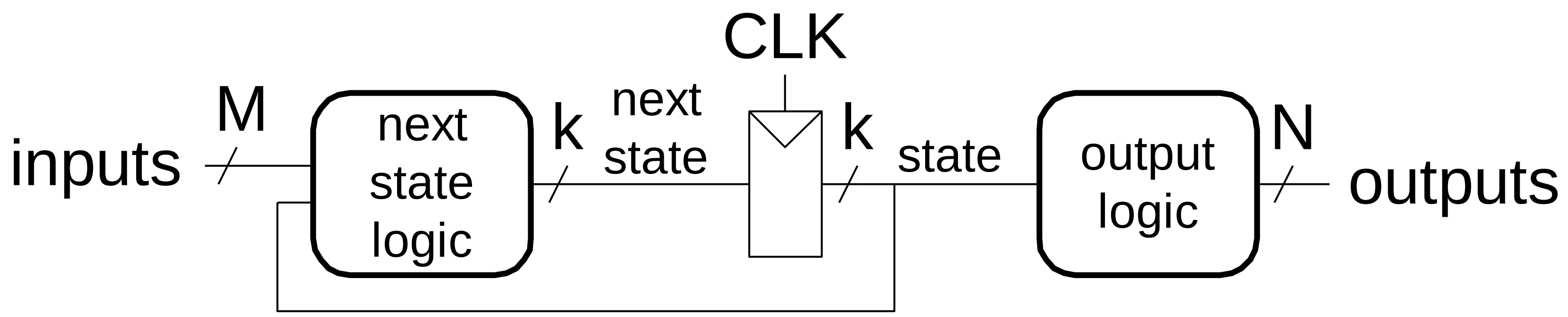
שפת תכנון חומרה
Verilog - ורילוג

Verilog – Finite State Machine (FSM)

Dr. Avihai Aharon

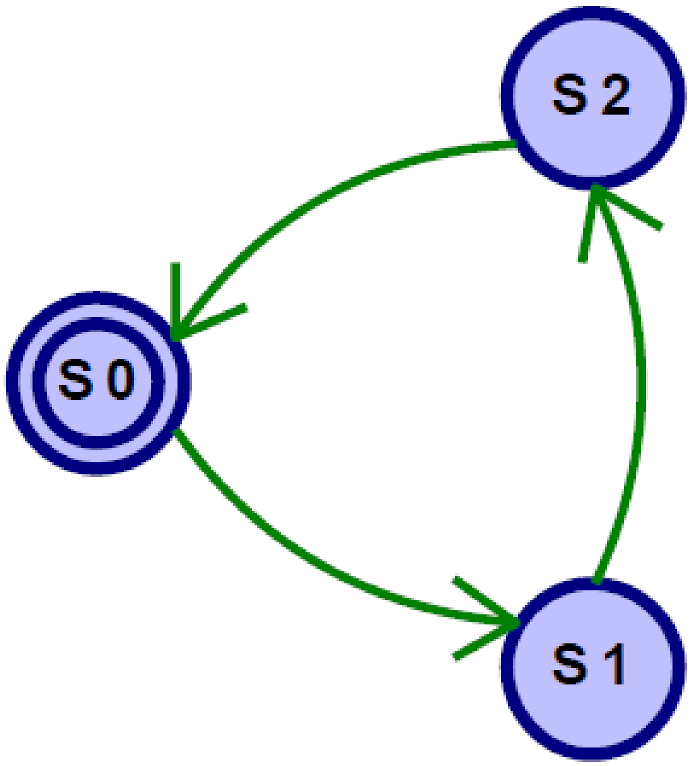
Finite State Machines (FSMs)

- **Each FSM consists of three separate parts:**
  - next state logic
  - state register
  - output logic



FSM Example: Divide by 3

- Output should be “1” every 3 clock cycles
    - state S0



FSM in Verilog - Definitions

```
module divideby3FSM (input clk,
                     input reset,
                     output q);

   reg  [1:0] state, nextstate;

   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;
```

- **We define state and nextstate as 2-bit reg**
- **The parameter descriptions are optional, it makes reading easier**

FSM in Verilog - State Register

```
// state register
   always @ (posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;
```

- This part defines the state register (memorizing process)
- Sensitive to only clk, reset
- In this example reset is active when ‘1’

FSM in Verilog - Next State Calculation

```
// next state logic
  always @ (*)
     case (state)
        S0:      nextstate = S1;
        S1:      nextstate = S2;
        S2:      nextstate = S0;
        default: nextstate = S0;
     endcase
```

- **Based on the value of state we determine the value of nextstate**
- **An always .. case statement is used for simplicity.**

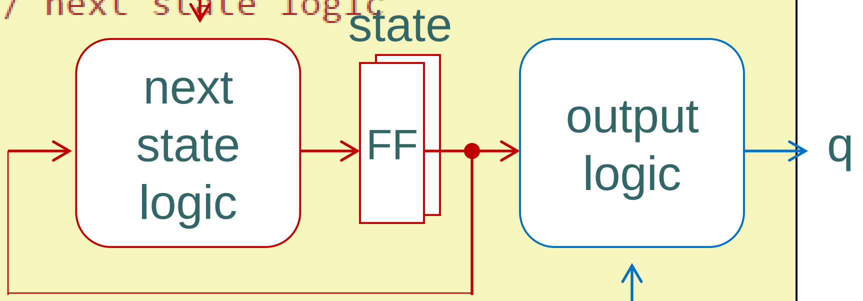
FSM in Verilog - Output Assignments

```
// output logic
   assign q = (state == S0);
```

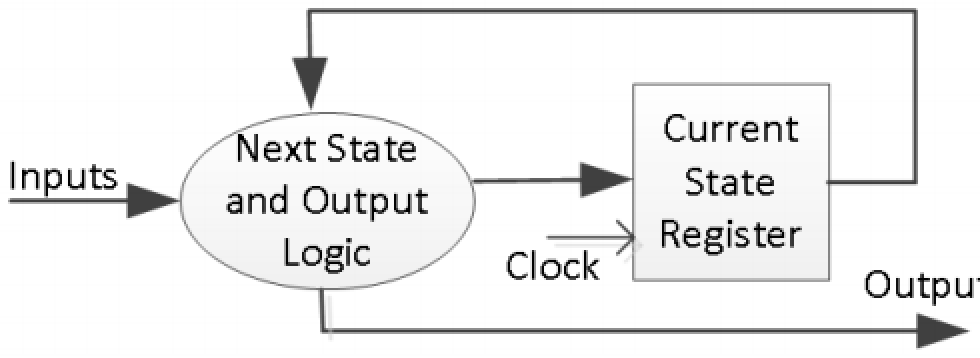
- In this example, output depends only on state
    - Moore type FSM
- We used a simple combinational assign

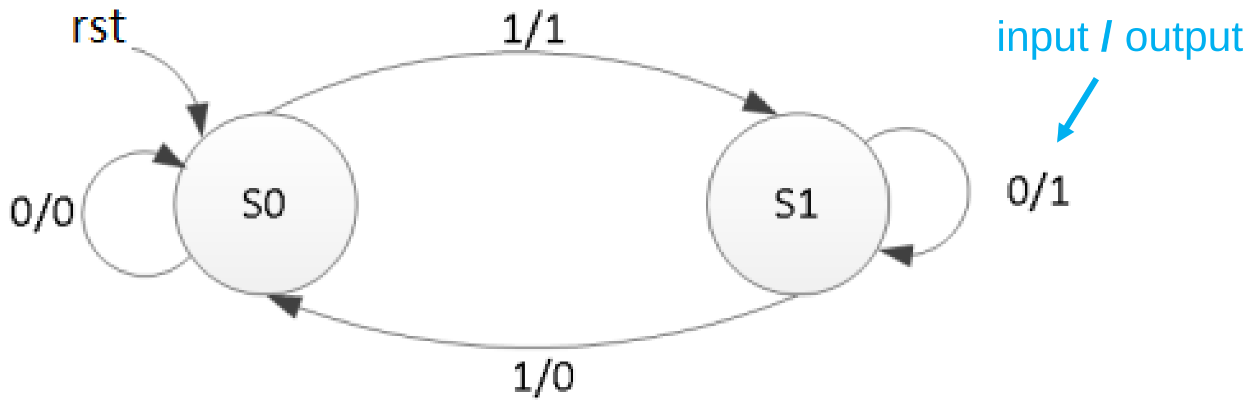
FSM in Verilog – Full code

```
module divideby3FSM (input clk, input reset, output q);
   reg  [1:0] state, nextstate;

   parameter S0 = 2'b00;
   parameter S1 = 2'b01;
   parameter S2 = 2'b10;

   always @ (posedge clk, posedge reset) // state register
      if (reset) state <= S0;
      else       state <= nextstate;
   always @ (*)                          // next state logic
      case (state)
         S0:      nextstate = S1;
         S1:      nextstate = S2;
         S2:      nextstate = S0;
         default: nextstate = S0;
      endcase
   assign q = (state == S0);            // output logic
endmodule
```



Parity checker – Mealy FSM





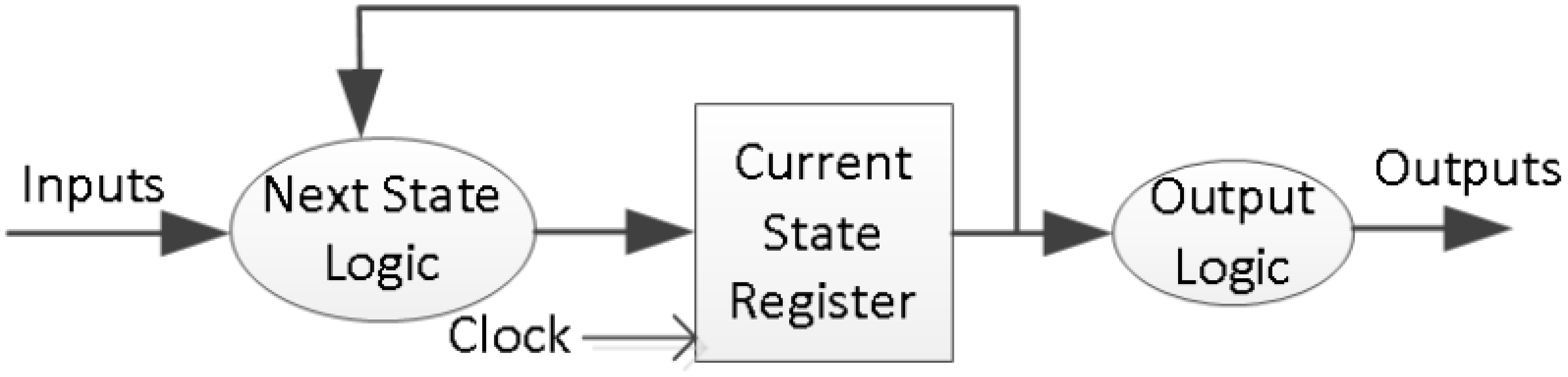
Parity checker – Mealy FSM

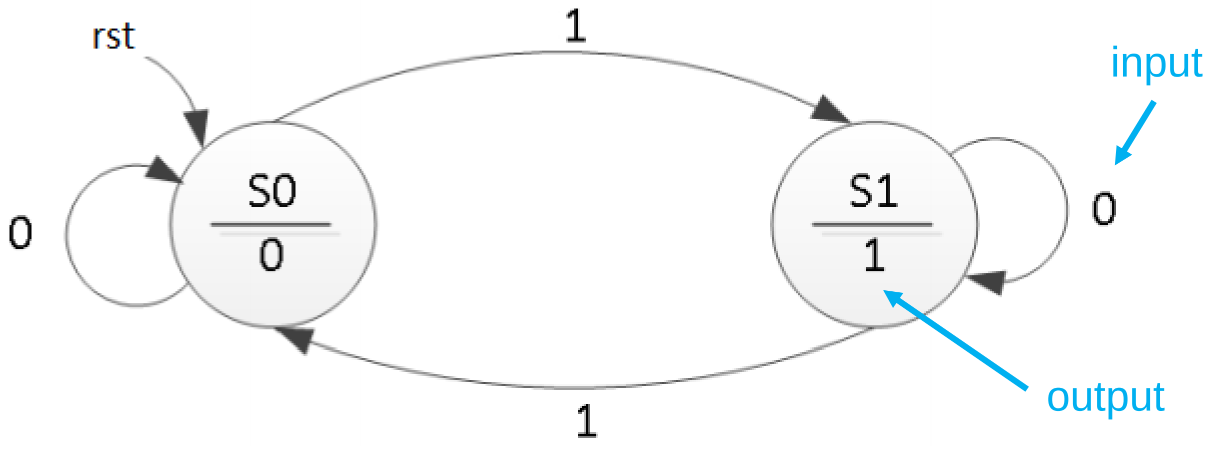
```
module mealy_2processes(input clk, input reset, input x, output reg parity);
    reg state, nextstate;
    parameter S0=0, S1=1;

always @(posedge clk or posedge reset)    // always block to update state
if (reset)
     state <= S0;
else
    state <= nextstate;

always @(state or x) // always block to compute both output & nextstate
begin
    parity = 1'b0;
    case(state)
       S0: if(x)
           begin
               parity = 1; nextstate = S1;
           end
           else
                nextstate = S0;
       S1: if(x)
                nextstate = S0;
           else
           begin
                parity = 1; nextstate = S1;
           end
       default:
           nextstate = S0;
    endcase
end
endmodule
```

Parity checker – Mealy FSM

```
module mealy_3processes(input clk, input reset, input x, output reg parity);
    ⋮

    always @(state or x)  // always block to compute output
    begin
       parity = 1'b0;
       case(state)
          S0: if(x)
                 parity = 1;
          S1: if(!x)
                 parity = 1;
       endcase
    end
    always @(state or x)  // always block to compute nextstate
    begin
       nextstate = S0;
       case(state)
          S0: if(x)
                 nextstate = S1;
          S1: if(!x)
                 nextstate = S1;
          endcase
    end
    endmodule
```

Parity checker – Moore FSM





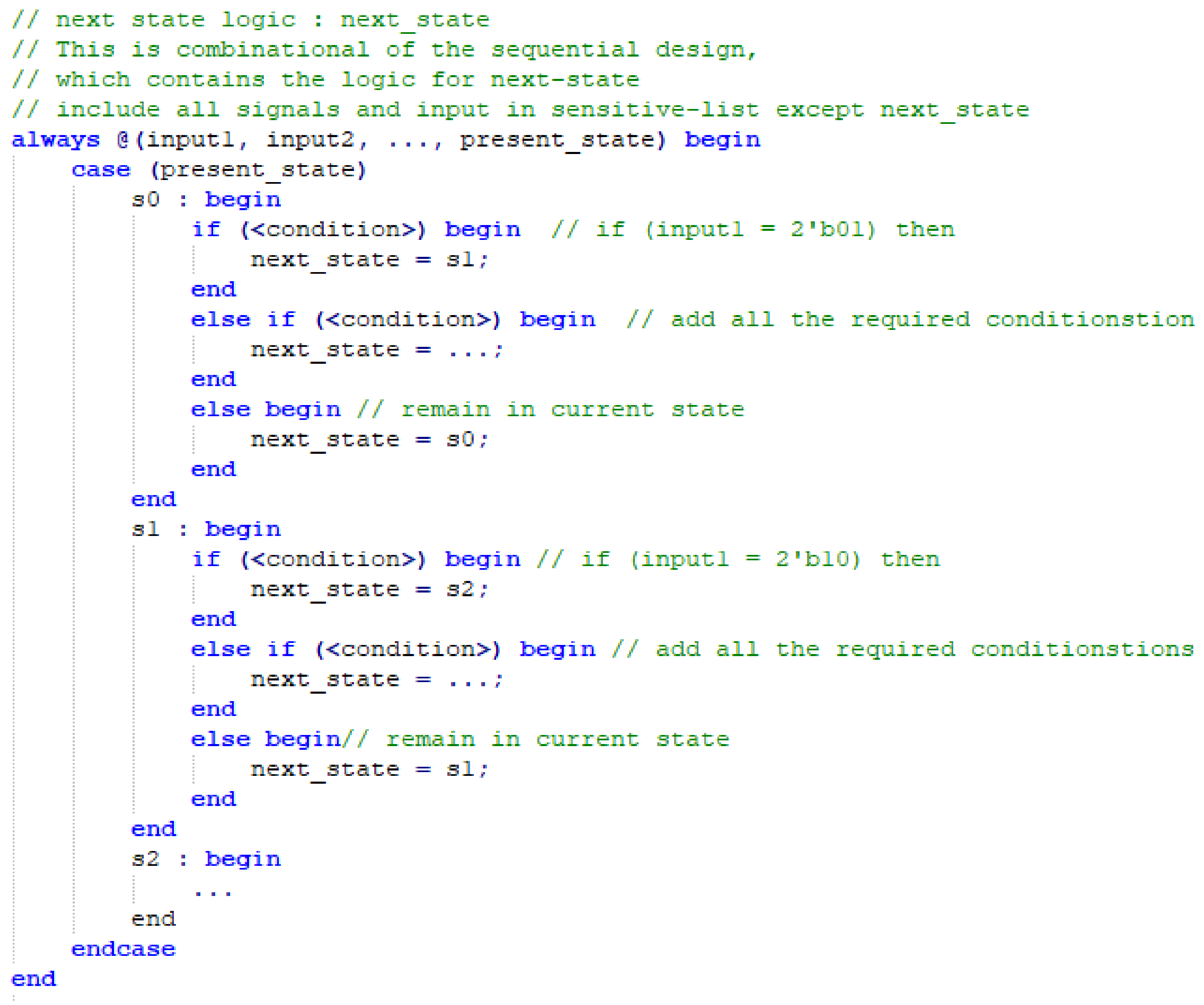
Parity checker – Moore FSM

```
module moore_3processes(input clk, input reset, input x, output reg parity);
    reg state, nextstate;
    parameter S0=0, S1=1;

  always @(posedge clk or posedge reset)  // always block to update state
  if (reset)
       state <= S0;
  else
      state <= nextstate;

  always @(state)                         // always block to compute output
  begin
      case(state)
         S0: parity = 0;
         S1: parity = 1;
      endcase
  end
  always @(state or x)                    // always block to compute nextstate
  begin
      nextstate = S0;
      case(state)
         S0: if(x)
                nextstate = S1;
         S1: if(!x)
                nextstate = S1;
      endcase
  end
  endmodule
```

FSM Moore template

```
module moore_regular_template
(
    input clk, reset,
    input [<size>] input1, input2, ...,
    output reg [<size>] output1, output2, ...,
);

    parameter [<size_state>] // for 4 states : size_state = 1:0
    s0 = 0,
    s1 = 1,
    s2 = 2,
    ... ;

    reg[<size_state>] present_state, next_state;


// state register : present_state
// This process contains sequential part and all the D-FF are
// included in this process. Hence, only 'clk' and 'reset' are
// required for this process.
always @(posedge clk, posedge reset) begin
    if (reset) begin
        present_state <= s0;
    end
    else begin
        present_state <= next_state;
    end
end
```

FSM Moore template



```
// next state logic : next_state
// This is combinational of the sequential design,
// which contains the logic for next-state
// include all signals and input in sensitive-list except next_state
always @(input1, input2, ..., present_state) begin
    case (present_state)
        s0 : begin
            if (<condition>) begin  // if (input1 = 2'b01) then
                next_state = s1;
            end
            else if (<condition>) begin  // add all the required conditionstion
                next_state = ...;
            end
            else begin // remain in current state
                next_state = s0;
            end
        end
        s1 : begin
            if (<condition>) begin // if (input1 = 2'b10) then
                next_state = s2;
            end
            else if (<condition>) begin // add all the required conditionstions
                next_state = ...;
            end
            else begin// remain in current state
                next_state = s1;
            end
        end
        s2 : begin
            ...
        end
    endcase
end
```

FSM Moore template

```
// combination output logic
// This part contains the output of the design
// no if-else statement is used in this part
// include the present_state in sensitive-list in moore FSM
always @(present_state) begin
    // default outputs
    output1 = <value>;
    output2 = <value>;
    ...
    case (present_state)
        s0 : begin
            output1 = <value>;
            output2 = <value>;
            ...
        end
        s1 : begin
            output1 = <value>;
            output2 = <value>;
            ...
        end
        s2 : begin
            ...
        end
    endcase
end
```

```
// optional D-FF to remove glitches
always @(posedge clk, posedge reset)
begin
    if (reset) begin
        new_output1 <= ... ;
        new_output2 <= ... ;
    end
    else begin
        new_output1 <= output1;
        new_output2 <= output2;
    end
end

endmodule
```

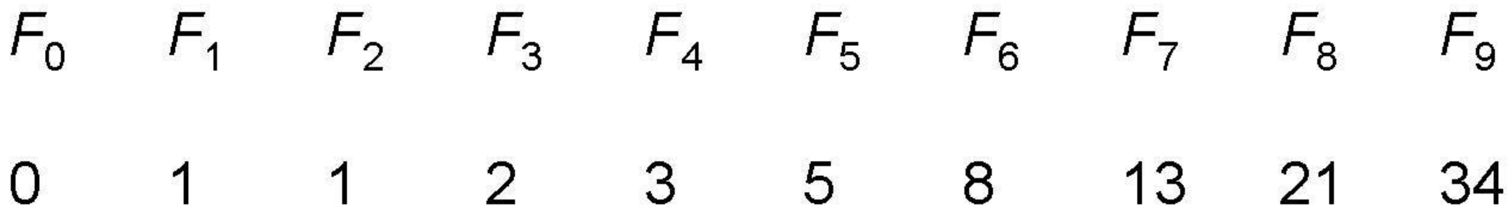
TB - for loop

```
module TB ();
.
.
Parameter data = 10’b0011101100;
Integer i;
.
.
initial
begin
.
.
.
      for(i=0; i<10; i=i+1)
      begin
        x = data[i];
        # 20
       end

.
.
endmodule
```

Fibonacci Calculator

- F(0) = 0, F(1) = 1
- F(n) = F(n – 1) + F(n – 2), when n > 1

- Examples:

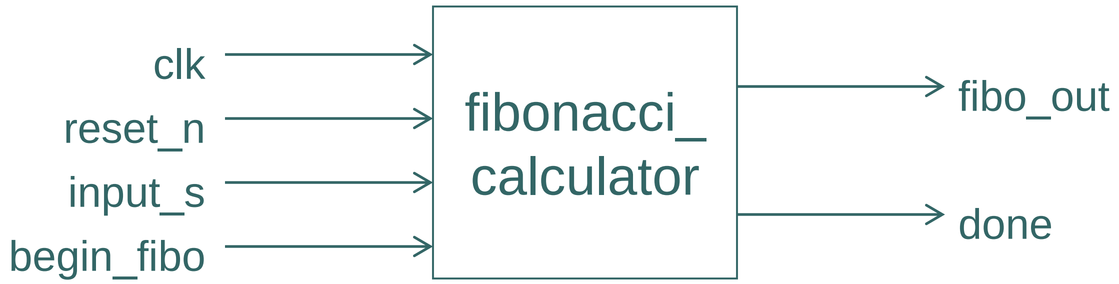


F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9
0	1	1	2	3	5	8	13	21	34

Fibonacci Calculator

- Design a FSM with the interface below.
- input_s is “n”, and fibo_out is “F(n)”.
- Wait in IDLE state until begin_fibo.
- When testbench sees done==1, it will check if fibo_out== F(input_s).

```
module fibonacci_calculator (input clk, reset_n,
                             input [4:0] input_s,
                             input begin_fibo,
                             output [15:0] fibo_out,
                             output done);

  ...
  always @(posedge clk, negedge reset_n)
  begin
    ...
  end
endmodule
```



Fibonacci Calculator

- Basic idea is to introduce 3 registers:

```
reg [4:0] counter;
reg [15:0] R0, R1;
```

- Set loop counter to “n”

```
counter = input_s;
```

- Repeat as long as counter is greater than 1 since we already know what F(0) and F(1) are:

```
counter = counter - 1;
R1 = R1 + R0;
R0 = R1;
```

- Finally, set output to “F(n)”

```
done = 1;
fibo_out = R0;
```

Fibonacci Calculator

```
module fibonacci_calculator (input clk, reset_n,
                             input [4:0] input_s,
                             input begin_fibo,
                            output [15:0] fibo_out,
                            output done);

  parameter IDLE=2'b00, COMPUTE=2'b01, DONE=2'b10;
  reg [1:0] state, Nxt_state;

  reg  [4:0] count;
  reg [15:0] R1, R0;

  //state register
  always @(posedge clk, negedge reset_n)
  begin
    if (!reset_n)
      state <= IDLE;
    else
      state <= Nxt_state;
  end

  // output logic
  assign done = (state== DONE);
  assign fibo_out = (state== DONE)? R1 : 16’d0;
```

```
    // nextstate logic
    always @(*)
        case (state)
          IDLE:
            if (begin_fibo) begin
              count = input_s;
              R1 = 1;
              R0 = 0;
              Nxt_state = COMPUTE;
            end
            else
              Nxt_state = IDLE;

          COMPUTE:
            if (count > 1) begin
              count = count - 1;
              R1 = R1 + R0;
              R0 = R1;
              Nxt_state = COMPUTE;
            end else begin
              Nxt_state = DONE;
            end

          DONE:
            Nxt_state <= IDLE;
        endcase
  endmodule
```