FSM Verilog HDL Coding

FSM Coding Goals : To determine what constitutes an efficient FSM coding style, we first need to identify HDL coding goals and why they are important. After the HDL coding goals have been identified, we can then quantify the capabilities of various FSM coding styles.

- The FSM coding style should be easily modified to change state encodings and FSM styles.
- The coding style should be compact.
- The coding style should be easy to code and understand.
- The coding style should facilitate debugging.
- The coding style should yield efficient synthesis results.

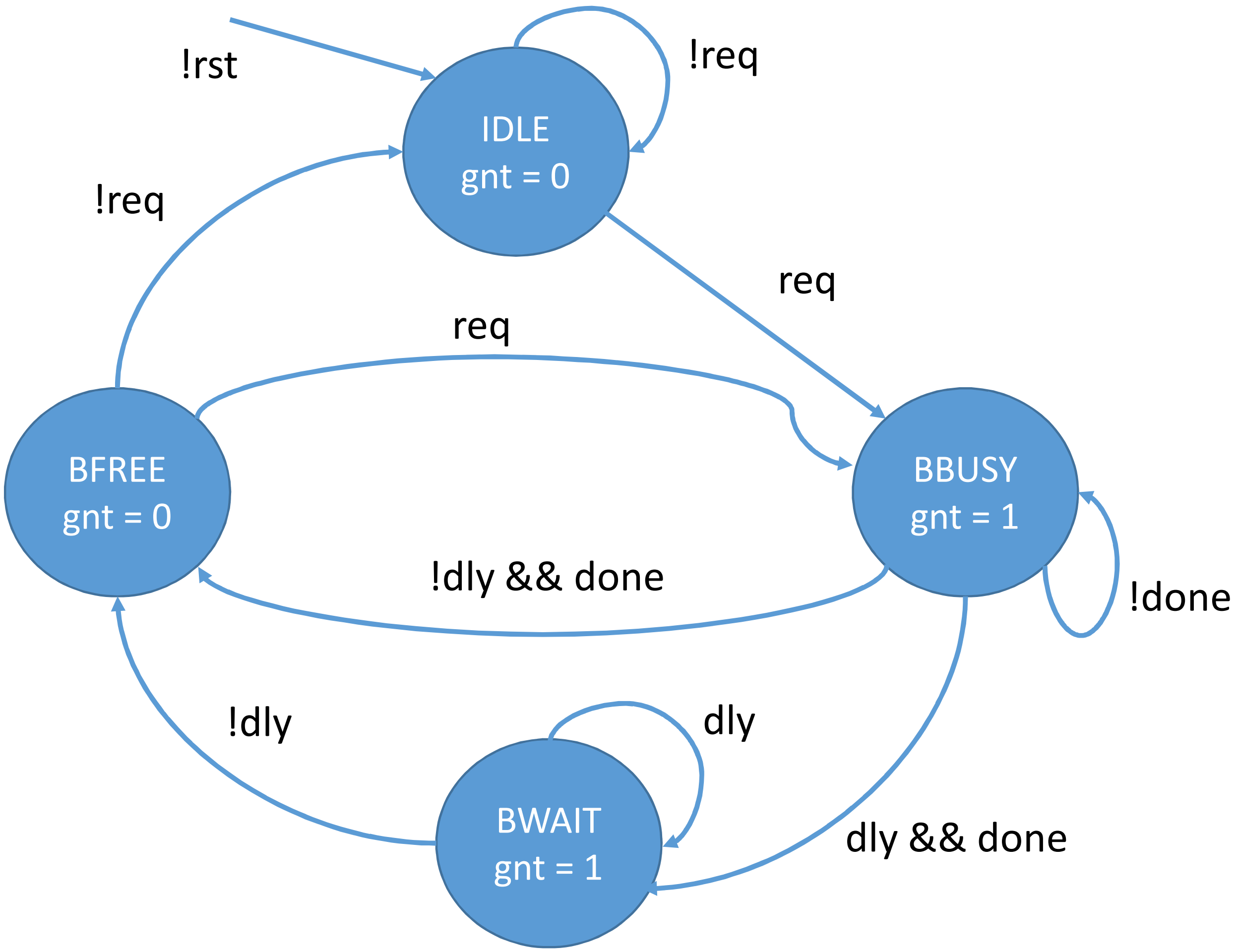
FSM Verilog HDL Coding

One of the best Verilog coding styles is to code the FSM design using two always blocks, one for the sequential state register and one for the combinational next-state and combinational output logic.

Two Always Block FSM Style (Good Style)?

```
module FSM (clk, rst, in, out) // module declaration

input clk; input in; input rst; output out; reg out;

reg [1:0] state; // State variable

reg [1:0] next_state; //Local variable

always @(posedge clk or posedge rst) //sequential state register

        if(rst)     state <= IDLE;

        else        state <= next_state;

always @(state or in) //combinational next state and output logic

        // compute next state and output logic whenever state or input changes

endmodule
```

FSM Example1



FSM Example1: Coding using two always block

One of the best Verilog coding styles is to code the FSM design using two always blocks, one for the sequential state register and one for the combinational next-state and combinational output logic.

```
module fsm1 (output reg gnt, input
dly, done, req, clk, rst_n);

parameter [1:0] IDLE = 2'b00,

BBUSY = 2'b01,

BWAIT = 2'b10,

BFREE = 2'b11;

reg [1:0] state, next;

always @(posedge clk or negedge
rst_n)

          if (!rst_n)        state <= IDLE;

          else               state <= next;
```

Declarations are made for **`state`** and **`next`** (next state) after the parameter assignments.

FSM Example1: Coding using two always block

```
always @(state or dly or done or req) begin
  next = 2'bx;
  gnt = 1'b0;
  case (state)
    IDLE : if (req) next = BBUSY;
           else next = IDLE;
    BBUSY: begin
            gnt = 1'b1;
            if (!done) next = BBUSY;
            else if ( dly) next = BWAIT;
            else next = BFREE;
           end
```

The combinational always block has a default **`next`** state assignment at the top of the always block

Default output assignments are made before coding the **`case`** statement (this eliminates latches and reduces the amount of code required to code the rest of the outputs in the **`case`** statement and highlights in the **`case`** statement exactly in which states the individual output(s) change).

FSM Example1: Coding using two always block

```
BWAIT: begin
         gnt = 1'b1;
         if (!dly) next = BFREE;
         else next = BWAIT;
       end
BFREE: if (req) next = BBUSY;
       else next = IDLE;
endcase
end
endmodule
```

FSM Example1: Coding using one always block

```
module fsm1 (output reg gnt,
input dly, done, req, clk, rst_n);
parameter [1:0] IDLE = 2'd0,
BBUSY = 2'd1,
BWAIT = 2'd2,
BFREE = 2'd3;
reg [1:0] state;
always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
         state <= IDLE;
         gnt <= 1'b0;
        end
    else    begin
         state <= 2'bx;
         gnt <= 1'b0;
```

```
case (state)
    IDLE : if (req) begin
            state <= BBUSY;
            gnt <= 1'b1;
          end
        else state <= IDLE;
    BBUSY: if (!done) begin
            state <= BBUSY;
            gnt <= 1'b1;
           end
        else if ( dly) begin
              state <= BWAIT;
              gnt <= 1'b1;
             end
        else state <= BFREE;
```

FSM Example1: Coding using one always block

```
BWAIT: if ( dly) begin
                state <= BWAIT;
                gnt <= 1'b1;
              end
        else state <= BFREE;
BFREE: if (req) begin
                state <= BBUSY;
                gnt <= 1'b1;
             end
        else state <= IDLE;
endcase
end
endmodule
```

FSM Example1: One hot FSM Coding Style (Good Style)

Efficient (small and fast) onehot state machines can be coded using an inverse case statement; a case statement where each case item is an expression that evaluates to true or false.

```
module fsm1(output reg gnt,
input dly, done, req, clk, rst_n);
parameter [3:0] IDLE = 0,
BBUSY = 1,
BWAIT = 2,
BFREE = 3;
reg [3:0] state, next;
always @(posedge clk or negedge rst_n)
if (!rst_n) begin
                state <= 4'b0;
                state[IDLE] <= 1'b1;
        end
else state <= next;
```

Index into the state register, not state encodings

Onehot requires larger declarations

Reset modification

FSM Example1: One hot FSM Coding Style (Good Style)

```
always @(state or dly or done or req) begin
        next = 4'b0;
        gnt = 1'b0;
        case (1'b1) // ambit synthesis case = full, parallel
          state[IDLE] : if (req) next[BBUSY] = 1'b1;
                        else next[IDLE] = 1'b1;
          state[BBUSY]: begin
                          gnt = 1'b1;
                          if (!done) next[BBUSY] = 1'b1;
                          else if ( dly) next[BWAIT] = 1'b1;
                          else next[BFREE] = 1'b1;
                        end
```

Must make all-0's assignment

Add "full" & "parallel" case

case (1'b1)

**state[*current_state*]
case items**

**Only update the
next[*next state*] bit**

FSM Example1: One hot FSM Coding Style (Good Style)

```
state[BWAIT]: begin
                        gnt = 1'b1;
                        if (!dly) next[BFREE] = 1'b1;
                        else next[BWAIT] = 1'b1;
                   end
state[BFREE]: begin
                        if (req) next[BBUSY] = 1'b1;
                        else next[IDLE] = 1'b1;
                  end
endcase
end
endmodule
```

FSM Example1: Registered FSM Outputs (Good Style)

Registering the outputs of an FSM design insures that the outputs are glitch-free and frequently improves synthesis results by standardizing the output and input delay constraints of synthesized modules FSM outputs are easily registered by adding a third always sequential block to an FSM module where output assignments are generated in a case statement with case items corresponding to the next state that will be active when the output is clocked.

```
module fsm1(output reg gnt,
input dly, done, req, clk, rst_n);
parameter [1:0] IDLE = 2'b00,
BBUSY = 2'b01,
BWAIT = 2'b10,
BFREE = 2'b11;
reg [1:0] state, next;
always @(posedge clk or negedge rst_n) // First always block
        if (!rst_n) state <= IDLE;
        else state <= next;
```

FSM Example1: Registered FSM Outputs (Good Style)

```
always @(state or dly or done or req) begin // Second always block
next = 2'bx;
case (state)
        IDLE :   if (req) next = BBUSY;
                 else next = IDLE;
        BBUSY:   if (!done) next = BBUSY;
                 else if ( dly) next = BWAIT;
                 else next = BFREE;
        BWAIT:   if (!dly) next = BFREE;
                 else next = BWAIT;
        BFREE:   if (req) next = BBUSY;
                 else next = IDLE;
endcase
end
```

FSM Example1: Registered FSM Outputs (Good Style)

```
always @(posedge clk or negedge rst_n) // Third always block
        if (!rst_n) gnt <= 1'b0;
        else
        begin
                gnt <= 1'b0;
          case (next)
                IDLE, BFREE: ; // default outputs
                BBUSY, BWAIT: gnt <= 1'b1;
          endcase
        end
endmodule
```

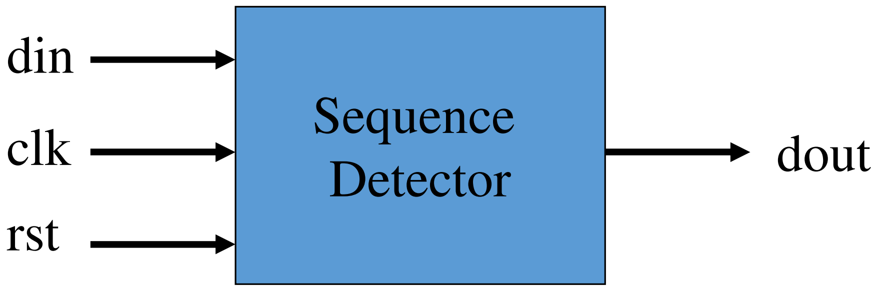
FSM Example1: Compare each Coding Effort

Coding Style	Two always block coding style	One always block coding style (12%-83% larger)	Onehot, two always block coding style	Three always block coding style w/ registered outputs
fsm1 (4 states, simple)	37 lines of code	47 lines of code (12%-27% larger)	42 lines of code	40 lines of code

we see that the one always block FSM coding style is the least efficient coding style with respect to the amount of RTL code required to render an equivalent design. In fact, the more outputs that an FSM design has and the more transition arcs in the FSM state diagram, and thus the faster the one always block coding style increases in size over comparable FSM coding styles.

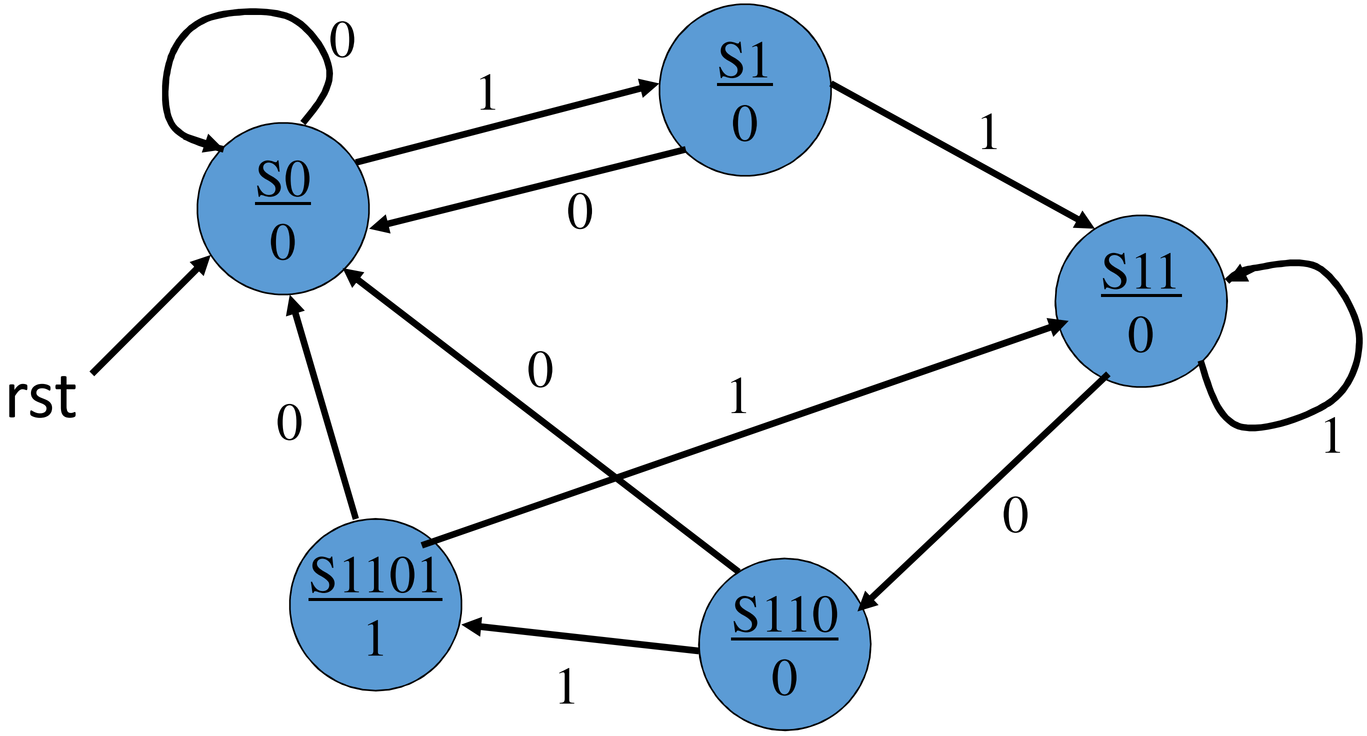
Example2: Detect input sequence 1101

The sequence detector will produce an output 1 whenever the input sequence 1101 occurs.

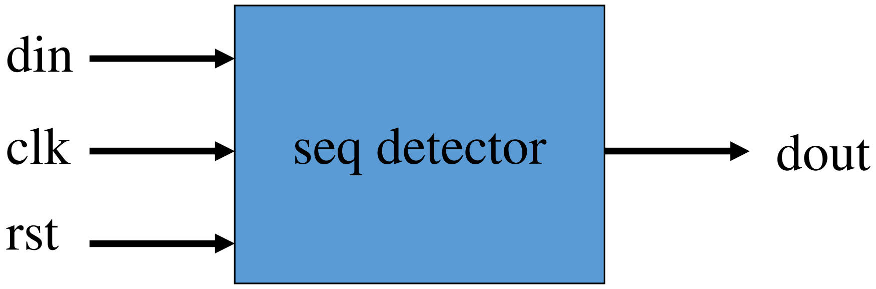


din	1	0	1	1	0	1	1	0	1	0	0	1	1	0	1	0
dout	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0

State Diagram to Detect input sequence 1101



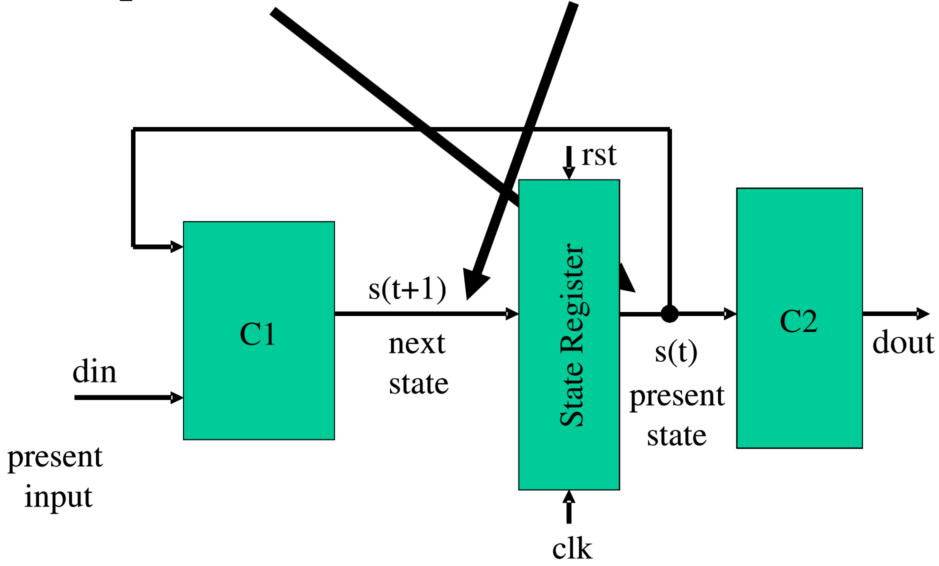
Detect input sequence 1101: Fsm Coding



```
// Sequence detector to Detect 1101 sequence
module seqdet(clk, rst, din, dout);
     input      clk, rst, din;
     output     dout;
     reg        dout;
```

Example2: Detect input sequence 1101

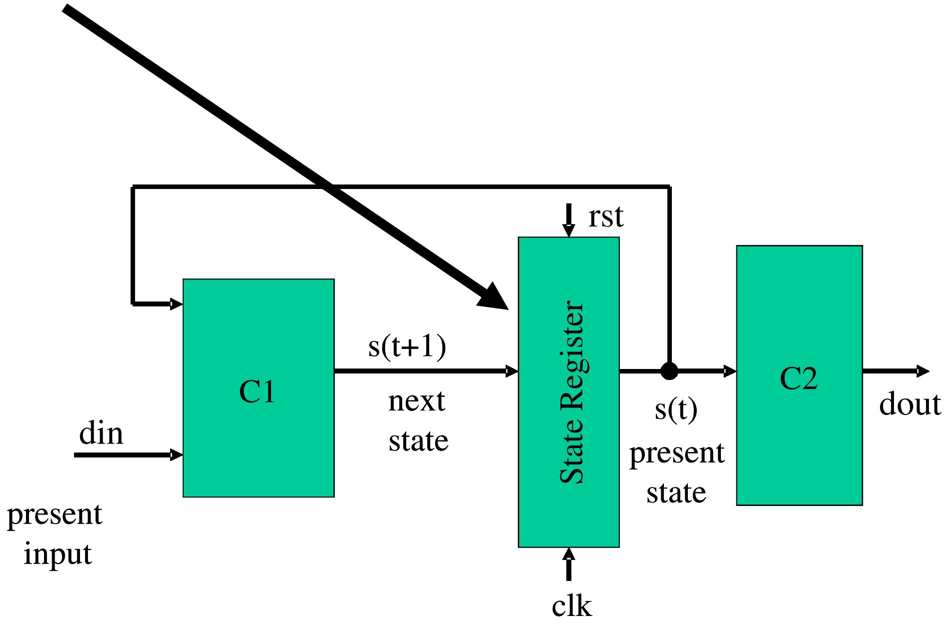
```
reg[2:0] present_state, next_state;
```



```
parameter S0 = 3'b000,S1 =3'b001, S11 = 3'b010,
          S110 = 3'b011, S1101 = 3'b100;
```

Example2: Detect input sequence 1101

```
always @(posedge clk or posedge rst)
       begin
              if (rst == 1)
                     present_state <= S0;
              else
                     present_state <= next_state;
       end
```

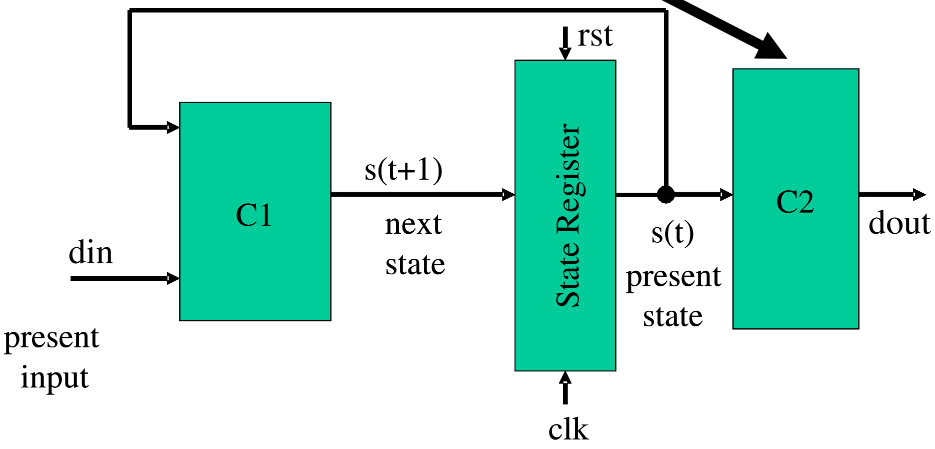


Example2: Detect input sequence 1101

```
// C1: Next State
always @(present_state or din)
 begin
   case(present_state)
       S0: if(din == 1)
             next_state = S1;
           else next_state = S0;
       S1: if(din == 1)
             next_state = S11;
           else next_state = S0;
       S11: if(din == 0)
             next_state = S110;
            else next_state = S11;
       S110: if(din == 1)
                       next_state = S1101;
                   else next_state = S0;
              S1101: if(din == 0)
                       next_state = S0;
                    else next_state = S11;
              default next_state = S0;
          endcase
        end
```

Example2: Detect input sequence 1101

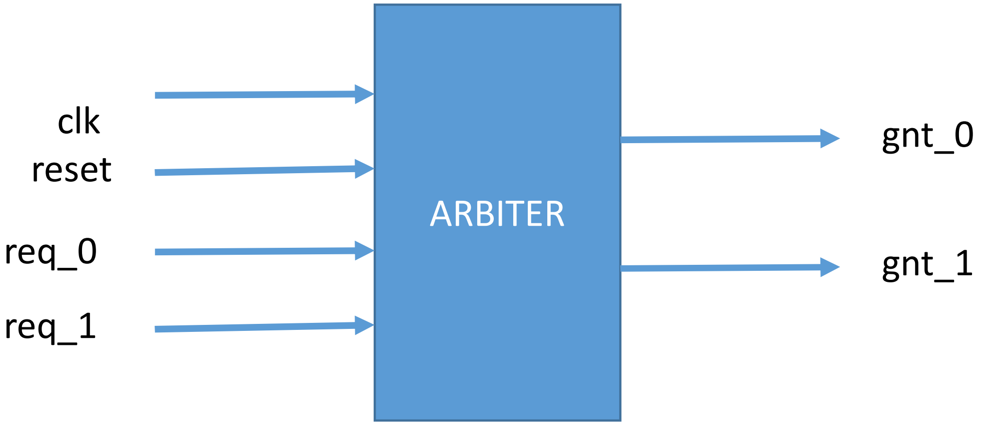
```
// C2: Outputs
        always @(present_state)
        begin
           if(present_state == S1101)
               dout =1;
           else
               dout = 0;
        end
endmodule
```



Example3: FSM for an Arbiter

Lets take a simple arbiter as the example; this has got two request inputs and two grant outputs, as shown in the signal diagram.

- When req_0 is asserted, gnt_0 is asserted
- When req_1 is asserted, gnt_1 is asserted
- When both req_0 and req_1 are asserted then gnt_0 is asserted i.e. higher priority is given to req_0 over req_1.

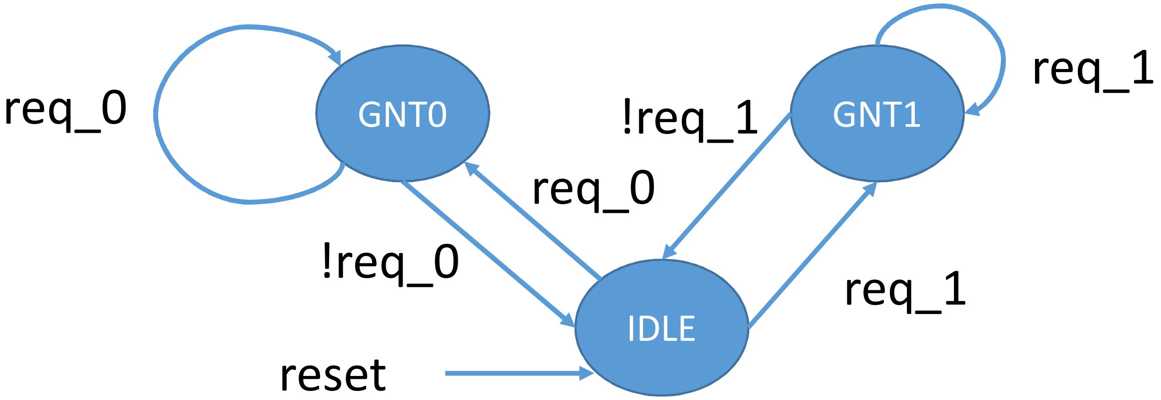


Example3: State Diagram Of Arbiter

IDLE : In this state FSM waits for the assertion of req_0 or req_1 and drives both gnt_0 and gnt_1 to inactive state (low). This is the default state of the FSM, it is entered after the reset and also during fault recovery condition.

GNT0 : FSM enters this state when req_0 is asserted, and remains here as long as req_0 is asserted. When req_0 is de-asserted, FSM returns to the IDLE state.

GNT1 : FSM enters this state when req_1 is asserted, and remains there as long as req_1 is asserted. When req_1 is de-asserted, FSM returns to the IDLE state



Example3: FSM Coding

```
module fsm( clock, reset, req_0, req_1, gnt_0, gnt_1);
input   clock,reset,req_0,req_1;
output  gnt_0,gnt_1;
wire    clock,reset,req_0,req_1;
reg     gnt_0,gnt_1;
parameter SIZE = 3          ;
parameter IDLE  = 3'b001,GNT0 = 3'b010,GNT1 = 3'b100;
reg   [SIZE-1:0]         state        ;// Seq part of the FSM
reg   [SIZE-1:0]         next_state   ;// combo part of FSM
//----------Code startes Here------------------------
always @ (state or req_0 or req_1)
begin : FSM_COMBO
 next_state = 3'b000;
 case(state)
```

```
IDLE : if (req_0 == 1'b1) begin
              next_state = GNT0;
             end else if (req_1 == 1'b1) begin
              next_state= GNT1;
             end else begin
              next_state = IDLE; end
  GNT0 : if (req_0 == 1'b1) begin
              next_state = GNT0;
             end else begin
              next_state = IDLE; end
  GNT1 : if (req_1 == 1'b1) begin
              next_state = GNT1;
             end else begin
              next_state = IDLE; end
  default : next_state = IDLE; endcase end
```

Example3: FSM Coding

```
always @ (posedge clock) // sequential logic
begin : FSM_SEQ
 if (reset == 1'b1) begin
  state <= #1 IDLE;
 end else begin
  state <= #1 next_state; end
end
//----------Output Logic----------------------------
always @ (posedge clock)
begin : OUTPUT_LOGIC
if (reset == 1'b1) begin
 gnt_0 <= #1 1'b0;
 gnt_1 <= #1 1'b0;
end
else begin
```

```
case(state)
  IDLE : begin
         gnt_0 <= #1 1'b0; gnt_1 <= #1 1'b0;
        end
 GNT0 : begin
         gnt_0 <= #1 1'b1; gnt_1 <= #1 1'b0;
        end
 GNT1 : begin
         gnt_0 <= #1 1'b0; gnt_1 <= #1 1'b1;
        end
 default : begin
          gnt_0 <= #1 1'b0; gnt_1 <= #1 1'b0;
         end
 endcase
end end endmodule
```