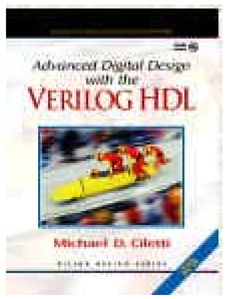
Advanced Digital Design with the Verilog HDL



M. D. Ciletti

Department
of
Electrical and Computer Engineering
University of Colorado
Colorado Springs, Colorado

ciletti@vlsic.uccs.edu

Draft: Chap 6b: Synthesis of Combinational and Sequential Logic

COURSE OVERVIEW

- Review of combinational and sequential logic design
- Modeling and verification with hardware description languages
- Introduction to synthesis with HDLs
- Programmable logic devices
- State machines, datapath controllers, RISC CPU
- Architectures and algorithms for computation and signal processing
- Synchronization across clock domains
- Timing analysis

Fault simulation and testing, JTAG, BIST

Synthesis of Sequential Logic with Flip-Flops

Note: Flip-flops are synthesized only from edge-sensitive cyclic behaviors

Questions:

- What determines the outcome of synthesis from an edge-sensitive behavior?
- How does a synthesis tool infer the need for a flip-flop?
- When does a register variable that is assigned value within an edge-sensitive behavior automatically synthesize to a flip-flop?
- How does the synthesis tool distinguish the synchronizing signal (clock) from other signals in the event control expression of a cyclic behavior?
- How can multiple flip-flops or registers be modeled to operate concurrently?

Inference of Flip-Flops

Each of the following conditions imply the need for memory and will synthesize a register variable to a flip flop:

- The variable is referenced outside of the behavior
- The variable is referenced within the behavior before it is assigned value
- The variable is assigned value in only some of the branches of activity within the behavior

Flip-Flops or Latches?

- Memory inferred for an edge-sensitive cyclic behavior will be synthesized as a flip flop
- Memory inferred for a level-sensitive cyclic behavior or a continuous assignment with feedback will be synthesized as a latch
- Note: an incomplete conditional statement (***if...else***) or a ***case*** statement in a level-sensitive cyclic behavior will synthesize to a latch.
- Note: an incomplete conditional statement statement (***if...else***) or a ***case*** statement in an edge-sensitive cyclic behavior will synthesize to a flip-flop with circuitry effectively implementing clock enable.

Synchronizing Signal (1 of 2)

- The order in which signals appear in the event control expression of an edge-sensitive cyclic behavior does not determine which signal is the synchronizing signal
- The sequence in which signals are decoded in the statement that follows the event control expression of an edge-sensitive cyclic behavior determines which of the edge-sensitive signals are control signals and which is the clock (i.e., the synchronizing signal).

Synchronizing Signal (2 of 2)

Note: If the event control expression is sensitive to the edge of more than one signal, an ***if*** statement must be the first statement in the behavior.

- The control signals that appear in the event control expression must be decoded explicitly in the branches of the ***if*** statement (e.g., decode the reset condition first).

- The synchronizing signal is not tested explicitly in the body of the if statement, but, by default, the last branch must describe the synchronous activity, independently of the actual names given to the signals.

Synthesis of a flip-flop

SYNTHESIS TIP

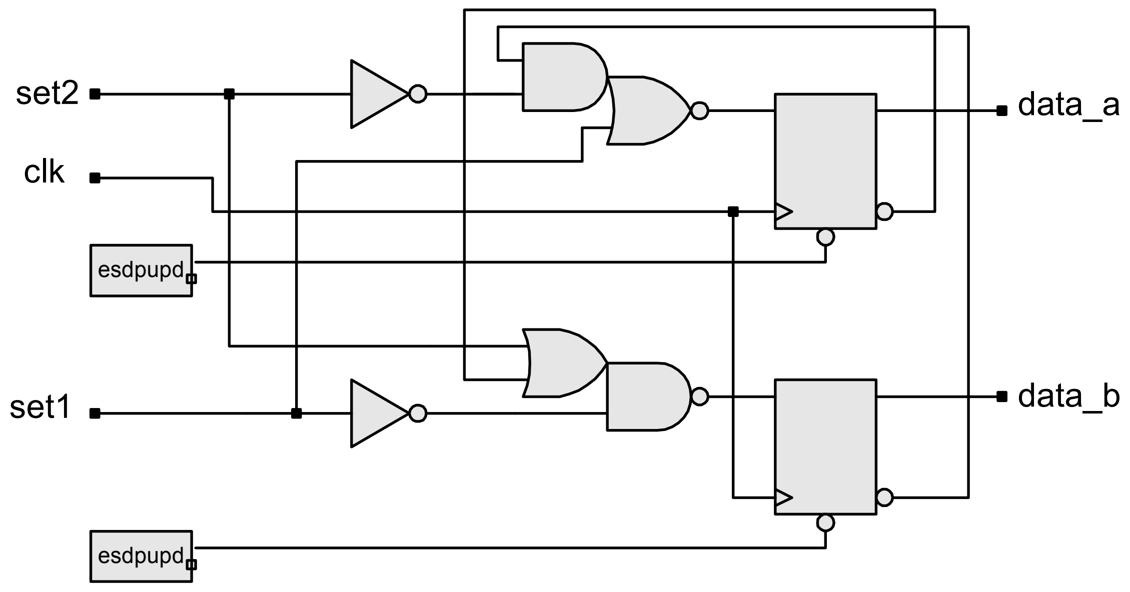
A variable that is referenced within an edge-sensitive behavior before it is assigned value by the behavior will be synthesized as the output of a flip-flop.

Example 6.23

```
module swap_synch (data_a, data_b, set1, set2, clk); // Typo at text
 output       data_a, data_b;                        // Typo at text
 input        clk, set1, set2;
 reg          data_a, data_b;

 always @  (posedge clk)
  begin
   if (set1) begin data_a <= 1; data_b <= 0; end else
    if (set2) begin data_a <= 0; data_b <= 1; end else
     else
      begin
       data_b <= data_a;
       data_a <= data_b;
      end
  end
endmodule
```

- The statements assigning value to *data_a* and *data_b* execute non-blocking assignments concurrently
- Both variables are sampled (referenced) before receiving value
- Both are synthesized as the output of a flip-flop
- Notice that *set1* and *set2* are explicitly decoded first. The last clause of the ***if*** statement assigns values to *data_a* and *data_b*. Those (non-blocking) assignments are synchronized to the rising edge of *clk*, which is not referenced explicitly in the ***if*** statement.

Synthesis result:



Example: Synthesis of a 4-bit Parallel Load Data Register

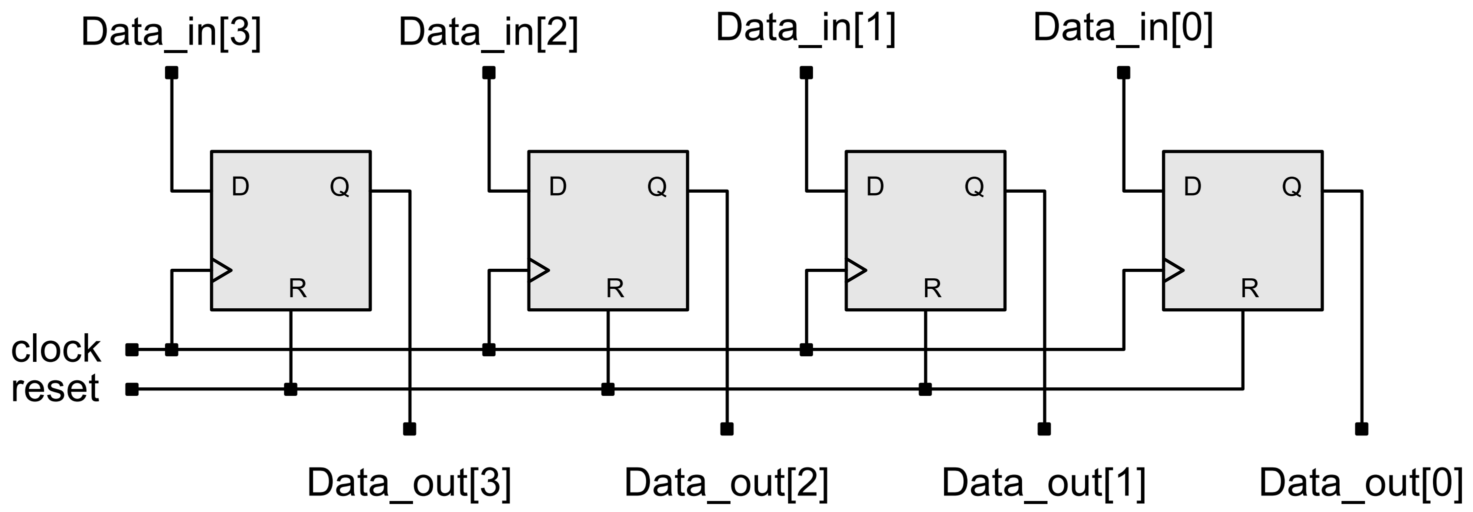
Example 6.24

```
module D_reg4_a  (Data_out, clock, reset, Data_in);
 output       [3: 0]     Data_out;
 input        [3: 0]     Data_in;
 input                   clock, reset;
 reg          [3: 0]     Data_out;

 always @  (posedge clock or posedge reset)
  begin
   if (reset == 1'b1) Data_out <= 4'b0;
    else Data_out <= Data_in;
   end
endmodule
```

Note:

- The positive edge of *reset* appears in the event control expression and appears in the first clause of the *if* statement
- The positive edge of *clock* also appears in the event control expression, but is not explicitly decoded by the branch statement that follows the event control expression.

Synthesis Result:



Rules for Synchronizing Signals

- Synchronize an edge-sensitive cyclic behavior to a single edge (***posedge*** or ***negedge***, but not both) of a single clock (synchronizing signal)
- Multiple behaviors need not have the same synchronizing signal, or be synchronized by the same edge of the same signal
- All of the synchronizing signals (clocks) must have the same period

SYNTHESIS TIP

A variable that is assigned value by a cyclic behavior before it is referenced within the behavior, but is not referenced outside the behavior, will be eliminated by the synthesis process.

SYNTHESIS TIP

A variable that is assigned value by an edge-sensitive behavior and is referenced outside the behavior will be synthesized as the output of a flip-flop.

Example (6.16):

```
module or4_behav (y, x_in);
  parameter  word_length = 4;
  output                             y;
  input      [word_length - 1: 0]    x_in;
  reg                                y;
  integer                            k;
```

```
   always @  x_in
     begin: check_for_1
       y = 0;
       for (k = 0; k <= word_length -1; k = k+1)
         if (x_in[k] == 1) begin
             y = 1;
             disable check_for_1;
         end
     end
 endmodule
```

Synthesis result: 4-input OR gate

Note: The register variable k is eliminated by the synthesis process.

Example 6.25

D_out is not referenced outside the scope of the behavior

A synthesis tool will eliminate *D_out*. output of a flip-flop.

```
module empty_circuit (D_in, clk);
  input   D_in;
  input   clk;
  reg     D_out;

  always @ (posedge clk)  begin
    D_out <= D_in;
  end
endmodule
```

Note: If *empty_circuit* is modified to declare *D_out* as an output port, *D_out* will be synthesized as the output of a flip-flop.

Synthesis of Explicit State Machines

Features of explicit state machines:

- Explicitly declared state register
- Explicit logic or state evolution

Explicit machines can be described by two behaviors

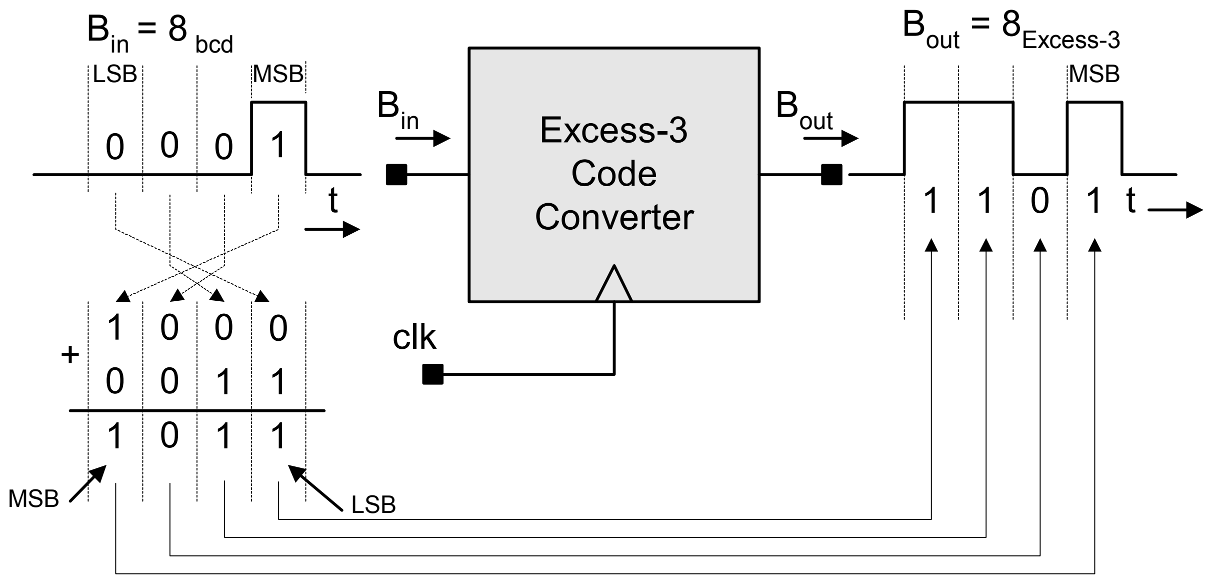
- Edge-sensitive behavior that synchronizes the evolution of the state
- Level-sensitive behavior that describes the next state and output logic.

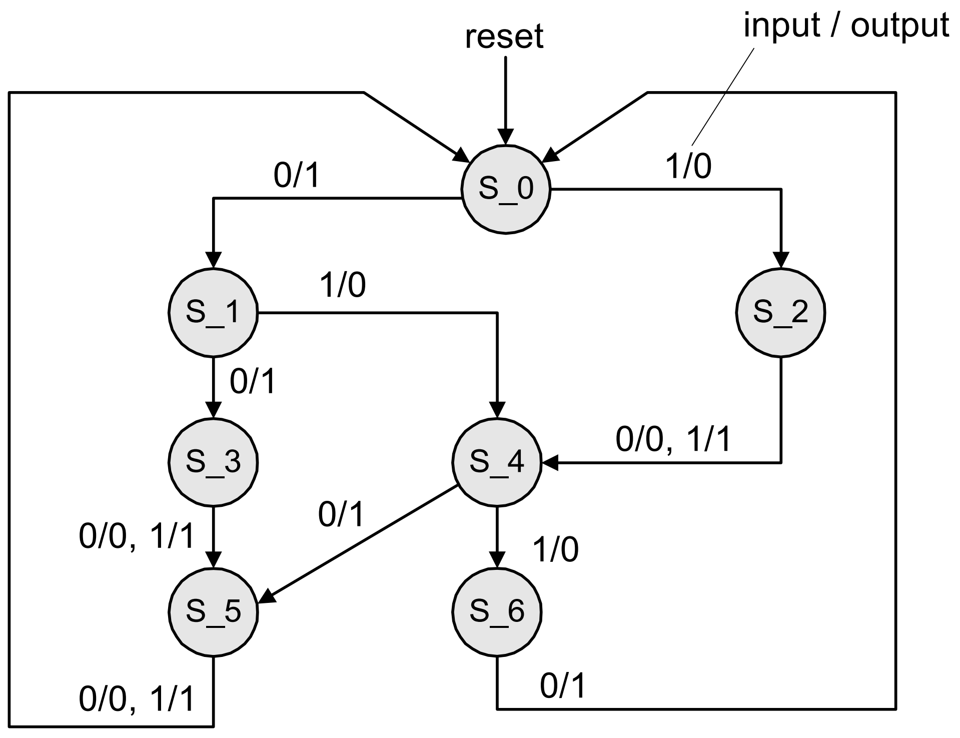
Example: BCD-to-Excess-3 Code Converter (Mealy)

See Example 3.2 for manual design

- A serially-transmitted BCD (8421 code) word is to be converted into an Excess-3 code
- An Excess-3 code word is obtained by adding *3* to the decimal value and taking the binary equivalent. Excess-3 code is self-complementing

Decimal Digit	8-4-2-1 Code (BCD)	Excess-3 Code
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100


$B_{in} = 8_{bcd}$
LSB
MSB
0 0 0 1
t
B_{in}
Excess-3 Code Converter
clk
B_{out}
$B_{out} = 8_{Excess-3}$
MSB
1 1 0 1
t
1 0 0 0
+
0 0 1 1
1 0 1 1
MSB
LSB

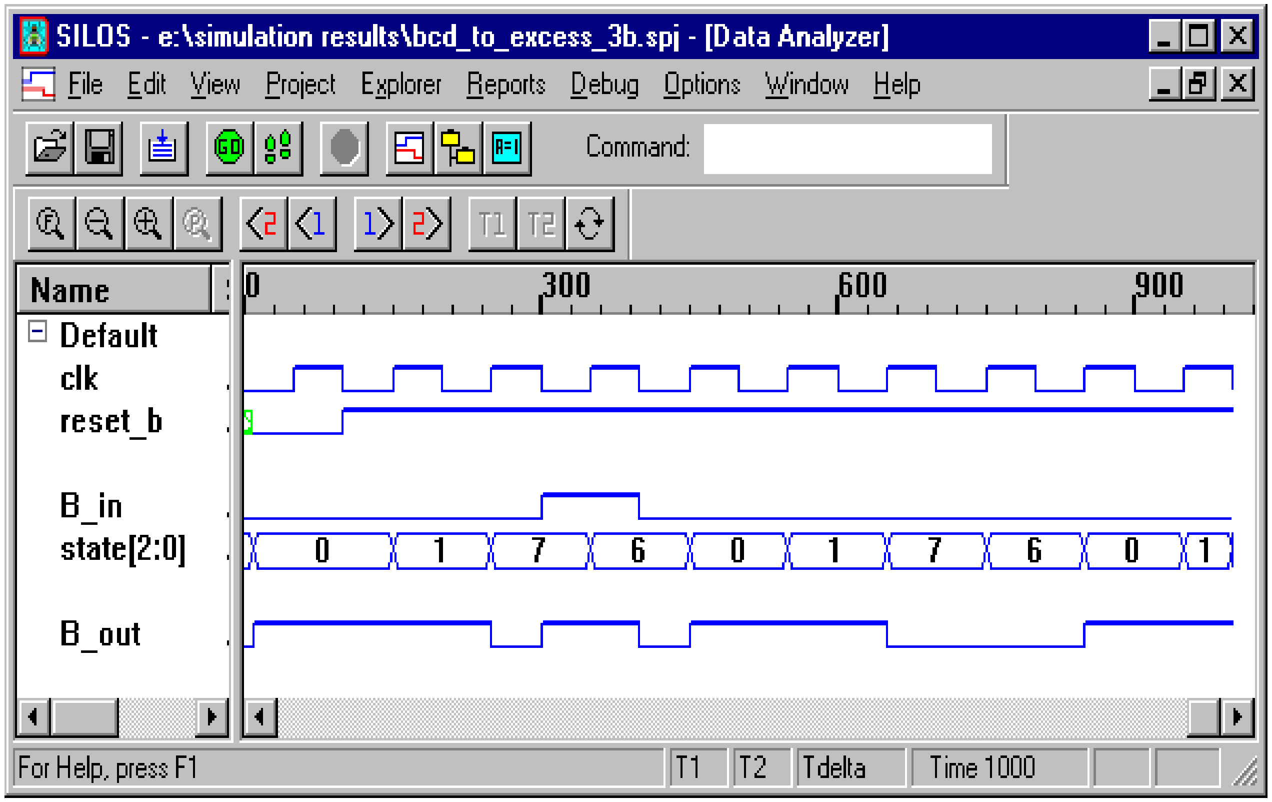

reset
input / output
S_0
S_1
S_2
S_3
S_4
S_5
S_6
0/1
1/0
1/0
0/1
0/0, 1/1
0/0, 1/1
0/1
1/0
0/0, 1/1
0/1

```
module BCD_to_Excess_3b (B_out, B_in, clk, reset_b);
 output      B_out;
 input       B_in, clk, reset_b;
 parameter   S_0 = 3'b000,          // State assignment
             S_1 = 3'b001,
             S_2 = 3'b101,
             S_3 = 3'b111,
             S_4 = 3'b011,
             S_5 = 3'b110,
             S_6 = 3'b010,
             dont_care_state = 3'bx,
             dont_care_out = 1'bx;

 reg[2: 0]   state, next_state;
 reg         B_out;

 always @ (posedge clk or negedge reset_b)
  if (reset_b == 0) state <= S_0; else state <= next_state;
```

```
always @ (state or B_in) begin
  B_out = 0;
  case (state)
    S_0: if (B_in == 0) begin next_state = S_1; B_out = 1; end
     else if (B_in == 1) begin next_state = S_2; end

    S_1: if (B_in == 0) begin next_state = S_3; B_out = 1; end
     else if (B_in == 1) begin next_state = S_4; end

    S_2: begin next_state = S_4; B_out = B_in; end
    S_3: begin next_state = S_5; B_out = B_in; end

    S_4: if (B_in == 0) begin next_state = S_5; B_out = 1; end
     else if (B_in == 1) begin next_state = S_6; end

    S_5: begin next_state = S_0; B_out = B_in; end

    S_6: begin next_state = S_0; B_out = 1; end
```

```
/*  Omitted for BCD_to_Excess_3b version
        Included for BCD_to_Excess_3c version
      default: begin next_state = dont_care_state;
                     B_out = dont_care_out; end
      */
    endcase
  end
endmodule
```

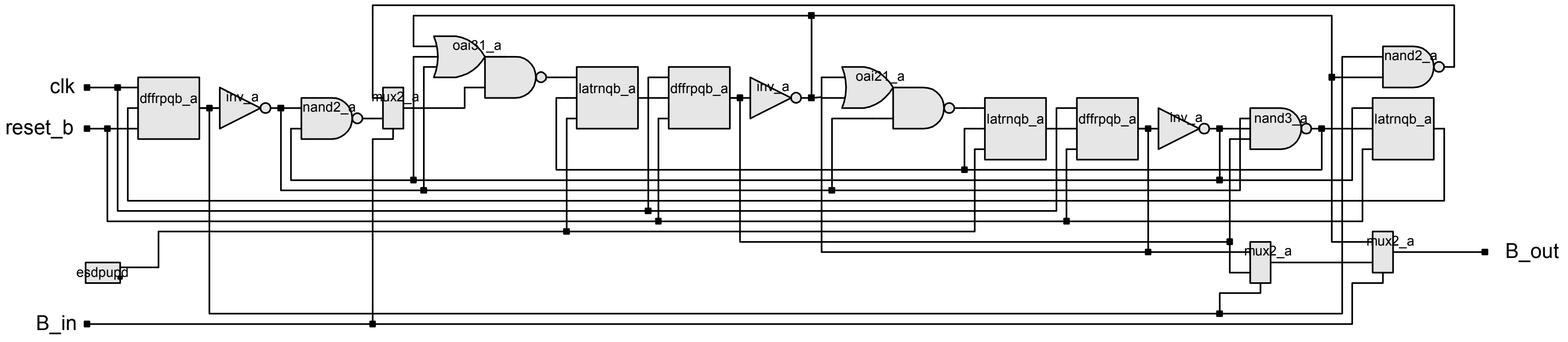
Simulation results for *BCD_to_Excess_3b*


SILOS - e:\simulation results\bcd_to_excess_3b.spj - [Data Analyzer]
File Edit View Project Explorer Reports Debug Options Window Help
Command:
Name
Default
clk
reset_b
B_in
state[2:0]
B_out
0
300
600
900
0 1 7 6 0 1 7 6 0 1
For Help, press F1
T1
T2
Tdelta
Time 1000

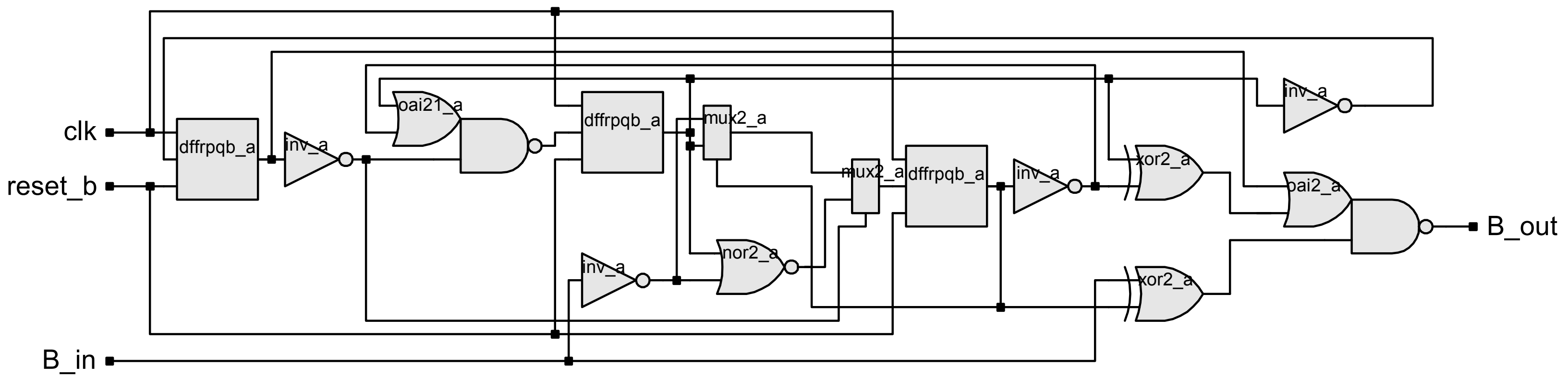
ASIC circuit synthesized from BCD_to_Excess_3b

Note: Latches result from failure to provide default state assignments

Note: Default values for B_out are hard-wired to 0

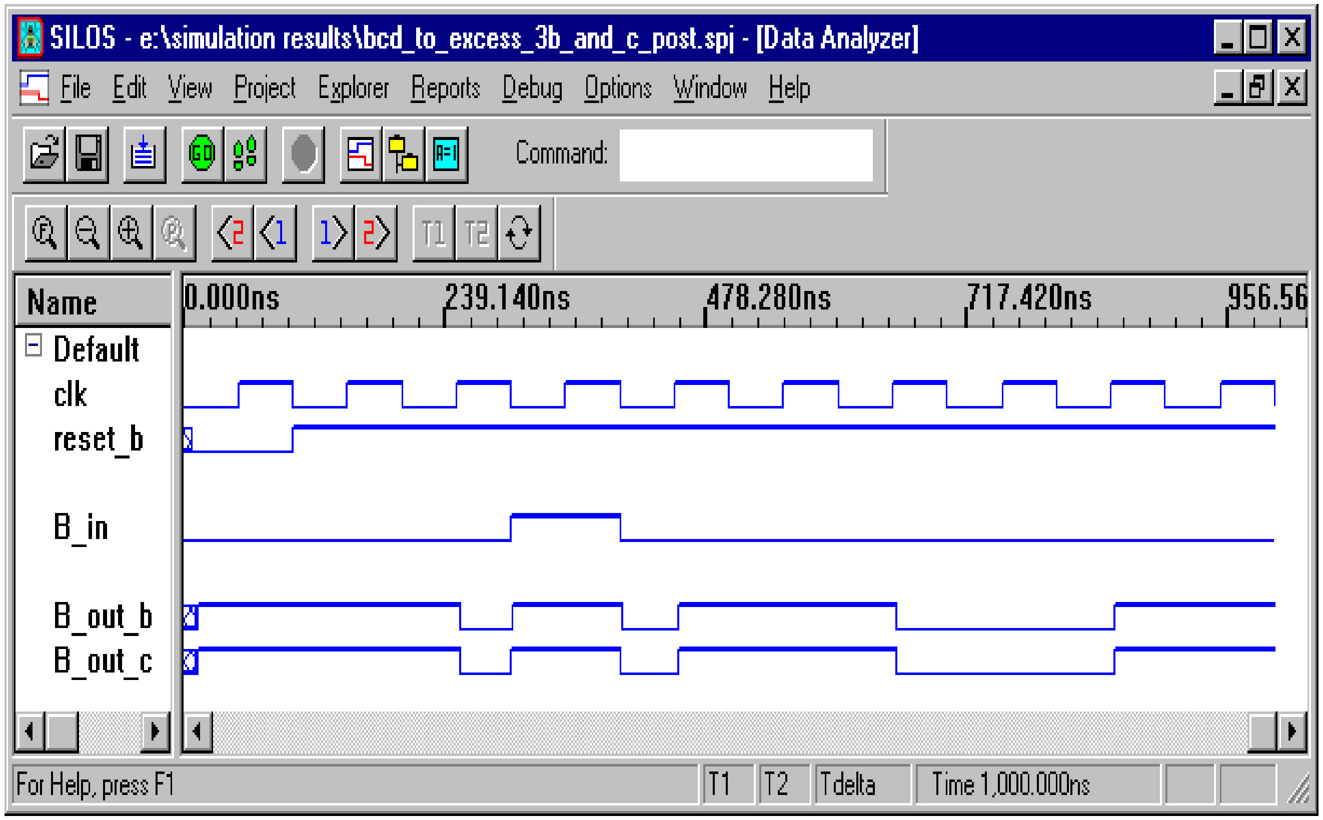


ASIC circuit synthesized from BCD_to_Excess_3c



Note: The model includes default don't care state assignments and default don't care assignments to B_out

Post-synthesis simulation results


SILOS - e:\simulation results\bcd_to_excess_3b_and_c_post.spj - [Data Analyzer]
File Edit View Project Explorer Reports Debug Options Window Help
Command:
Name
0.000ns
239.140ns
478.280ns
717.420ns
956.56
Default
clk
reset_b
B_in
B_out_b
B_out_c
For Help, press F1
T1
T2
Tdelta
Time 1,000.000ns

Note: The circuits have same functionality, and simulation results match the manual design, but BCD_to_Excess_3c wastes hardware

SYNTHESIS TIP

Use two cyclic behaviors to describe an explicit state machine: a level-sensitive behavior to describe the combinational logic for the next state and outputs, and an edge-sensitive behavior to synchronize the state transitions.

SYNTHESIS TIP

Use the procedural assignment operator (**=**) in the level-sensitive cyclic behaviors describing the combinational logic of a finite state machine.

SYNTHESIS TIP

Use the non-blocking assignment operator (**<=**) in the edge-sensitive cyclic behaviors describing the state transitions of a finite state machine and the register transfers of the datapath of a sequential machine.

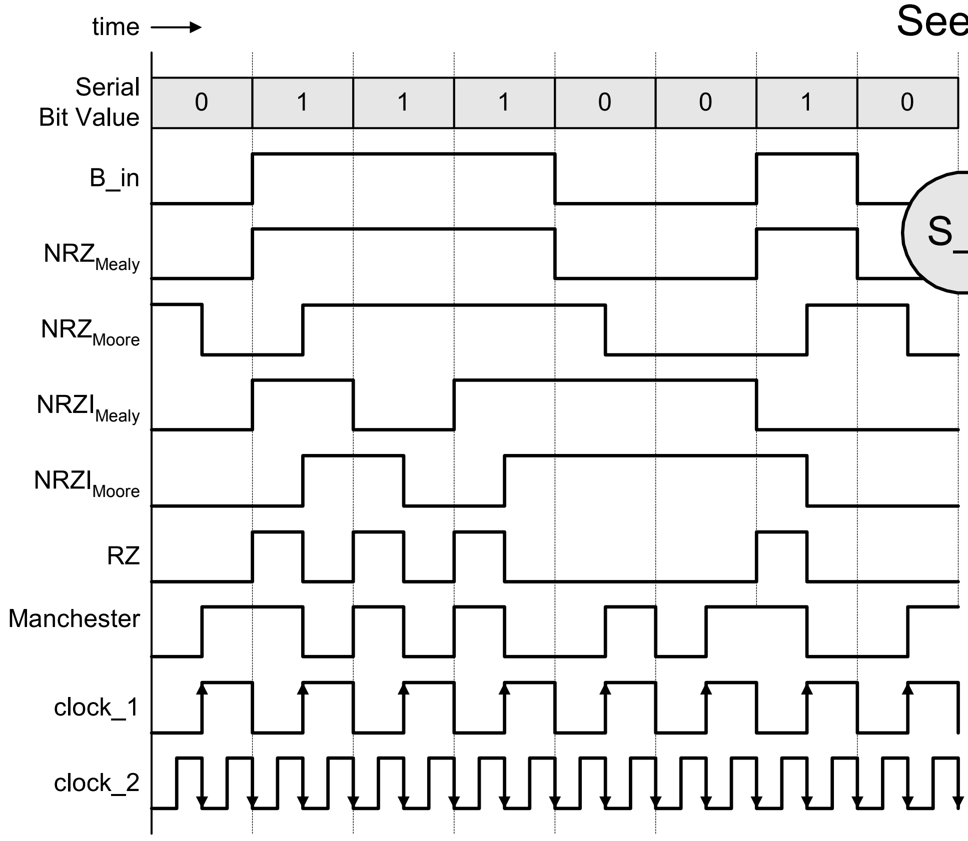
SYNTHESIS TIP

Decode all possible states in a level-sensitive behavior describing the combinational next state and output logic of an explicit state machine.

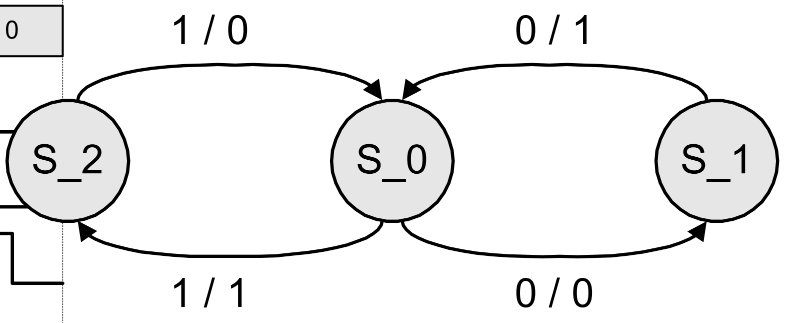
Additional Rules for Synthesis

- Assign value to the entire state vector (no bit or part select allowed)
- Asynchronous control signals (e.g., set and reset) in the event control expression must be scalars
- The value that is assigned to the state register must be either a constant or a variable that evaluates to a constant after static evaluation (i.e., the state transition diagram must specify a fixed relationship)

Example: Mealy-Type NRZ-to-Manchester Line Code Converter

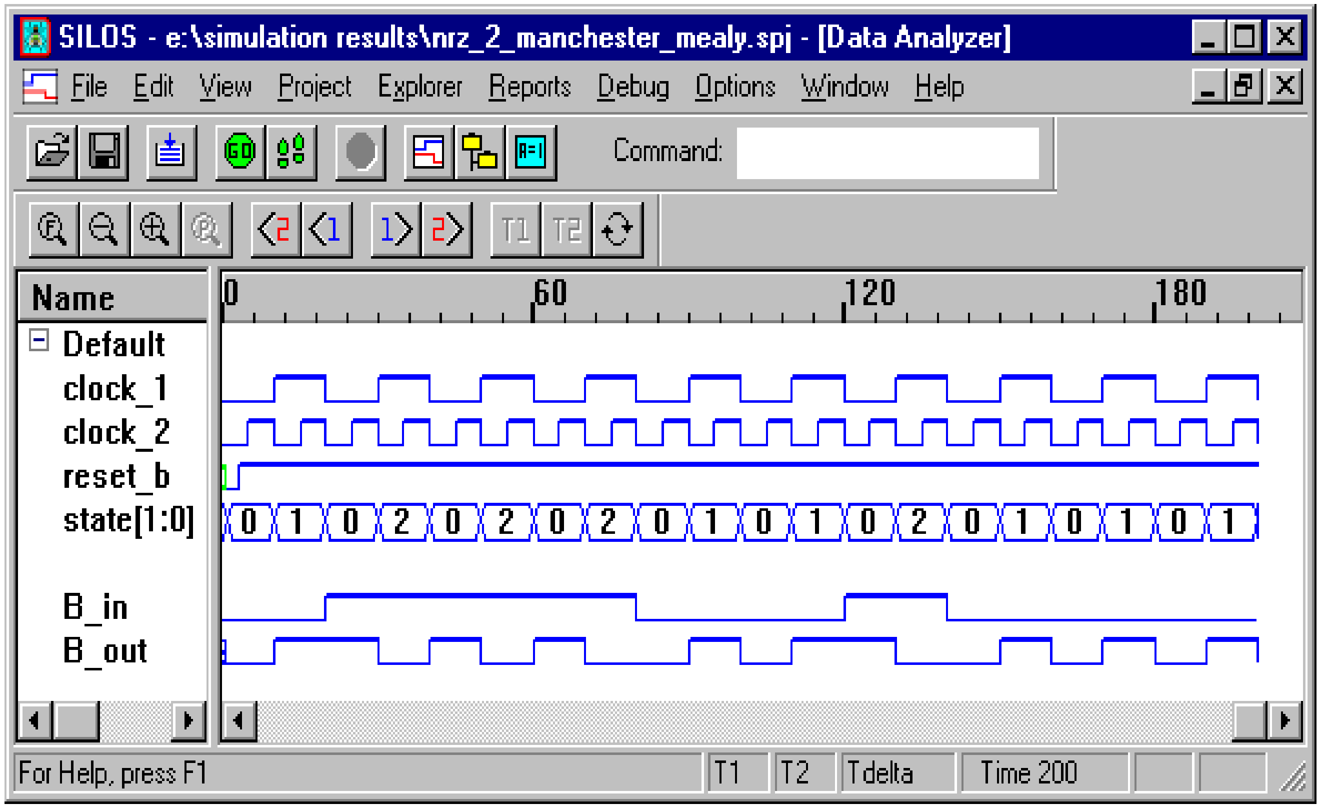


See Chapter 3



```
module NRZ_2_Manchester_Mealy (B_out, B_in, clock, reset_b);
  output       B_out;
  input        B_in;
  input        clock, reset_b;
  reg [1: 0]   state, next_state;
  reg          B_out;
  parameter    S_0 = 0,
               S_1 = 1,
               S_2 = 2,
               dont_care_state = 2'bx,
               dont_care_out = 1'bx;

  always @ (negedge clock or negedge reset_b)
   if (reset_b == 0) state <= S_0; else state <= next_state;
```

```
 always @ (state or B_in ) begin
  B_out = 0;
  case (state)                                    // Partially decoded
   S_0: if (B_in == 0) next_state = S_1;
        else if (B_in == 1) begin next_state = S_2; B_out = 1; end
   S_1: begin next_state = S_0; B_out = 1;  end
   S_2: begin next_state = S_0;  end
   default:  begin next_state = dont_care_state;
                   B_out = dont_care_out;  end
  endcase
 end
endmodule
```



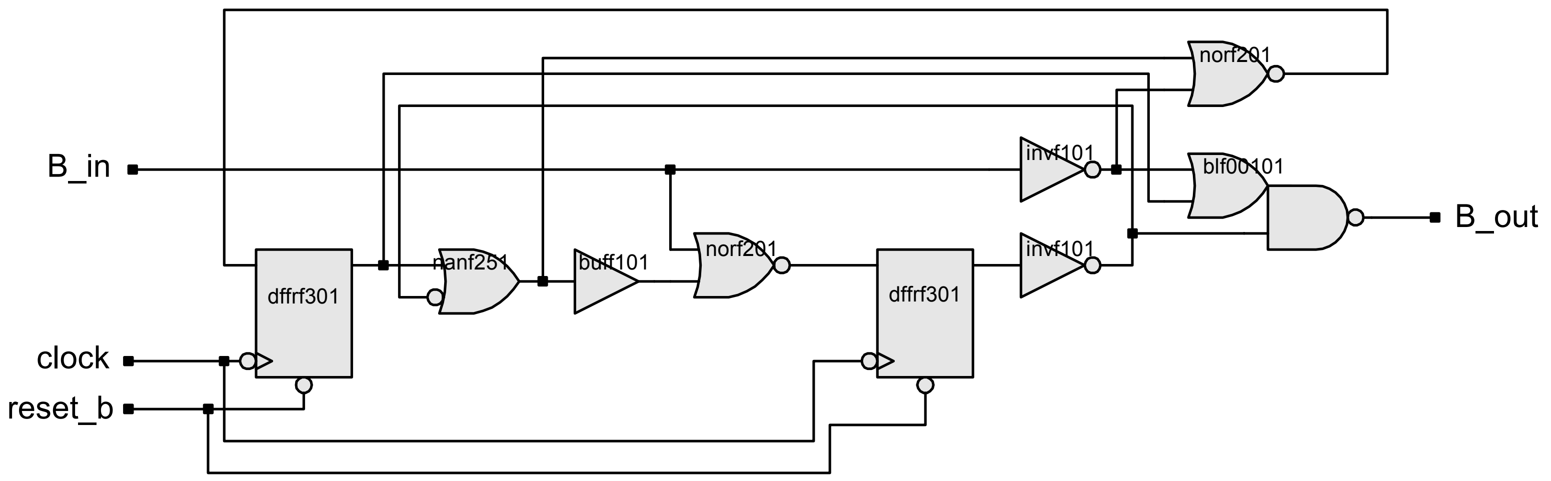
Note:

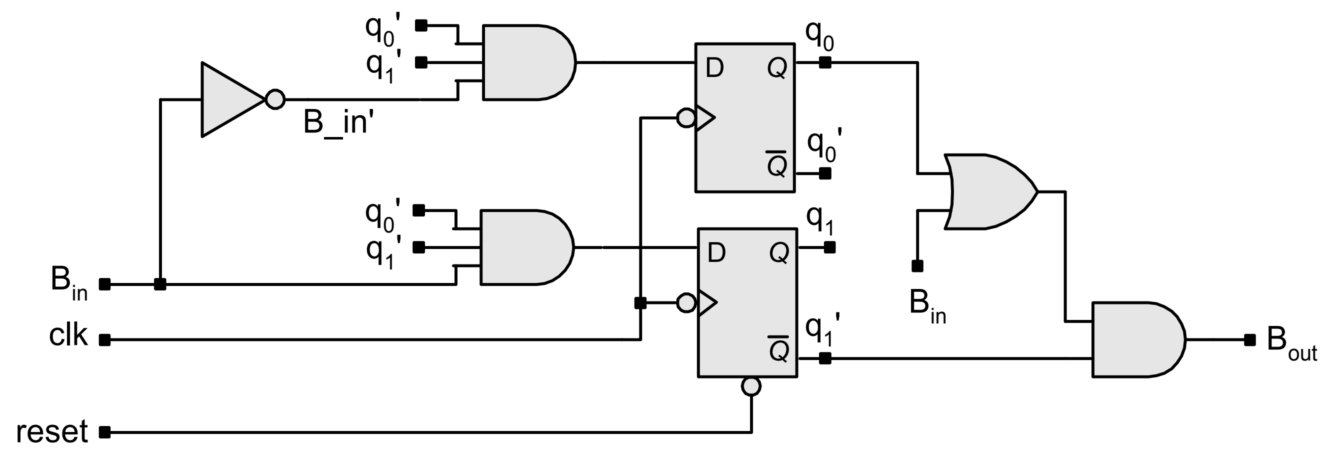
- *B_in* is switching on active edges of *clock_1 (*alternate active edges of *clock_2)*
- *B_in* changes at the same time as the state
- General rule: avoid having the inputs change at the same time that the state changes *unless it happens that the inputs are treated as don't-cares* at those edges

Synthesis Netlist

```
module NRZ_2_Manchester_Mealy ( B_out, B_in, clock, reset );
input  B_in, clock, reset;
output B_out;
wire \next_state<1> , \next_state<0> , \state<1> , \state<0> , n80, n81, n82, n83;
  buff101 U26 ( .A1(n81), .O(n80) );
  norf201 U27 ( .A1(n81), .B1(n82), .O(\next_state<1> ) );
  norf201 U28 ( .A1(B_in), .B1(n80), .O(\next_state<0> ) );
  blf00101 U29 ( .A1(n83), .B2(\state<1> ), .C2(n82), .O(B_out) );
  nanf251 U30 ( .A1(\state<1> ), .B2(n83), .O(n81) );
  invf101 U31 ( .A1(B_in), .O(n82) );
  invf101 U32 ( .A1(\state<0> ), .O(n83) );
  dfrf301 \state_reg<1>  ( .DATA1(\next_state<1> ), .CLK2(clock), .RST3(
      reset), .Q(\state<1> ) );
  dfrf301 \state_reg<0>  ( .DATA1(\next_state<0> ), .CLK2(clock), .RST3(
      reset), .Q(\state<0> ) );
endmodule
```

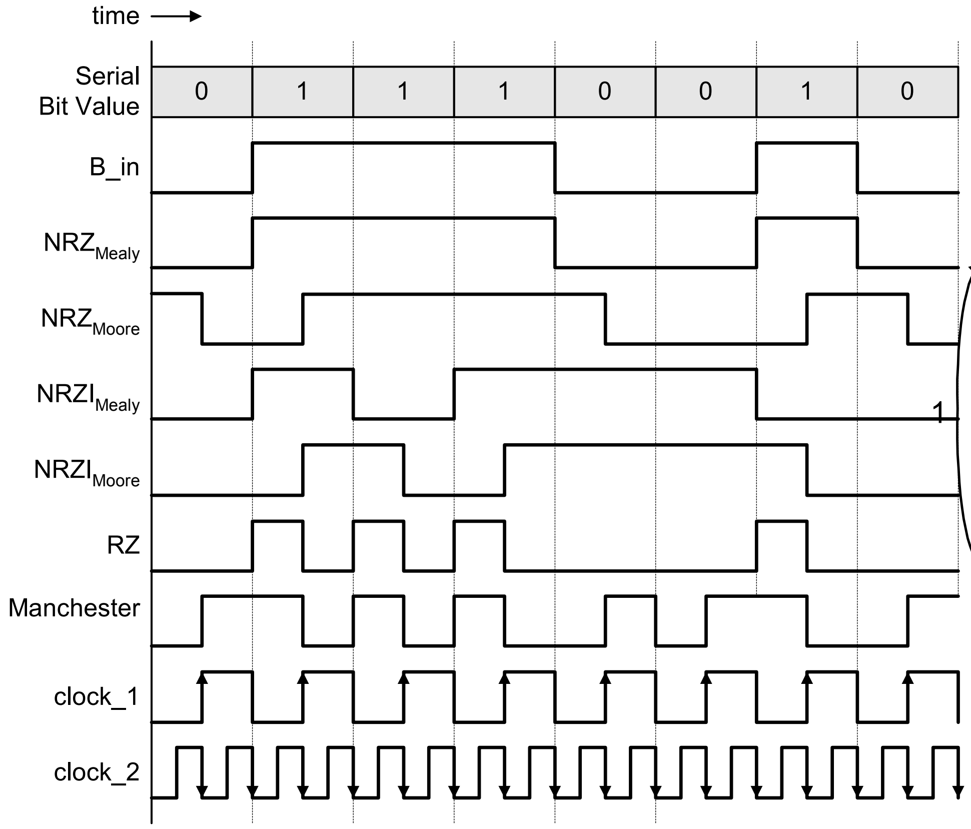
Synthesized Circuit (Compared to Chapter 3 result)

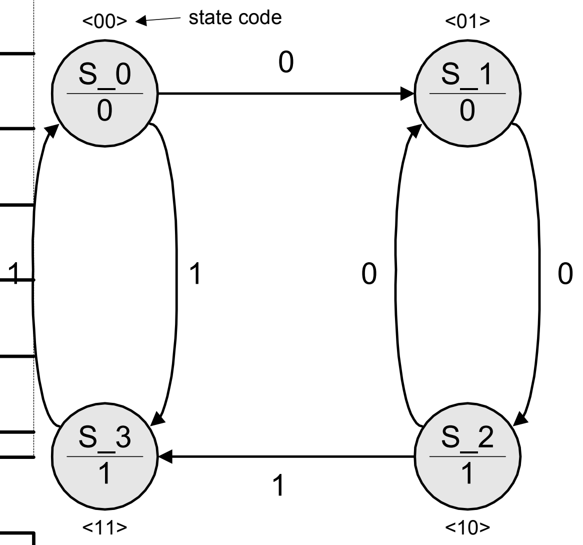




Note: Errata for missing reset on flip-flop

Moore-Type NRZ-to-Manchester Line Code Converter

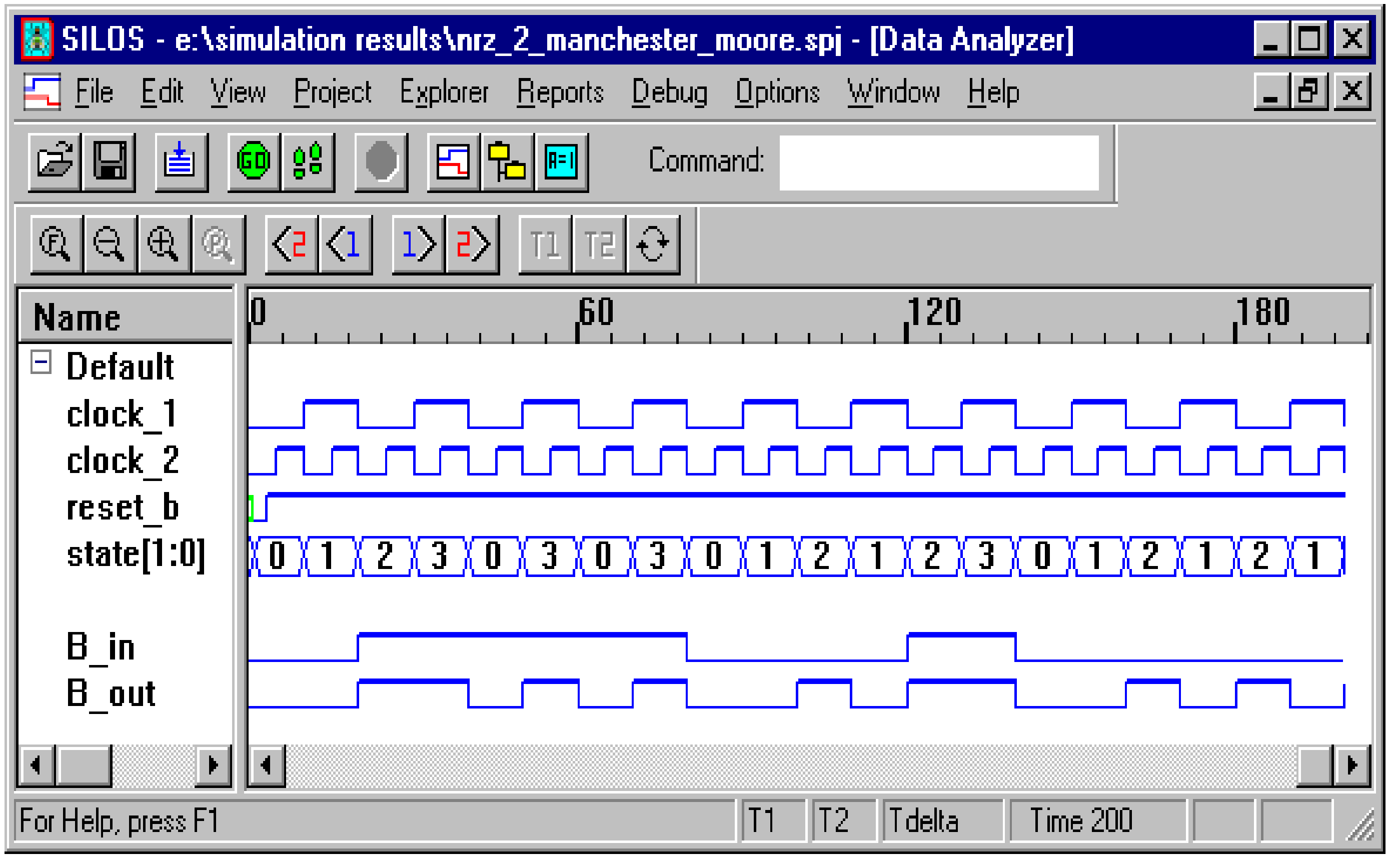




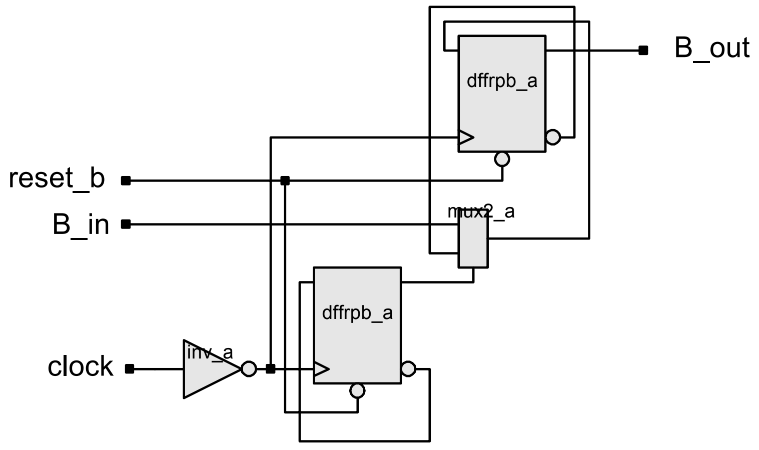
```
module NRZ_2_Manchester_Moore (B_out, B_in, clock, reset_b);
 output       B_out;
 input        B_in;
 input        clock, reset_b;
 reg [1: 0]   state, next_state;
 reg          B_out;
 parameter    S_0 = 0,
          S_1 = 1,
          S_2 = 2,
          S_3 = 3;

always @ (negedge clock or negedge reset_b)
  if (reset_b == 0) state <= S_0; else state <= next_state;
```

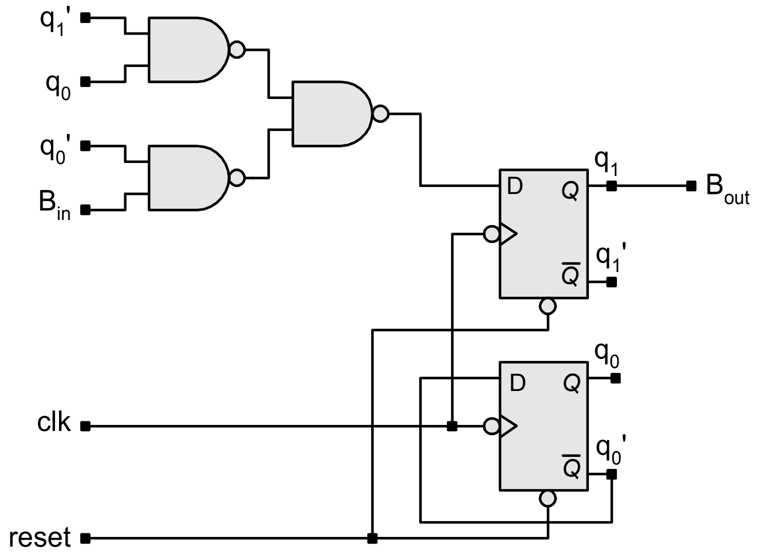
```
always @ (state or B_in ) begin
    B_out = 0;
    case (state)                                    // Fully decoded
      S_0: begin if (B_in == 0) next_state = S_1; else next_state = S_3; end
      S_1: begin next_state = S_2; end
      S_2: begin B_out = 1; if (B_in == 0) next_state = S_1; else next_state =
S_3; end
      S_3: begin B_out = 1; next_state = S_0; end
   endcase
  end
endmodule
```

Simulation Results:



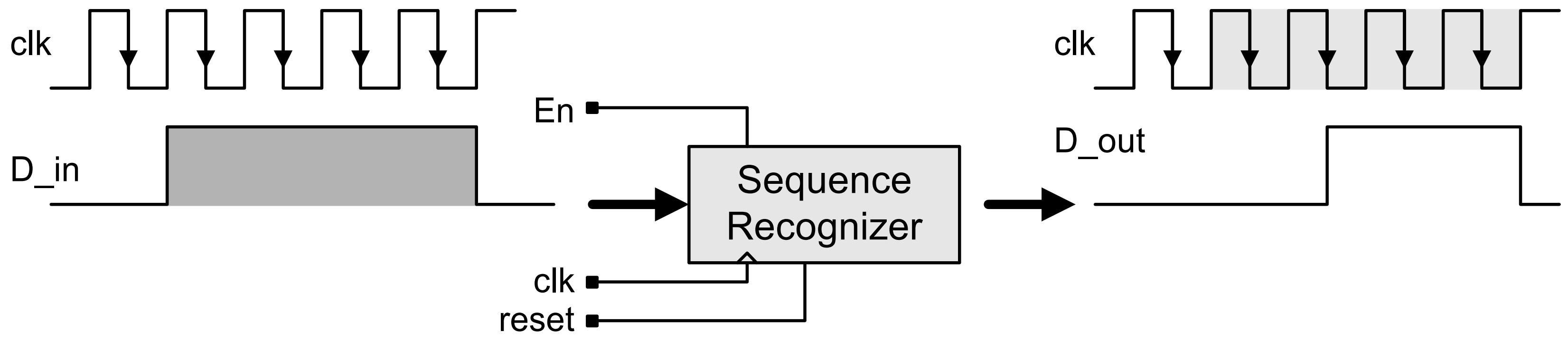
Synthesis Results (Compared to Chapter 3):





Example: Sequence Recognizer

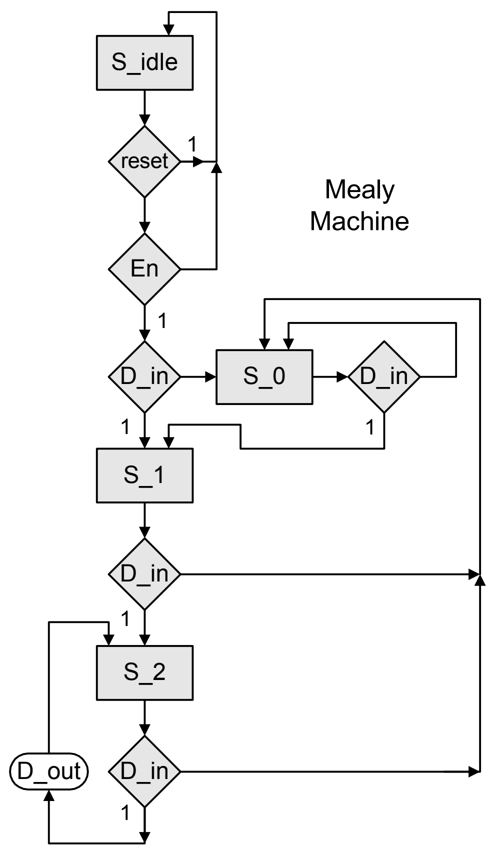
Detect three successive 1s

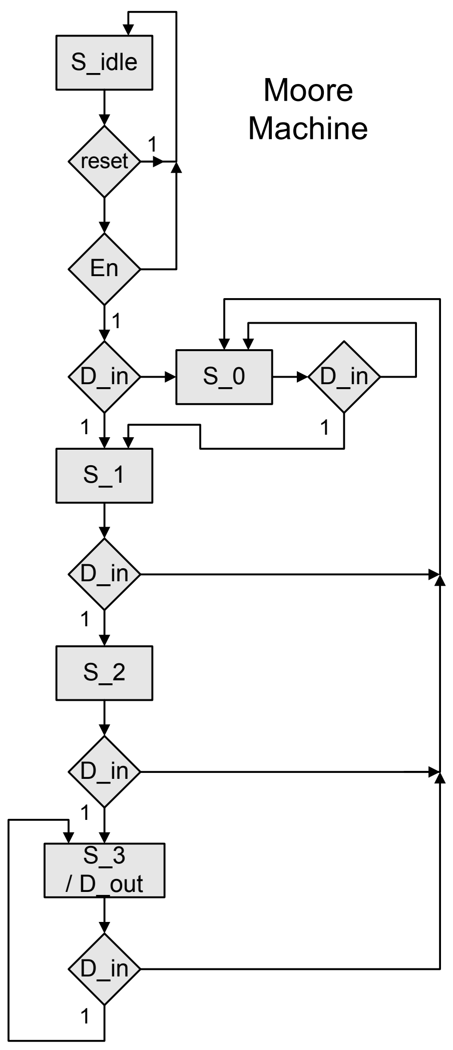


- Assert *D_out* when a given pattern of consecutive bits has been received in its serial input stream, *D_in*
- Apply data on the rising edge of the clock if the state transitions are to occur on the falling edge of the clock, and visa-versa.

Conventions:

- The output of a Mealy machine is valid immediately before the active edge of the clock controlling the machine
- Successive values inputs are received in successive clock cycles.
- A *non-resetting* machine continues to assert its output if the input bit pattern is overlapping
- A resetting machine asserts for one cycle after detecting the input sequence, and then de-asserts for one cycle before detecting the next sequence of bits


S_idle
reset
1
En
1
Mealy
Machine
D_in
S_0
D_in
1
1
S_1
D_in
1
S_2
D_out
D_in
1


S_idle
reset
1
Moore
Machine
En
1
D_in
S_0
D_in
1
1
S_1
D_in
1
S_2
D_in
1
S_3
/ D_out
D_in
1

```
module Seq_Rec_3_1s_Mealy (D_out, D_in, En, clk, reset);
 output        D_out;
 input         D_in, En;
 input         clk, reset;

 parameter     S_idle =   0;                // Binary code
 parameter     S_0 =      1;
 parameter     S_1 =      2;
 parameter     S_2 =      3;
 reg       [1: 0] state, next_state;

 always @ (negedge clk)
  if (reset == 1) state <= S_idle; else state <= next_state;
 always @ (state or D_in or En) begin
  case (state)                                // Partially decoded
   S_idle:     if ((En == 1) && (D_in == 1))   next_state = S_1; else
               if ((En  == 1) && (D_in == 0))  next_state = S_0;
               else                            next_state = S_idle;

   S_0:        if (D_in == 0)              next_state = S_0; else
               if (D_in == 1)              next_state = S_1;
               else                        next_state = S_idle;

   S_1:        if (D_in == 0)              next_state = S_0; else
               if (D_in == 1)              next_state = S_2;
               else                        next_state = S_idle;

   S_2:        if (D_in == 0)              next_state = S_0; else
               if (D_in == 1)              next_state = S_2;
               else                        next_state = S_idle;
   default:                                next_state = S_idle;
  endcase
 end

 assign D_out = ((state == S_2) && (D_in == 1 )); // Mealy output
endmodule
```

```
module Seq_Rec_3_1s_Moore (D_out, D_in, En, clk, reset);
 output         D_out;
 input          D_in, En;
 input          clk, reset;

 parameter      S_idle =  0;            // Binary code
 parameter      S_0 =     1;
 parameter      S_1 =     2;
 parameter      S_2 =     3;
 parameter      S_3 =     4;

 reg        [2: 0]      state, next_state;

 always @ (negedge clk)
  if (reset == 1) state <= S_idle; else state <= next_state;
 always @ (state or D_in) begin
  case (state)
   S_idle:     if ((En == 1) && (D_in == 1))   next_state = S_1; else
               if ((En  == 1) && (D_in == 0))  next_state = S_0;
               else                            next_state = S_idle;

   S_0:        if (D_in == 0)              next_state = S_0; else
               if (D_in == 1)              next_state = S_1;
               else                        next_state = S_idle;

   S_1:        if (D_in == 0)              next_state = S_0; else
               if (D_in == 1)              next_state = S_2;
               else                        next_state = S_idle;

   S_2, S_3:   if (D_in == 0)              next_state = S_0; else
               if (D_in == 1)              next_state = S_3;
               else                        next_state = S_idle;
   default:                                next_state = S_idle;
  endcase
 end

 assign D_out = (state == S_3);                // Moore output
endmodule
```

```
module t_Seq_Rec_3_1s ();
 reg D_in_NRZ, D_in_RZ, En, clk, reset;

 wire Mealy_NRZ;
 wire Mealy_RZ;
 wire Moore_NRZ;
 wire Moore_RZ;

 Seq_Rec_3_1s_Mealy M0 (Mealy_NRZ, D_in_NRZ, En, clk, reset);
 Seq_Rec_3_1s_Mealy M1 (Mealy_RZ, D_in_RZ, En, clk, reset);
 Seq_Rec_3_1s_Moore M2 (Moore_NRZ, D_in_NRZ, En, clk, reset);
 Seq_Rec_3_1s_Moore M3 (Moore_RZ, D_in_RZ, En, clk, reset);

 initial #275 $finish;

 initial begin #5 reset = 1; #1 reset = 0; end
 initial begin
   clk = 0; forever #10 clk = ~clk;
 end
 initial begin
   #5 En = 1;
  #50 En = 0;
 end
```

```
initial fork
   begin #10 D_in_NRZ = 0;    #25 D_in_NRZ = 1;    #80 D_in_NRZ = 0; end
   begin #135 D_in_NRZ = 1; #40 D_in_NRZ = 0; end
   begin #195 D_in_NRZ = 1'bx; #60 D_in_NRZ = 0; end
  join

 initial fork
   #10 D_in_RZ = 0;
   #35 D_in_RZ = 1;    #45 D_in_RZ = 0;
   #55 D_in_RZ = 1;    #65 D_in_RZ = 0;
   #75 D_in_RZ = 1;    #85 D_in_RZ = 0;
   #95 D_in_RZ = 1;    #105 D_in_RZ = 0;
   #135 D_in_RZ = 1; #145 D_in_RZ = 0; #155 D_in_RZ = 1; #165 D_in_RZ = 0;
   #195 D_in_RZ = 1'bx; #250 D_in_RZ = 0;
 join
endmodule
```

Note:

The Mealy machine is non-resetting

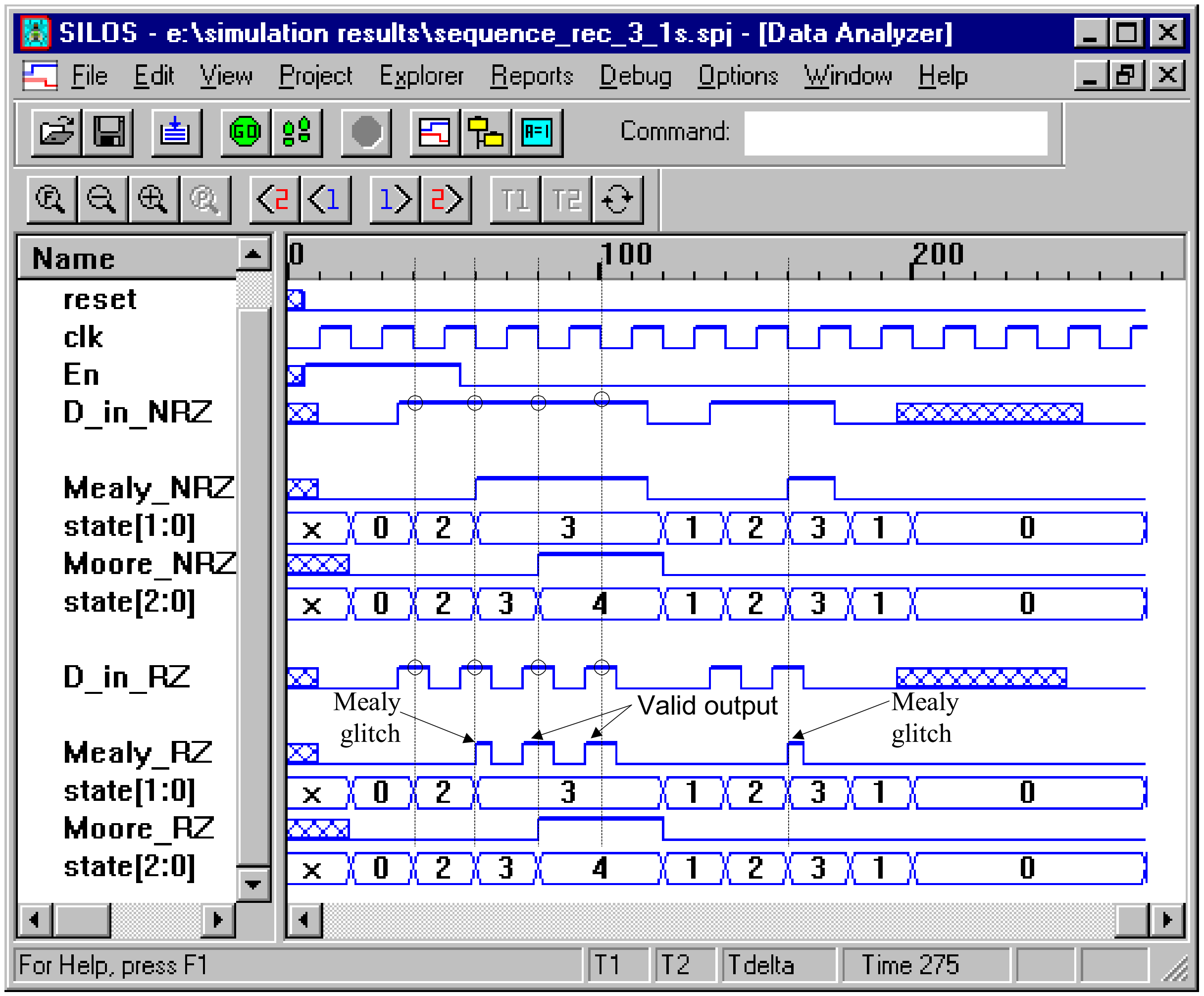
The Moore machine does not anticipate *D_in*

The Mealy machine anticipates *D_in* and asserts *D_out* before the third clock

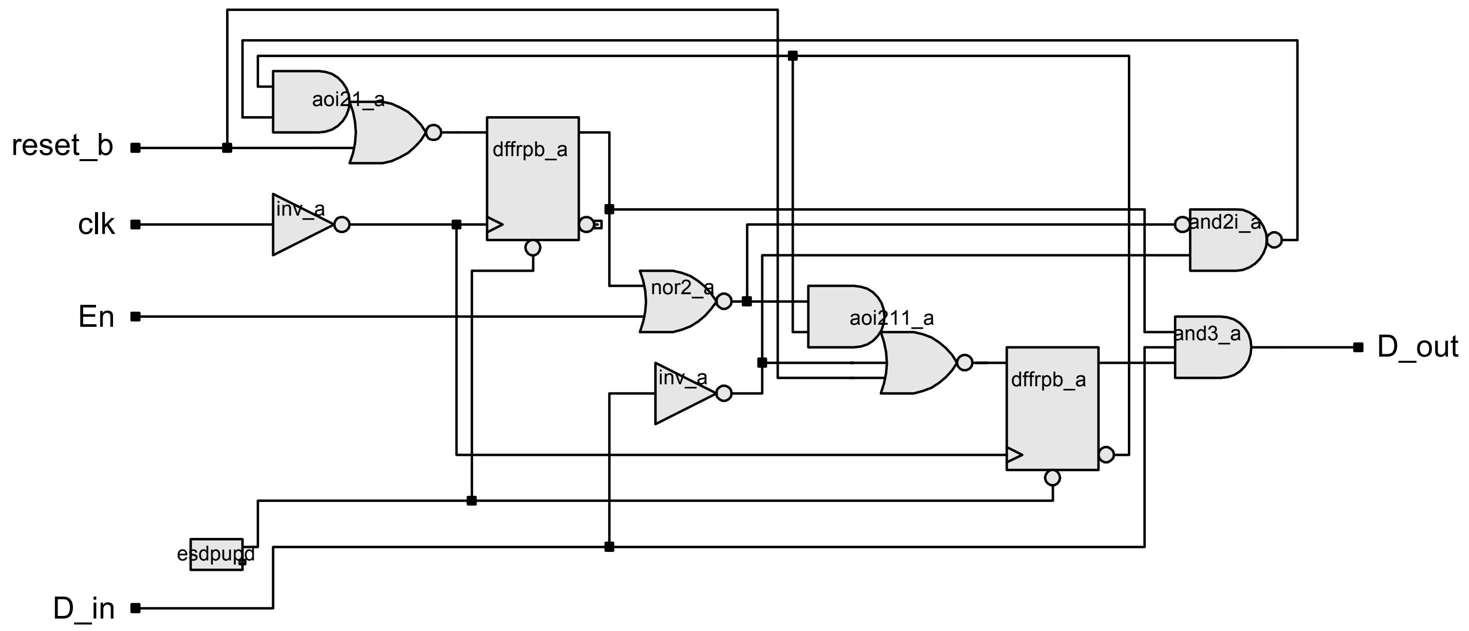
The Mealy machine asserts D_out in the state reached after the third active edge of the clock

Simulation results

- Testbench includes RZ and NRZ input formats
- The Mealy machine has an invalid assertion of its output when the input has an RZ format, an apparent glitch
- Mealy glitch occurs immediately after the second clock and persists until *D_in* is de-asserted
- The valid Mealy output is 0, which is the value of *Mealy_RZ* immediately before the second clock
- The value of *Mealy_RZ* immediately before the third clock is 1(valid output)
- Notice behavior if B_in = x

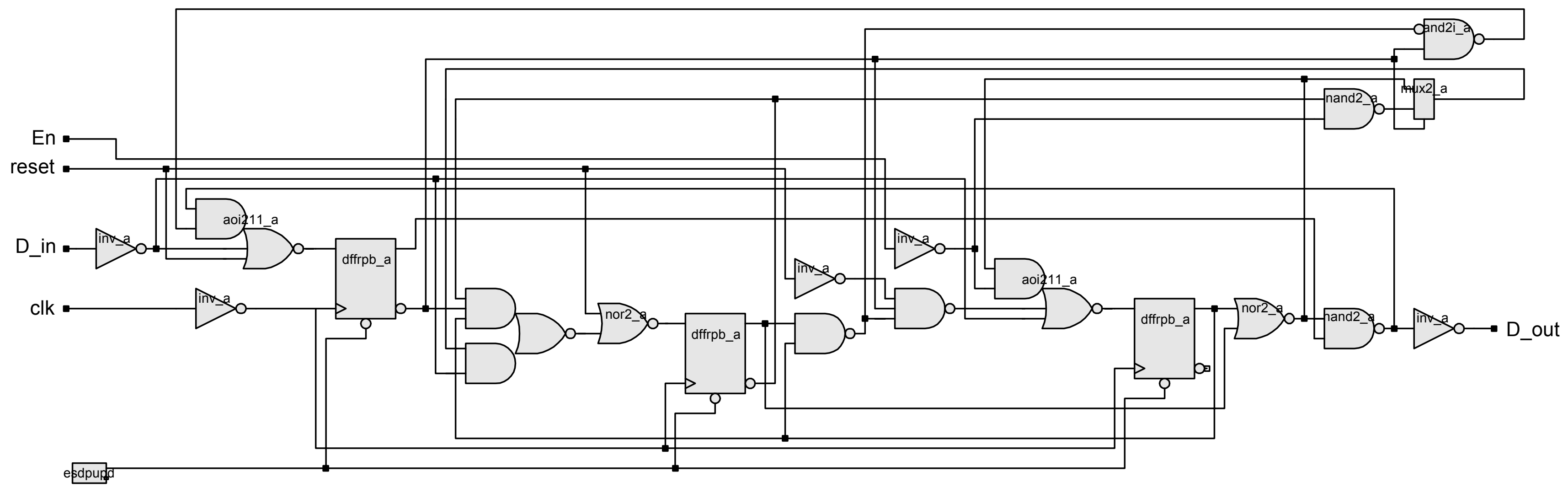

SILOS - e:\simulation results\sequence_rec_3_1s.spj - [Data Analyzer]
File Edit View Project Explorer Reports Debug Options Window Help
Command:
Name
reset
clk
En
D_in_NRZ
Mealy_NRZ
state[1:0]
Moore_NRZ
state[2:0]
D_in_RZ
Mealy_RZ
state[1:0]
Moore_RZ
state[2:0]
0
100
200
Mealy glitch
Valid output
Mealy glitch
For Help, press F1
T1
T2
Tdelta
Time 275

Synthesis Result: Mealy Machine



Synthesis Result: Moore Machine

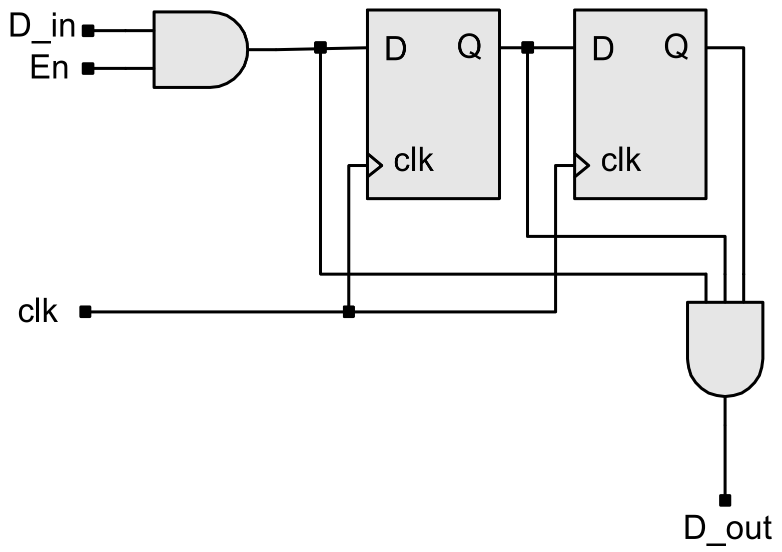
Note: Circuit includes logic to default to *S_idle* if *En* is de-asserted



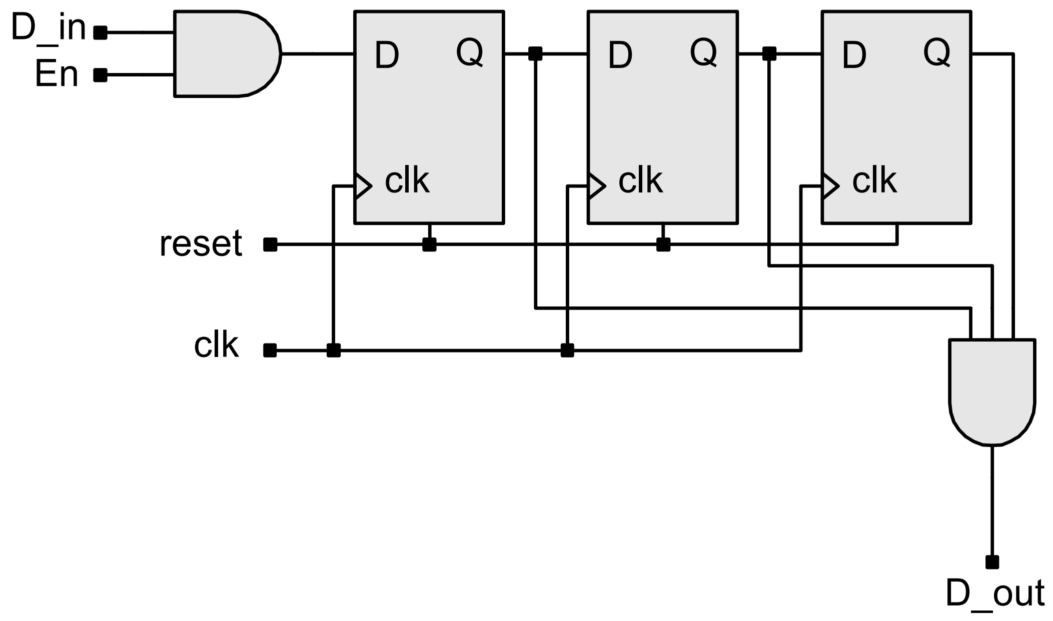
(b)

Alternative approach: Shift input bits through a register and detect contents

Note: The Mealy machine below differs from the previous implementation by gating the datapath with *En*



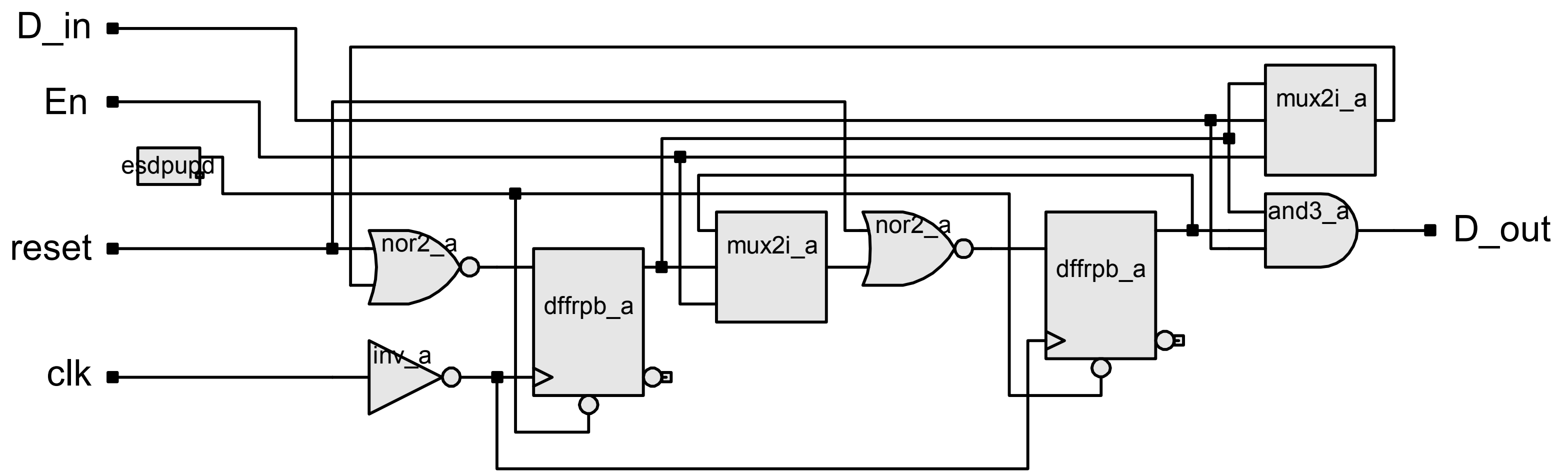
Moore Machine:

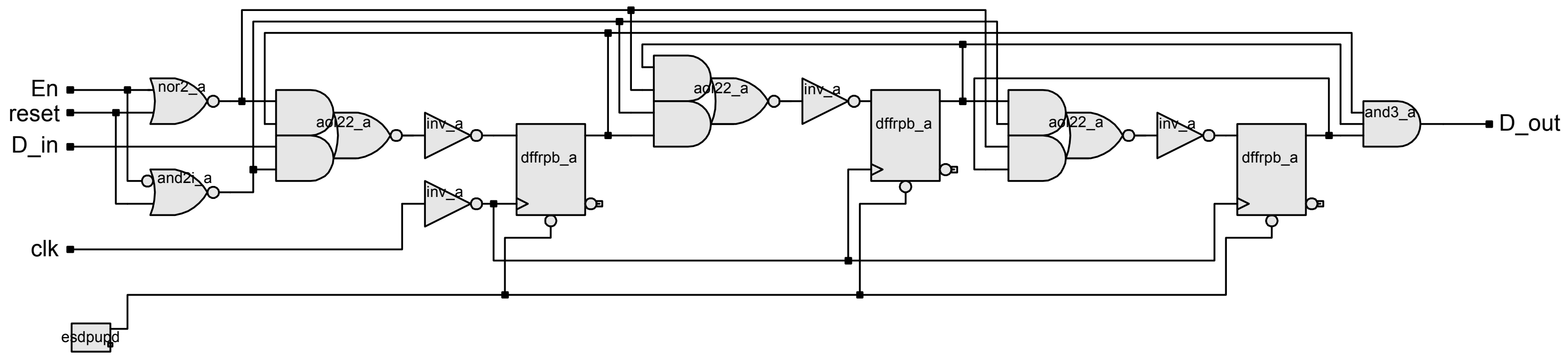


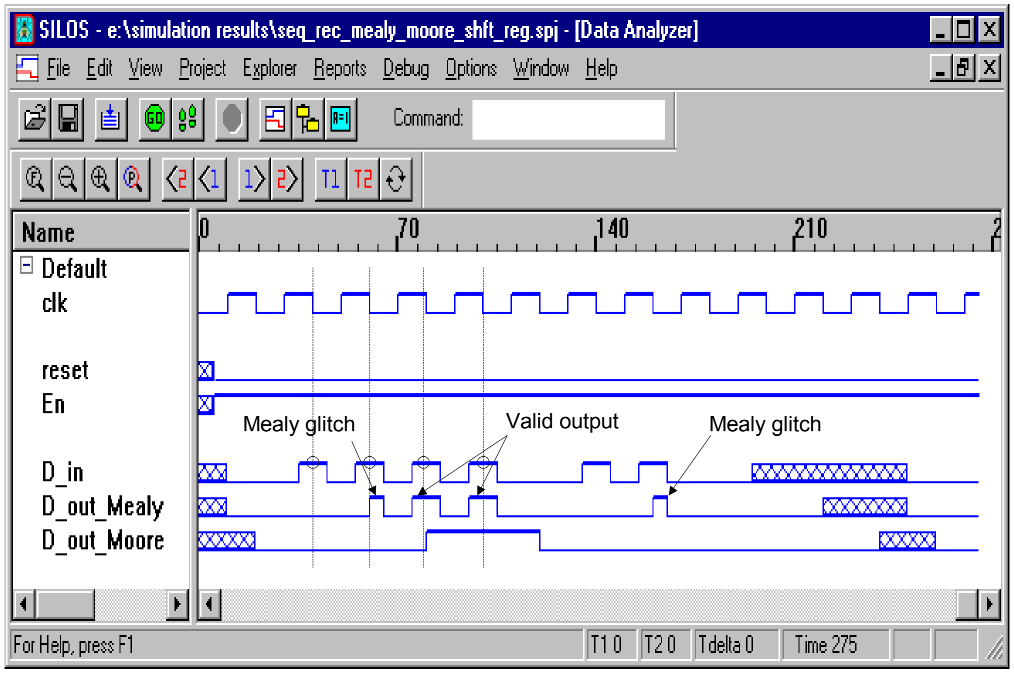
Note: an explicit state machine implementation of a sequence recognizer is not necessarily the most efficient implementation

```
 module Seq_Rec_3_1s_Mealy_Shft_Reg (D_out, D_in, En, clk, reset);
  output       D_out;
  input        D_in, En;
  input        clk, reset;
  parameter  Empty = 2'b00;
  reg      [1: 0]     Data;

  always @ (negedge clk)
    if (reset == 1) Data <= Empty; else if (En  == 1) Data <= {D_in, Data[1]};

  assign D_out = ((Data == 2'b11) && (D_in == 1 ));    // Mealy output
endmodule
```

```
module Seq_Rec_3_1s_Moore_Shft_Reg (D_out, D_in, En, clk, reset);
  output       D_out;
  input        D_in, En;
  input        clk, reset;
  parameter    Empty = 2'b00;
  reg      [2: 0]      Data;

  always @ (negedge clk)
    if (reset == 1) Data <= Empty; else if (En == 1) Data <= {D_in, Data[2:1]};

  assign D_out = (Data == 3'b111);     // Moore output
endmodule
```


D_in
En
esdpupd
reset
clk
nor2_a
inv_a
dffrpb_a
mux2i_a
nor2_a
dffrpb_a
mux2i_a
and3_a
D_out


En
reset
D_in
clk
nor2_a
and2i_a
aoi22_a
inv_a
inv_a
dffrpb_a
aoi22_a
inv_a
dffrpb_a
aoi22_a
inv_a
dffrpb_a
and3_a
D_out
esdpupd

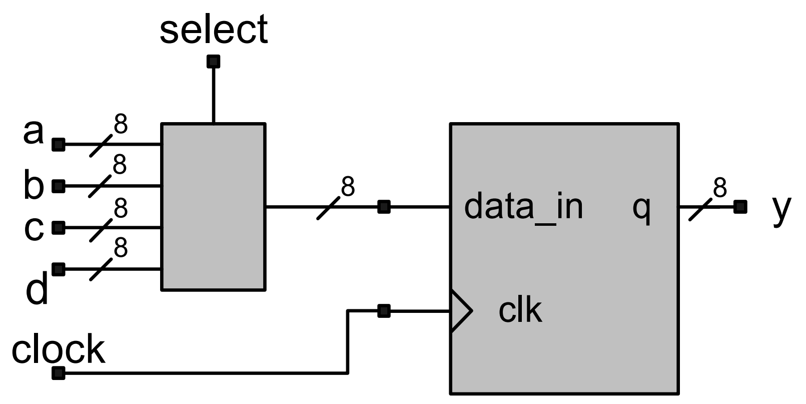

SILOS - e:\simulation results\seq_rec_mealy_moore_shft_reg.spj - [Data Analyzer]
File Edit View Project Explorer Reports Debug Options Window Help
Command:
Name
Default
clk
reset
En
D_in
D_out_Mealy
D_out_Moore
0
70
140
210
Mealy glitch
Valid output
Mealy glitch
For Help, press F1
T1 0
T2 0
Tdelta 0
Time 275

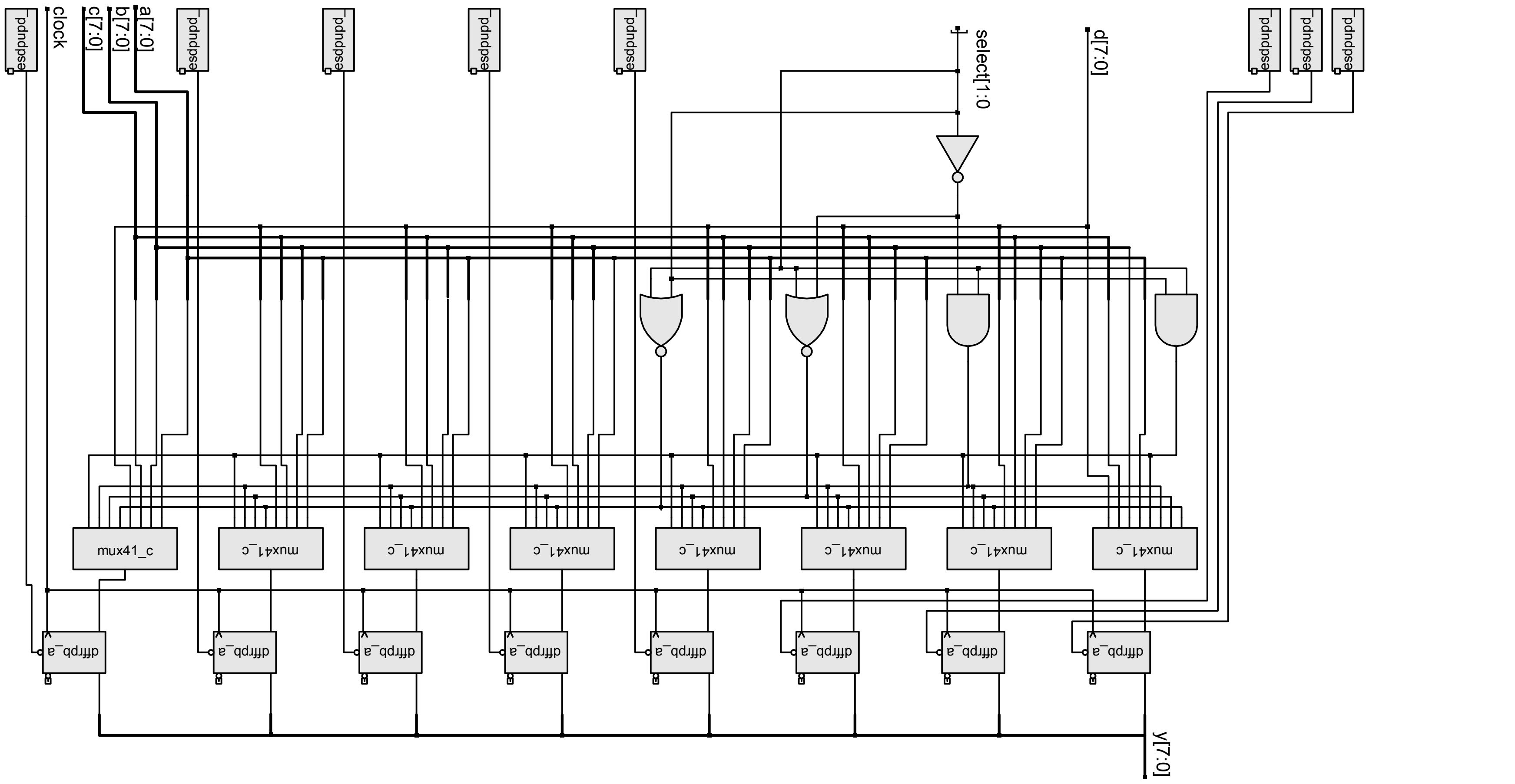
Registered Logic

- Variables whose values are assigned synchronously with a clock signal are said to be *registered*.
- Registered signals are updated at the active edges of the clock and are stable otherwise

Example: Registered Logic

```
module mux_reg (y, a, b, c, d, select, clock);
 output [7: 0]     y;
 input  [7: 0]     a, b, c, d;
 input  [1: 0]     select;
 input             clock;
 reg    [7: 0]     y;

always @ (posedge clock)
 case (select)
    0: y <= a; // non-blocking
    1: y <= b;
    2: y <= c;
    3: y <= d;
    default y <= 8'bx;
  endcase
endmodule
```

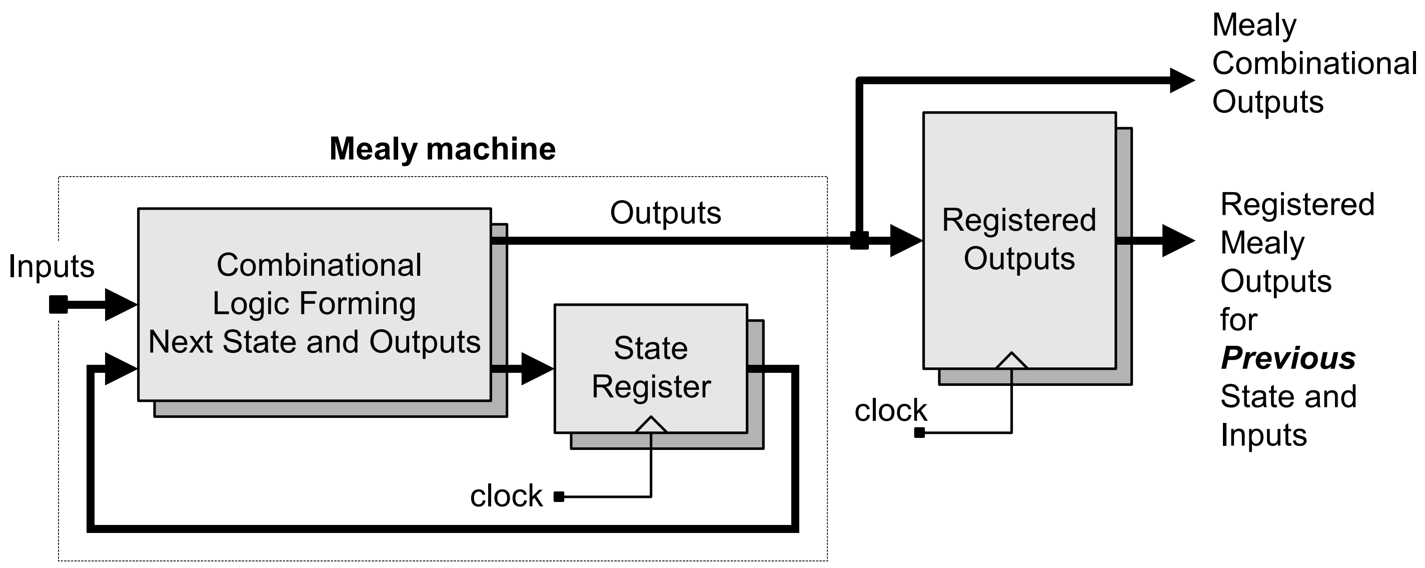




esdpupd
clock
c[7:0]
b[7:0]
a[7:0]
select[1:0]
d[7:0]
mux41_c
dffrpb_a
y[7:0]

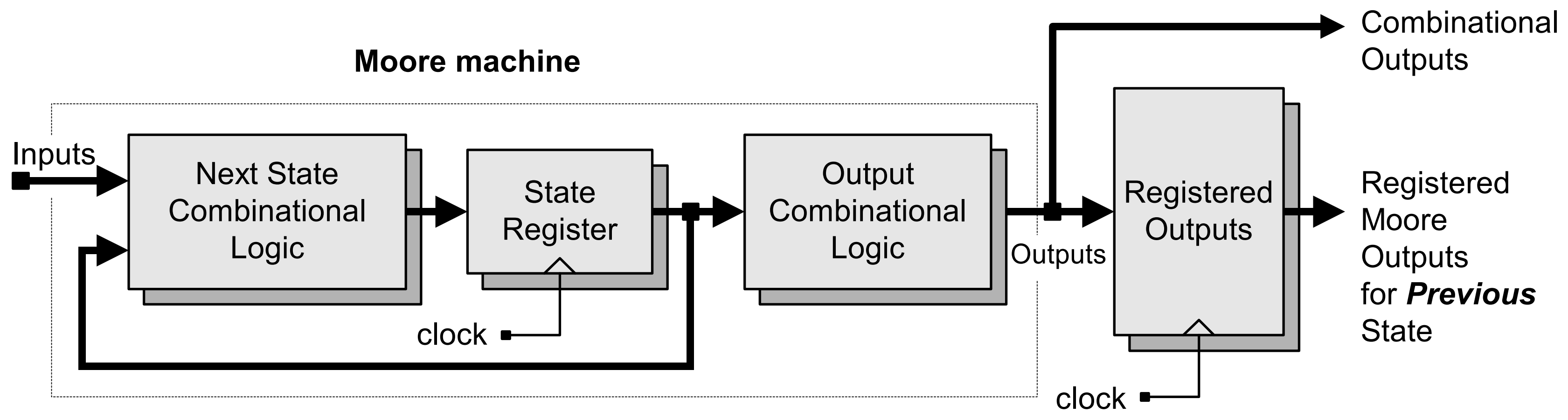
Sequential Machines with Registered Outputs

- Outputs can be registered to prevent glitches from affecting the logic driven by the outputs of the machine.
- Several options


Mealy machine
Inputs
Combinational Logic Forming Next State and Outputs
Outputs
State Register
clock
Registered Outputs
clock
Mealy Combinational Outputs
Registered Mealy Outputs for ***Previous*** State and Inputs

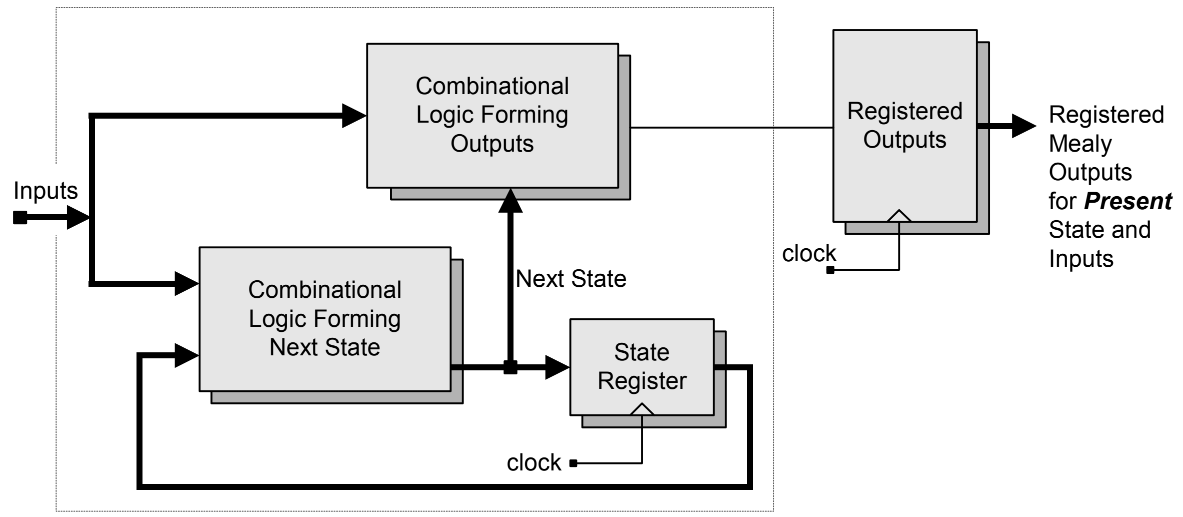
Note:

- The output of the storage register lags the combinational values by one clock cycle
- The output of the register corresponds to the state of the machine in the previous cycle


Moore machine
Inputs
Next State Combinational Logic
State Register
clock
Output Combinational Logic
Outputs
Registered Outputs
clock
Combinational Outputs
Registered Moore Outputs for ***Previous*** State

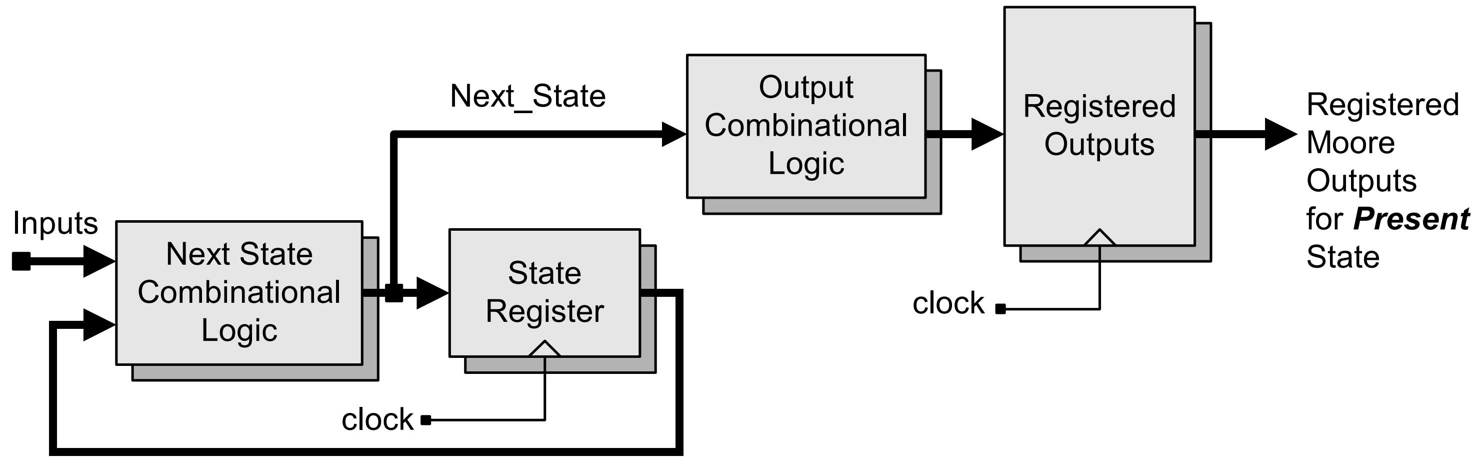
Note:

- The output of the storage register lags the combinational values by one clock cycle
- The output of the register corresponds to the state of the machine in the previous cycle


Combinational Logic Forming Outputs
Inputs
Combinational Logic Forming Next State
Next State
State Register
clock
Registered Outputs
clock
Registered Mealy Outputs for ***Present*** State and Inputs

Note:

- The registered outputs are formed in the same cycle as the state
- The Mealy outputs are formed from the next state and the inputs at the active edge of the clock
- The value stored in the output register corresponds to the state reached at the clock transition and the inputs that caused the transition
-


Next_State
Output Combinational Logic
Registered Outputs
Registered Moore Outputs for ***Present*** State
Inputs
Next State Combinational Logic
State Register
clock
clock

Note: The value stored in the output register will correspond to the state that is stored in the state register

Example: Sequence Recognizer with Registered Outputs

Include the following code in *Seq_Rec_3_1s_Mealy*:[1]

```
reg D_out_reg;
always @ (negedge clk)
  if (reset == 1) D_out_reg <= 0;
  else D_out_reg <= ((state == S_2) && (next_state == S_2) && (D_in
== 1 ));
```

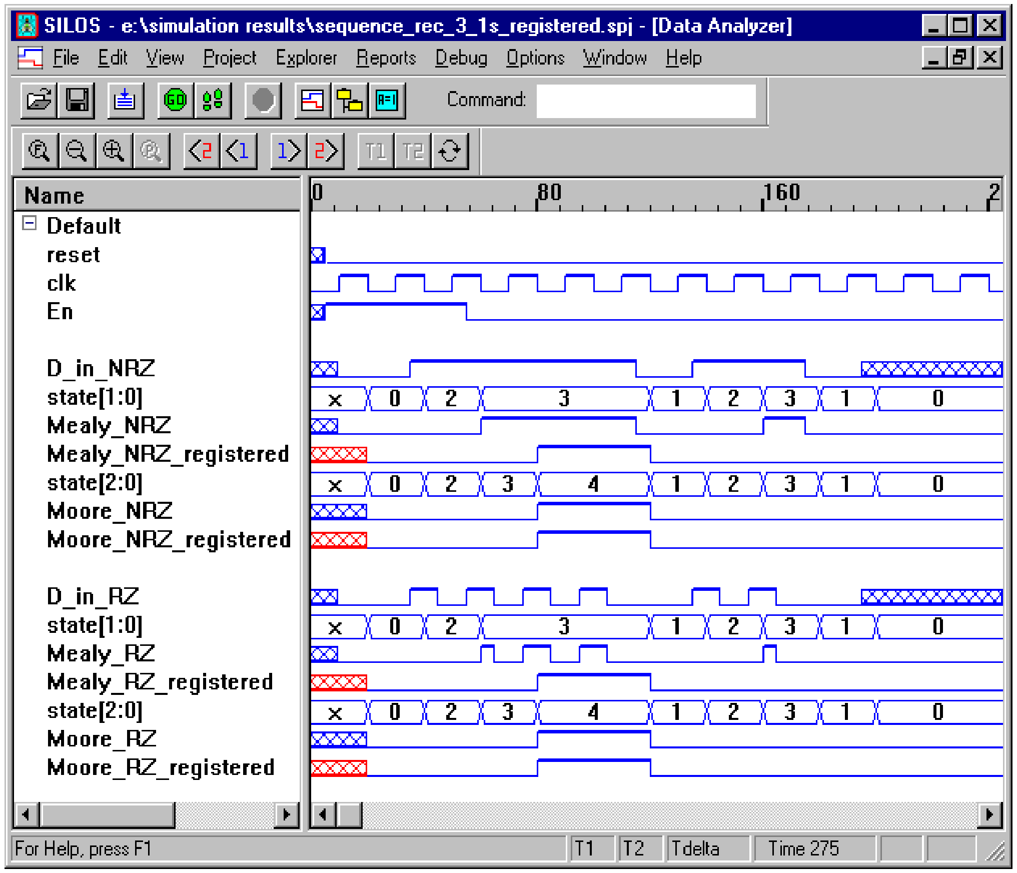
Notice that the clause *(state == S_2)* is included in the logic to prevent a premature assertion while the state of the machine is *S_1* (see the ASM chart in Figure 6.38b).

[1] The port declarations of each machine must be modified to include the registered output.

Include the following code in *Seq_Rec_3_1s_Moore*:

```
reg D_out_reg;
 always @ (negedge clk)
   if (reset == 1) D_out_reg <= 0; else D_out_reg <= (next_state ==
S_3);
```

Simulation Results:



Note:

- Registered and unregistered outputs for NRZ and RZ formatted serial inputs
- The unregistered output of the Mealy machine changes with the input
- The registered output does not change with the input to the machine
- The value of the registered Mealy output corresponds to the value implied by the input and next state at the active (falling) edge of the clock
- The unregistered output anticipates the clock
- The output of the registered machine does not
- The waveforms of the registered and unregistered Moore outputs are identical
- The output of the unregistered machine is formed by combinational logic
- The output of the registered machine the output is the output of a register

State Encoding

#	Binary	One-Hot	Gray	Johnson
0	0000	0000000000000001	0000	00000000
1	0001	0000000000000010	0001	00000001
2	0010	0000000000000100	0011	00000011
3	0011	0000000000001000	0010	00000111
4	0100	0000000000010000	0110	00001111
5	0101	0000000000100000	0111	00011111
6	0110	0000000001000000	0101	00111111
7	0111	0000000010000000	0100	01111111
8	1000	0000000100000000	1100	11111111
9	1001	0000001000000000	1101	11111110
10	1010	0000010000000000	1111	11111100
11	1011	0000100000000000	1110	11111000
12	1100	0001000000000000	1010	11110000
13	1101	0010000000000000	1011	11100000
14	1110	0100000000000000	1001	11000000
15	1111	1000000000000000	1000	10000000

Note:

- A binary coded decimal format (BCD uses the minimal number of flip-flops, but does not necessarily lead to an optimal realization of the combinational logic used to decode the next state and output of the machine.

- If a machine has more than 16 states, a binary code will result in a relatively large amount of next-state logic; the machine's speed will also be slower than alternative encoding.

- A Gray code uses the same number of bits as a binary code, but has the feature that two adjacent codes differ by only one bit, which can reduce the electrical noise in a circuit.

- A Johnson code has the same property, but uses more bits.
- A code that changes by only one bit between adjacent codes will reduce the simultaneous switching of adjacent physical signal lines in a circuit, thereby minimizing the possibility of electrical crosstalk.

- These codes also minimize transitions through intermediate states.

One-Hot Codes

- One flip-flop for each state
- Reduces the decoding logic for next state and output
- Complexity does not increase as states are added to the machine
- Tradeoff: speed is not compromised by the time required to decode the state
- Cost: area of the additional flip flops and signal routing
- A one-hot encoding with an ***if*** statement that tests individual bits might provide simpler decoding logic than decoding with a case statement
- One-hot encoding usually does not correspond to the optimal state assignment
- Use one-hots in Xilinx to reduce the use of CLBs

- Note: in large machines, one-hot encoding will have several unused states, in addition to requiring more registers than alternative encoding
- Gray encoding is recommended for machines having more than 32 states because it requires fewer flip-flops than one-hot encoding, and is more reliable than binary encoding because fewer bits change simultaneously.
- Caution: if a state assignment does not exhaust the possibilities of a code, then additional logic will be required to detect and recover from transitions into unused states.