ECE 4514
Digital Design II
Spring 2008

Lecture 15:
FSM-based Control

A *Design* Lecture

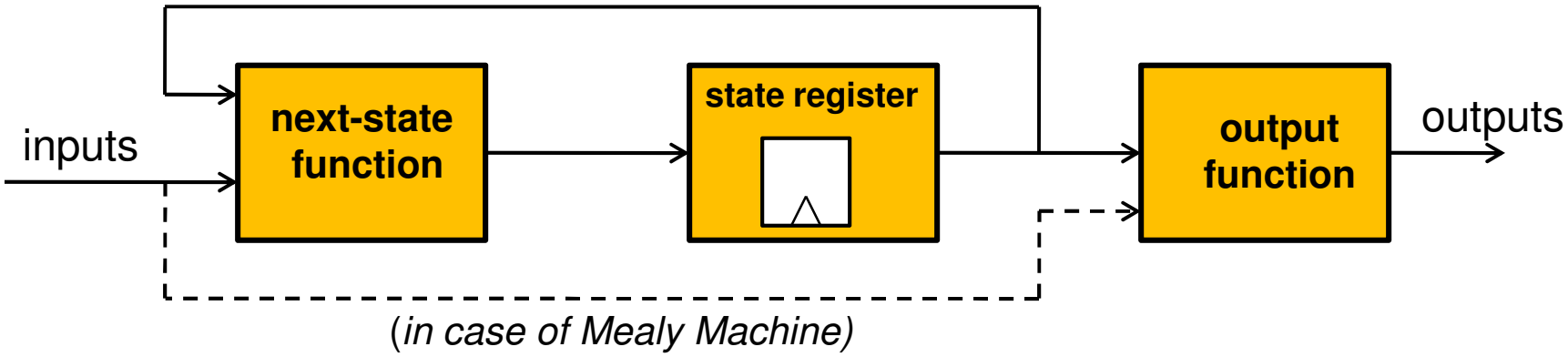
Patrick Schaumont

Overview

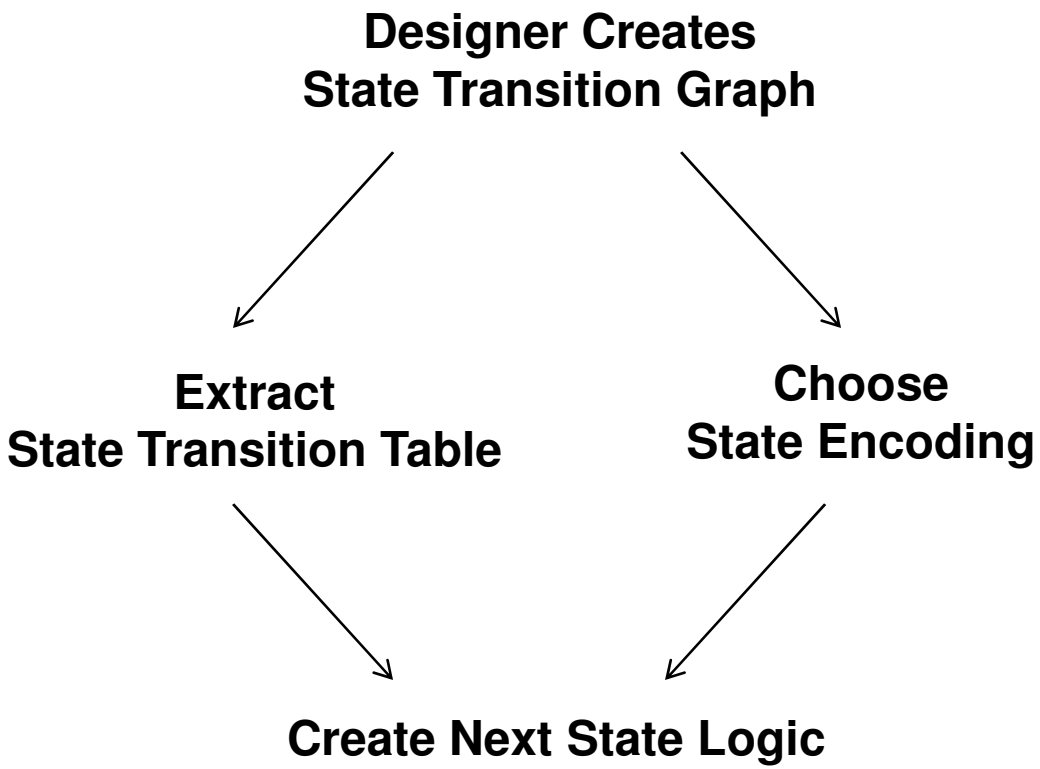
- Finite State Machines
  - Verilog Mapping: one, two, three always blocks
- State Encoding
  - User-defined or tool-defined
  - State encoding techniques: one-hot, user-defined, gray
- Synthesis Issues
  - Default state assignment
  - Safe Implementations
  - Next-state logic: RAM or LUT
- FSM-based control of Datapath (gcd)

Finite State Machine Template

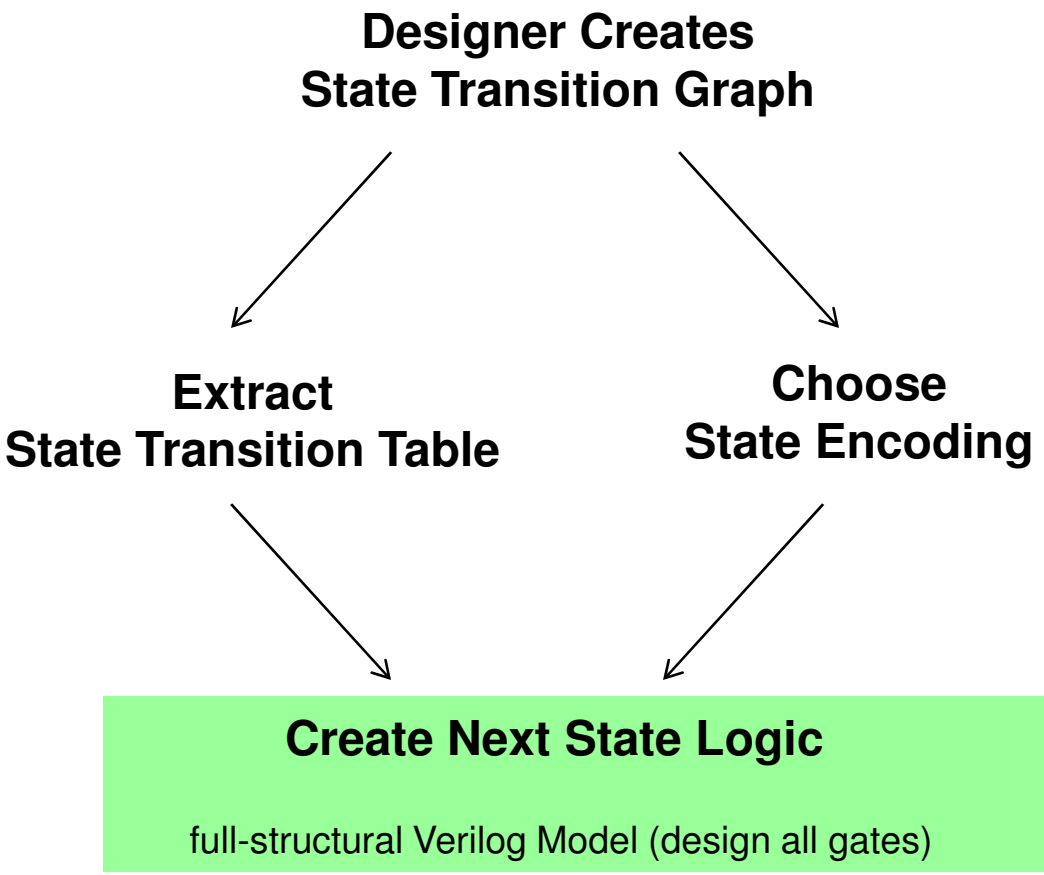
- Sequential Machine defined by
  - Set of Inputs and Outputs
  - Set of States
  - Initial State (reset function)
  - State Transitions



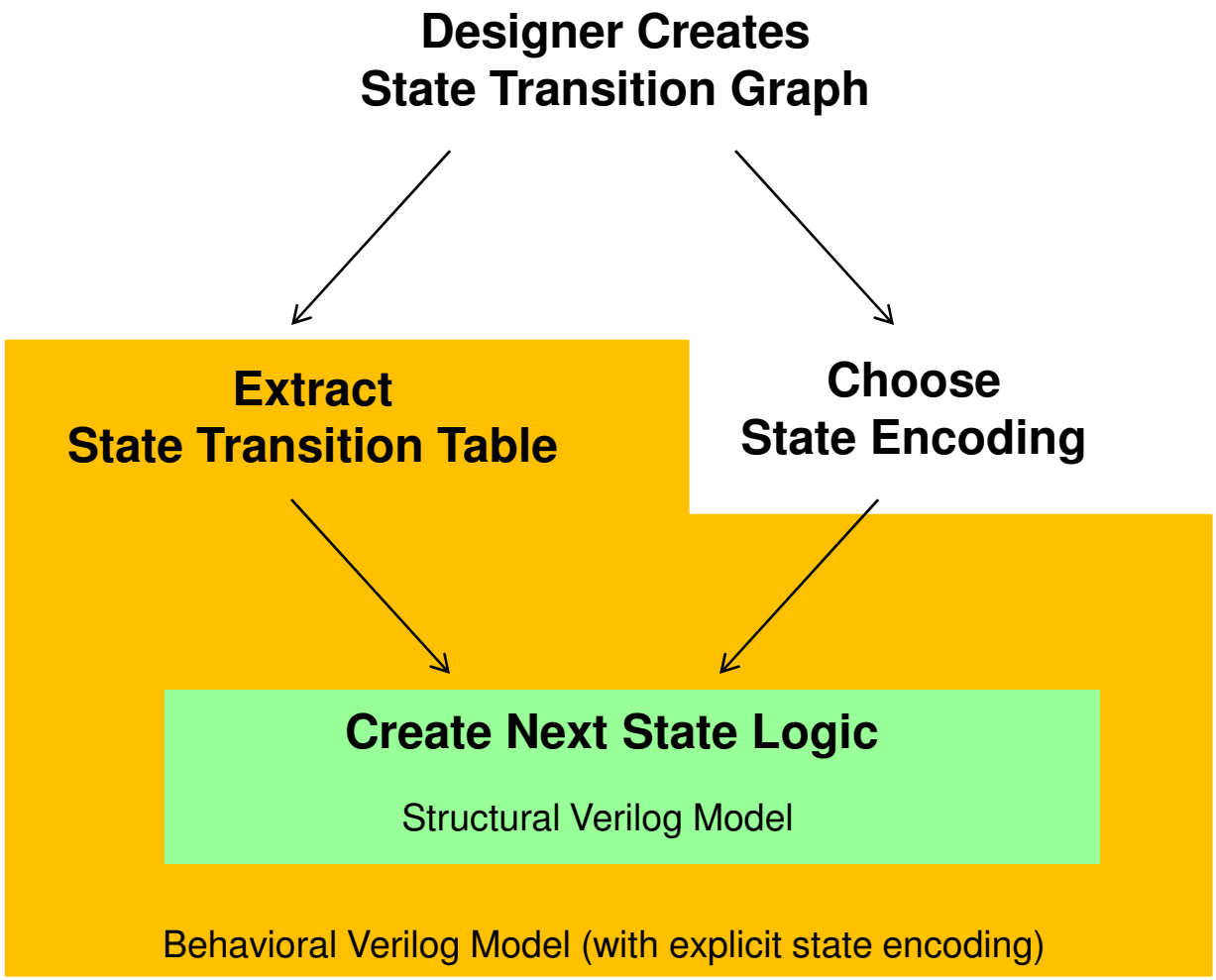
Design Process



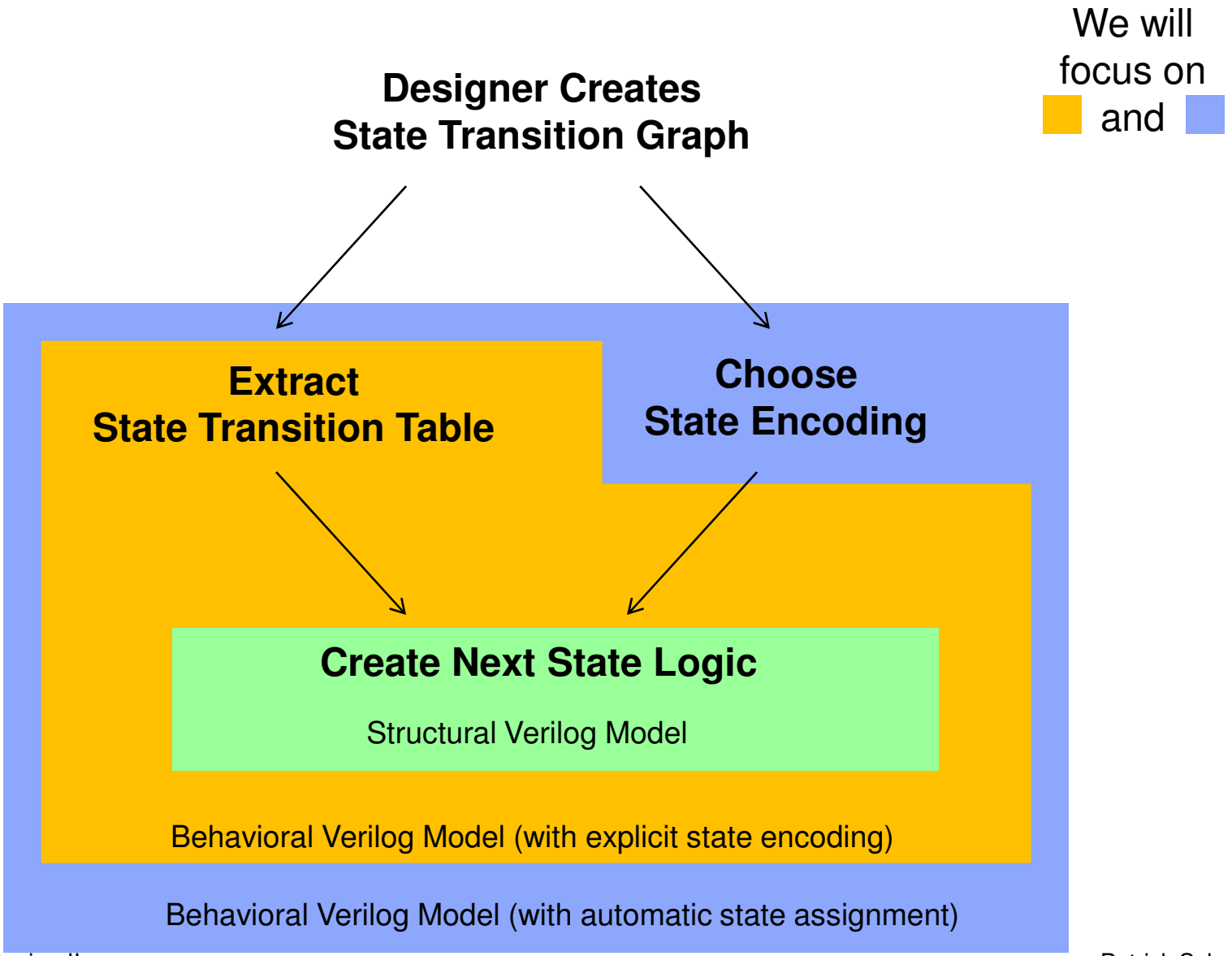
Design Process



Design Process



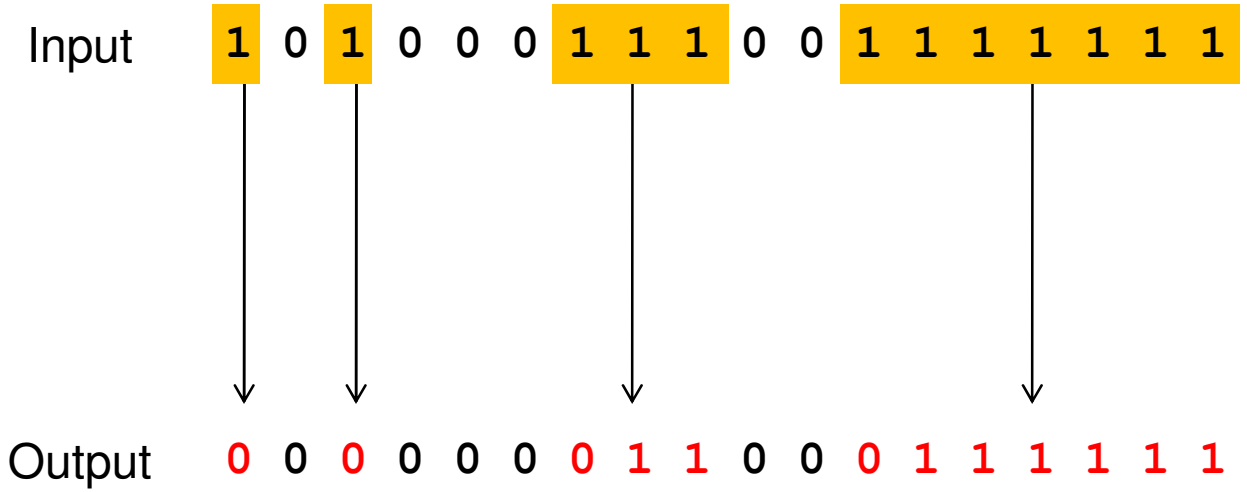
Design Process



Example FSM Design

- *Create an FSM that turns the first '1' of a string of consecutive '1' into a '0'.*

  Each clock cycle, a new input is provided



Example FSM Design

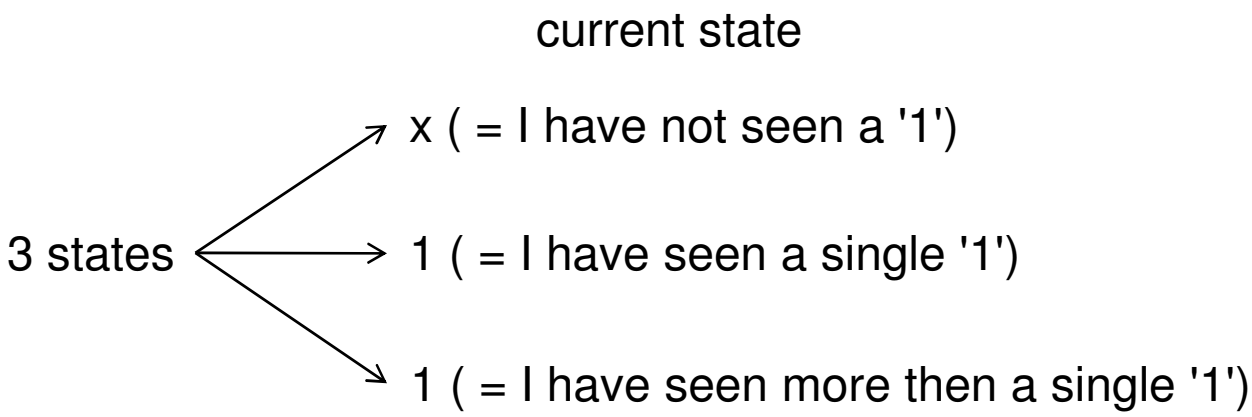
- How many states?

Input	1	0	1	0	0	0	1	1	1	0	0	1	1	1	1	1	1	1
Output	0	0	0	0	0	0	0	1	1	0	0	0	1	1	1	1	1	1

Example FSM Design

- How many states?

Input	1	0	1	0	0	0	1	1	1	0	0	1	1	1	1	1	1	1
Output	0	0	0	0	0	0	0	1	1	0	0	0	1	1	1	1	1	1



Example FSM Design

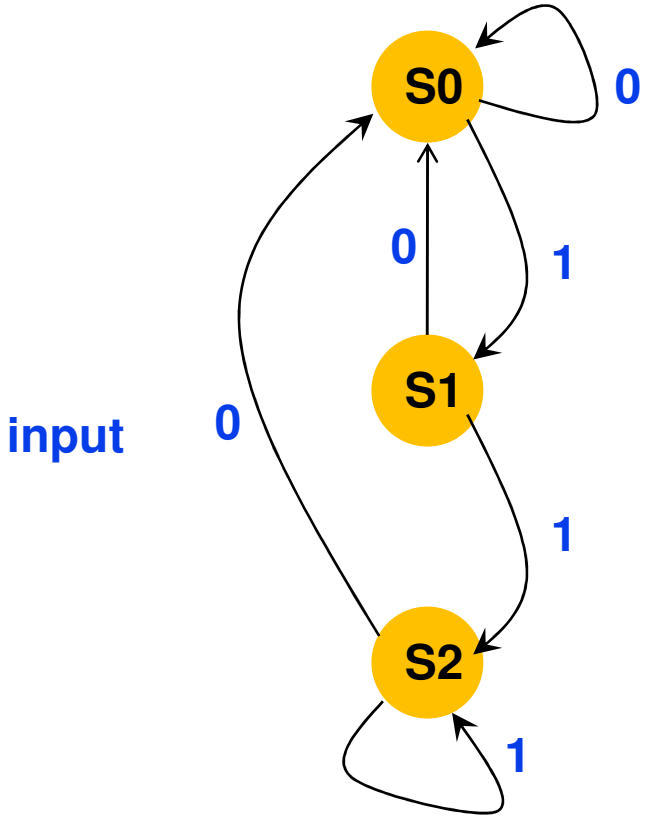
- How many states?

Input	1	0	1	0	0	0	1	1	1	0	0	1	1	1	1	1	1	1
Output	0	0	0	0	0	0	0	1	1	0	0	0	1	1	1	1	1	1

Example FSM Design

- How many states?

Input	1	0	1	0	0	0	1	1	1	0	0	1	1	1	1	1	1	1
Output	0	0	0	0	0	0	0	1	1	0	0	0	1	1	1	1	1	1



output

S0 -> 0
S1 -> 0
S2 -> 1

Verilog Mapping of FSM - Generic Ideas

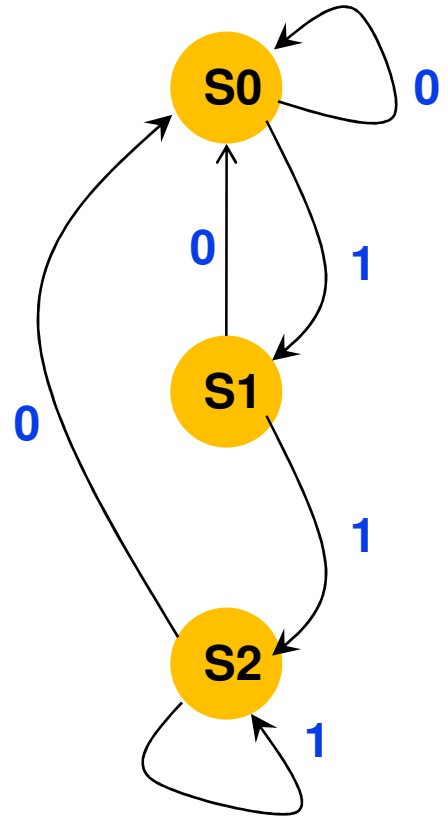
- Use 'parameter' to express state encoding

```
parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10;
```

- This is a *symbolic* encoding, not necessarily the encoding that will be used by the synthesis tool
- Use standard conventions for coding logic
  - Use non-blocking <= for registers, specify correct reset and edge-triggered behavior
  - Use blocking = for combinational logic, make sure sensitivity lists are complete

Example FSM Design - Verilog Mapping I

```
module fsm(q, i, clk, rst);
  input i, clk, rst;
  output q;
  reg q;
  reg [1:0] state;
  parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10;

  always @(posedge clk or posedge rst)
    if (rst) begin
      state <= s0; q <= 1'b0;
    end else begin
      case (state)
        s0: if (i == 1'b1) begin
              state <= s1;
              q     <= 1'b0;
            end else begin
              state <= s0;
              q     <= 1'b0;
            end
        s1: ..
        s2: ..
      endcase
    end
endmodule
```

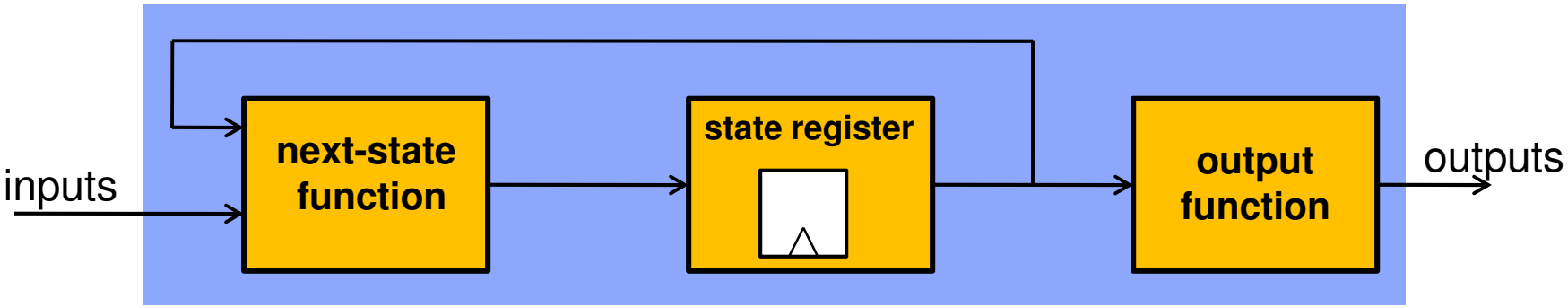


output

S0 -> 0
S1 -> 0
S2 -> 1

Example FSM Design - Verilog Mapping I

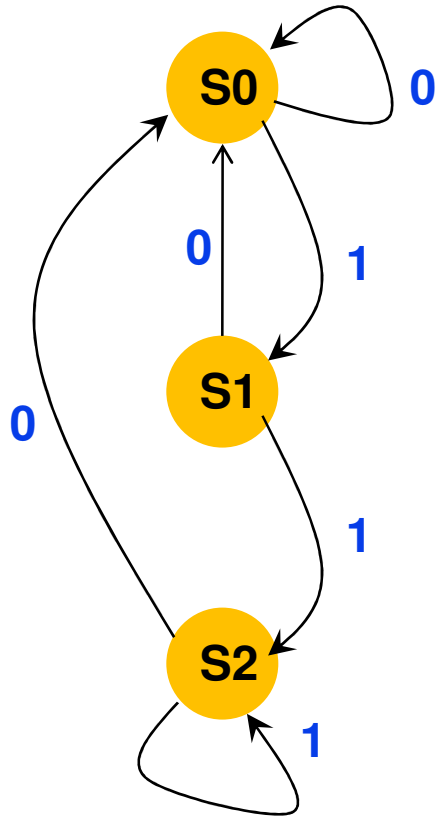
- One single always block
  - Reset and update in same block
  - Output will always have a register (not always desired ..)



Example FSM Design - Verilog Mapping II

```
reg q;
reg [1:0] state;
parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10;

always @(posedge clk or posedge rst)
  if (rst)
    state <= s0;
  else
    case (state)
      s0: if (i == 1'b1)
            state <= s1;
          else
            state <= s0;
      s1: ..
    endcase
```

```
always @(state)
   q = 1'b0;    // default assignment
   case (state)
     s0: q = 1'b0;
     s1: q = 1'b0;
     s2: q = 1'b1;
   endcase
endmodule
```



output

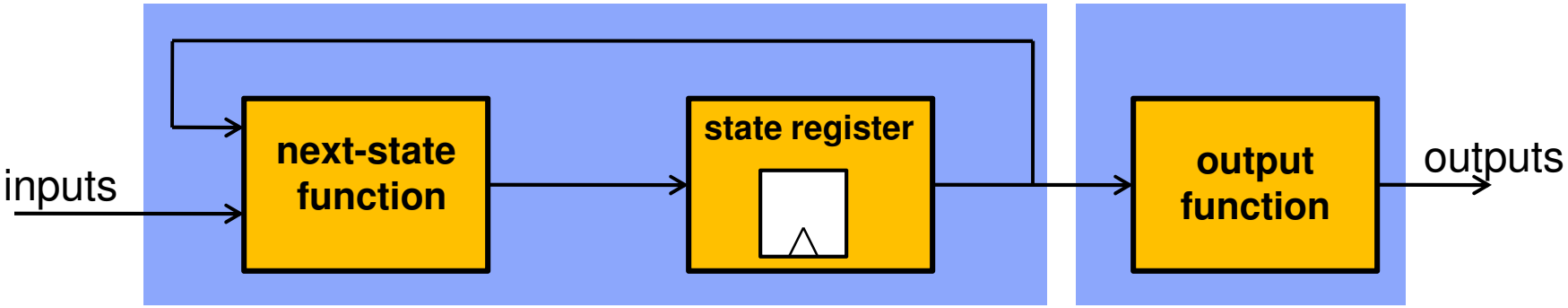
S0 -> 0

S1 -> 0

S2 -> 1

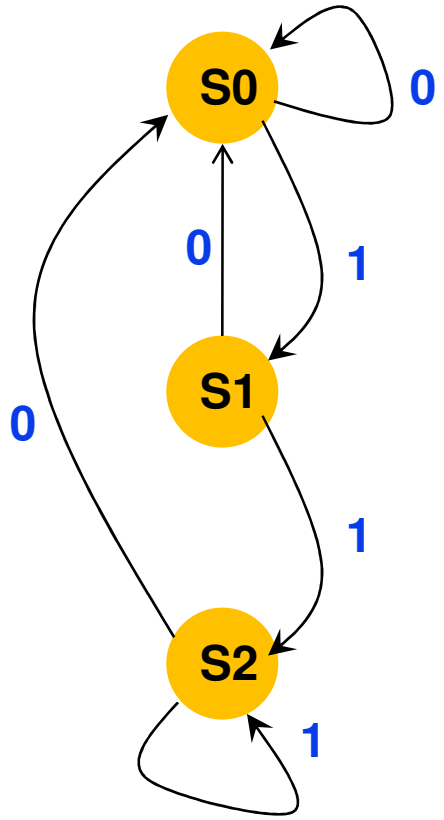
Example FSM Design - Verilog Mapping II

- Two always block
  - Reset and update still in same block
  - Output can be combinational



Example FSM Design - Verilog Mapping III

```
reg q;
reg [1:0] state, next_state;
parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10;

always @(posedge clk or posedge rst)
  if (rst)
    state <= s0;
  else
    state <= next_state;

always @(state or i) begin
    next_state = s0;
    case (state)
      s0: if (i == 1'b1)
            next_state = s1;
          else
            next_state = s0;
      s1: ..
    endcase
end

always @(state)
  // output encoding
endmodule
```



output

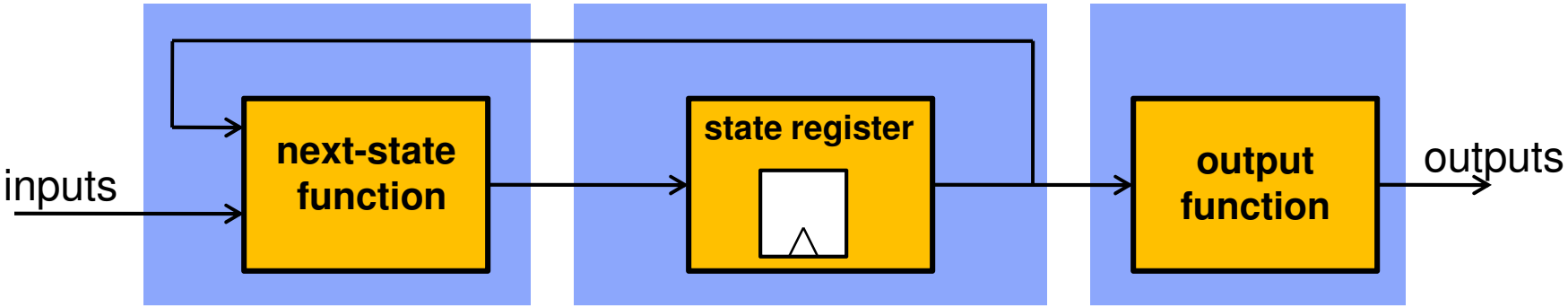
S0 -> 0

S1 -> 0

S2 -> 1

Example FSM Design - Verilog Mapping III

- Three always block
  - Reset and update in different blocks
  - Output can be combinational



One, two, three always block

	1 always block	2 always block	3 always block
Combinational Output		√	√
Reset/Update separate from next-state Logic			√

Synthesis of FSM

- Output of XST on Example 1 (single always block):

```
=========================================================================
*                            HDL Synthesis                              *
=========================================================================

Performing bidirectional port resolution...

Synthesizing Unit <myfsm>.
    Related source file is "myfsm.v".
    Found finite state machine <FSM_0> for signal <state>.
    -----------------------------------------------------------------------
    | States             | 3                                              |
    | Transitions        | 6                                              |
    | Inputs             | 1                                              |
    | Outputs            | 3                                              |
    | Clock              | clk (rising_edge)                              |
    | Reset              | rst (positive)                                 |
    | Reset type         | asynchronous                                   |
    | Reset State        | 00                                             |
    | Encoding           | automatic                                      |
    | Implementation     | LUT                                            |
    -----------------------------------------------------------------------
    Found 1-bit register for signal <q>.
    Summary:
          inferred   1 Finite State Machine(s).
          inferred   1 D-type flip-flop(s).
Unit <myfsm> synthesized.
```

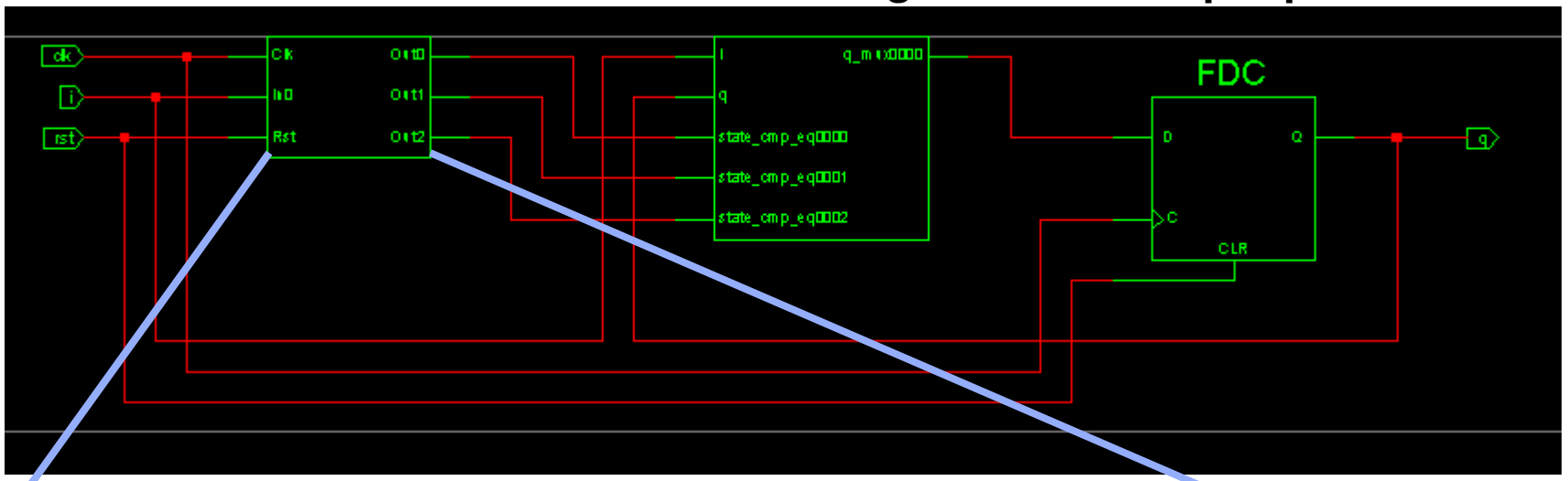
Synthesis tool treats FSM as a separate entity

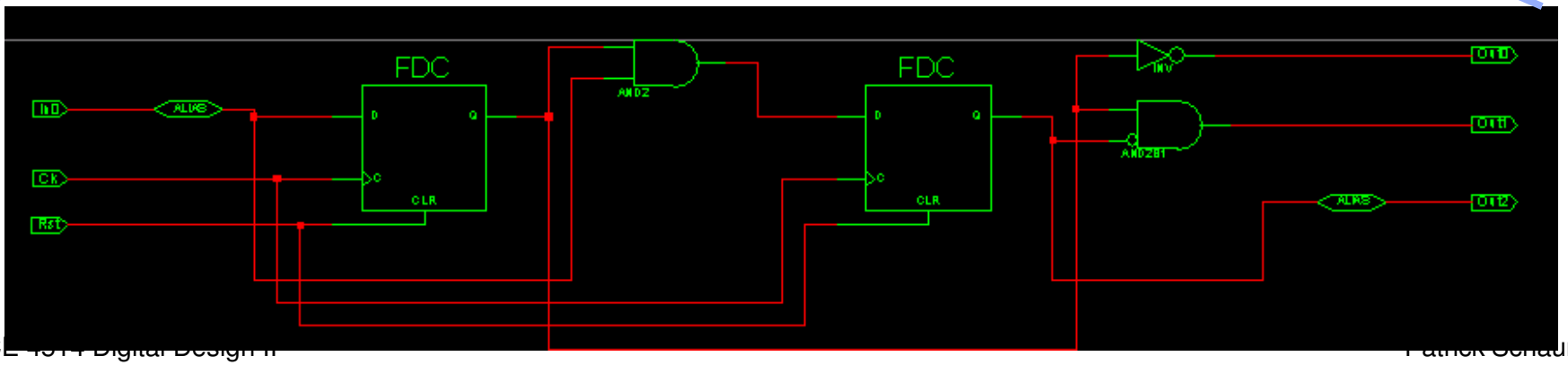
RTL Schematic

State Machine

Output Encoding

Output Flipflop





Synthesis of FSM

```
=========================================================================
*                        Advanced HDL Synthesis                         *
=========================================================================

Analyzing FSM <FSM_0> for best encoding.
Optimizing FSM <state> on signal <state[1:2]> with gray encoding.
-------------------
 State  | Encoding
-------------------
 00     | 00
 01     | 01
 10     | 11
-------------------
```

This encoding style was chosen by the tool (encoding is selected to be 'automatic')

This encoding is used in actual implementation

This state was chosen in parameter entry

```
parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10;
```

Gray Encoding ?

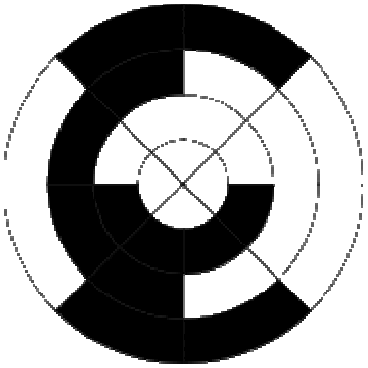
- A binary code in which subsequent words differ by only a single bit

1-bit gray code

0
1

2-bit gray code

00
01
mirror
11
10



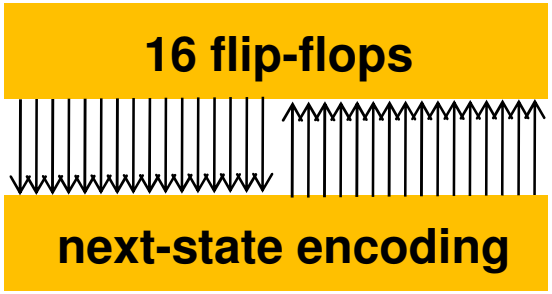
3-bit gray code

000
001
011
010
mirror
110
111
101
100

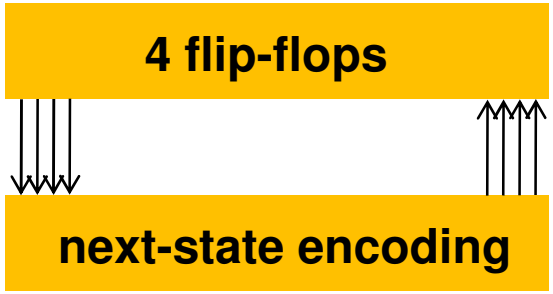
Gray Codes require minimal logic transitions to go from one state to the next.
(= mininal power consumption, minimal risk for glitches & hazards)

What other types of encoding are common ?

- One-hot:
  - Select one bit for each state
  - Each state transition flips only two bits
  - Good for FPGA (with lots of flip-flops)
  - Not so efficient with large number of states or large number of state transitions, because state-decoding becomes too big



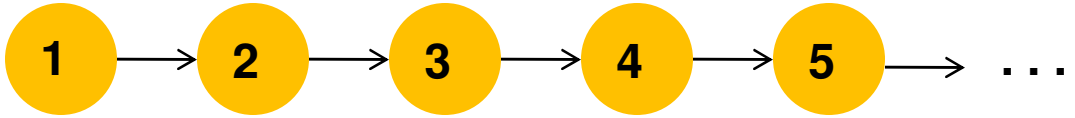
one-hot encoding
with 16 states



normal encoding
with 16 states

What other types of encoding are common ?

- Sequential:
  - Encode successive states in sequence
  - Simplifies next-state logic if there are long sequences



- User-defined:
  - Keeps the same encoding as specified by the designer

```
parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10;
```

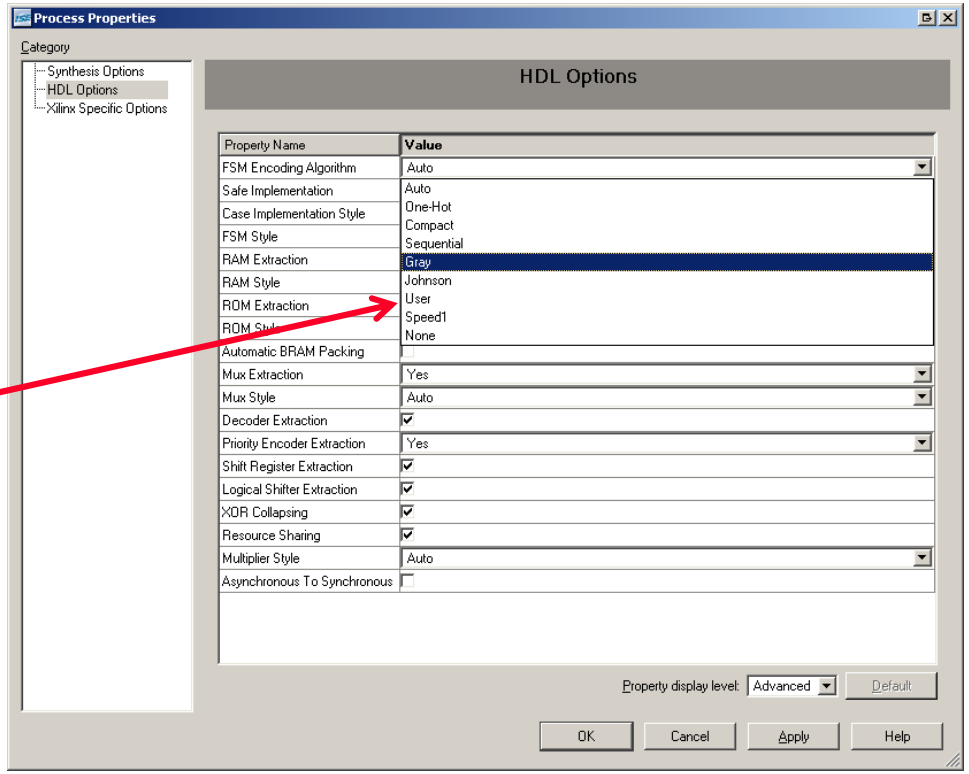
- Synthesis tool tries to select a good encoding automatically
  - This choice is only a heuristic !

Can we enforce the encoding?

- What if we insist that the encoding should be User-defined:

```
parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10;
```

- Solution 1: Change encoding option in the synthesis tool



Select User Encoding

Can we enforce the encoding?

- What if we insist that the encoding should be User-defined:

```
parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10;
```

- Solution 2: Provide pragma in the Verilog Code
  (This is also called a *synthesis constraint*)

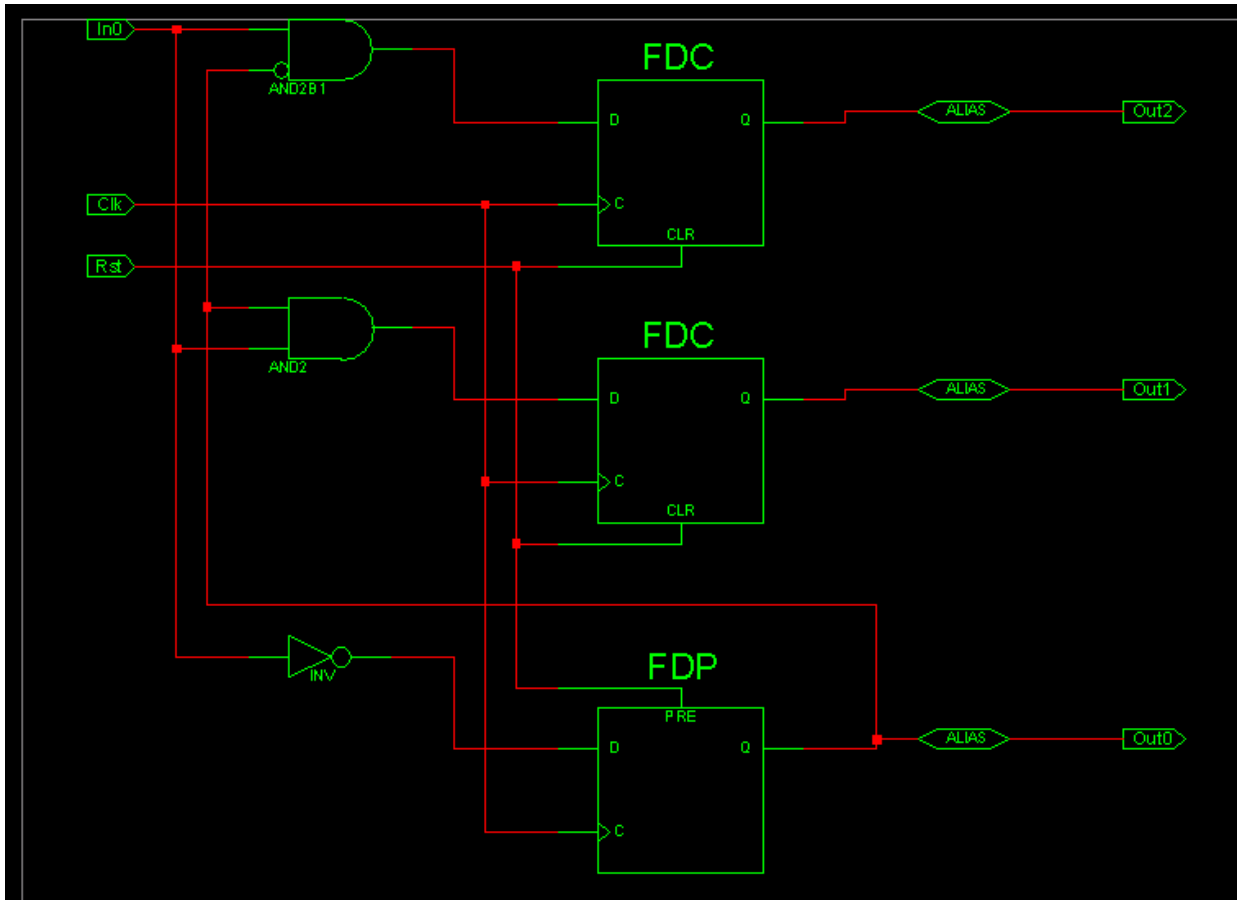
```
(* fsm_encoding = user *)

module fsm(q, i, clk, rst);
  input i, clk, rst;
  output q;
  reg q;
  reg [1:0] state;
  parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10;
  ...
endmodule
```

Example 1 with one-hot encoding

```
(* fsm_encoding = one-hot *)

module fsm(q, i, clk, rst);
  input i, clk, rst;
  output q;
  reg q;
  reg [1:0] state;
  parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10;
  ...
endmodule
```

```
=========================================================================
*                        Advanced HDL Synthesis                         *
=========================================================================

Optimizing FSM <state> on signal <state[1:3]> with one-hot encoding.
-------------------
 State | Encoding
-------------------
 00    | 001
 01    | 010
 10    | 100
-------------------
```

Example 1 with one-hot encoding



Another way of writing one-hot state encoding

```
reg q;
reg [2:0] state, next_state;
parameter s0 = 3'd0, s1 = 3'd1, s2 = 3'd2;

always @(posedge clk or posedge rst)
  if (rst)
    state <= 3'd1;
  else
    state <= next_state;

always @(state or i) begin
    next_state = 3'd0;
    case (1'b1)
      state[s0]: if (i == 1'b0)
                   next_state[s1] = 1'd1;
                 else
                   next_state[s0] = 1'd1;
      state[s1]: ..
    endcase
end
```

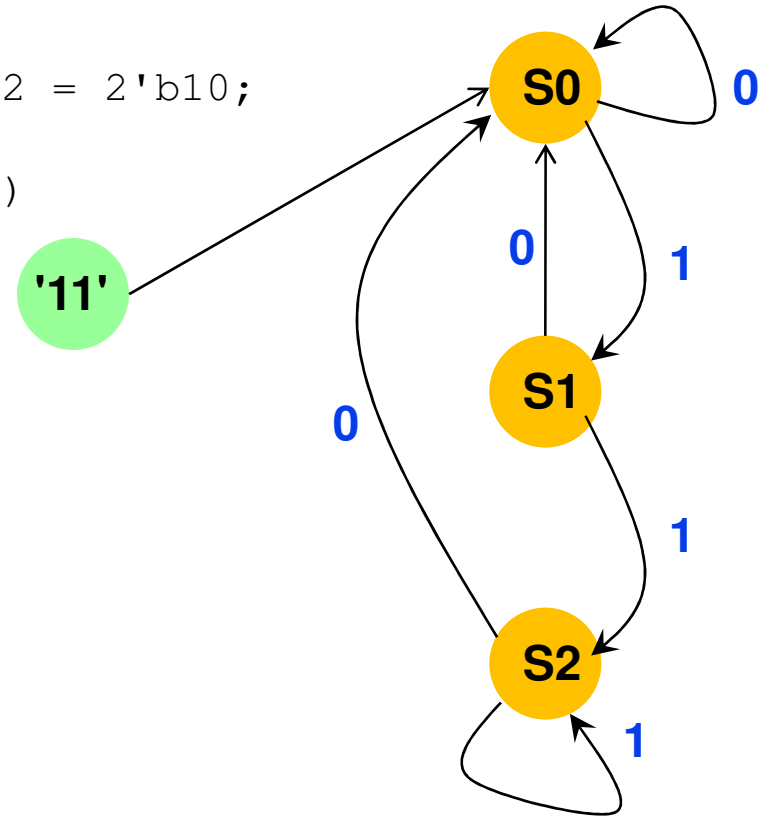
index into state register

next-state selection

This example is not recognized as a finite state machine by XST, so that the user-specified encoding will be used.

Default State Assignment

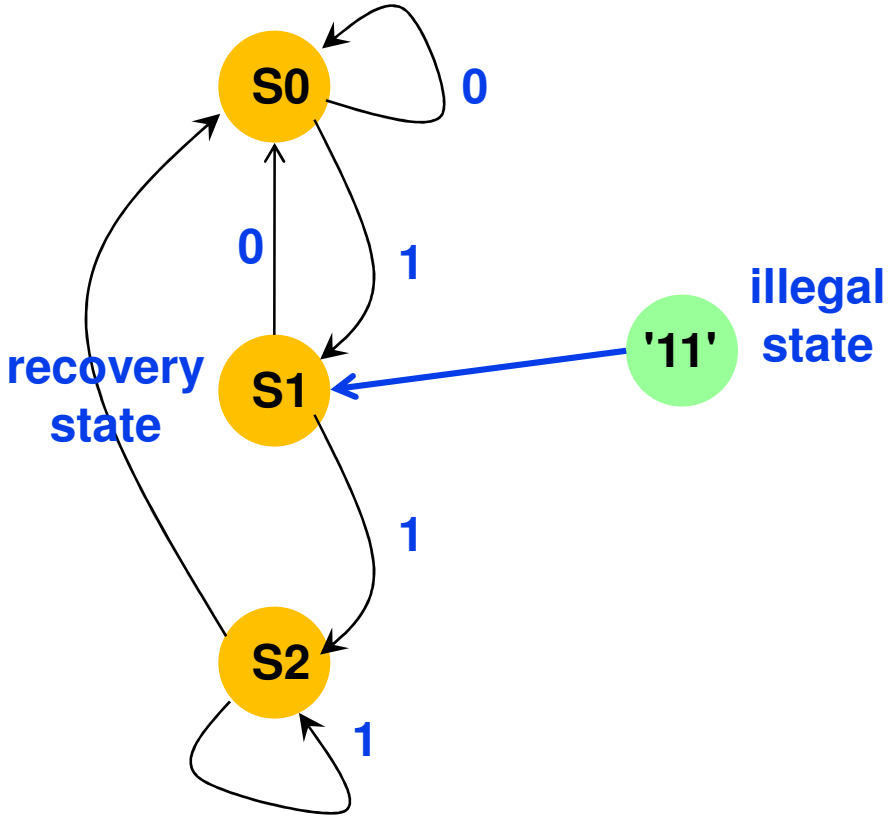
```
reg q;
reg [1:0] state, next_state;
parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10;

always @(posedge clk or posedge rst)
  if (rst)
    state <= s0;
  else
    state <= next_state;

always @(state or i) begin
    next_state = s0;
    case (state)
      s0: if (i == 1'b0)
            next_state = s1;
          else
            next_state = s0;
      s1: ..
    endcase
end

always @(state)
  // output encoding
endmodule
```



Default next-state assignment may deal with improper initialization and faults

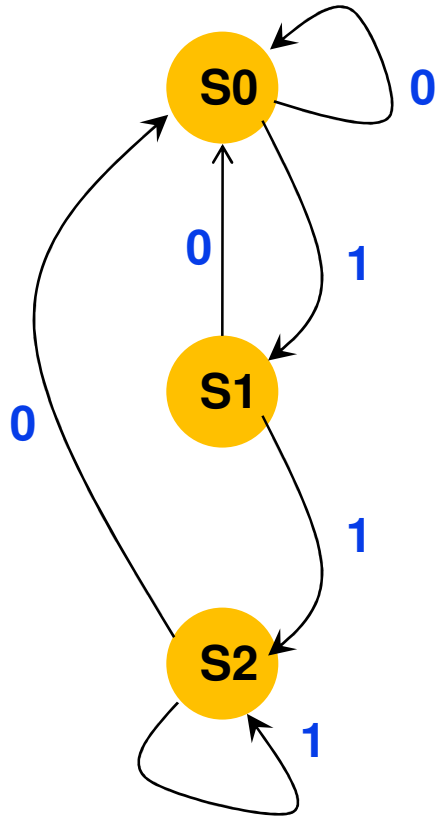
Safe Implementation

- A 'Safe FSM' is an FSM that will convert each invalid state automatically to a recovery state (usually the reset state)



Default (unsafe) implementation

```
module fsm(q, i, clk, rst);
  input i, clk, rst;
  output q;
  reg q;
  reg [1:0] state;
  parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10;

  always @(posedge clk or posedge rst)
    if (rst) begin
      state <= s0; q <= 1'b0;
    end else begin
      case (state)
        s0: if (i == 1'b0) begin
              state <= s1;
              q     <= 1'b0;
            end else begin
              state <= s0;
              q     <= 1'b0;
            end
        s1: ..
        s2: ..
      endcase
    end
endmodule
```



output

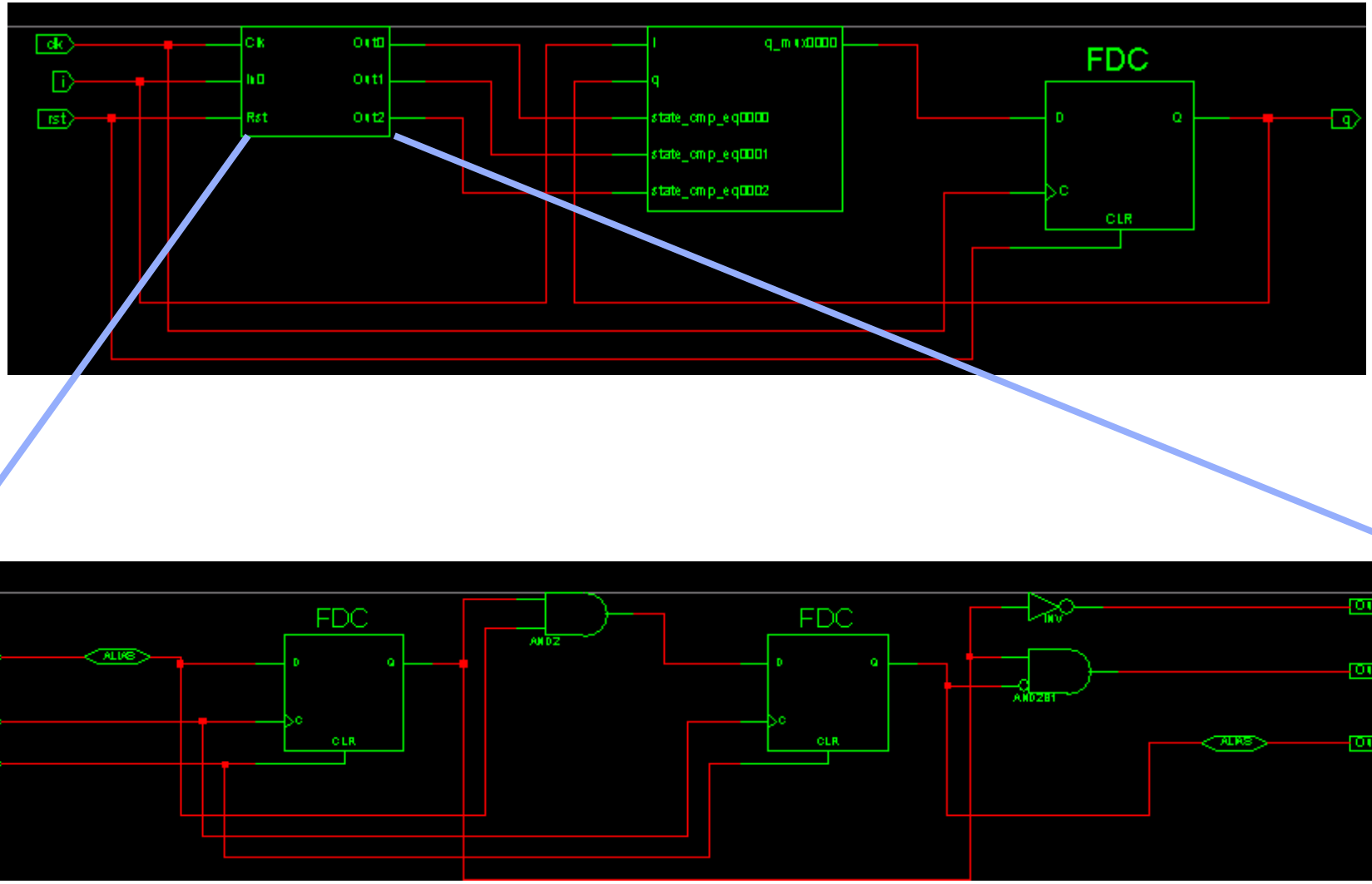
S0 -> 0

S1 -> 0

S2 -> 1

Cannot 'escape' from state '11'

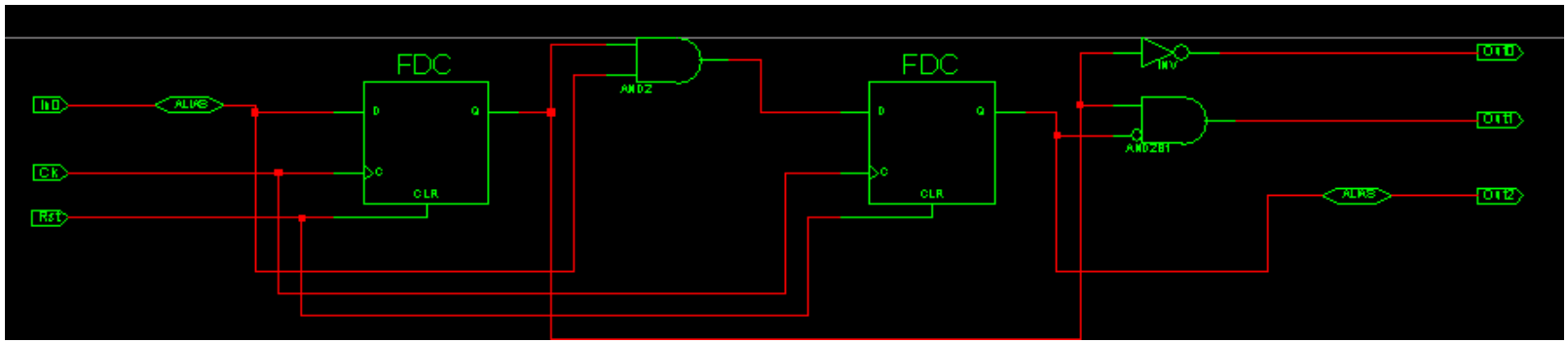
Default (unsafe) implementation

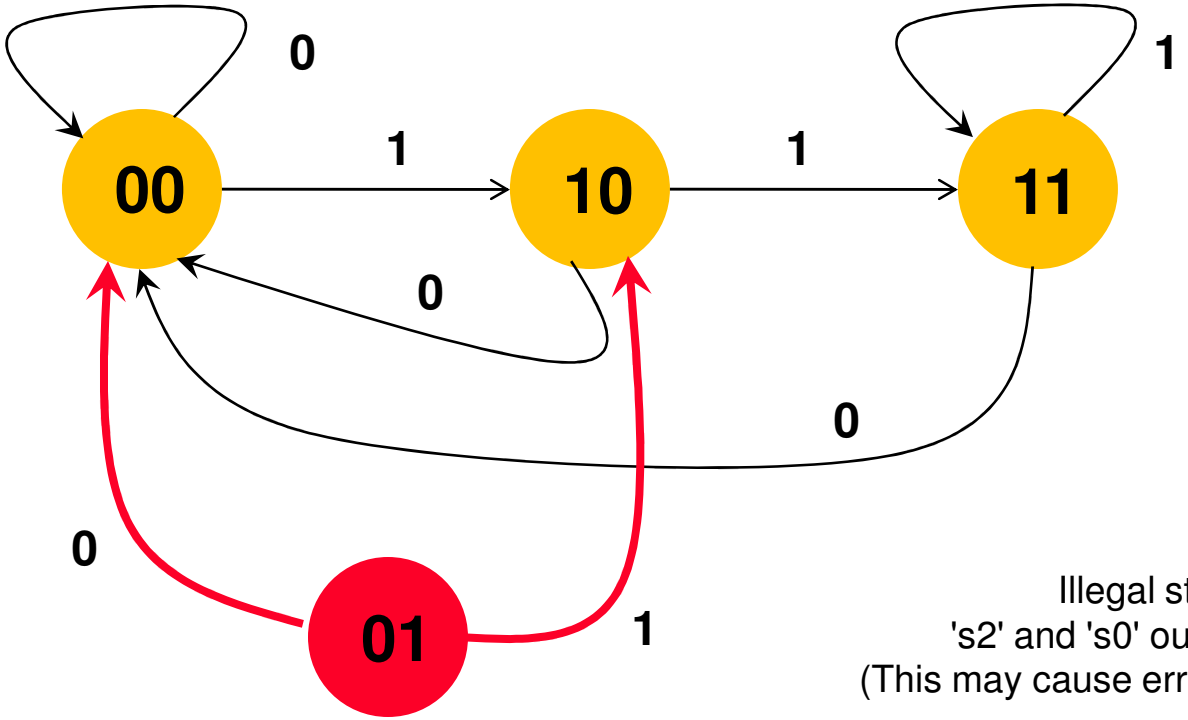


state flip-flop

state flip-flop

Default (unsafe) implementation

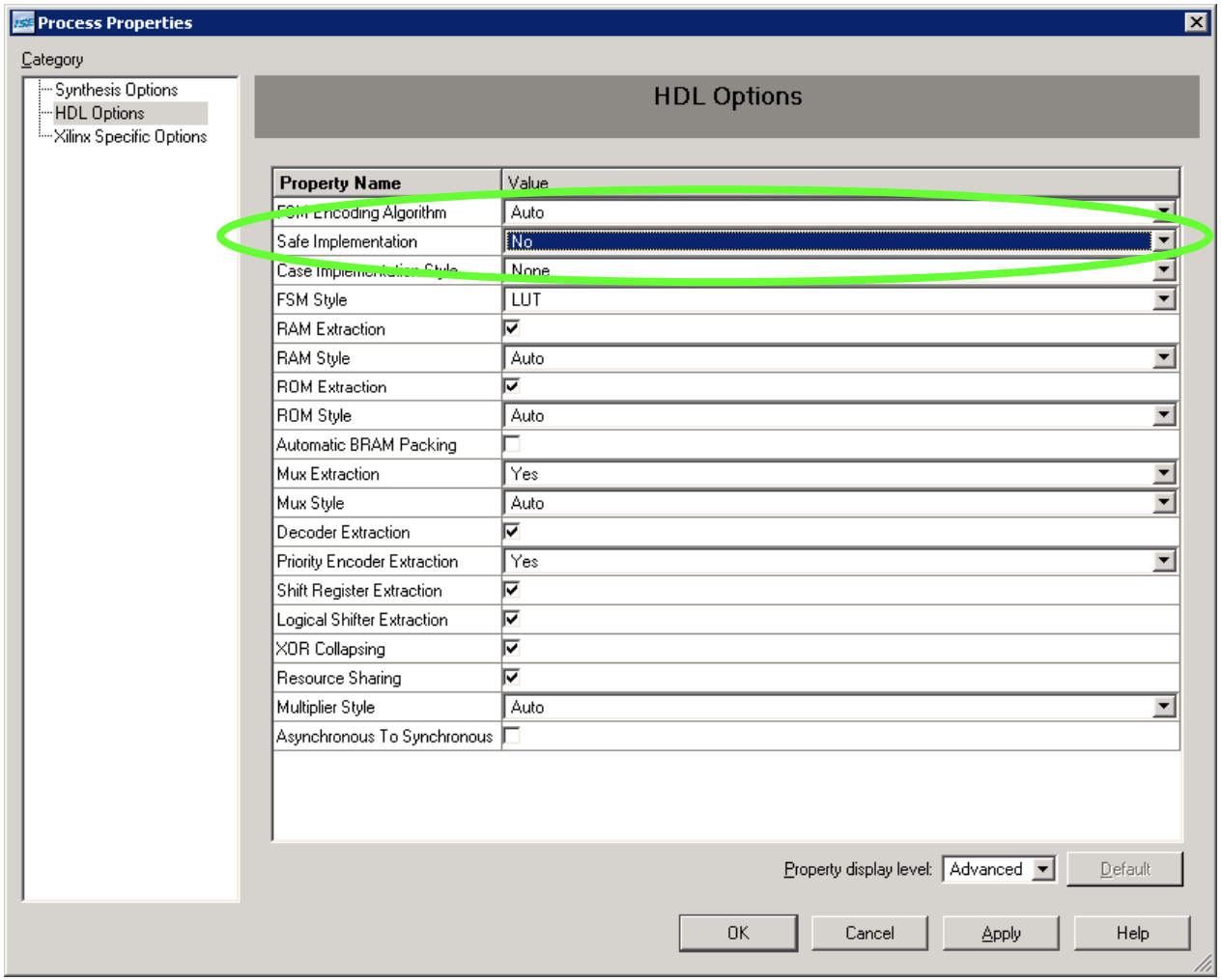




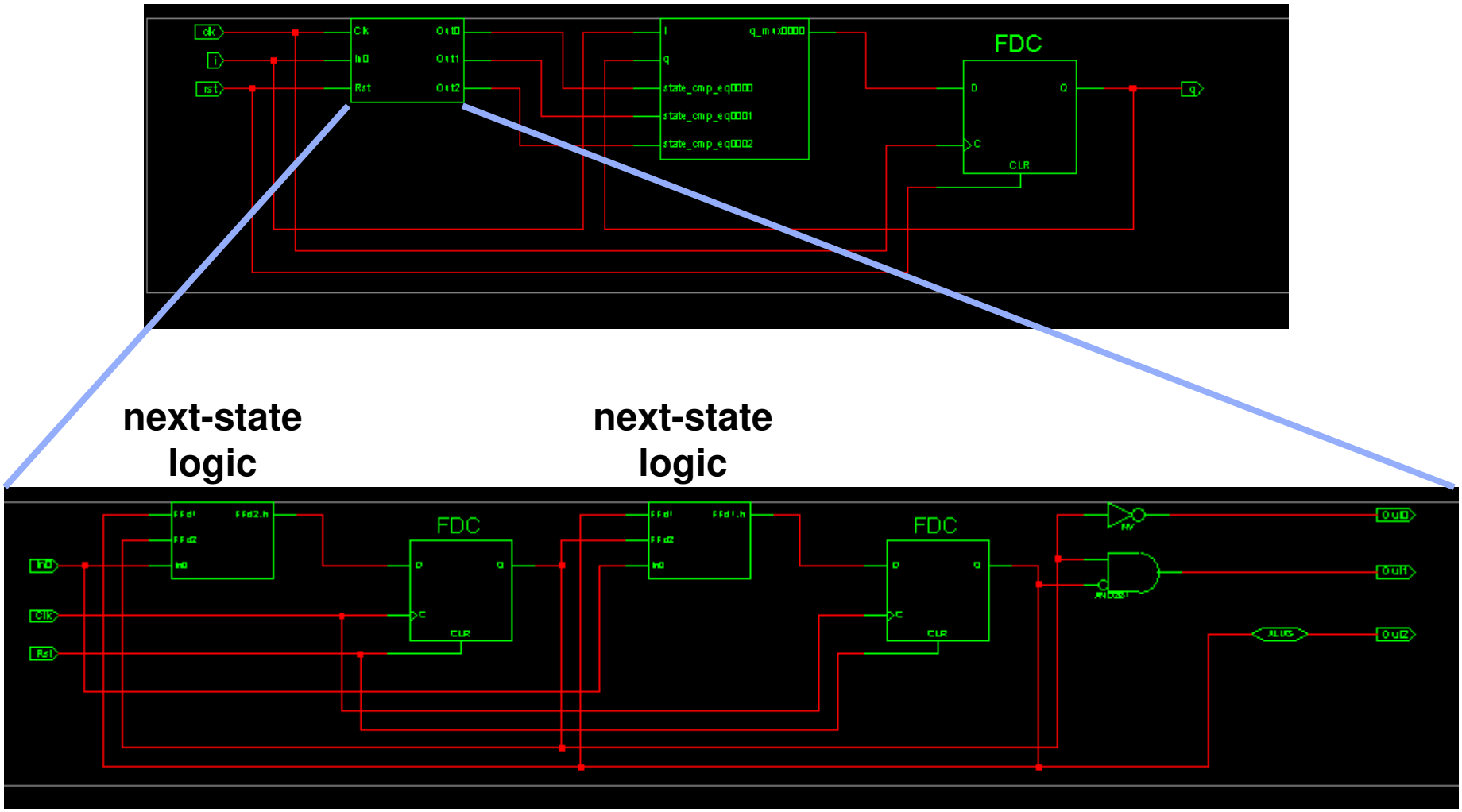
Illegal state '01' will drive
's2' and 's0' output simultaneouly high
(This may cause errors in output encoding block)

Safe Implementation

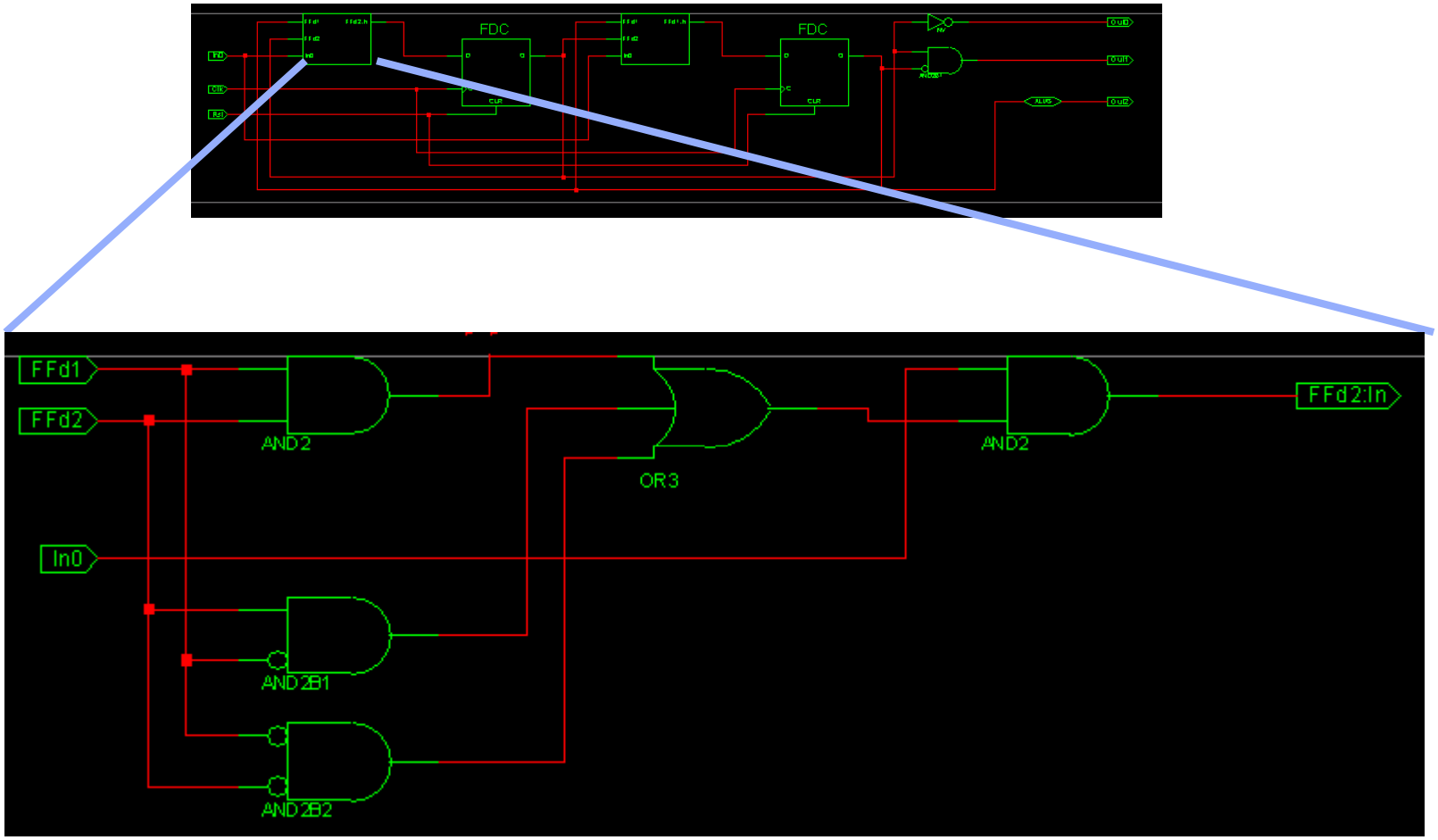
- Specified with tool option or Verilog code pragma



Safe Implementation

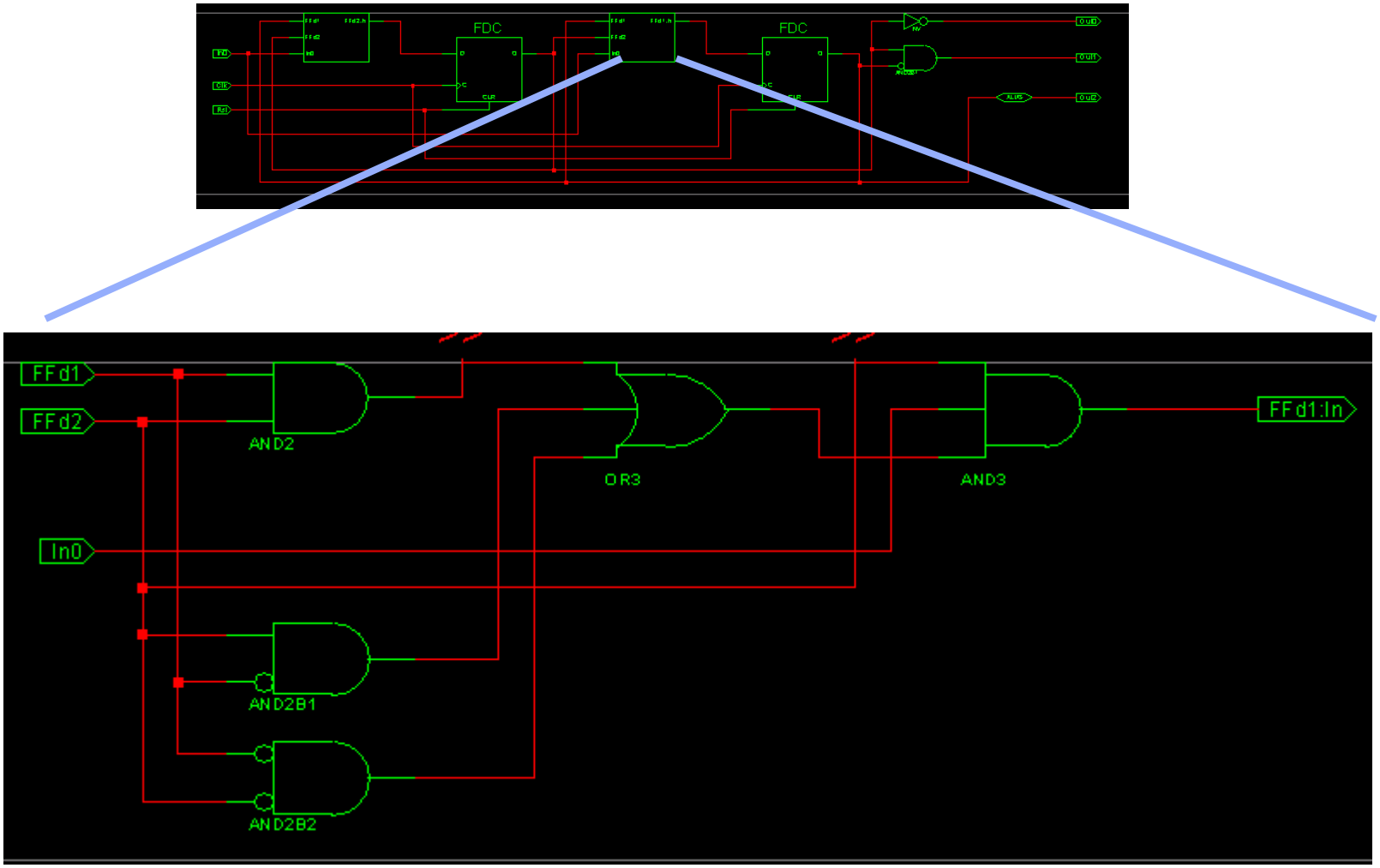


Safe Implementation



f2 = (f1.f2 + !f1.f2 + !f1.!f2) . (in1)

Safe Implementation



f1 = (f1.f2 + !f1.f2 + !f1.!f2) . (in.f2)

Safe Implementation

f1 = (f1.f2 + !f1.f2 + !f1.!f2) . (in.f2)

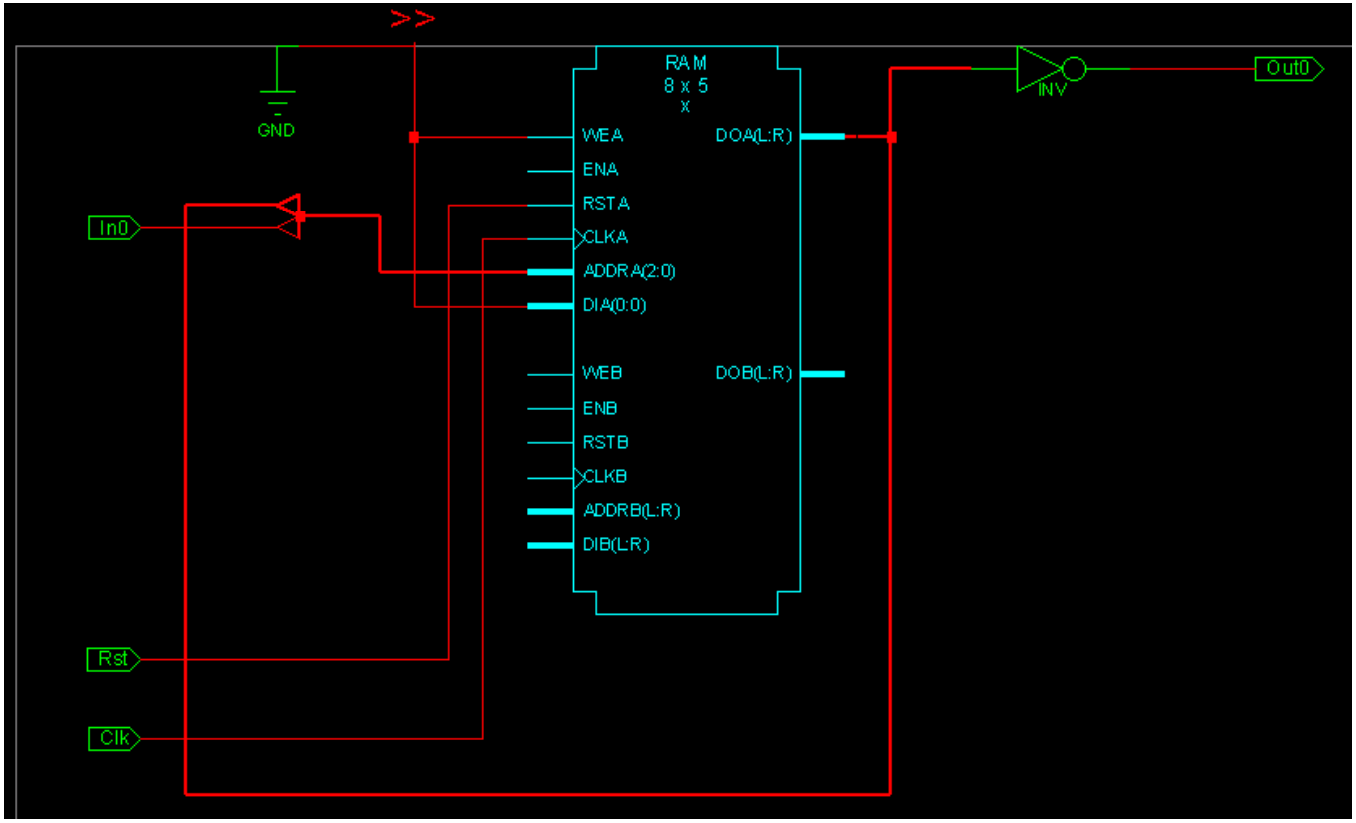
f2 = (f1.f2 + !f1.f2 + !f1.!f2) . (in1)

⟶ **f1. f2 will always be one of {00, 01, 11}.**
State 10 cannot appear (will be immediately converted into 00)

- Safe Implementations are used when proper, precisely defined operation is more important than cost (area, performance)
- Related concept: fault tolerance.
  - Design of digital architectures that can withstand the presence of faults
  - Usually implemented using redundancy (replicating functions in time or space).

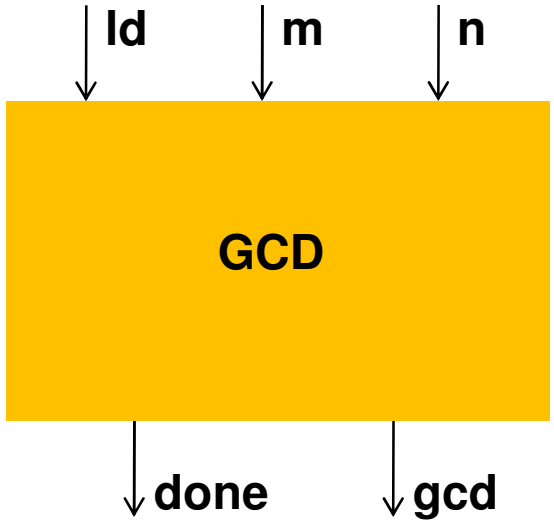
RAM-based next-state logic

- In FPGA, on-chip synchronous BRAM can be used for next-state logic and state register
- Selected as a synthesis option; default is LUT-based FSM



Mixing Datapath and FSM

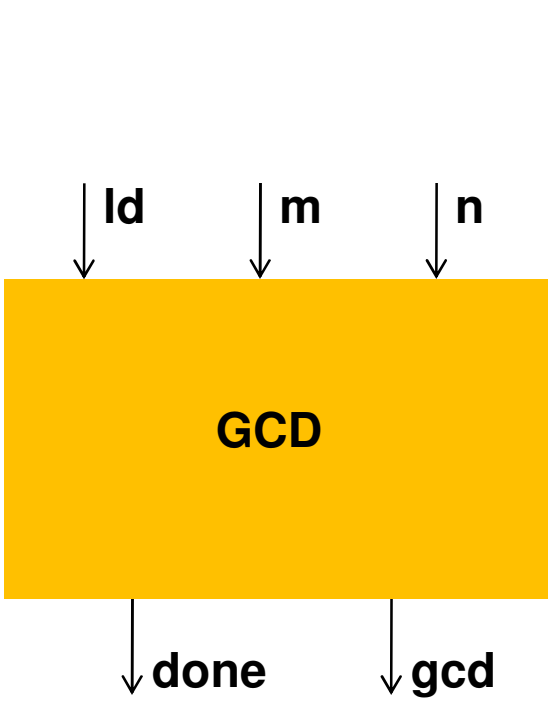
- Example - Greatest Common Divisor

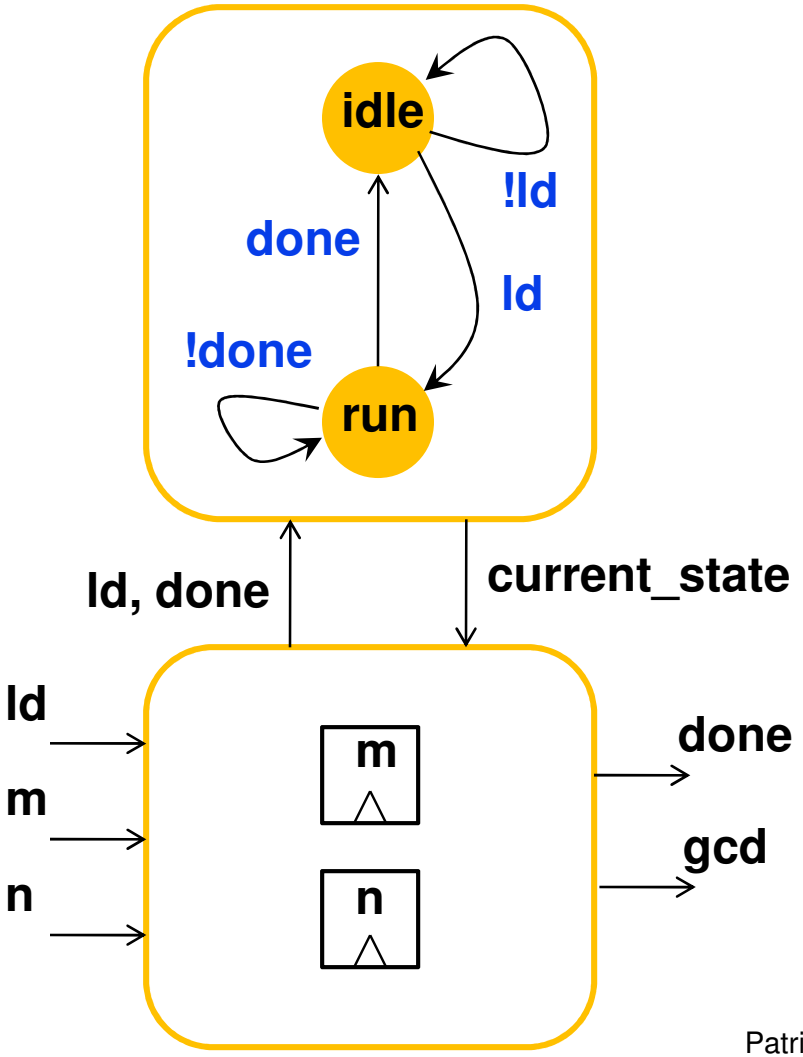



```
GCD: while (m != n) {
       if (m > n)
         m = m - n;
       else
         n = n - m;
     }
     output = m;
```

Mixing Datapath and FSM

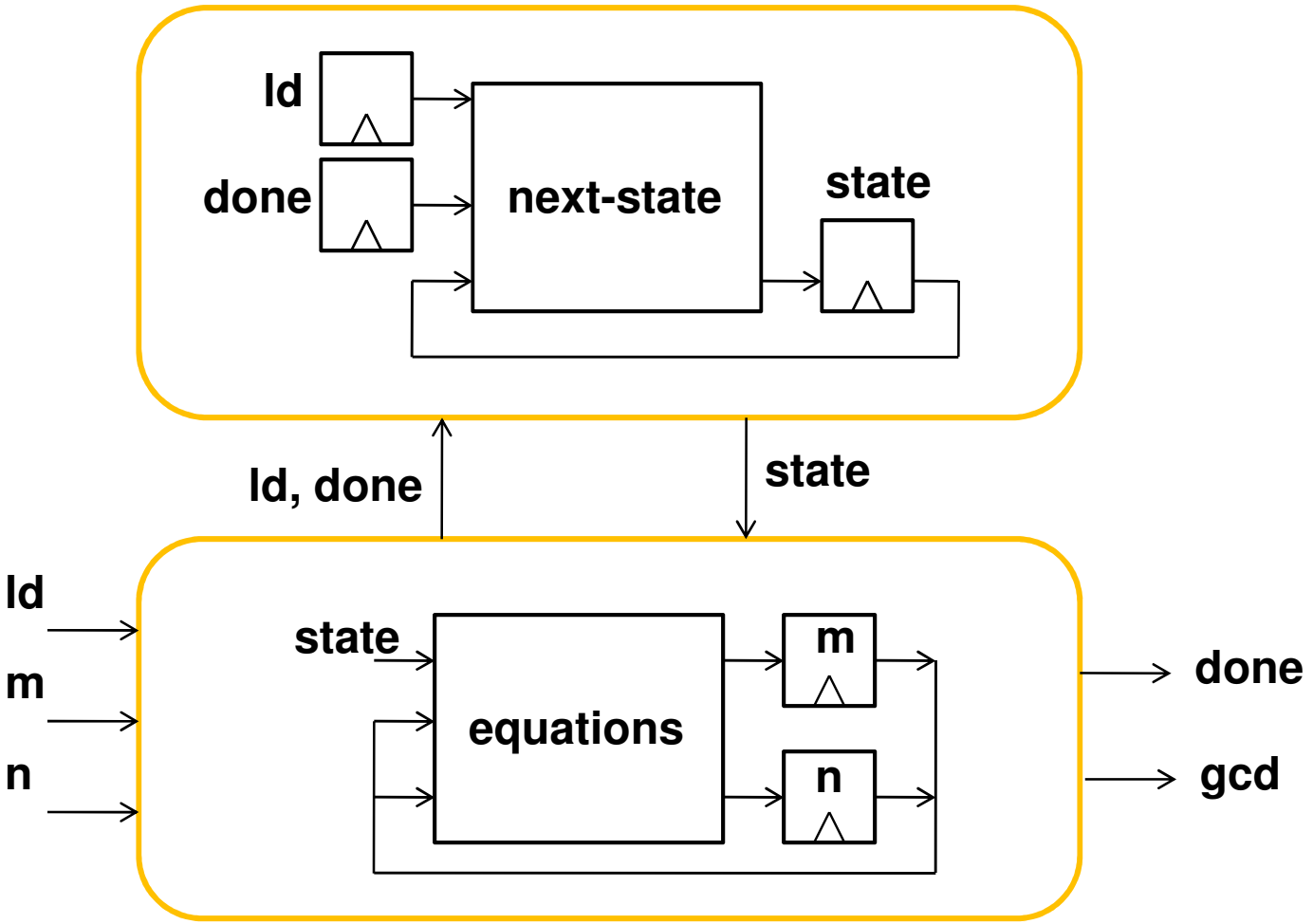
- Example - Greatest Common Divisor





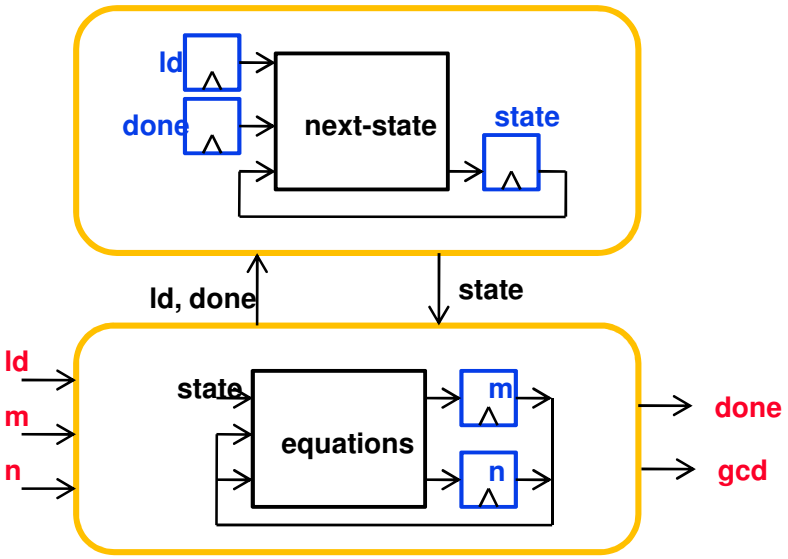
Mixing Datapath and FSM

- Example - Greatest Common Divisor



GCD I/O and State

```
module gcd(q, done, m, n, ld, clk, rst);
  output [9:0] q;
  reg [9:0] q;
  output done;
  reg done;
  input [9:0] m, n;
  input ld;
  input clk;
  input rst;

  reg state;
  reg [9:0] reg_m, reg_n;
  reg reg_ld;
  parameter swait = 1'b0, srun = 1'b1;
```



GCD Single-case statement design

```
module gcd(q, done, m, n, ld, clk, rst);
...
always @(posedge clk) begin

  state  <= swait;  // default reg assignment (essential for control-related
  reg_ld <= ld;     // state, optional for datapath-related state)
  done   <= 1'b0;

  case (state)

    swait: begin
      if (reg_ld)
        state <= srun;
      else
        state <= swait;
      reg_m <= m;   // read new input
      reg_n <= n;
      end

    srun:  begin
      if (done)
        state <= swait;
      else
        state <= srun;
      reg_m    <= (reg_m > reg_n) ? reg_m - reg_n : reg_m;  // GCD iteration
      reg_n    <= (reg_m > reg_n) ? reg_n : reg_n - reg_m;
      q        <=  reg_m;
      done     <= (reg_m == reg_n);
      end
   endcase
end
```

GCD Single-case statement design

```
module gcd(q, done, m, n, ld, clk, rst);
...
always @(posedge clk) begin

  state  <= swait;  // default reg assignment (essential for control-related
  reg_ld <= ld;     // state, optional for datapath-related state)
  done   <= 1'b0;

  case (state)

    swait: begin
      if (reg_ld)
        state <= srun;
      else
        state <= swait;
      reg_m <= m;   // read new input
      reg_n <= n;
      end

    srun:  begin
      if (done)
        state <= swait;
      else
        state <= srun;
      reg_m    <= (reg_m > reg_n) ? reg_m - reg_n : reg_m;  // GCD iteration
      reg_n    <= (reg_m > reg_n) ? reg_n : reg_n - reg_m;
      q        <=  reg_m;
      done     <= (reg_m == reg_n);
      end
  endcase
end
```

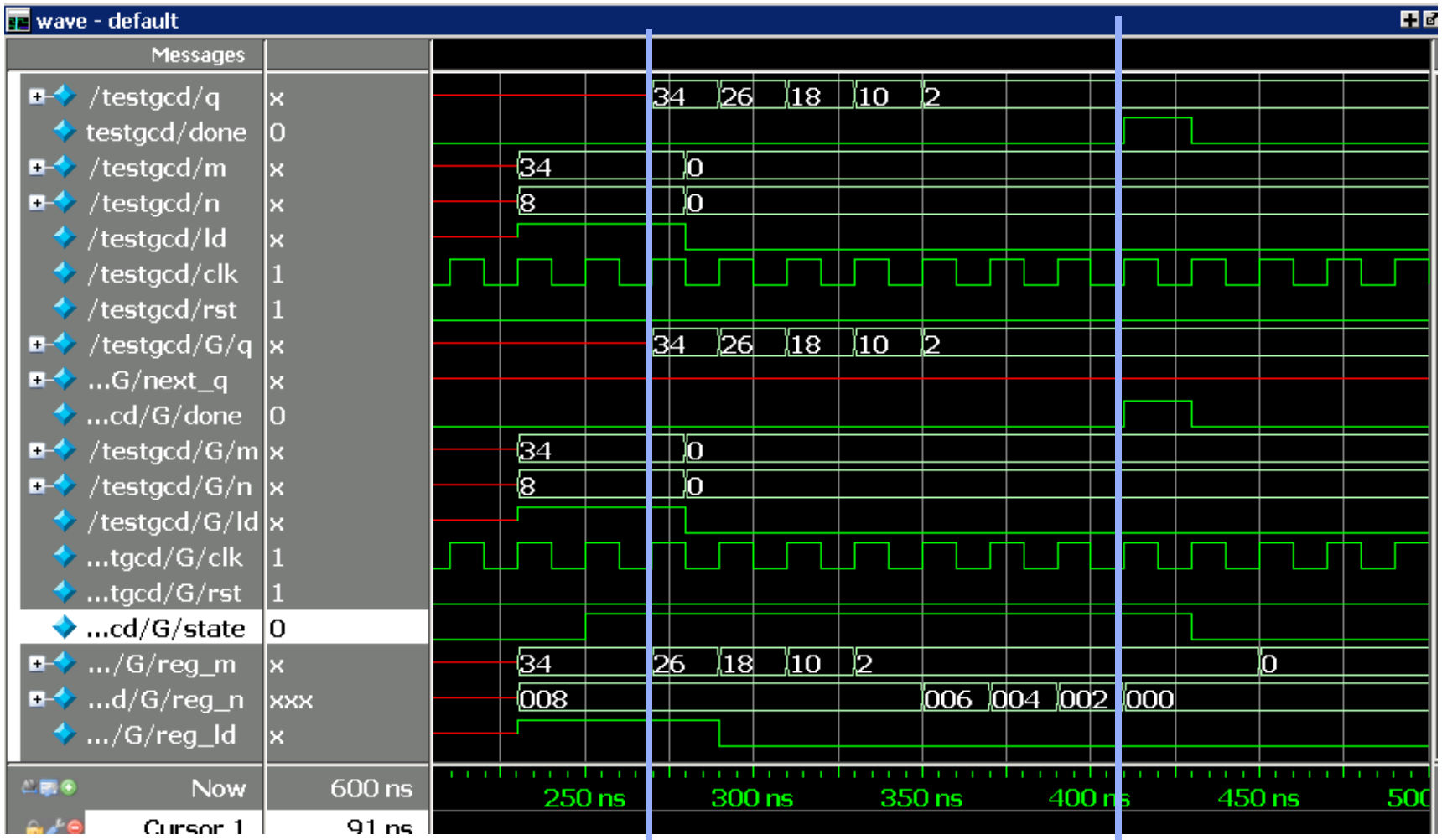
Each state specifies state transitions + datapath operations

Synthesis

```
Advanced HDL Synthesis Report

Macro Statistics
# Adders/Subtractors                                   : 2
 10-bit subtractor                                     : 2
# Registers                                            : 33
 Flip-Flops                                            : 33
# Comparators                                          : 2
 10-bit comparator equal                               : 1
 10-bit comparator greater                             : 1
```

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	22	9,312	1%	
Number of 4 input LUTs	78	9,312	1%	
Logic Distribution				
Number of occupied Slices	40	4,656	1%	
Number of Slices containing only related logic	40	40	100%	
Number of Slices containing unrelated logic	0	40	0%	
Total Number of 4 input LUTs	78	9,312	1%	
Number of bonded IOBs	33	92	35%	
IOB Flip Flops	11			
Number of GCLKs	1	24	4%	
Total equivalent gate count for design	948			
Additional JTAG gate count for IOBs	1,584			

GCD Single-case statement design



start
GCD(34, 8)

done
GCD = 2

Summary

- Finite State Machines
  - Verilog Mapping: one, two, three always blocks
- State Encoding
  - User-defined or tool-defined
  - State encoding techniques: one-hot, user-defined, gray
- Synthesis Issues
  - Default state assignment
  - Safe Implementations
  - Next-state logic: RAM or LUT
- FSM-based control of Datapath (gcd)