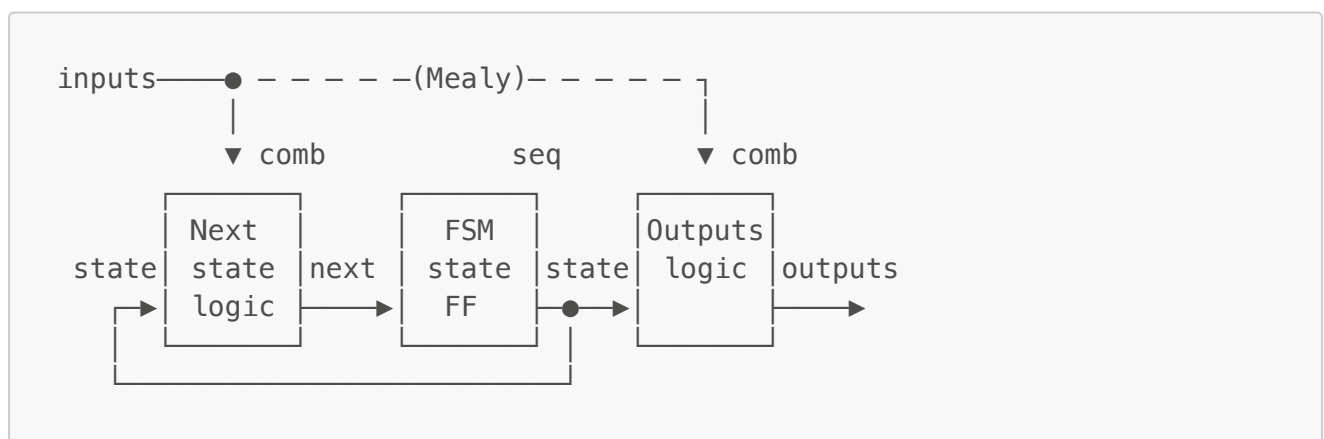
Goals	2 Always, comb outputs	1 Always, reg outputs	3 Always, reg outputs	4 Always, reg outputs
	 Not generally recommended but useful for simple designs where combinational logic for outputs is manageable	 Recommended for general use	 Efficient for FSMs with many states and transitions; however, the 4 Always, reg outputs is often preferable	 Initially complex but ideal for large-scale FSMs; may be overkill for smaller FSMs
Conclusion				

Summary of Coding Goals

4. State machine types

4.1. Moore and Mealy

A Moore state machine is classified as an FSM where the outputs are only a function of the present state, while Mealy state machines have one or more outputs that are a function of the present state and one or more FSM inputs.



In industry, Moore state machines are often preferred because their outputs have a full clock cycle to settle through combinatorial logic, making it easier to meet required timing constraints. In contrast, Mealy state machines allow an input to affect the output mid-cycle. This requires the input to traverse the combinatorial logic and meet setup timing within the same cycle, adding complexity and risk to timing closure.

When a design needs an input to be processed and reflected in the output within a single cycle, Mealy outputs are sometimes used. However, such cases are typically rare, as they come with stricter timing demands. The phrase "Moore is Less" reflects that Moore state machines depend only on the current state, while Mealy state machines depend on both the current state and one or more inputs.

In general, Mealy FSM designs are avoided unless strictly necessary to meet specific design requirements.

4.2. State encoding

FSM (Finite State Machine) encoding refers to the method of representing the states of a finite state machine in binary form. This representation is essential for implementing FSMs in digital circuits, as it determines how the states are stored and how transitions between states occur based on input conditions.

Key Aspects of FSM Encoding:

- 1. State Representation:** Each state of the FSM is assigned a unique binary code. The choice of encoding scheme directly affects how efficiently states can be represented and how easily they can be transitioned between.
- 2. Transition Logic:** The way states are encoded influences the complexity of the combinational logic required to transition from one state to another. Different encoding schemes can simplify or complicate the transition logic based on how states are represented.
- 3. Output Generation:** In many FSM implementations, outputs depend on the current state. The encoding scheme affects how outputs are generated based on the state values.
- 4. Design Constraints:** The encoding scheme chosen can impact various design constraints, including resource usage (number of flip-flops, gates), timing performance (how quickly the FSM can change states), and debugging complexity.

Choosing the appropriate encoding scheme is crucial for the overall performance and reliability of digital systems. It affects:

- Resource Utilization:** Efficient encoding can minimize the number of flip-flops and gates needed, which is especially important in resource-constrained designs like FPGAs.
- Timing Performance:** Certain encoding schemes allow for faster transitions and simpler logic, helping to meet timing constraints in high-speed applications.
- Debugging Ease:** Well-structured encoding schemes can simplify the debugging process, making it easier to identify and correct errors in the FSM.

In summary, FSM encoding is a fundamental aspect of digital design that directly impacts the efficiency, performance, and reliability of finite state machines in various applications. Each FSM encoding scheme comes with its own set of advantages and disadvantages, which influence the choice of scheme based on specific project requirements. Here's a more detailed breakdown of the trade-offs of some common encoding scheme:

Encoding Scheme	Description	Advantages	Disadvantages	Common Use Cases
Binary	Uses the minimum number of bits to represent all states in binary form.	Minimizes flip-flops and memory usage	Complex decoding logic	Small or resource-constrained designs

Encoding Scheme	Description	Advantages	Disadvantages	Common Use Cases
One-Hot	Uses one flip-flop per state, with only one bit high (1) per state.	Simple decoding; fast transitions	High resource usage	High-speed designs, FPGAs
Gray	Each state transition changes only one bit.	Reduces transient errors	More complex encoding/decoding	Designs with timing constraints, sequential

Comparison of FSM Encoding Schemes

Support of enumeration encoding styles differs between RTL synthesis tools, so have a look at the manual of yours for supported styles.

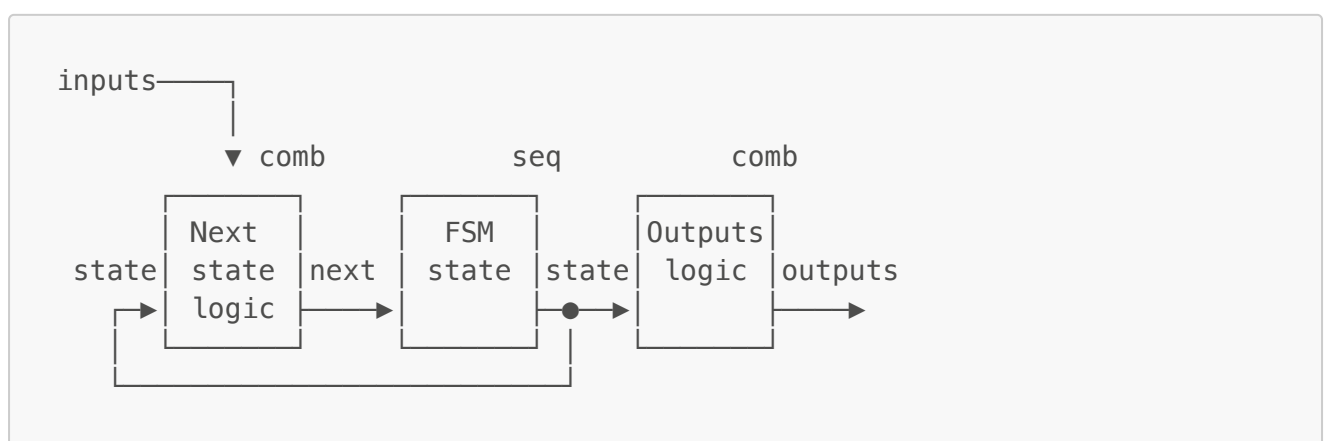
5. FSM coding style

5.1. Two Always Block FSM coding style - combinatorial outputs

The **2-always block FSM coding style** separates the state register and the next-state logic into two distinct blocks:

- The first block is a concise `always_ff` state register, typically only three lines of code, which updates the current state on each clock cycle.
- The second block is an `always_comb` block that combines both the next-state logic and combinatorial output logic.

For added flexibility, outputs can optionally be moved out of the main `always_comb` block and placed in a separate `always_comb` block or defined through continuous assignment statements. This modular approach helps in managing and organizing the next-state and output logic distinctly.



```

module fsm (
    input  clk, rst,
    output rd,
    ...

```

```

);

reg [BW-1:0] current_state, next_state;

always_ff @(posedge clk, posedge rst) begin
    if (rst) current_state <= IDLE;          // reset state
    else     current_state <= next_state;    // update state
end

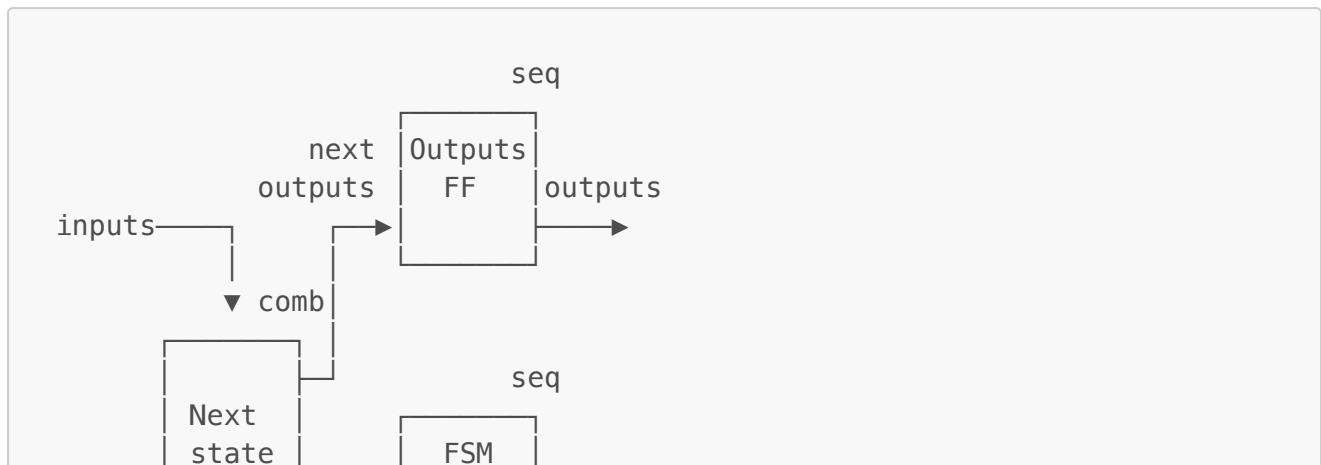
always_comb begin
    next_state = X; // next state
    rd         = 0; // output
    ...
    case (current_state)
        IDLE: begin
            // logic to change state
            if(start) next_state = READ;
            else       next_state = IDLE;
        end
        READ: begin
            next_state = ...; // logic to change state
            rd         = 1; // output depends on current_state
            ...
        end
        ...
    endcase
end

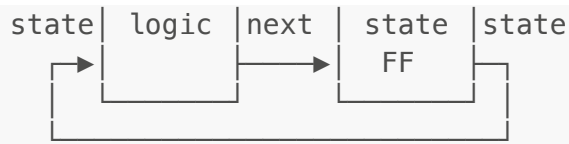
endmodule

```

5.2. One Always Block FSM coding style - registered outputs

The **1-always block FSM coding style** is a method that uses a single `always_ff` block to manage the state register, next-state assignments, and output assignments for each transition. In this approach, output assignments are set based on the *next state* rather than the current state. Therefore, when specifying outputs, designers must assign the outputs within each transition arc, not just once per state. This often results in more verbose code due to the need for explicit output assignments for every possible transition.





```

module fsm (
    input  clk, rst,
    output rd,
    ...
);

reg [BW-1:0] current_state;

always_ff @(posedge clk, posedge rst) begin
    if (rst) begin
        current_state <= IDLE;
        outputs       <= 0;
        ...
    end
    else begin
        current_state <= X;
        outputs       <= 0;
        ...
        case (current_state)
            IDLE: begin
                // logic to change state and next outputs
                if(start) begin
                    current_state <= READ; // next state being
registered          rd          <= 1;    // next output rd being
registered
                    ...
                end else begin
                    current_state <= IDLE;
                end
            end
            READ: begin
                current_state <= ...;
                ...
            end
            ...
        endcase
    end
end

endmodule

```

Advantages:

- Improved synthesis results: By assigning the next state and next outputs within the same `always_ff` block, the design generates these values in parallel, improving timing efficiency over the 2-always and 3-always block styles.

Drawbacks:

- Verbosity: The 1-always block style is more verbose because output assignments must be specified for each state transition.

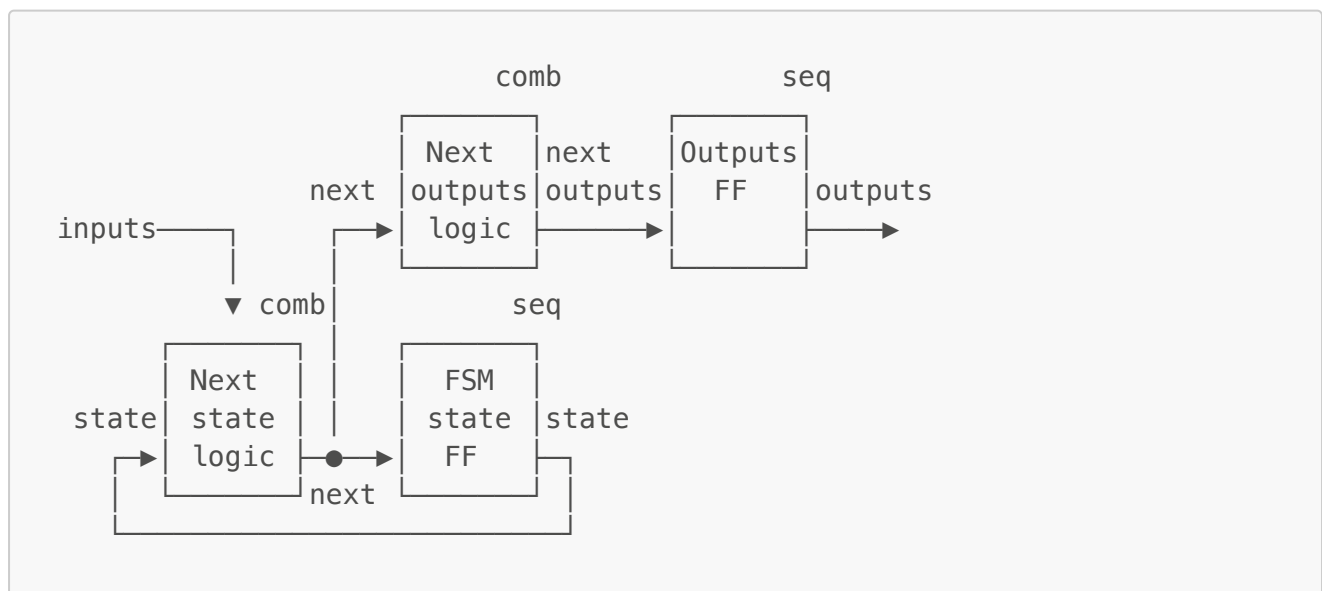
The **4-always block FSM coding style** addresses the synthesis inefficiencies of the 3-always block style by using a separate `always_comb` block for next-state outputs, which are then registered in a final `always_ff` block. This style provides a balance, combining the benefits of the 1-always style with reduced verbosity and improved synthesis results.

5.3. Three Always Block FSM coding style - registered outputs

The **3-always block FSM coding style** structures the FSM code into three distinct blocks:

1. An `always_ff` block as the state register, typically three lines of code, which updates the current state.
2. An `always_comb` block to handle the next-state logic.
3. A second `always_ff` block to compute and register the next outputs.

A key feature of this style is that outputs are assigned based on the *next state* rather than the current state. The final `always_ff` block uses a case statement to evaluate the next state and set output values accordingly.



```
module fsm (
    input  clk, rst,
    output rd,
    ...
);

    reg [BW-1:0] current_state, next_state;
```



```

always_ff @(posedge clk, posedge rst) begin
    if (rst) current_state <= IDLE;          // reset state
    else     current_state <= next_state; // update state
end

always_comb begin
    next_state = X; // next state
    case (current_state)
        IDLE: begin
            // logic to change state
            if(start) next_state = READ;
            else      next_state = IDLE;
        end
        READ: begin
            next_state = ...; // logic to change state
        end
        ...
    endcase
end

always_ff @(posedge clk, posedge rst) begin
    if (rst) begin
        rd      <= 0; // reset rd output
        outputs <= 0; // reset outputs
        ...
    end
    else begin
        outputs <= 0;
        ...
        case (next_state)
            READ: begin
                rd <= 1; // if next_state is READ, rd will be 1 the next
cycle
                ...
            end
            ...
        endcase
    end
end
endmodule

```

Synthesis Considerations:

Since outputs are calculated from the next-state logic, synthesizing this design often results in additional combinatorial logic. The next-state logic, already computed from input conditions, feeds into a second combinatorial logic block for calculating next outputs. This chaining effect can increase the complexity and delay of the combinatorial logic.

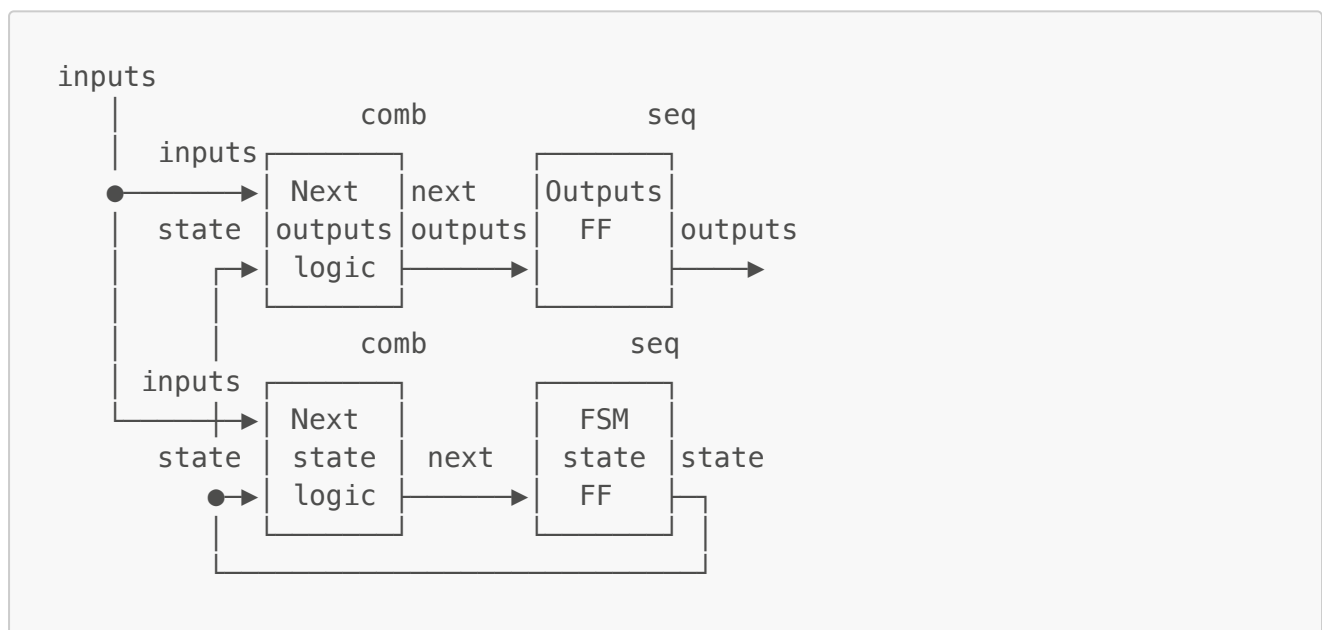
In comparison, the **1-always block coding style** evaluates both next-state and next-output values in parallel using the inputs and state variables, which can reduce the size and delay of combinatorial logic.

The **4-always block coding style** further addresses this inefficiency by separating next outputs into a dedicated `always_comb` block, which can then be registered in a final `always_ff` block, improving synthesis results.

5.4. Four Always Block FSM coding style - registered outputs

During their analysis, they observed that the **1-always block FSM coding style** required much more code but consistently achieved better synthesis results than the **3-always block style**. This difference became clear once they noted that the 3-always block style's next-output logic depends on an additional combinatorial block to compute the next state. In contrast, the 1-always block style performs next-state and next-output calculations in parallel within the same combinatorial block.

To improve efficiency, they introduced a **4-always block coding style**. Here, an `always_comb` block handles next-output values based on both the registered state and FSM inputs. The design resembles the 3-always block style but splits the final `always_ff` block into an `always_comb` for next-output logic and an `always_ff` for registering outputs. This modification delivers synthesis results comparable to the 1-always block style while requiring significantly less code.



```
module fsm (
    input  clk, rst,
    output rd,
    ...
);

reg [BW-1:0] current_state, next_state;

always_ff @(posedge clk, posedge rst) begin
    if (rst) current_state <= IDLE;      // reset state
    else    current_state <= next_state; // update state
end

always_comb begin
    next_state = X; // next state
end
```

```

case (current_state)
  IDLE: begin
    // logic to change state
    if(start) next_state = READ;
    else      next_state = IDLE;
  end
  READ: begin
    next_state = ...; // logic to change state
  end
  ...
endcase
end

always_comb begin
  next_rd      = 0;
  next_outputs = 0;
  ...
  case (current_state)
    READ: begin
      next_rd <= 1; // if current_state is READ, next_rd will be 1
      ...
    end
    ...
  endcase
end

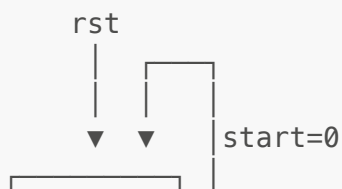
always_ff @(posedge clk, posedge rst) begin
  if (rst) begin
    rd      <= 0; // reset rd output
    outputs <= 0; // reset outputs
    ...
  end
  else begin
    rd      <= next_rd; // update rd output
    outputs <= next_outputs; // update outputs
    ...
  end
end

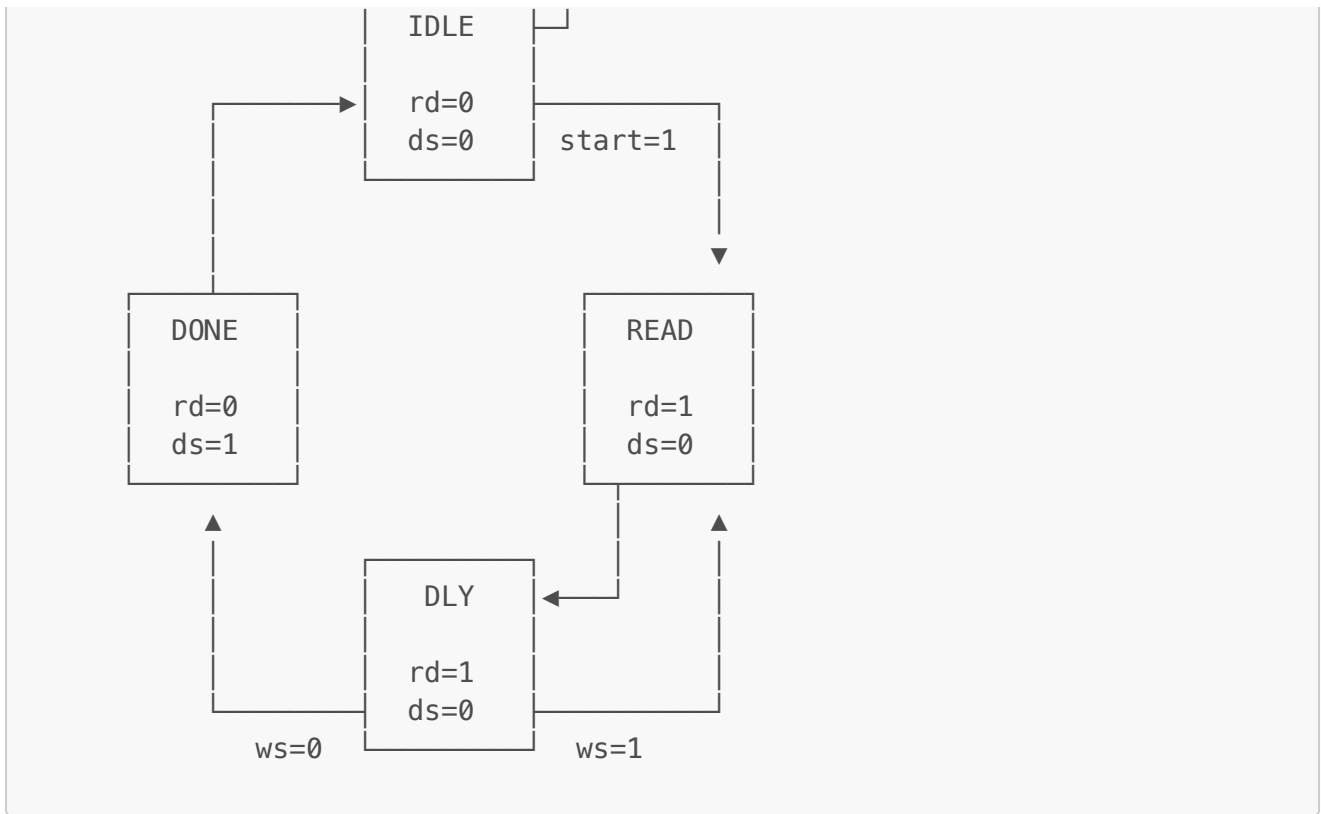
endmodule

```

6. Examples:

FSM:





Example code for schemes:

- One Always Block FSM coding style - registered outputs
 - File: [fsm_1_always.sv](#)
- Two Always Block FSM coding style - combinatorial outputs
 - File: [fsm_2_always.sv](#)
- Three Always Block FSM coding style - registered outputs
 - File: [fsm_3_always.sv](#)
- Four Always Block FSM coding style - registered outputs
 - File: [fsm_4_always.sv](#)

6.1. One Always Block FSM coding style - registered outputs

- File: [fsm_1_always.sv](#)

```

module fsm_1_always (
    output reg rd,
    output reg ds,

    input clk,
    input rst,

    input start,
    input ws
);

localparam IDLE = "IDLE";
localparam READ = "READ";

```

```

localparam DLY = "DLY";
localparam DONE = "DONE";

localparam UNKW = 'x;

reg [4*8-1:0] cstate;

always_ff @(posedge clk, posedge rst) begin
    if (rst) begin
        cstate <= IDLE;
        rd      <= 0;
        ds      <= 0;
    end else begin
        cstate <= UNKW; // Need @LB
        rd <= 0;
        ds <= 0;
        case (cstate)
            IDLE:
                if (start) begin
                    cstate <= READ;
                    rd      <= 1;
                end else cstate <= IDLE; // @LB
            READ: begin
                cstate <= DLY;
                rd      <= 1;
            end
            DLY:
                if (ws) begin
                    cstate <= READ;
                    rd      <= 1;
                end else begin
                    cstate <= DONE;
                    ds      <= 1;
                end
            DONE: cstate <= IDLE;
            default: begin
                cstate <= UNKW;
                rd      <= UNKW;
                ds      <= UNKW;
            end
        endcase
    end
end
endmodule

```

6.2. Two Always Block FSM coding style - combinatorial outputs

- File: [fsm_2_always.sv](#)

```

module fsm_2_always (
    output reg rd,
    output reg ds,

    input clk,
    input rst,

    input start,
    input ws
);

localparam IDLE = "IDLE";
localparam READ = "READ";
localparam DLY = "DLY";
localparam DONE = "DONE";

localparam UNKW = 'x;

reg [4*8-1:0] cstate;
reg [4*8-1:0] nstate;

always_ff @(posedge clk, posedge rst)
    if (rst) cstate <= IDLE;
    else cstate <= nstate;

// /*
always_comb begin
    nstate = UNKW; // Need @LB
    rd = 0;
    ds = 0;
    case (cstate)
        IDLE: nstate = start ? READ : IDLE;
        READ: begin
            nstate = DLY;
            rd      = 1;
        end
        DLY: begin
            nstate = ws ? READ : DONE;
            rd      = 1;
        end
        DONE: begin
            nstate = IDLE;
            ds      = 1;
        end
        default: begin
            nstate = UNKW;
            rd      = UNKW;
            ds      = UNKW;
        end
    endcase
end
// */

```

```

/*
// For added flexibility, outputs can optionally be moved out of the
main `always_comb` block
// and placed in a separate `always_comb` block or defined through
continuous assignment statements.
// This modular approach helps in managing and organizing the next-
state and output logic distinctly.

always_comb begin
    nstate = UNKW; // @LB nstate = cstate;
    case (cstate)
        IDLE:    nstate = start ? READ : IDLE;
        READ:    nstate = DLY;
        DLY:     nstate = ws ? READ : DONE;
        DONE:    nstate = IDLE;
        default: nstate = UNKW;
    endcase
end

always_comb begin
    rd = 0;
    ds = 0;
    case (cstate)
        READ:    rd = 1;
        DLY:     rd = 1;
        DONE:    ds = 1;
        default: {rd, ds} = UNKW;
    endcase
end
// */

endmodule

```

6.3. Three Always Block FSM coding style - registered outputs

- File: [fsm_3_always.sv](#)

```

module fsm_3_always (
    output reg rd,
    output reg ds,

    input clk,
    input rst,

    input start,
    input ws
);

localparam IDLE = "IDLE";

```

```

localparam READ = "READ";
localparam DLY = "DLY";
localparam DONE = "DONE";

localparam UNKW = 'x;

reg [4*8-1:0] cstate;
reg [4*8-1:0] nstate;

always_ff @(posedge clk, posedge rst)
    if (rst) cstate <= IDLE;
    else cstate <= nstate;

always_comb begin
    nstate = UNKW; // @LB nstate = cstate;
    case (cstate)
        IDLE:    nstate = start ? READ : IDLE;
        READ:    nstate = DLY;
        DLY:     nstate = ws ? READ : DONE;
        DONE:    nstate = IDLE;
        default: nstate = UNKW;
    endcase
end

always_ff @(posedge clk, posedge rst) begin
    if (rst) begin
        rd <= 0;
        ds <= 0;
    end else begin
        rd <= 0;
        ds <= 0;
        case (nstate)
            IDLE:    ;
            READ:    rd <= 1;
            DLY:     rd <= 1;
            DONE:    ds <= 1;
            default: {rd, ds} <= UNKW;
        endcase
    end
end

endmodule

```

6.4. Four Always Block FSM coding style - registered outputs

- File: [fsm_4_always.sv](#)

```

module fsm_4_always (
    output reg rd,
    output reg ds,

```



```

    input clk,
    input rst,

    input start,
    input ws
);

localparam IDLE = "IDLE";
localparam READ = "READ";
localparam DLY = "DLY";
localparam DONE = "DONE";

localparam UNKW = 'x;

reg [4*8-1:0] cstate;
reg [4*8-1:0] nstate;

always_ff @(posedge clk, posedge rst)
    if (rst) cstate <= IDLE;
    else cstate <= nstate;

always_comb begin
    nstate = UNKW; // @LB nstate = cstate;
    case (cstate)
        IDLE:    nstate = start ? READ : IDLE;
        READ:    nstate = DLY;
        DLY:     nstate = ws ? READ : DONE;
        DONE:    nstate = IDLE;
        default: nstate = UNKW;
    endcase
end

reg nrd; // next rd
reg nds; // next ds

always_comb begin
    nrd = 0;
    nds = 0;
    case (cstate)
        IDLE:    if(start) nrd = 1;
        READ:    nrd = 1;
        DLY: begin
            if (ws) nrd = 1;
            else nds = 1;
        end
        DONE: ;
        default: {nrd, nds} = UNKW;
    endcase
end

always_ff @(posedge clk, posedge rst) begin
    if (rst) begin

```

```
        rd <= 0;  
        ds <= 0;  
    end else begin  
        rd <= nrd;  
        ds <= nds;  
    end  
end  
  
endmodule
```