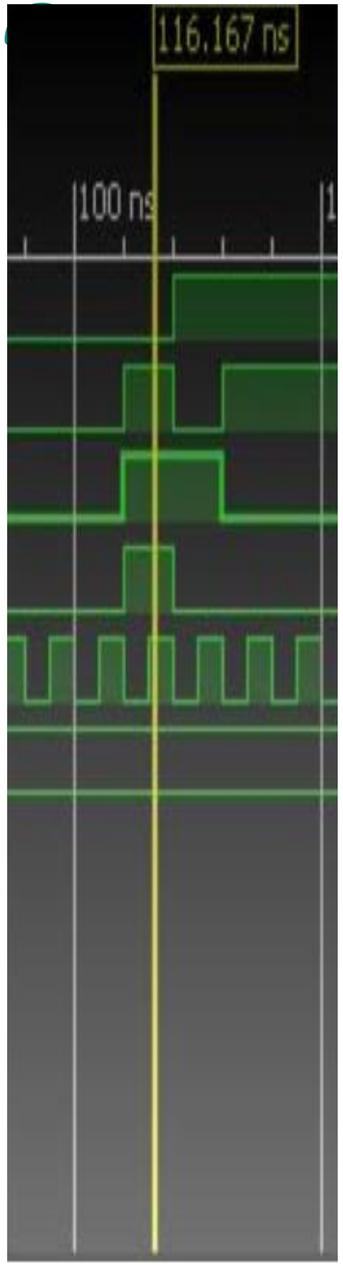
Finite State Machines

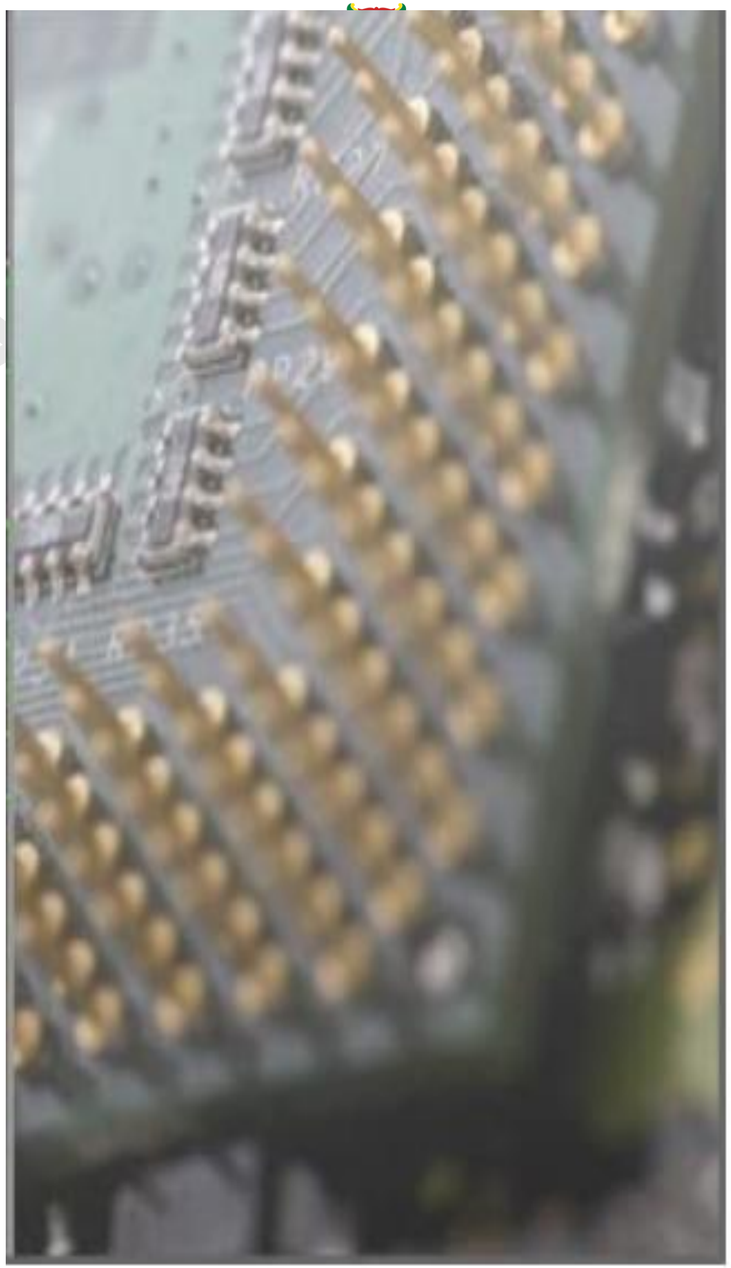
MC. Martin González Pérez

Agenda

- Modeling FSMs
  - FSM introduction
  - One always block FSM
  - Two always block FSM
  - Three always block FSM
  - Four always block FSM
- Apendix A

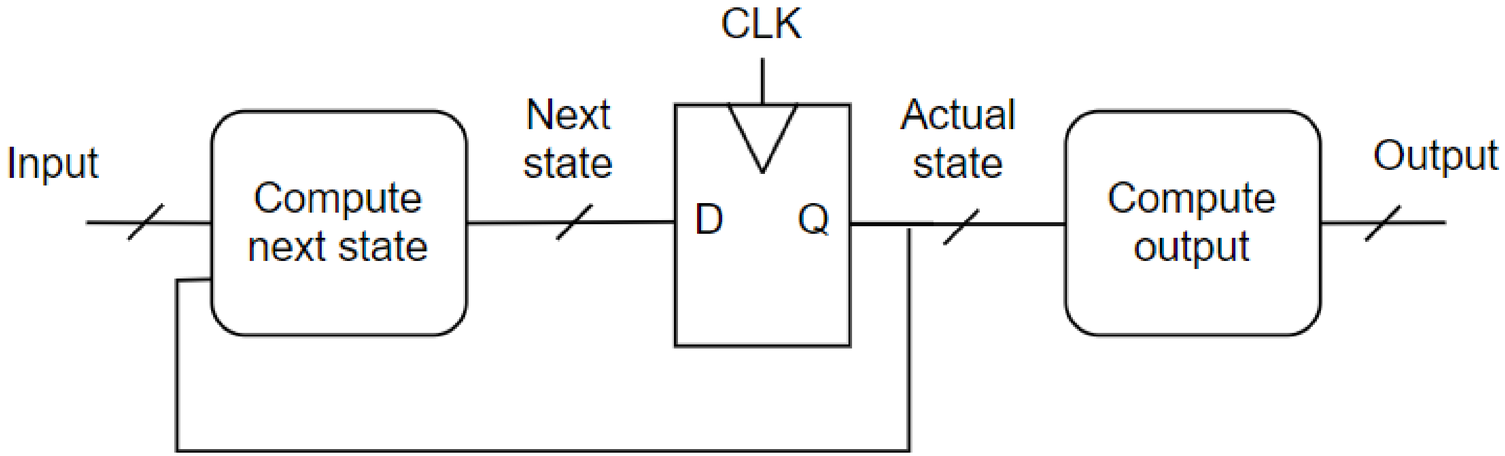


Modeling FSM



Finite State Machines

A sequential logic circuit is composed of combinational circuits and registers. The outputs of the registers represent the current state **(Q)**, while the inputs of the registers **(D)** represent the next state. A system that includes **n registers** can be in one of a finite number of **states (2^n)**, which is why these sequential systems are called Finite State Machines (**FSMs**).



Finite State Machines

We know the basics of state machines and methods to design them using:

- State diagrams
- State tables
- Boolean equations
- Schematics

This method is useful for small state machines and understanding their operation but designing larger and complex state machines using this methodology can become time-consuming. Using the HDL Verilog allows us to model state machines in a simpler way.

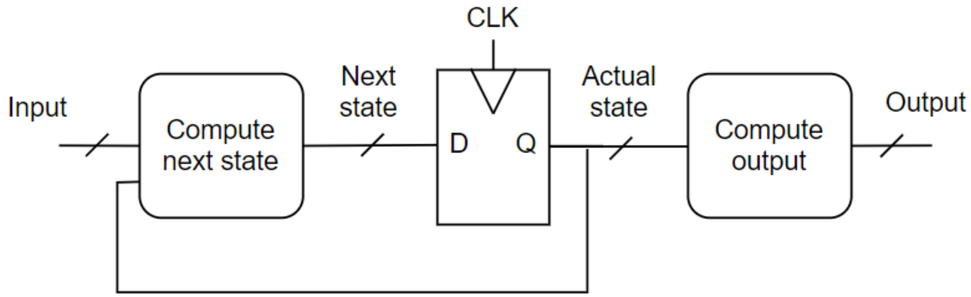
Modeling FSM

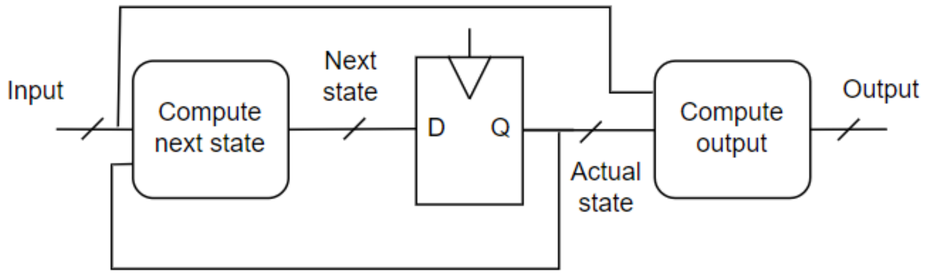
The objective of this class is to learn how to model FSMs using HDL Verilog in different ways and to analyze their advantages and disadvantages.

FSM Classification

FSM can be classified according to their **output dependency.**

- Moore FSM: Outputs depend only on current state.
- Mealy FSM: Outputs depend on current state and inputs.





*Moore FSM are preferred due to the outputs reduced logical depth.

FSM Classification

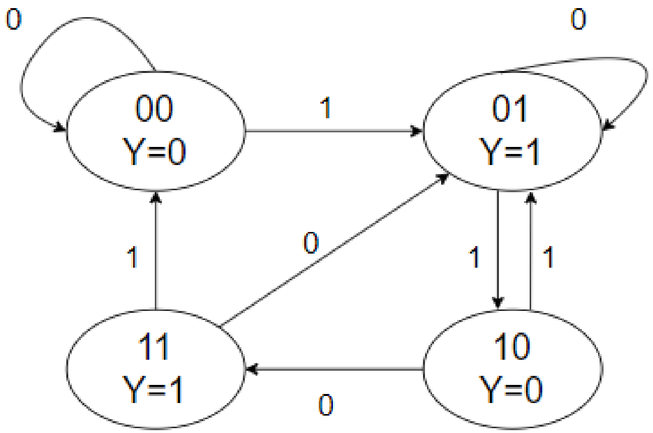
FSM can be classified according to their **state encoding.**

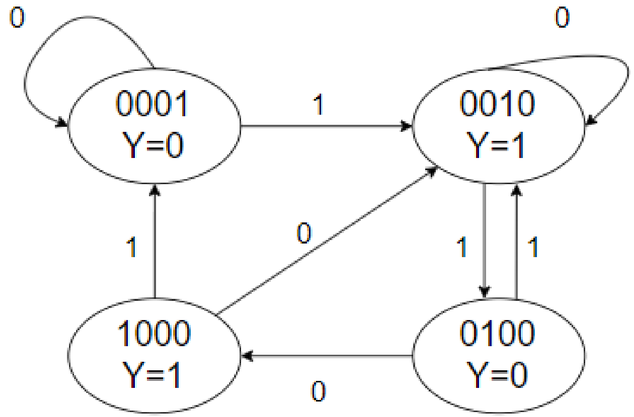
- Binary encoded (Highly encoded):

Use **less flip-flops** but need **more combinational logic** to encode states.

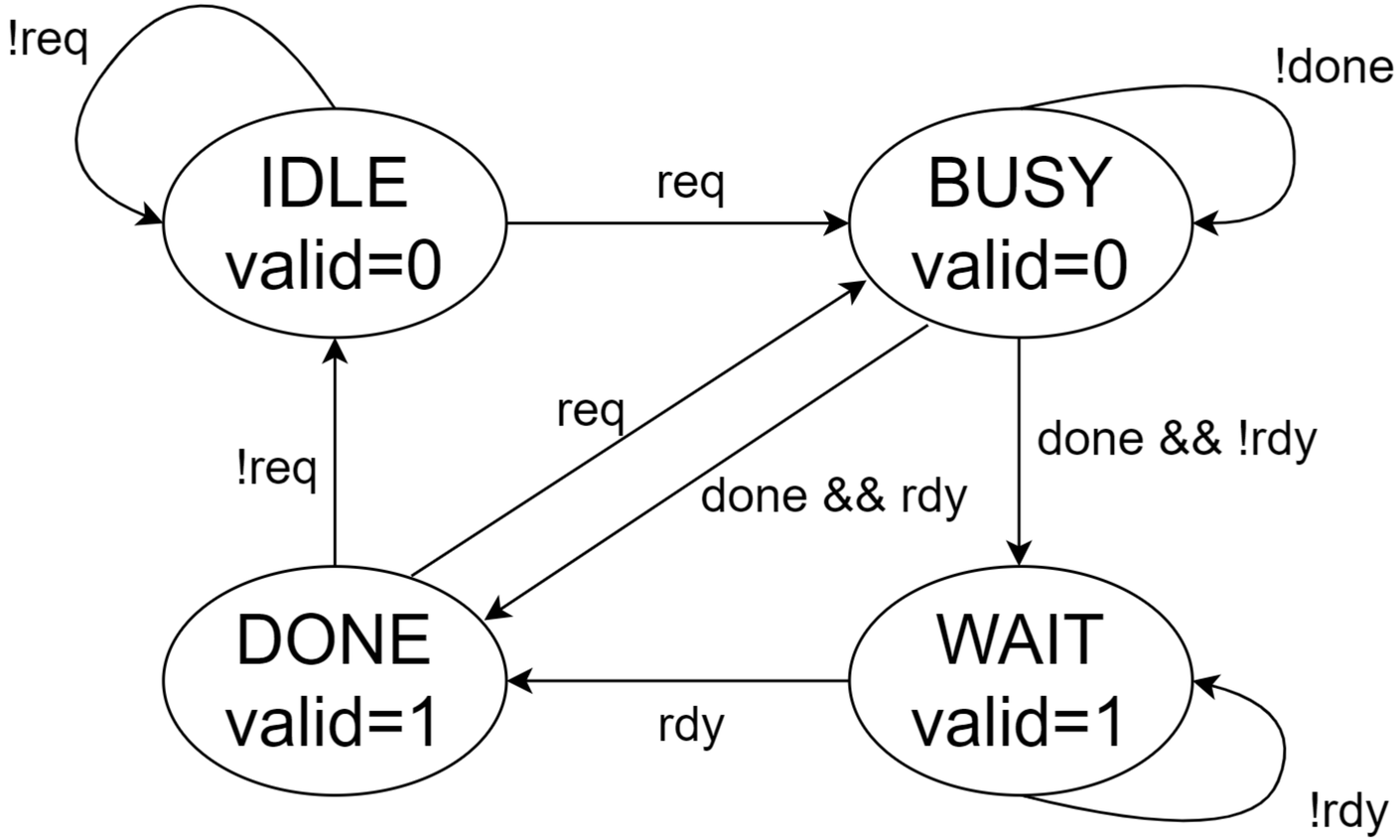
- One-hot encoded:

Use **more flip-flops** but need **less combinational logic** to encode states.





FSM Example



Inputs
req
rdy
done

Outputs
valid

One always block FSM

This style uses only one always procedural block that handles **every state**, the **next state**, and **outputs for the next state**. To do this, you need to know the next state for every current state and the output assignment for the next state.
It's important to set the outputs for every next state (even if the next state is the same that the current state).

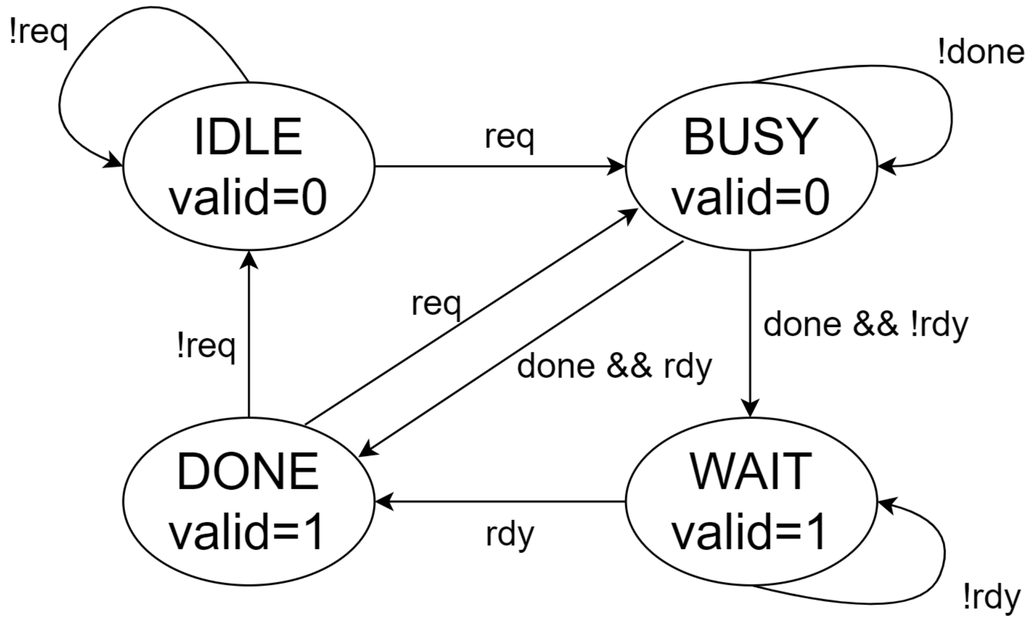
One always block FSM

```
always @(posedge clk, negedge arstn) begin
    if (!arstn) begin
        state <= STATE_IDLE;
        valid <= 1'b0;
    end else begin
    valid <= 1'b0;
    case (state)
        STATE_IDLE: begin
            if (req) begin
                state <= STATE_BUSY;
            end else begin
                state <= STATE_IDLE;
            end
        end
        STATE_BUSY: begin
            if (!done) begin
                state <= STATE_BUSY;
            end else if (rdy) begin
                state <= STATE_DONE;
                valid <= 1'b1;
            end else begin
                state <= STATE_WAIT;
                valid <= 1'b1;
            end
        end
```

State

Next_state

Next_output



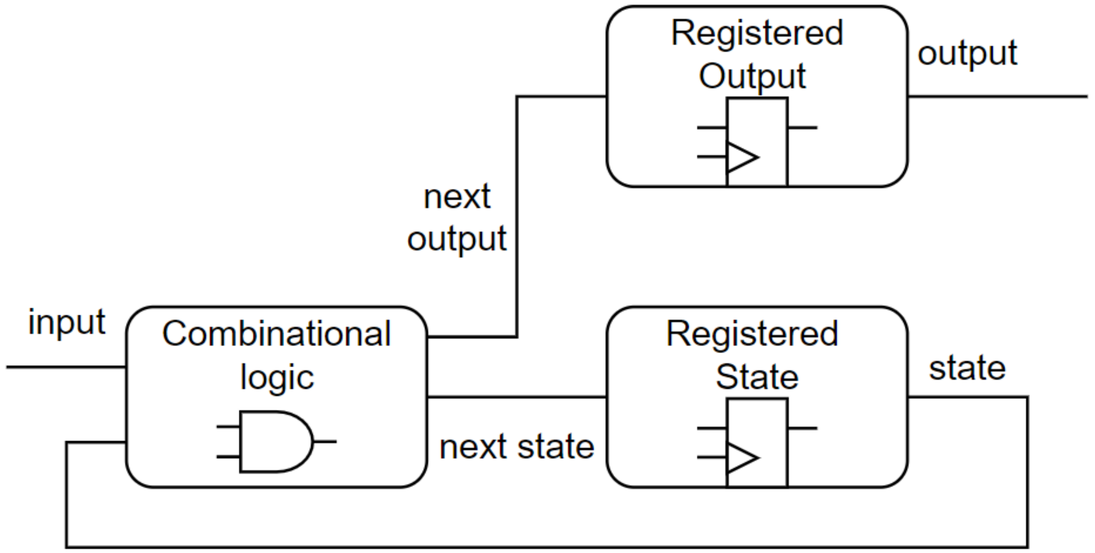
One always block FSM

```
always @(posedge clk, negedge arstn) begin
    if (!arstn) begin
        state <= STATE_IDLE;
        valid <= 1'b0;
    end else begin
    valid <= 1'b0;
    case (state)
        STATE_IDLE: begin
            if (req) begin
                state <= STATE_BUSY;
            end else begin
                state <= STATE_IDLE;
            end
        end
        STATE_BUSY: begin
            if (!done) begin
                state <= STATE_BUSY;
            end else if (rdy) begin
                state <= STATE_DONE;
                valid <= 1'b1;
            end else begin
                state <= STATE_WAIT;
                valid <= 1'b1;
            end
        end
```

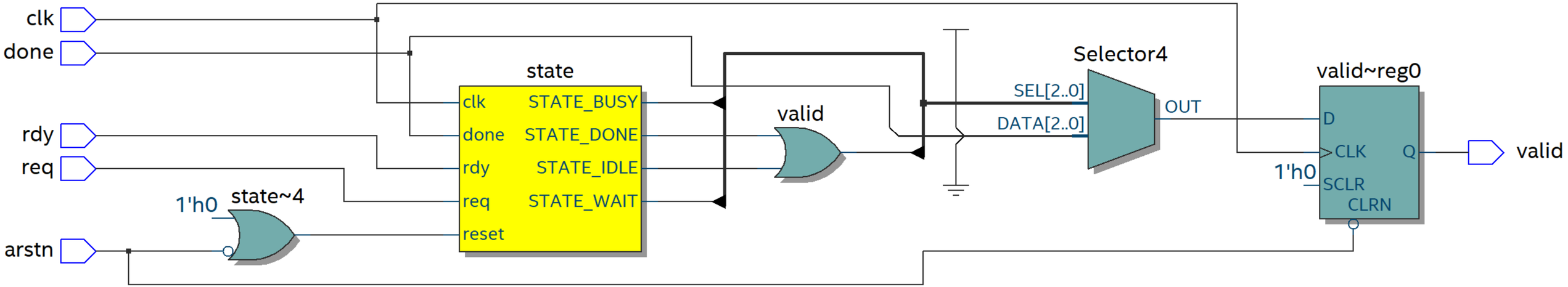
State

Next_state

Next_output



One always block FSM



One always block FSM

Main advantage
The synthesis results are good.

Main disadvantage
Code is verbose and error prone for complex FSMs.

Two always block FSM

This method uses a **sequential block** and a **combinational block**. The sequential block only **update the state**, while combinational block computes the **next state and outputs**.

Using this method, the outputs are not registered.

Two always block FSM

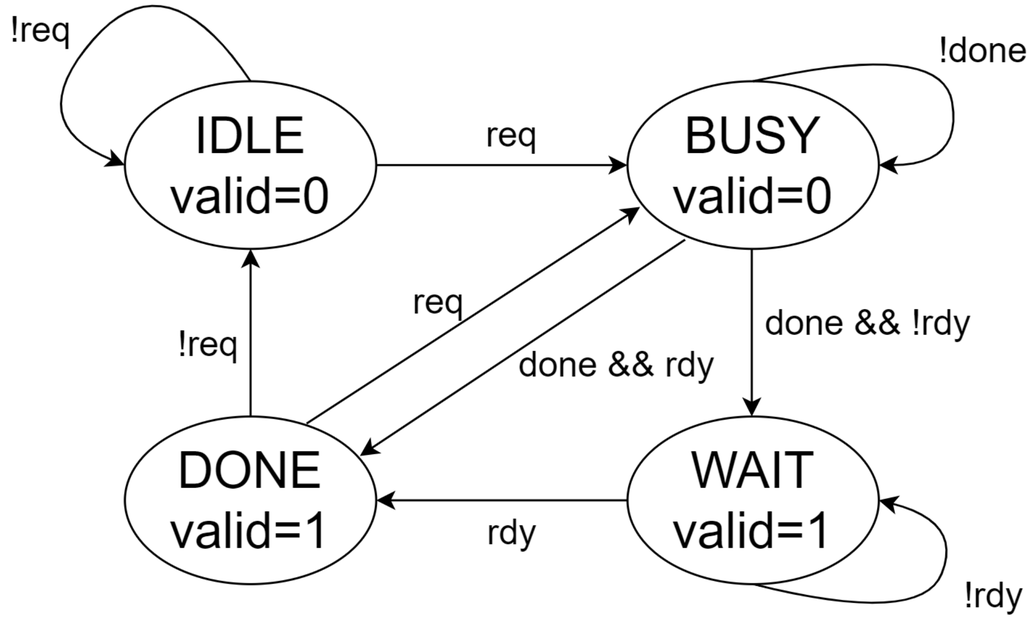
Update state

```
always @(posedge clk, negedge arstn) begin
    if (!arstn) begin
        state <= STATE_IDLE;
    end else begin
        state <= next_state;
    end
end
```

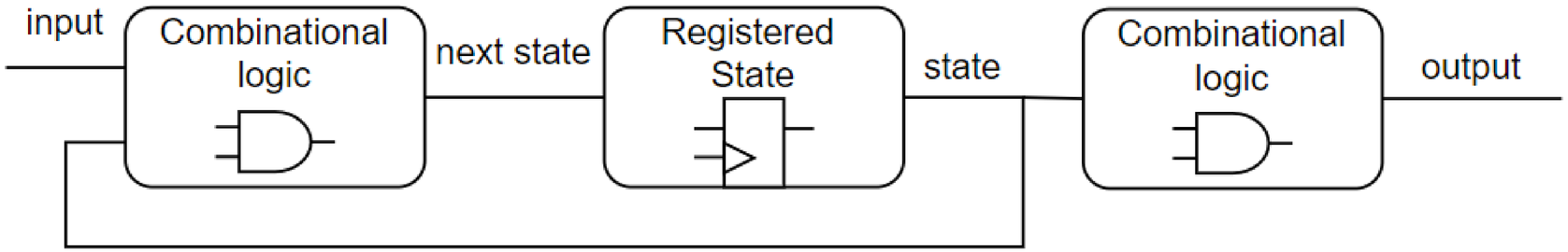
Next state and outputs

```
always @(*) begin
    valid = 1'b0;
    case (state)
        STATE_IDLE: begin
            if (req) begin
                next_state = STATE_BUSY;
            end else begin
                next_state = STATE_IDLE;
            end
        end
        STATE_BUSY: begin
            if (!done) begin
                next_state = STATE_BUSY;
            end else if (rdy) begin
                next_state = STATE_DONE;
            end else begin
                next_state = STATE_WAIT;
            end
        end
        STATE_WAIT: begin
            valid = 1'b1;
            if (!rdy) begin
                next_state = STATE_WAIT;
            end else begin
                next_state = STATE_DONE;
            end
        end
```

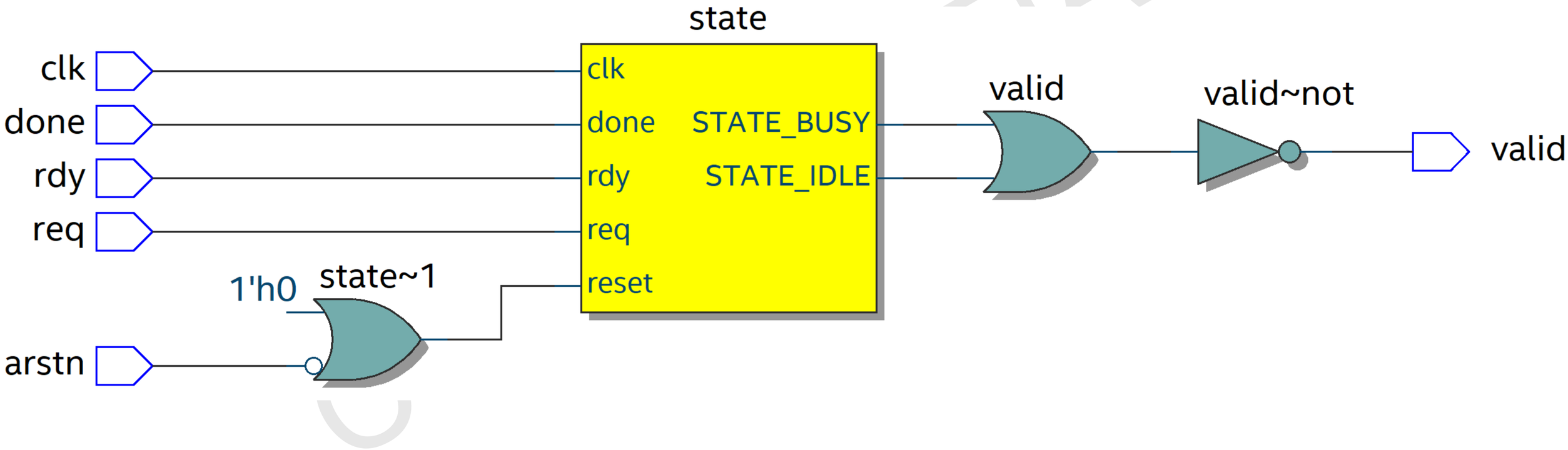
Output assignment



Two always block FSM



Two always block FSM



Two always block FSM

Main advantage
Easy to code and read.

Main disadvantage
Synthesis results not good as one always block method.
Outputs are not registered.

Three always block FSM

This method is similar to the two always blocks method, with the difference that **a third sequential block is added to register the output**. The added block computes the next state output.

Three always block FSM

Update state

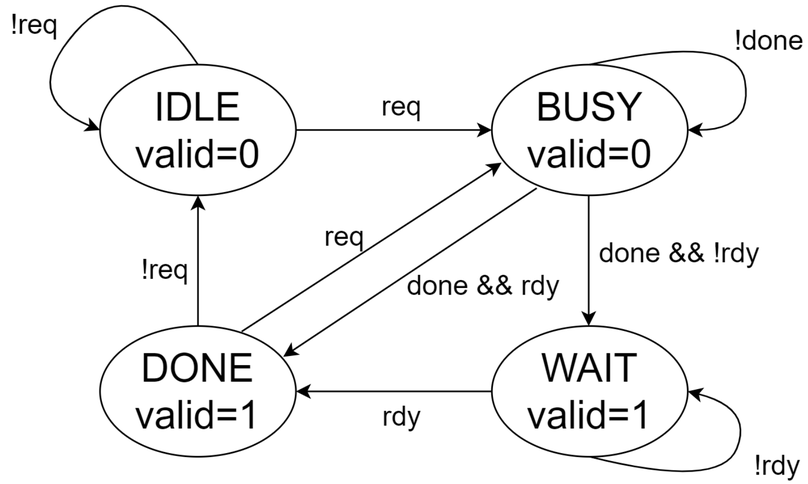
```
always @(posedge clk, negedge arstn) begin
    if (!arstn) begin
        state <= STATE_IDLE;
    end else begin
        state <= next_state;
    end
end
```

Next state

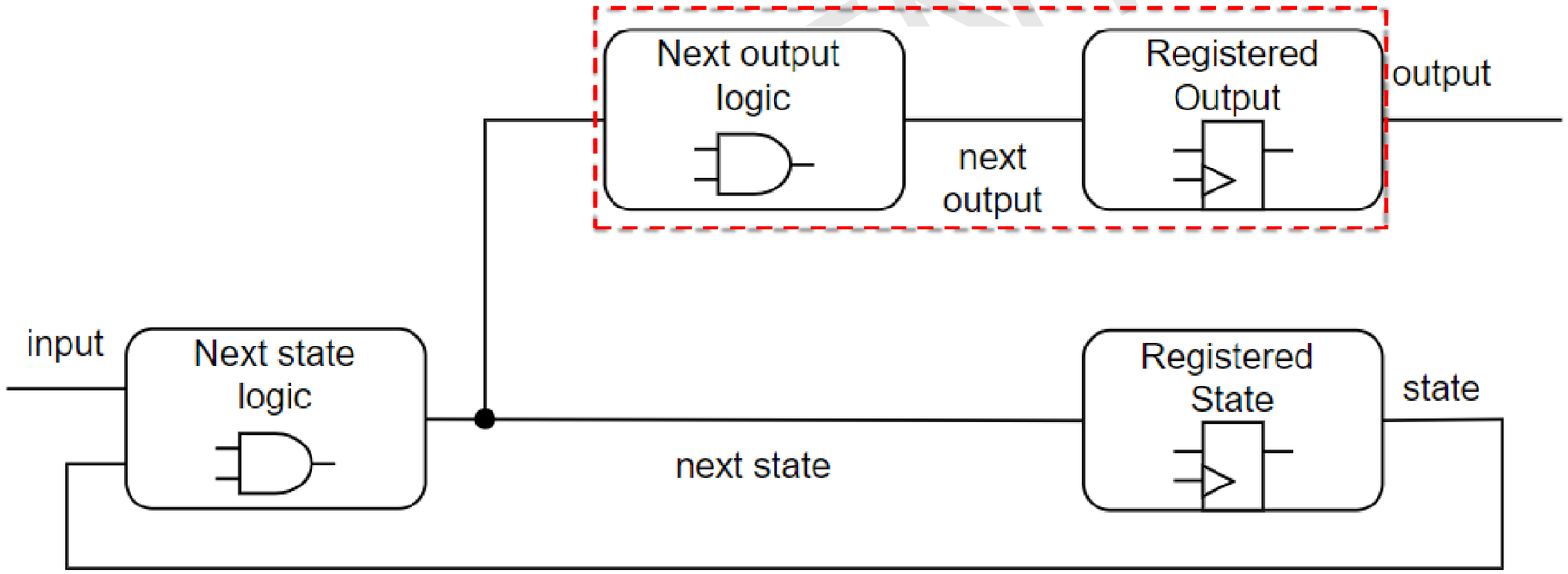
```
always @(*) begin
    case (state)
        STATE_IDLE: begin
            if (req) begin
                next_state = STATE_BUSY;
            end else begin
                next_state = STATE_IDLE;
            end
        end
        STATE_BUSY: begin
            if (!done) begin
                next_state = STATE_BUSY;
            end else if (rdy) begin
                next_state = STATE_DONE;
            end else begin
                next_state = STATE_WAIT;
            end
        end
```

Next output

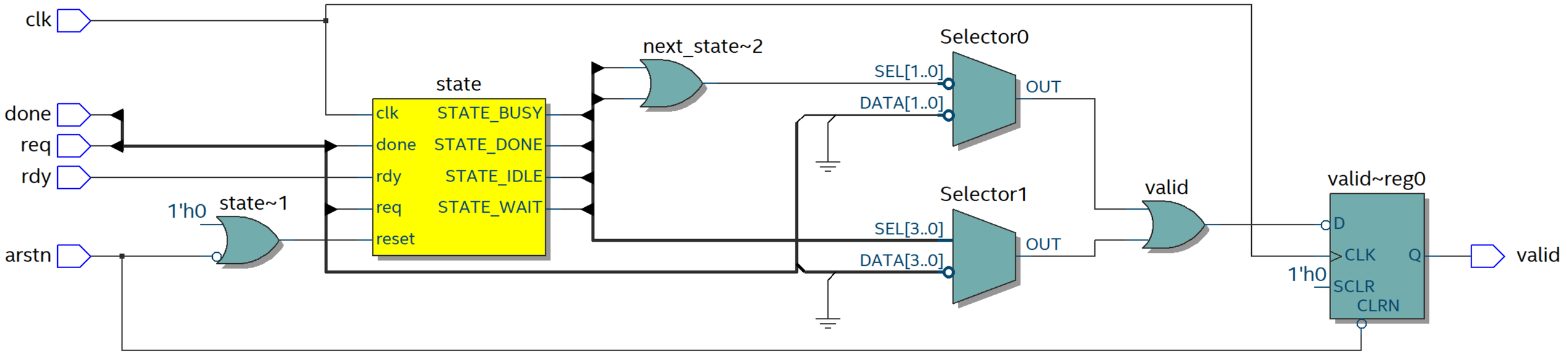
```
always @(posedge clk, negedge arstn) begin
    if (!arstn) begin
        valid <= 1'b0;
    end else begin
        valid <= 1'b0;
        case (next_state)
            STATE_IDLE: valid <= 1'b0;
            STATE_BUSY: valid <= 1'b0;
            STATE_WAIT: valid <= 1'b1;
            STATE_DONE: valid <= 1'b1;
        endcase
    end
end
```



Three always block FSM



Three always block FSM



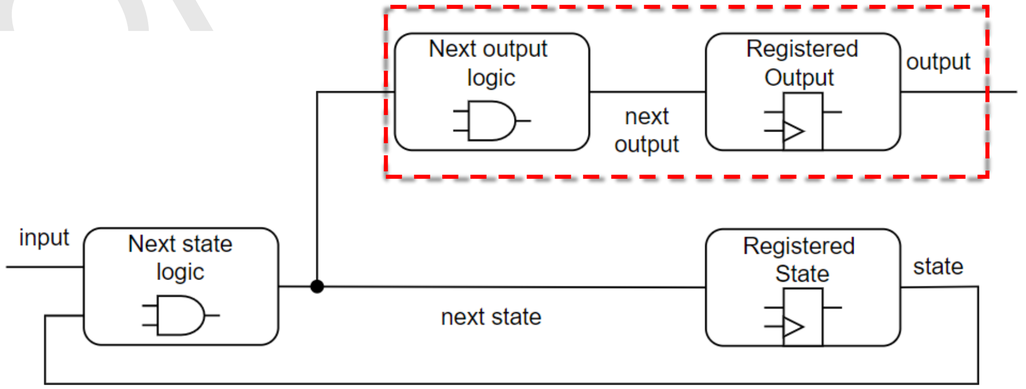
Three always block FSM

Main advantage
Easy to code and read.

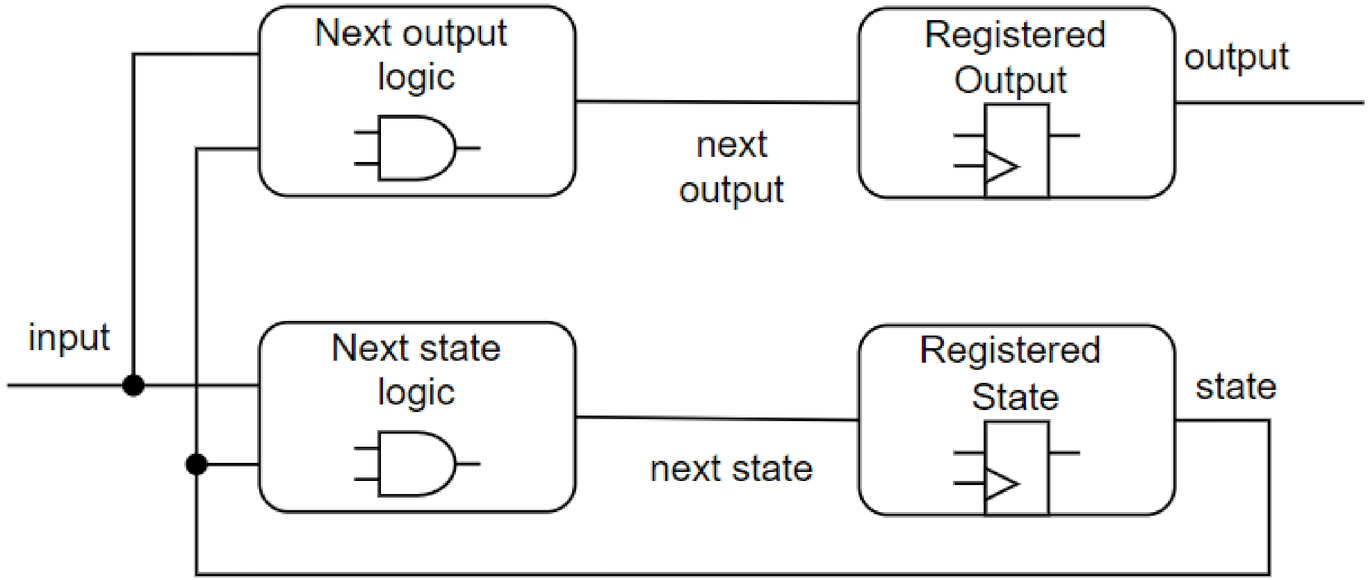
Main disadvantage
Synthesis results not good as one always block.

Four always block FSM

The one always method gave better synthesis results than the three always method. In the three always block method, the block used to compute the next outputs depends on the block that computes the next state.
We can add a fourth block that simplifies the next output assignments and computes the next output using combinational logic in parallel (using the current state instead of the next state) with the next state combinational logic.



Four always block FSM



Four always block FSM

Update state

```
always @(posedge clk, negedge arstn) begin
    if (!arstn) begin
        state <= STATE_IDLE;
    end else begin
        state <= next_state;
    end
end
```

Next state

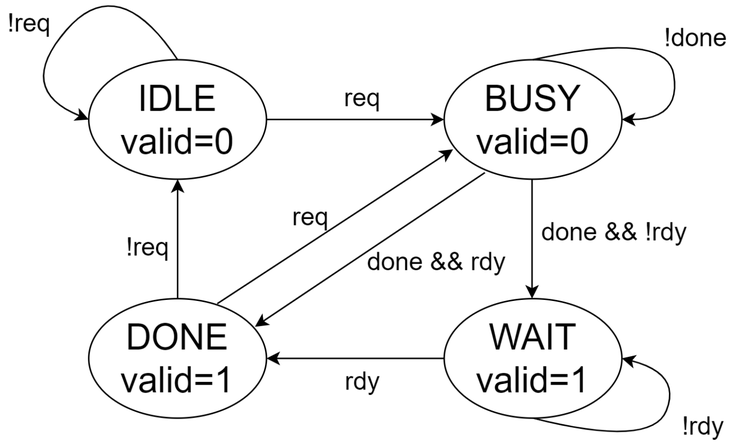
```
always @(*) begin
    case (state)
        STATE_IDLE: begin
            if (req) begin
                next_state = STATE_BUSY;
            end else begin
                next_state = STATE_IDLE;
            end
        end
        STATE_BUSY: begin
            if (!done) begin
                next_state = STATE_BUSY;
            end else if (rdy) begin
                next_state = STATE_DONE;
            end else begin
                next_state = STATE_WAIT;
            end
        end
```

Next output

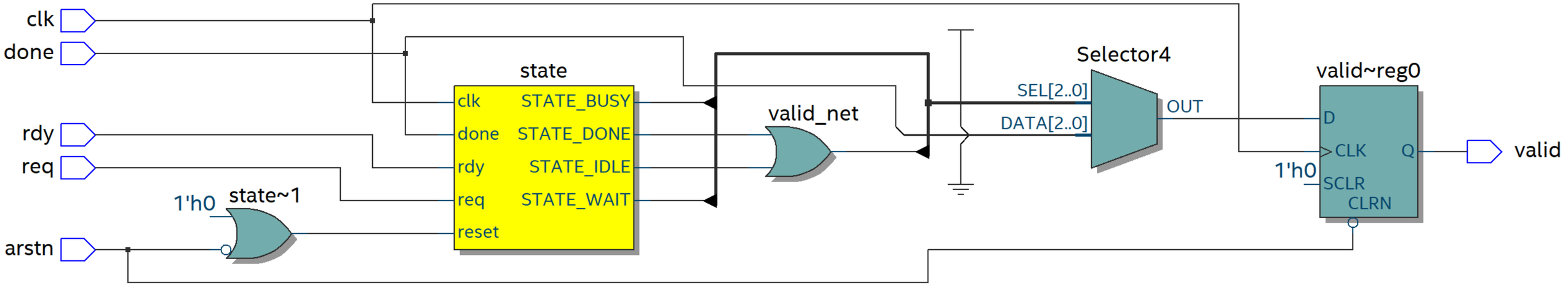
```
always @(*) begin
    case (state)
        STATE_IDLE: valid_net = 1'b0;
        STATE_BUSY: if (done)
            valid_net = 1'b1;
        else
            valid_net = 1'b0;
        STATE_WAIT: valid_net = 1'b1;
        STATE_DONE: valid_net = 1'b0;
    endcase
end
```

Update output

```
always @(posedge clk, negedge arstn) begin
    if (!arstn) begin
        valid <= 1'b0;
    end else begin
        valid <= valid_net;
    end
end
```



Four always block FSM



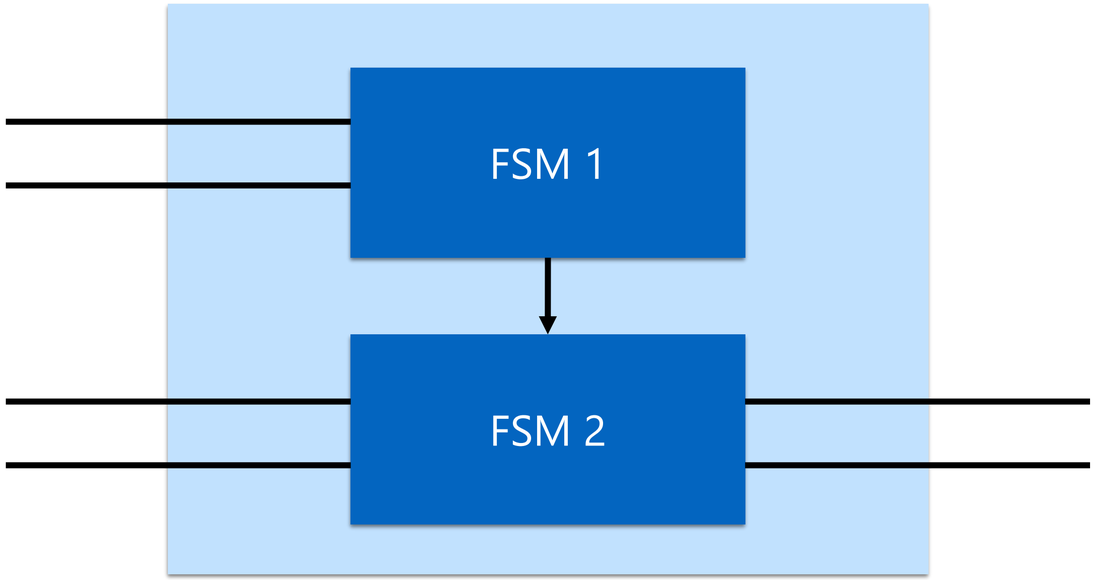
Four always block FSM

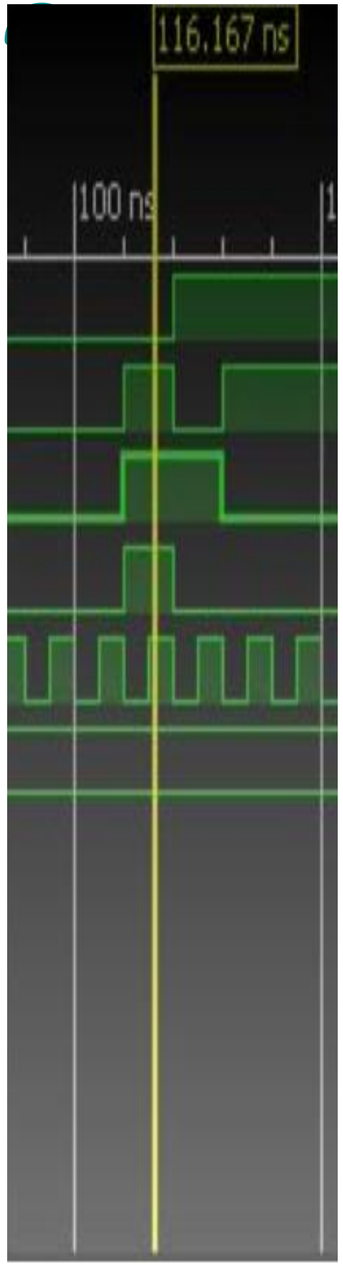
Advantage
Easy to code and read that one always block.
Synthesis results similar to one always block.

Disadvantage
The method has more steps.

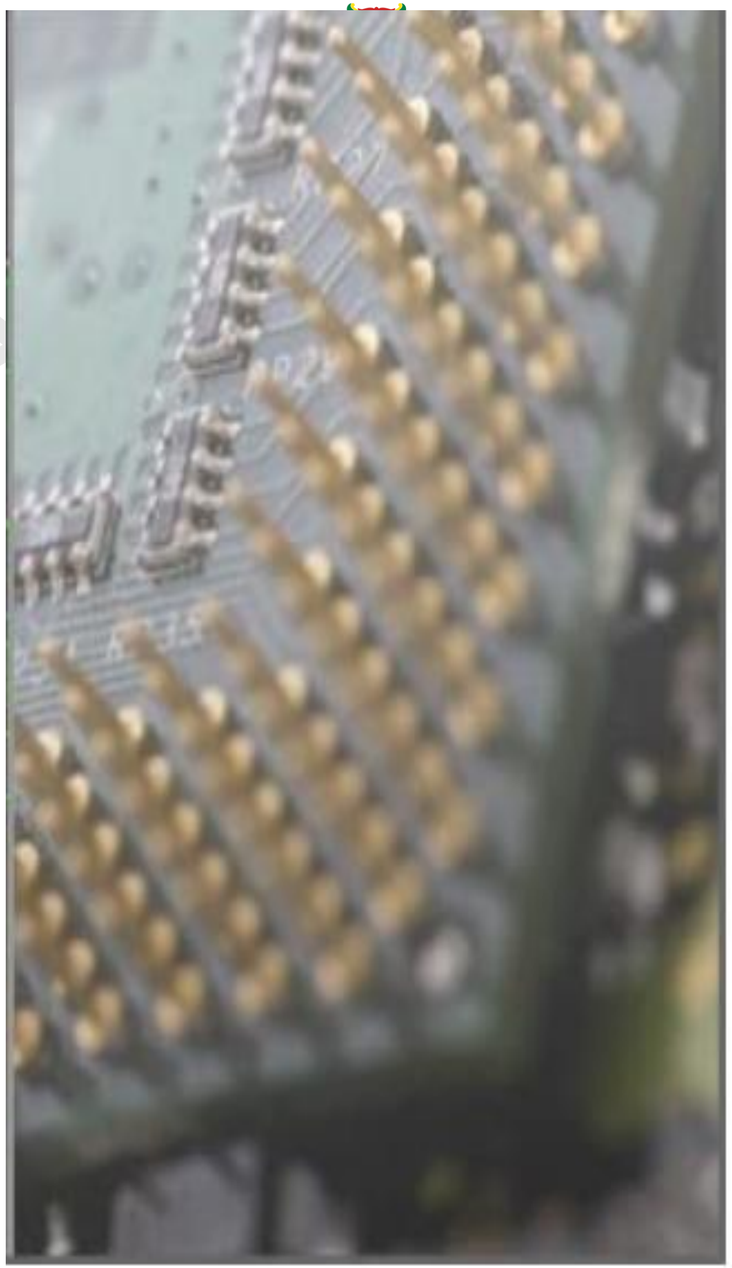
Factored FSM

Complex FSMs can be broken down into multiple interacting simpler FSMs, such that the outputs of some FSMs are the inputs of others. This application of hierarchy and modularity is called factoring.





Apendix A



One always block FSM

```
module one_always_block_fsm(
    input clk,
    input arstn,
    input req,
    input rdy,
    input done,
    output reg valid
);

    reg [1:0]state;
    parameter STATE_IDLE = 2'b00;
    parameter STATE_BUSY = 2'b01;
    parameter STATE_WAIT = 2'b10;
    parameter STATE_DONE = 2'b11;

    always @(posedge clk, negedge arstn) begin
        if (!arstn) begin
            state <= STATE_IDLE;
            valid <= 1'b0;
        end else begin
        valid <= 1'b0;
        case (state)
            STATE_IDLE: begin
                if (req) begin
                    state <= STATE_BUSY;
                end else begin
                    state <= STATE_IDLE;
                end
            end
            STATE_BUSY: begin
                if (!done) begin
                    state <= STATE_BUSY;
                end else if (rdy) begin
                    state <= STATE_DONE;
                    valid <= 1'b1;
                end else begin
                    state <= STATE_WAIT;
                    valid <= 1'b1;
                end
            end
            STATE_WAIT: begin
                valid <= 1'b1;
                if (!rdy) begin
                    state <= STATE_WAIT;
                end else begin
                    state <= STATE_DONE;
                end
            end
            STATE_DONE: begin
                if (req) begin
                    state <= STATE_BUSY;
                end else begin
                    state <= STATE_IDLE;
                end
            end
        endcase
        end
    end
endmodule
```

Two always block FSM

```
module two_always_block_fsm(
    input clk,
    input arstn,
    input req,
    input rdy,
    input done,
    output reg valid
);
    reg [1:0] state;
    reg [1:0] next_state;
    parameter STATE_IDLE = 2'b00;
    parameter STATE_BUSY = 2'b01;
    parameter STATE_WAIT = 2'b10;
    parameter STATE_DONE = 2'b11;

    always @(posedge clk, negedge arstn) begin
        if (!arstn) begin
            state <= STATE_IDLE;
        end else begin
            state <= next_state;
        end
    end

    always @(*) begin
        valid = 1'b0;
        case (state)
            STATE_IDLE: begin
                if (req) begin
                    next_state = STATE_BUSY;
                end else begin
                    next_state = STATE_IDLE;
                end
            end
            STATE_BUSY: begin
                if (!done) begin
                    next_state = STATE_BUSY;
                end else if (rdy) begin
                    next_state = STATE_DONE;
                end else begin
                    next_state = STATE_WAIT;
                end
            end
            STATE_WAIT: begin
                valid = 1'b1;
                if (!rdy) begin
                    next_state = STATE_WAIT;
                end else begin
                    next_state = STATE_DONE;
                end
            end
            STATE_DONE: begin
                valid = 1'b1;
                if (req) begin
                    next_state = STATE_BUSY;
                end else begin
                    next_state = STATE_IDLE;
                end
            end
        endcase
    end
endmodule
```

Three always block FSM

```
module three_always_block_fsm(
    input clk,
    input arstn,
    input req,
    input rdy,
    input done,
    output reg valid
);
    reg [1:0] state;
    reg [1:0] next_state;
    parameter STATE_IDLE = 2'b00;
    parameter STATE_BUSY = 2'b01;
    parameter STATE_WAIT = 2'b10;
    parameter STATE_DONE = 2'b11;

    always @(posedge clk, negedge arstn) begin
        if (!arstn) begin
            state <= STATE_IDLE;
        end else begin
            state <= next_state;
        end
    end

    always @(*) begin
        case (state)
            STATE_IDLE: begin
                if (req) begin
                    next_state = STATE_BUSY;
                end else begin
                    next_state = STATE_IDLE;
                end
            end
            STATE_BUSY: begin
                if (!done) begin
                    next_state = STATE_BUSY;
                end else if (rdy) begin
                    next_state = STATE_DONE;
                end else begin
                    next_state = STATE_WAIT;
                end
            end
            STATE_WAIT: begin
                if (!rdy) begin
                    next_state = STATE_WAIT;
                end else begin
                    next_state = STATE_DONE;
                end
            end
            STATE_DONE: begin
                if (req) begin
                    next_state = STATE_BUSY;
                end else begin
                    next_state = STATE_IDLE;
                end
            end
        endcase
    end

    always @(posedge clk, negedge arstn) begin
        if (!arstn) begin
            valid <= 1'b0;
        end else begin
            valid <= 1'b0;
            case (next_state)
                STATE_IDLE: valid <= 1'b0;
                STATE_BUSY: valid <= 1'b0;
                STATE_WAIT: valid <= 1'b1;
                STATE_DONE: valid <= 1'b1;
            endcase
        end
    end
endmodule
```

Four always block FSM

```
module four_always_block_fsm(
    input clk,
    input arstn,
    input req,
    input rdy,
    input done,
    output reg valid
);
    reg [1:0] state;
    reg [1:0] next_state;
    reg valid_net;
    parameter STATE_IDLE = 2'b00;
    parameter STATE_BUSY = 2'b01;
    parameter STATE_WAIT = 2'b10;
    parameter STATE_DONE = 2'b11;

    always @(posedge clk, negedge arstn) begin
        if (!arstn) begin
            state <= STATE_IDLE;
        end else begin
            state <= next_state;
        end
    end

    always @(*) begin
        case (state)
            STATE_IDLE: begin
                if (req) begin
                    next_state = STATE_BUSY;
                end else begin
                    next_state = STATE_IDLE;
                end
            end
            STATE_BUSY: begin
                if (!done) begin
                    next_state = STATE_BUSY;
                end else if (rdy) begin
                    next_state = STATE_DONE;
                end else begin
                    next_state = STATE_WAIT;
                end
            end
            STATE_WAIT: begin
                if (!rdy) begin
                    next_state = STATE_WAIT;
                end else begin
                    next_state = STATE_DONE;
                end
            end
            STATE_DONE: begin
                if (req) begin
                    next_state = STATE_BUSY;
                end else begin
                    next_state = STATE_IDLE;
                end
            end
        endcase
    end

    always @(*) begin
        case (state)
            STATE_IDLE: valid_net = 1'b0;
            STATE_BUSY: if (done)
                valid_net = 1'b1;
            else
                valid_net = 1'b0;
            STATE_WAIT: valid_net = 1'b1;
            STATE_DONE: valid_net = 1'b0;
        endcase
    end

    always @(posedge clk, negedge arstn) begin
        if (!arstn) begin
            valid <= 1'b0;
        end else begin
            valid <= valid_net;
        end
    end
endmodule
```