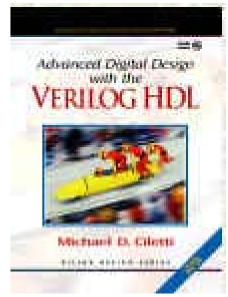
# Advanced Digital Design with the Verilog HDL




M. D. Ciletti

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

**ciletti@vlsic.uccs.edu**

## Draft: Chap 5: Logic Design with Behavioral Models of Combinational and Sequential Logic (Rev 9/23/2003)

**Note to the instructor: These slides are provided <u>solely for classroom use</u> in academic institutions by the instructor using the text, *Advance Digital Design with the Verilog HDL* by Michael Ciletti, published by Prentice Hall. This material may not be used in off-campus instruction, resold, reproduced or generally distributed in the original or modified format for any purpose without the permission of the <u>Author.</u> This material may <u>not</u> be placed on any server or network, and is protected under all copyright laws, as they currently exist. I am providing these slides to you subject to your agreeing that you will not provide them to your students in hardcopy or electronic format or use them for off-campus instruction of any kind. <u>Please email to me your agreement to these conditions.</u>**

**I will greatly appreciate your assisting me by calling to my attention any errors or any other revisions that would enhance the utility of these slides for classroom use.**

# 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

# Data Types

- Two families of data types for variables:

  Nets: wire, tri, wand, triand, wor, trior, supply0, supply1

  Registers: reg, integer, real, time, realtime

- Nets establish structural connectivity
- Register variables act as storage containers for the waveform of a signal
- Default size of a net or reg variable is a signal bit
- An **integer** is stored at a minimum of 32 bits
- **time** is stored as 64 bit integer
- **real** is stored as a real number
- **realtime** stores the value of time as a real number

# Behavioral Models

- Behavioral models are abstract descriptions of functionality.
- Widely used for quick development of model
- Follow by synthesis
- We'll consider two types:
  - Continuous assignment (Boolean equations)
  - Cyclic behavior (more general, e.g. algorithms)

# Example: Abstract Models of Boolean Equations

- Continuous assignments (Keyword: **assign)** are the Verilog counterpart of Boolean equations
- Hardware is implicit (i.e. combinational logic)

**Example** 5.1 (p 145): Revisit the AOI circuit in Figure 4.7

```
module AOI_5_CA0 (y_out, x_in1, x_in2, x_in3, x_in4, x_in5);
  input        x_in1, x_in2, x_in3, x_in4, x_in5;
  output       y_out;

  assign y_out =  ~((x_in1 & x_in2) | (x_in3 & x_in4 & x_in5));

endmodule
```

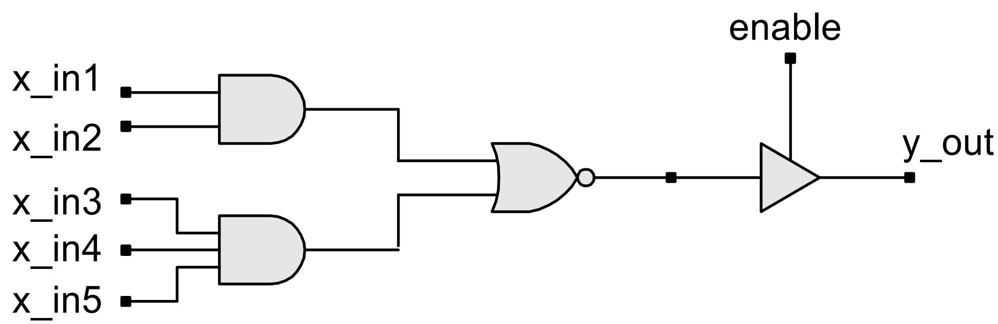
- The LHS variable is monitored automatically and updates when the RHS expression changes value

# Example 5.2 (p 146)

```
module AOI_5_CA1 (y_out, x_in1, x_in2, x_in3, x_in4, x_in5, enable);
  input        x_in1, x_in2, x_in3, x_in4, x_in5, enable;
  output       y_out;

  assign y_out = enable ? ~((x_in1 & x_in2) | (x_in3 & x_in4 & x_in5)) : 1'bz;

endmodule
```

- The *conditional operator (? :)* acts like a software if-then-else switch that selects between two expressions.
- Must provide both expressions

# Equivalent circuit:

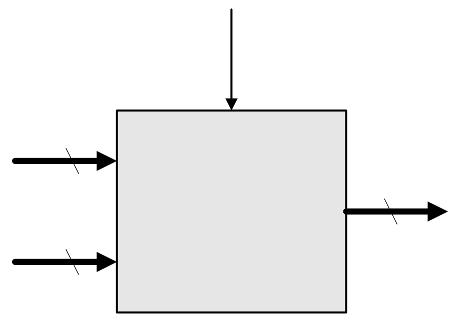


# Implicit Continuous Assignment

**Example 5.3**

```
module AOI_5_CA2 (y_out, x_in1, x_in2, x_in3, x_in4, x_in5, enable);
  input         x_in1, x_in2, x_in3, x_in4, x_in5, enable;
  output        y_out;

  wire y_out = enable ? ~((x_in1 & x_in2) | (x_in3 & x_in4 & x_in5)) : 1'bz;

endmodule
```

# Example 5.4 (p 148)



```
module Mux_2_ 32_CA ( mux_out, data_1, data_0, select);
  parameter                                word_size = 32;
  output          [word_size -1: 0]        mux_out;
  input           [word_size-1: 0]         data_1, data_0;
  input                                    select;

  assign mux_out = select ? data_1 : data_0;
endmodule
```

# Propagation Delay for Continuous Assignments

Example 5.3 (Note: Three-state behavior)

```
module AOI_5 _CA2 (y_out, x_in1, x_in2, x_in3, x_in4);
  input        x_in1, x_in2, x_in3, x_in4;
  output       y_out;

  wire #1 y1 = x_in1 & x_in2;    // Bitwise and operation
  wire #1 y2 = x_in3 & x_in_4;
  wire #1 y_out = ~ (y1 | y2);   // Complement the result of bitwise OR operation
endmodule
```

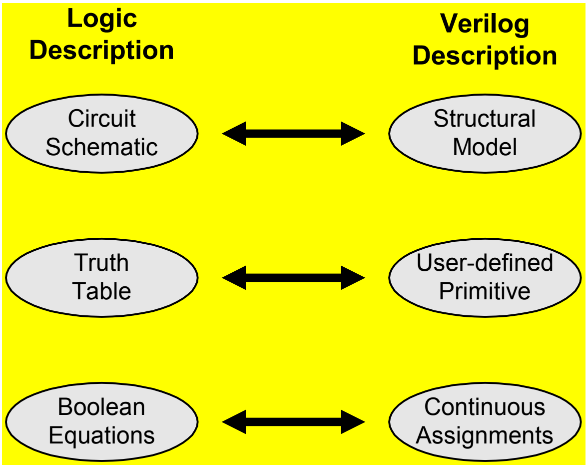
# Multiple Continuous Assignments

- Multiple continuous assignments are active concurrently

```
module compare_2_CA0 (A_lt_B, A_gt_B, A_eq_B, A1, A0, B1, B0);
  input    A1, A0, B1, B0;
  output   A_lt_B, A_gt_B, A_eq_B;

  assign A_lt_B = (~A1) & B1 | (~A1) & (~A0) & B0 | (~A0) & B1 & B0;
  assign A_gt_B = A1 & (~B1) | A0 & (~B1) & (~B0) | A1 & A0 & (~B0);
  assign A_eq_B = (~A1) & (~A0) & (~B1) & (~B0) | (~A1) & A0 & (~B1) & B0
      | A1 & A0 & B1 & B0 | A1 & (~A0) & B1 & (~B0);

endmodule
```

- Note: this style can become unwieldy and error-prone

# Review of Modeling Styles for Combinational Logic



# Latched and Level-Sensitive Behavior

- Avoid explicit or implicit structural feedback
- It simulates but won't synthesize
- Timing analyzers won't work either

**Example**

```
assign q = set ~& qbar;
assign qbar = rst ~& q;
```

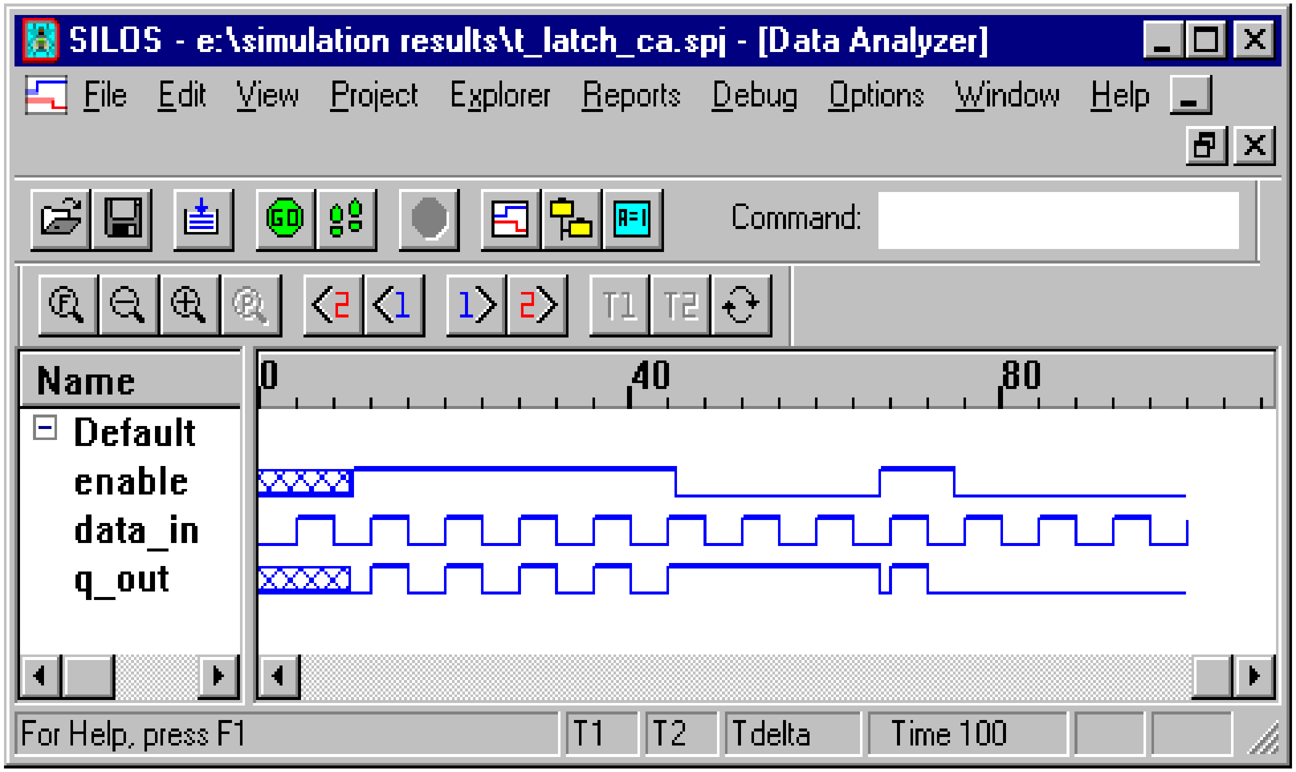
# Recommended Style for Transparent Latch

- **Use a continuous assignment with feedback to model a latch**
- **Synthesis tools understand this model**

**Example 5.7**

```
module Latch_CA (q_out, data_in, enable);
  output          q_out;
  input           data_in, enable;

  assign q_out = enable ? data_in : q_out;
endmodule
```

Simulation results:

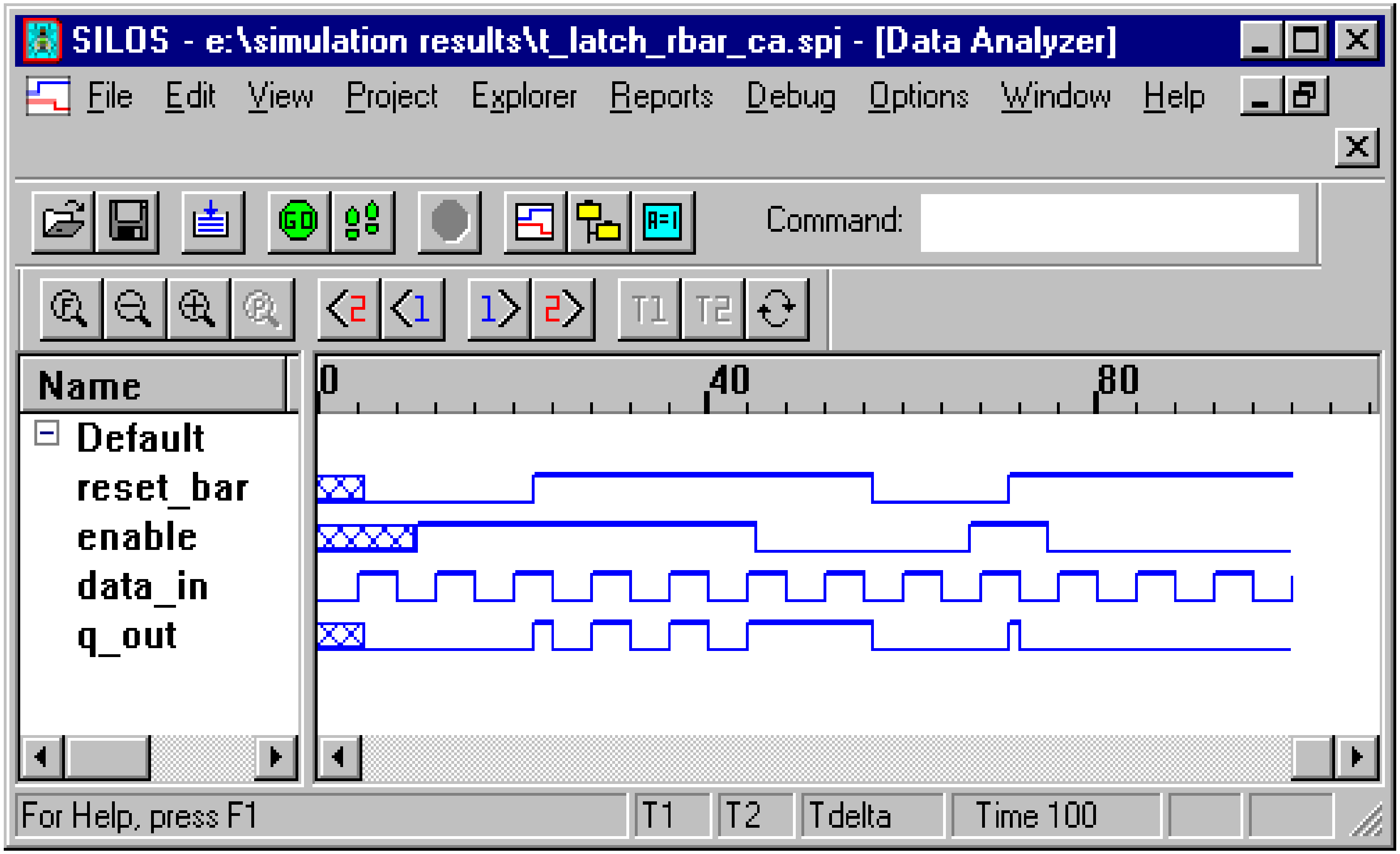


# Example: T-Latch with Active-Low Reset

Example 5.8: T-latch with active-low reset (nested conditional operators)

```
module Latch_Rbar_CA (q_out, data_in, enable, reset_bar);
  output        q_out;
  input         data_in, enable, reset_bar;

  assign q_out = !reset_bar ? 0 : enable ? data_in : q_out;
endmodule
```

## Simulation results:



# Abstract Modeling with Cyclic Behaviors

- Cyclic behaviors assign values to register variables to describe the behavior of hardware
- Model level-sensitive and edge-sensitive behavior
- Synthesis tool selects the hardware
- Note: Cyclic behaviors re-execute after executing the last procedural statement executes (subject to timing controls – more on this later)

# Example 5.9: D-type Flip-Flop

```
module df_behav (q, q_bar, data, set, reset, clk);
input           data, set, clk, reset;
  output        q, q_bar;
  reg           q;

  assign q_bar = ~ q;

  always @  (posedge clk)  // Flip-flop with synchronous set/reset
  begin
    if (reset == 0) q <= 0;         // <= is the nonblocking assignment operator
      else if (set ==0) q <= 1;
        else q <= data;
  end
endmodule
```

# Example 5.10 (Asynchronous reset)

```
module asynch_df_behav (q, q_bar, data, set, clk, reset );
  input          data, set, reset, clk;
  output         q, q_bar;
  reg            q;

  assign q_bar = ~q;

  always @  (negedge set or negedge reset or posedge clk)
    begin
      if (reset == 0) q <= 0;
      else if (set == 0) q <= 1;
        else q <= data;              // synchronized activity
     end
endmodule
```

Note: See discussion in text

Note: Consider simultaneous assertion of set and reset).

# Example: Transparent Latch (Cyclic Behavior)

```
module tr_latch (q_out, enable, data);
  output q_out;
  input enable, data;
  reg q_out;

  always @  (enable or data)
    begin
      if (enable) q_out = data;
    end
endmodule
```

# Alternative Behavioral Models

**Example 5. 12** (Two-bit comparator)

```
module compare_2_CA1  (A_lt_B, A_gt_B, A_eq_B, A1, A0, B1, B0);
  input            A1, A0, B1, B0;
  output           A_lt_B, A_gt_B, A_eq_B;

  assign   A_lt_B = ({A1,A0} < {B1,B0});
  assign   A_gt_B = ({A1,A0} > {B1,B0});
  assign   A_eq_B = ({A1,A0} == {B1,B0});
endmodule
```

**Example 5.13** (Clarity!)

```
module compare_2_CA1 (A_lt_B, A_gt_B, A_eq_B, A, B);
  input    [1: 0]      A, B;
  output               A_lt_B, A_gt_B, A_eq_B;

  assign  A_lt_B = (A < B);     // The RHS expression is true (1) or false (0)
  assign  A_gt_B = (A > B);
  assign  A_eq_B = (A == B);
endmodule
```

**Example 5.14** (Parameterized and reusable model)

```
module compare_32_CA (A_gt_B, A_lt_B, A_eq_B, A, B);
  parameter                              word_size = 32;
  input       [word_size-1: 0]           A, B;
  output                                 A_gt_B, A_lt_B, A_eq_B;

  assign  A_gt_B = (A > B),              // Note: list of multiple assignments
          A_lt_B = (A < B),
          A_eq_B = (A == B);
endmodule
```

# Dataflow – RTL Models

- Dataflow (register transfer level) models of combinational logic describe concurrent operations on datapath signals, usually in a synchronous machine

## Example 5.15

```
module compare_2_RTL (A_lt_B, A_gt_B, A_eq_B, A1, A0, B1, B0);
  input           A1, A0, B1, B0;
  output          A_lt_B, A_gt_B, A_eq_B;
  reg             A_lt_B, A_gt_B, A_eq_B;

  always @ (A0 or A1 or B0 or B1) begin
    A_lt_B =      ({A1,A0} < {B1,B0});
    A_gt_B =      ({A1,A0} > {B1,B0});
    A_eq_B =      ({A1,A0} == {B1,B0});
  end
endmodule
```

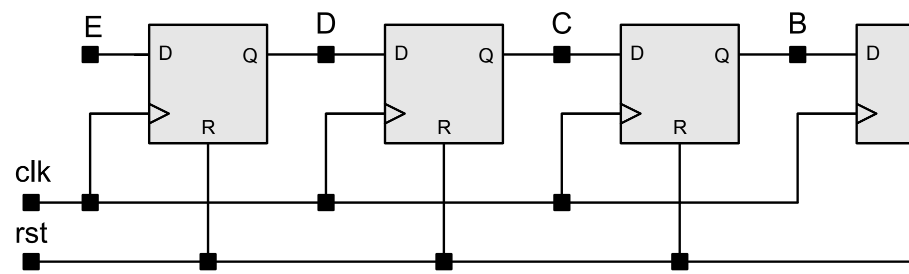
# Modeling Trap

- The order of execution of procedural statements in a cyclic behavior may depend on the order in which the statements are listed
- A procedural assignment cannot execute until the previous statement executes
- Expression substitution is recognized by synthesis tools

## Example 5.16

```
module shiftreg_PA (E, A, clk, rst);
  output   A;
  input    E;
  input    clk, rst;
  reg A, B, C, D;

  always @ (posedge clk or posedge rst) begin
    if (reset) begin A = 0; B = 0; C = 0; D = 0; end
    else begin
      A = B;
      B = C;
      C = D;
      D = E;
    end
  end
endmodule
```

**Result of synthesis:**



**Reverse the order of the statements:**

```
module shiftreg_PA_rev (A, E, clk, rst);
  output   A;
  input    E;
  input    clk, rst;
  reg A, B, C, D;

  always @ (posedge clk or posedge rst) begin
    if (rst) begin A = 0; B = 0; C = 0; D = 0; end
    else begin
      D = E;
      C = D;
      B = C;
      A = B;
    end
  end
endmodule
```

**Result of synthesis:**

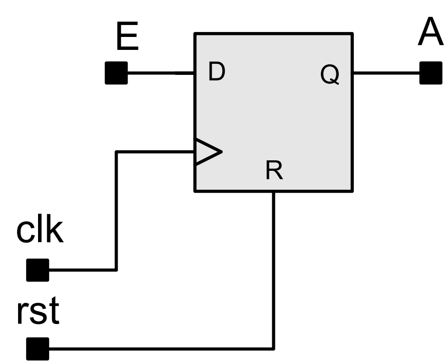



**Figure 5.8** Circuit synthesized as a result of expression substitution in an incorrect model of a 4-bit serial shift register.

# Nonblocking Assignment Operator and Concurrent Assignments

- Nonblocking assignment statements execute *concurrently* (in parallel) rather than sequentially
- The order in which nonblocking assignments are listed has no effect.
- Mechanism: the RHS of the assignments are sampled, then assignments are updated
- Assignments are based on values held by RHS before the statements execute
- Result: No dependency between statements

# Example: Shift Register

**Example 5.17**

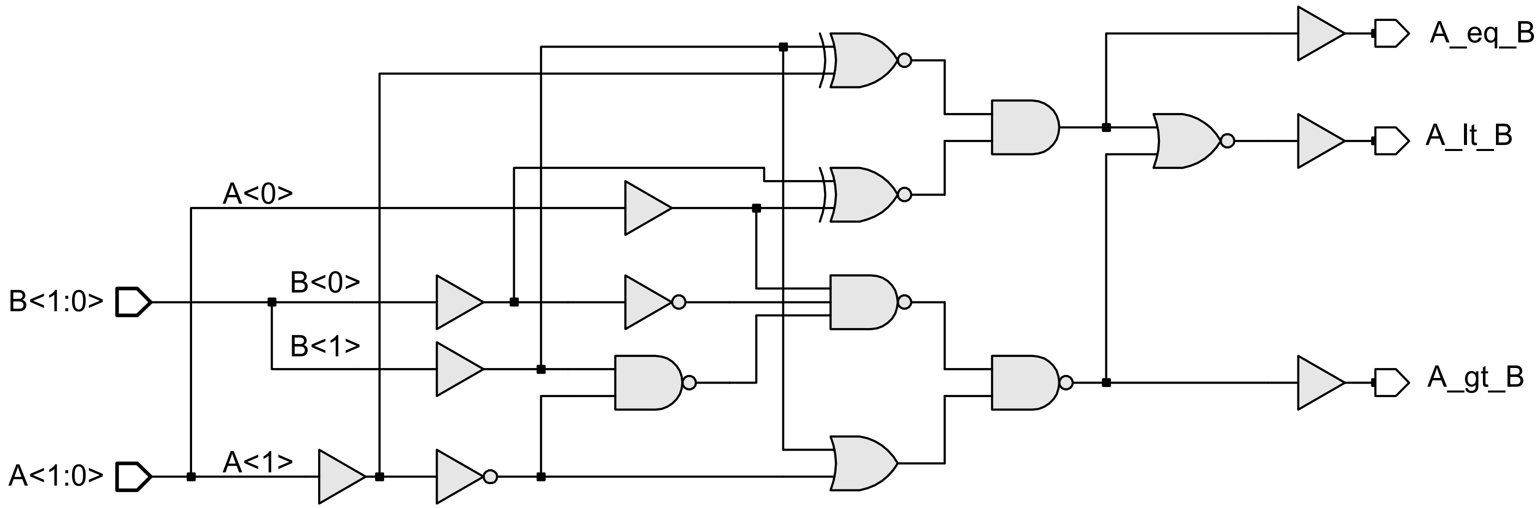
```
module shiftreg_nb (A, E, clk, rst);
  output  A;
  input   E;
  input   clk, rst;
  reg     A, B, C, D;

always @ (posedge clk or posedge rst) begin
  if (rst) begin A <= 0; B <= 0; C <= 0; D <= 0; end
  else begin
    A <= B;        //     D <= E;
    B <= C;        //     C <= D;
    C <= D;        //     B <= D;
    D <= E;        //     A <= B;
  end
  end
endmodule
```

# Algorithm-Based Models

**Example 5.18**

```
module compare_2_algo (A_lt_B, A_gt_B, A_eq_B, A, B);
 output              A_lt_B, A_gt_B, A_eq_B;
 input     [1: 0]    A, B;

 reg                 A_lt_B, A_gt_B, A_eq_B;

 always @  (A or B)       // Level-sensitive behavior
  begin
   A_lt_B = 0;
   A_gt_B = 0;
   A_eq_B = 0;
   if (A == B)       A_eq_B = 1;    // Note: parentheses are required
   else if (A > B)   A_gt_B = 1;
   else              A_lt_B = 1;
  end
endmodule
```

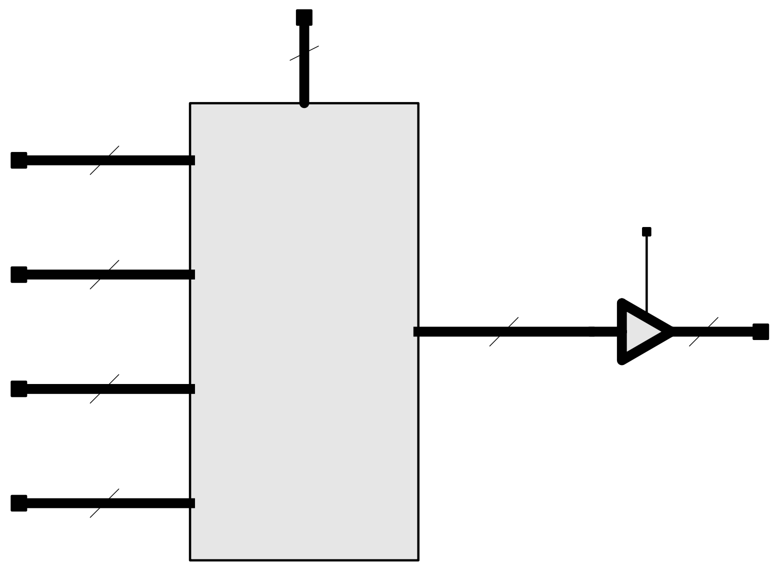
# Result of synthesis:



# Simulation with Behavioral Models

See discussion in text – p 165

# Example 5.19: Four-Channel Mux with Three-State Output



```
module Mux_4_32_case   (mux_out, data_3, data_2, data_1, data_0, select, enable);
     output   [31: 0]     mux_out;
     input    [31: 0]     data_3, data_2, data_1, data_0;
     input    [1: 0]      select;
     input                enable;
     reg      [31: 0]     mux_int;

     assign mux_out = enable ? mux_int : 32'bz;

     always @ ( data_3 or data_2 or data_1 or data_0 or select)
      case (select)
       0:          mux_int = data_0;
       1:          mux_int = data_1;
       2:          mux_int = data_2;
       3:          mux_int = data_3;
       default:    mux_int = 32'bx;          // May execute in simulation
      endcase
    endmodule
```

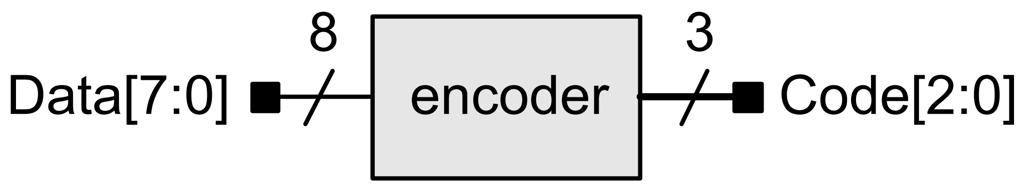
# Example 5.20: Alternative Model

```
module Mux_4_32_if
  (mux_out, data_3, data_2, data_1, data_0, select, enable);
  output   [31: 0]    mux_out;
  input    [31: 0]    data_3, data_2, data_1, data_0;
  input    [1: 0]     select;
  input               enable;
  reg      [31: 0]    mux_int;

  assign mux_out = enable ? mux_int : 32'bz;

  always @ (data_3 or data_2 or data_1 or data_0 or select)
    if (select == 0) mux_int = data_0; else
      if (select == 1) mux_int = data_1; else
        if (select == 2) mux_int = data_2; else
          if (select == 3) mux_int = data_3; else mux_int = 32'bx;
endmodule
```

# Example 5.21: Alternative Model

```
module Mux_4_32_CA (mux_out, data_3, data_2, data_1, data_0, select, enable);
    output     [31: 0]   mux_out;
    input      [31: 0]   data_3, data_2, data_1, data_0;
    input      [1: 0]    select;
    input             enable;
    wire       [31: 0]   mux_int;

    assign mux_out = enable ? mux_int : 32'bz;
    assign mux_int = (select == 0)  ? data_0 :
                         (select == 1) ? data_1:
                             (select == 2) ? data_2:
                                 (select == 3) ? data_3: 32'bx;
  endmodule
```

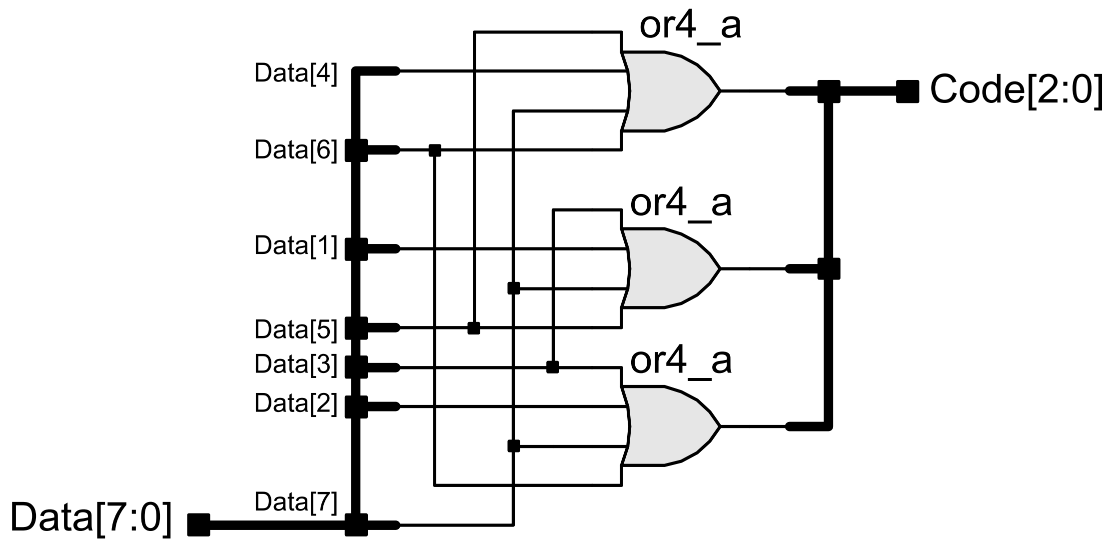
# Example 5.22: Encoder




```
module encoder (Code, Data);
  output        [2: 0] Code;
  input         [7: 0] Data;
  reg           [2: 0] Code;

  always @  (Data)
    begin
      if (Data == 8'b00000001) Code = 0; else
      if (Data == 8'b00000010) Code = 1; else
      if (Data == 8'b00000100) Code = 2; else
      if (Data == 8'b00001000) Code = 3; else
      if (Data == 8'b00010000) Code = 4; else
      if (Data == 8'b00100000) Code = 5; else
      if (Data == 8'b01000000) Code = 6; else
      if (Data == 8'b10000000) Code = 7; else Code = 3'bx;
    end
```

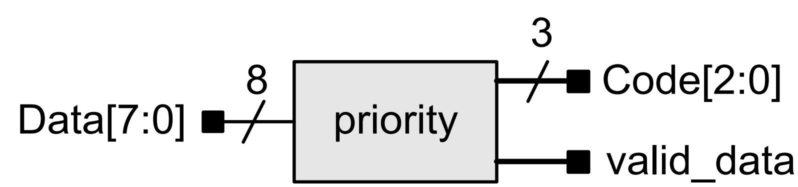
```
/* Alternative description is given below

always @  (Data)
  case (Data)
      8'b00000001     : Code = 0;
      8'b00000010     : Code = 1;
      8'b00000100     : Code = 2;
      8'b00001000     : Code = 3;
      8'b00010000     : Code = 4;
      8'b00100000     : Code = 5;
      8'b01000000     : Code = 6;
      8'b10000000     : Code = 7;
      default         : Code = 3'bx;
  endcase
*/
endmodule
```

Synthesis result (standard cells):



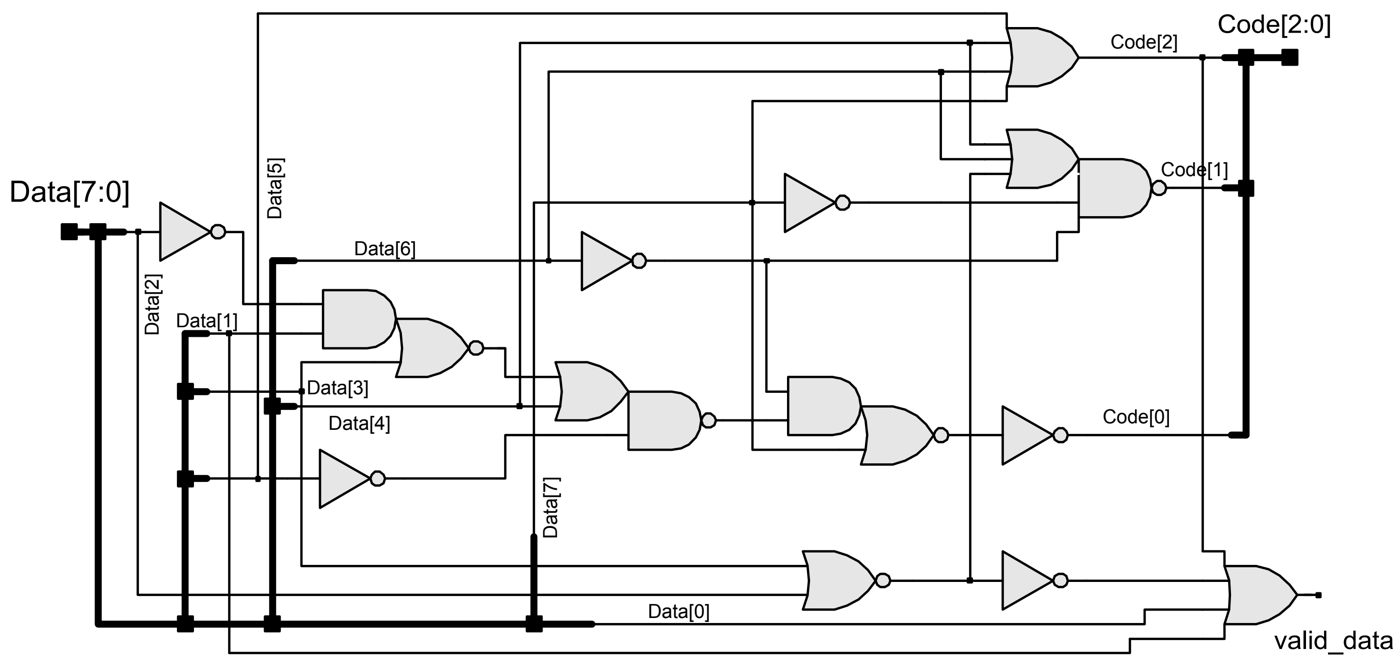
# Example 5.23: Priority Encoder

```
module priority (Code, valid_data, Data);
  output       [2: 0] Code;
  output              valid_data;
  input        [7: 0] Data;
  reg          [2: 0] Code;

  assign       valid_data = |Data; // "reduction or" operator

  always @  (Data)
    begin
      if (Data[7]) Code = 7; else
      if (Data[6]) Code = 6; else
      if (Data[5]) Code = 5; else
      if (Data[4]) Code = 4; else
      if (Data[3]) Code = 3; else
      if (Data[2]) Code = 2; else
      if (Data[1]) Code = 1; else
      if (Data[0]) Code = 0; else
                   Code = 3'bx;
    end
```

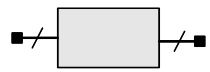


```
/*// Alternative description is given below

always @  (Data)
  casex (Data)
       8'b1xxxxxxx     : Code = 7;
       8'b01xxxxxx     : Code = 6;
       8'b001xxxxx     : Code = 5;
       8'b0001xxxx     : Code = 4;
       8'b00001xxx     : Code = 3;
       8'b000001xx     : Code = 2;
       8'b0000001x     : Code = 1;
       8'b00000001     : Code = 0;
       default         : Code = 3'bx;
  endcase
*/
endmodule
```

# Synthesis Result:



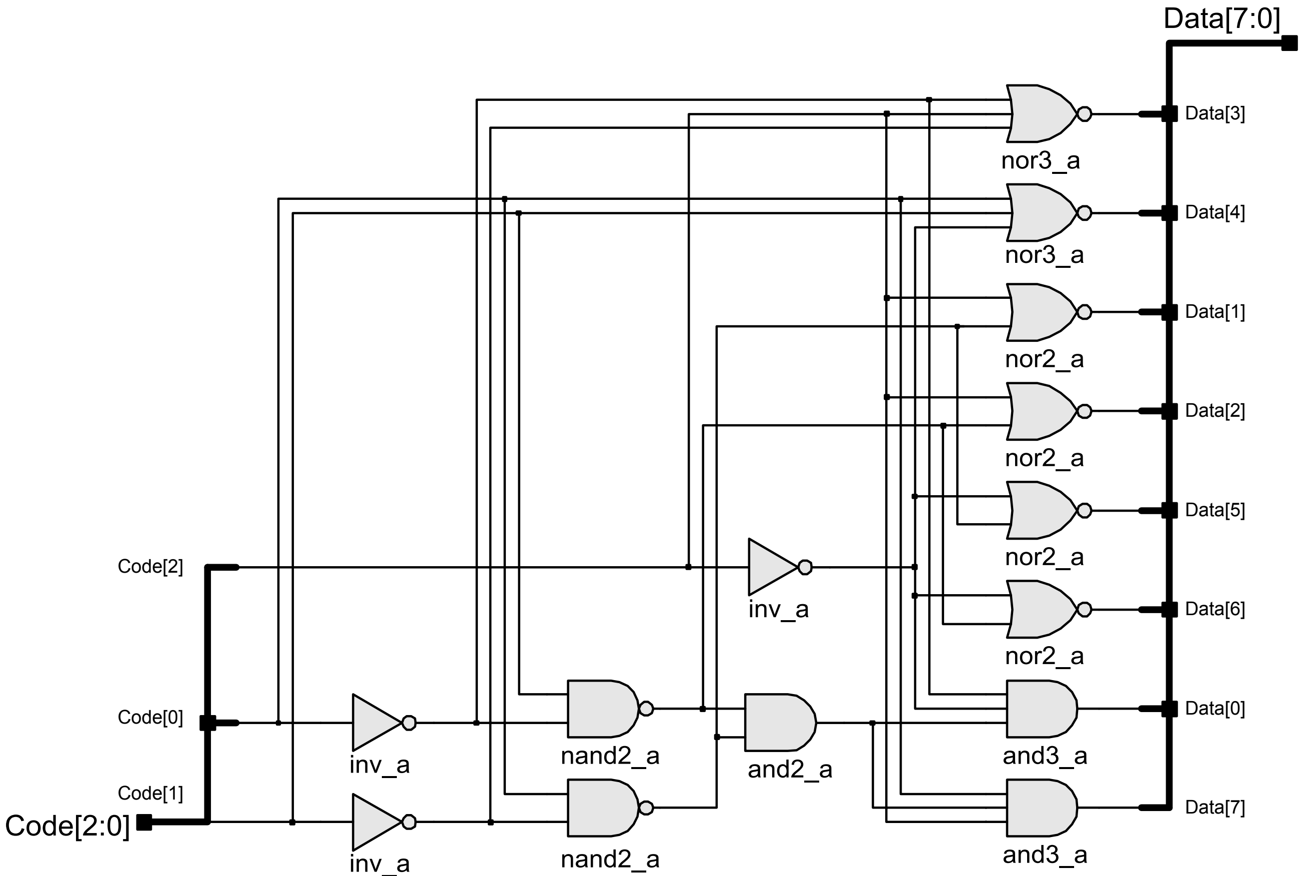
# Example 5.24: Decoder



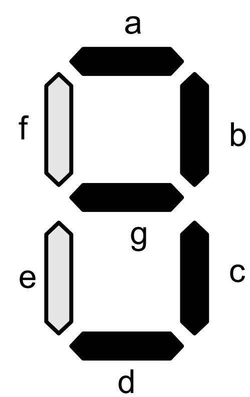
```
module decoder (Data, Code);
  output         [7: 0] Data;
  input          [2: 0] Code;
  reg            [7: 0] Data;

  always @  (Code)
    begin
      if (Code == 0) Data = 8'b00000001; else
      if (Code == 1) Data = 8'b00000010; else
      if (Code == 2) Data = 8'b00000100; else
      if (Code == 3) Data = 8'b00001000; else
      if (Code == 4) Data = 8'b00010000; else
      if (Code == 5) Data = 8'b00100000; else
      if (Code == 6) Data = 8'b01000000; else
      if (Code == 7) Data = 8'b10000000; else
                     Data = 8'bx;
    end
```

```
/* Alternative description is given below
always @  (Code)
 case (Code)
   0         : Data = 8'b00000001;
   1         : Data = 8'b00000010;
   2         : Data = 8'b00000100;
   3         : Data = 8'b00001000;
   4         : Data = 8'b00010000;
   5         : Data = 8'b00100000;
   6         : Data = 8'b01000000;
   7         : Data = 8'b10000000;
   default   : Data = 8'bx;
 endcase
*/
endmodule
```

# Synthesis Result:



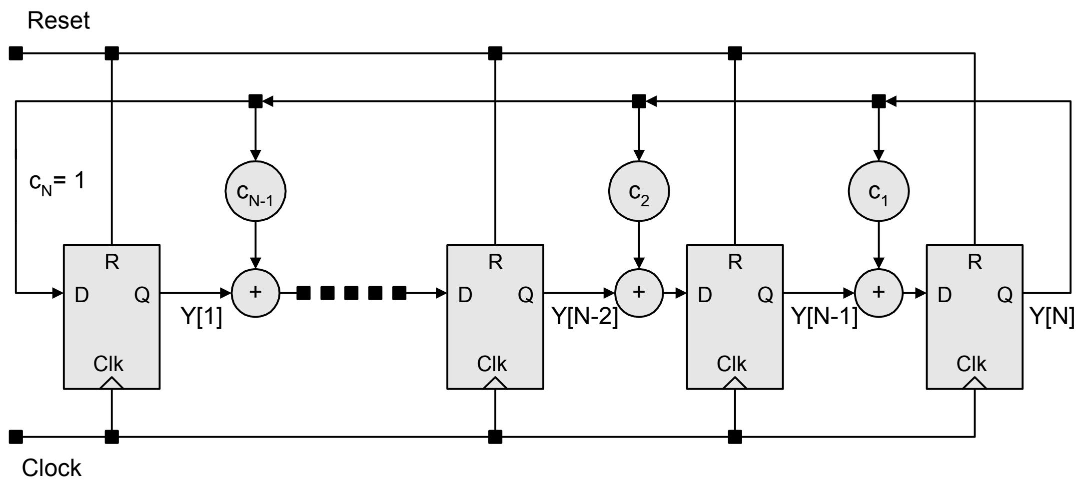
# Example 5.25: Seven Segment Display




```
module Seven_Seg_Display (Display, BCD);
  output   [6: 0]Display;
  input    [3: 0]BCD;
  reg      [6: 0]Display;
  //                          abc_defg
  parameter    BLANK    = 7'b111_1111;
  parameter    ZERO     = 7'b000_0001;        // h01
  parameter    ONE      = 7'b100_1111;        // h4f
  parameter    TWO      = 7'b001_0010;        // h12
  parameter    THREE    = 7'b000_0110;        // h06
```

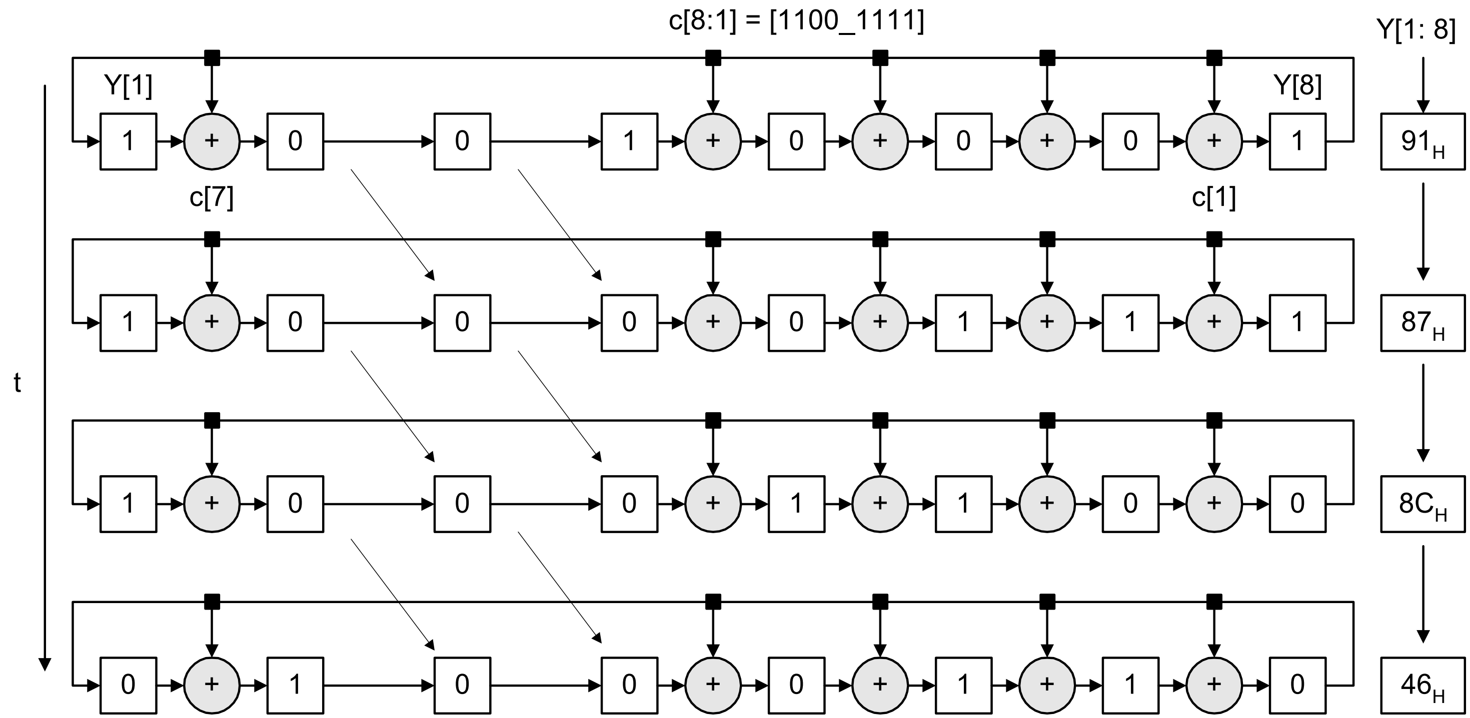
```
parameter   FOUR    = 7'b100_1100;        // h4c
parameter   FIVE    = 7'b010_0100;        // h24
parameter   SIX     = 7'b010_0000;        // h20
parameter   SEVEN   = 7'b000_1111;        // h0f
parameter   EIGHT   = 7'b000_0000;        // h00
parameter   NINE    = 7'b000_0100;        // h04

always @ (BCD or)
  case (BCD)
     0:        Display = ZERO;
     1:        Display = ONE;
     2:        Display = TWO;
     3:        Display = THREE;
     4:        Display = FOUR;
     5:        Display = FIVE;
     6:        Display = SIX;
     7:        Display = SEVEN;
     8:        Display = EIGHT;
     9:        Display = NINE;
     default:  Display = BLANK;
  endcase
endmodule
```

# Example 5.26: LFSR (RTL – Dataflow)




```
module Auto_LFSR_RTL (Y, Clock, Reset);
  parameter                   Length = 8;
  parameter                   initial_state = 8'b1001_0001;   // 91h
  parameter    [1: Length]    Tap_Coefficient = 8'b_1111_1100;

  input                       Clock, Reset;
  output       [1: Length]    Y;
  reg          [1: Length]    Y;
```

```
always @  (posedge Clock)
  if (!Reset) Y <= initial_state;        // Active-low reset to initial state
    else begin
      Y[1] <= Y[8];
      Y[2] <= Tap_Coefficient[7] ? Y[1] ^ Y[8] : Y[1];
      Y[3] <= Tap_Coefficient[6] ? Y[2] ^ Y[8] : Y[2];
      Y[4] <= Tap_Coefficient[5] ? Y[3] ^ Y[8] : Y[3];
      Y[5] <= Tap_Coefficient[4] ? Y[4] ^ Y[8] : Y[4];
      Y[6] <= Tap_Coefficient[3] ? Y[5] ^ Y[8] : Y[5];
      Y[7] <= Tap_Coefficient[2] ? Y[6] ^ Y[8] : Y[6];
      Y[8] <= Tap_Coefficient[1] ? Y[7] ^ Y[8] : Y[7];

  end
endmodule
```


c[8:1] = [1100_1111]
Y[1: 8]
Y[1]
Y[8]
c[7]
c[1]
t
91H
87H
8CH
46H

# Example 5.27: LFSR (RTL – Algorithm)

```
module Auto_LFSR_ALGO (Y, Clock, Reset);
  parameter    Length = 8;
  parameter    initial_state = 8'b1001_0001;
  parameter    [1: Length] Tap_Coefficient = 8'b1111_1100;
  input        Clock, Reset;
  output       [1: Length] Y;
  integer      Cell_ptr;
  reg          [1: Length] Y;   // Redundant declaration for some compilers

  always @  (posedge Clock)
    begin
      if (Reset == 0) Y <= initial_state;            // Arbitrary initial state, 91h
      else begin for (Cell_ptr = 2; Cell_ptr <= Length; Cell_ptr = Cell_ptr +1)
        if (Tap_Coefficient [Length - Cell_ptr + 1] == 1)
          Y[Cell_ptr] <= Y[Cell_ptr -1]^ Y [Length];
        else
          Y[Cell_ptr] <= Y[Cell_ptr -1];
          Y[1] <= Y[Length];
      end
    end
endmodule
```

# Example 5.28: repeat Loop

```
...
word_address = 0;
repeat (memory_size)
  begin
    memory [ word_address] = 0;
    word_address = word_address + 1;
  end
...
```

# Example 5.29: for Loop

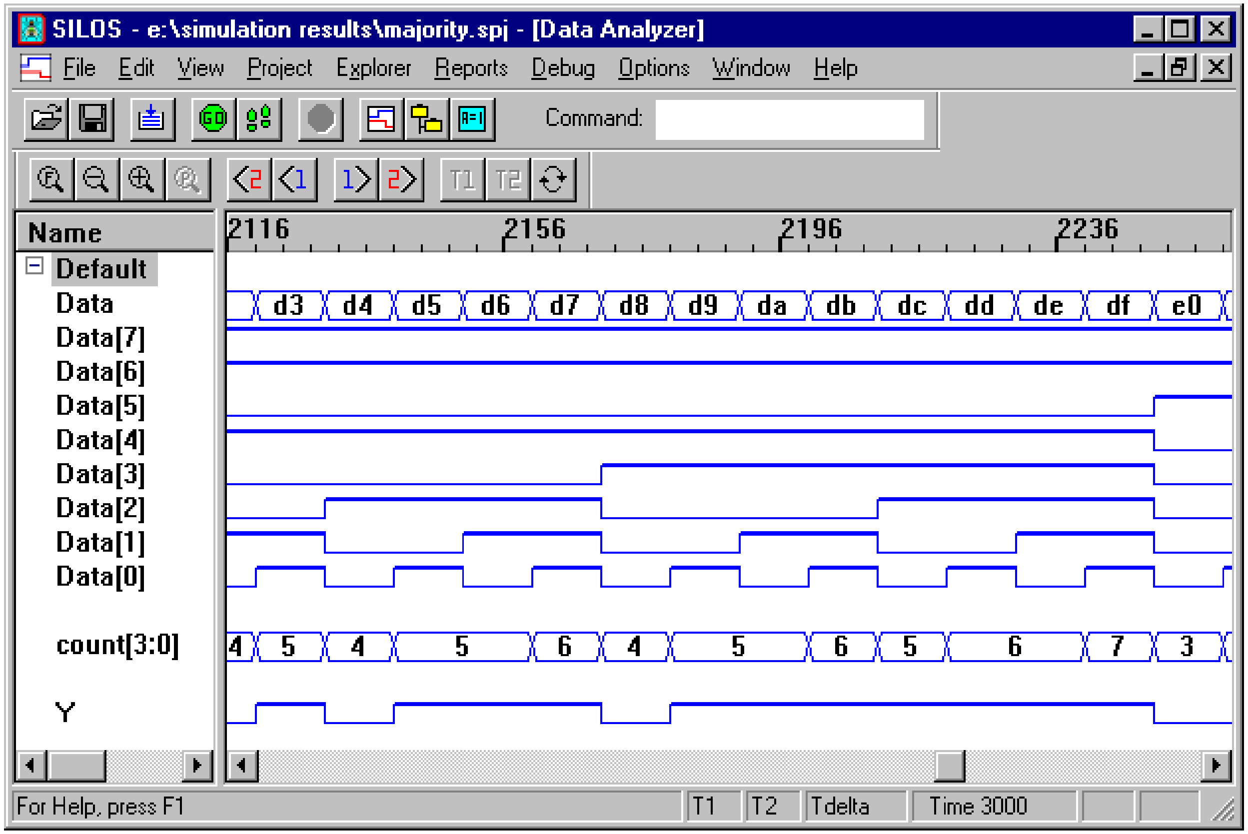
```
reg [15: 0] demo_register;
integer K;

…
for (K = 4; K; K = K - 1)
  begin
    demo_register [K + 10] = 0;
    demo_register [K + 2] = 1;
  end
…
```

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| x | 0 | 0 | 0 | 0 | x | x | x | x | 1 | 1 | 1 | 1 | x | x | x |

# Example 5.30: Majority Circuit

```
module Majority_4b (Y, A, B, C, D);
  input    A, B, C, D;
  output   Y;
  reg      Y;
  always @ (A or B or C or D) begin
    case ({A, B,C, D})
      7, 11, 13, 14, 15:      Y = 1;
      default                 Y = 0;
    endcase
  end
endmodule

module Majority (Y, Data);
  parameter    size = 8;
  parameter    max = 3;
  parameter    majority = 5;
  input        [size-1: 0] Data;
  output                   Y;
  reg                      Y;
  reg          [max-1: 0]  count;
  integer                  k;
```

```
  always @ (Data) begin
    count = 0;
    for (k = 0; k < size; k = k + 1) begin
      if (Data[k] == 1) count = count + 1;
    end
    Y = (count >= majority);
  end
endmodule
```


SILOS - e:\simulation results\majority.spj - [Data Analyzer]
File Edit View Project Explorer Reports Debug Options Window Help
Command:
Name
2116
2156
2196
2236
Default
Data
Data[7]
Data[6]
Data[5]
Data[4]
Data[3]
Data[2]
Data[1]
Data[0]
count[3:0]
Y
d3 d4 d5 d6 d7 d8 d9 da db dc dd de df e0
4 5 4 5 6 4 5 6 5 6 7 3
For Help, press F1
T1
T2
Tdelta
Time 3000

# Example 5.31 Parameterized Model of LFSR

```
module Auto_LFSR_Param (Y, Clock, Reset);
  parameter                Length = 8;
  parameter                initial_state = 8'b1001_0001;   // Arbitrary initial state
  parameter [1: Length]    Tap_Coefficient = 8'b1111_1100;

  input                    Clock, Reset;
  output   [1: Length]     Y;
  reg      [1: Length]     Y;
  integer                  k;

  always @  (posedge Clock)
    if (!Reset) Y <= initial_state;
      else begin
        for (k = 2; k <= Length; k = k + 1)
          Y[k] <= Tap_Coefficient[Length-k+1] ? Y[k-1] ^ Y[Length] : Y[k-1];
          Y[1] <= Y[Length];
      end
 endmodule
```

# Example 5.32: Ones Counter

```
begin: count_of_1s          // count_of_1s declares a named block of statements
  reg [7: 0] temp_reg;

  count = 0;
  temp_reg = reg_a; // load a data word
  while (temp_reg)
    begin
      if (temp_reg[0]) count = count + 1;
      temp_reg = temp_reg >> 1;
    end
end
```

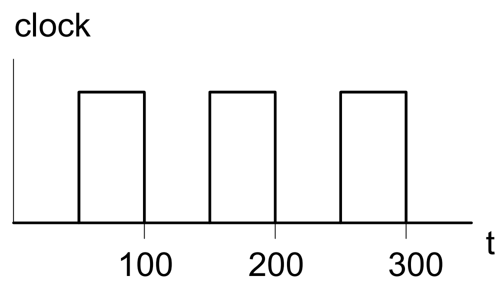
Alternative Description:

```
begin: count_of_1s
  reg [7: 0] temp_reg;

  count = 0;
  temp_reg = reg_a; // load a data word
  while (temp_reg)
    begin
      count = count + temp_reg[0];
      temp_reg = temp_reg >> 1;
    end
  end
```

Note: Verilog 2001 includes arithmetic shift operators (See Appendix I)

# Example 5.32: Clock Generator

```
parameter half_cycle = 50;

initial
  begin: clock_loop  // Note: clock_loop is a named block of statements
    clock = 0;
    forever
      begin
        #half_cycle clock = 1;
        #half_cycle clock = 0;
      end
  end

initial
  #350 disable clock_loop;
```

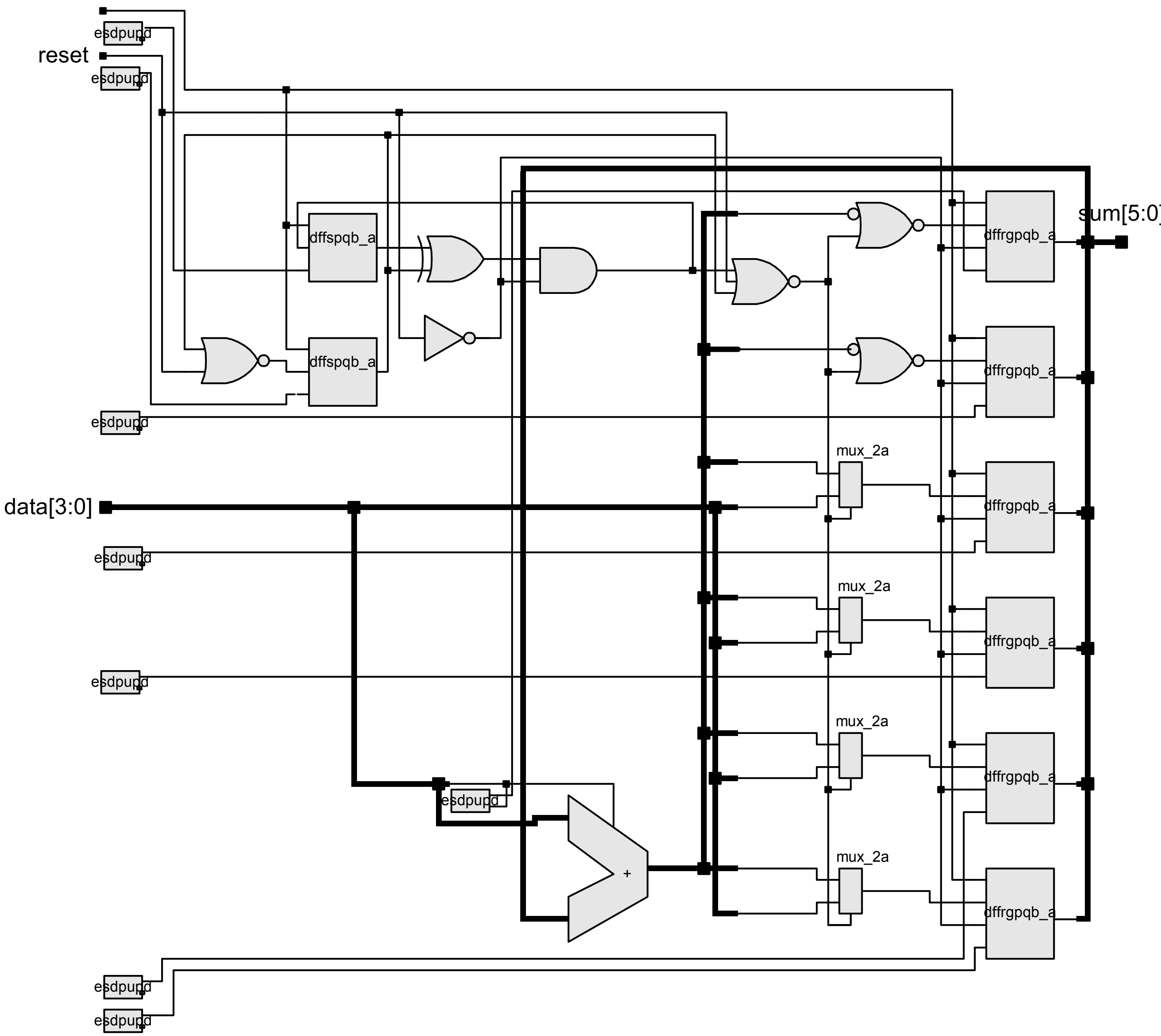

clock
t
100
200
300

# Example 5.34

```
module find_first_one (index_value, A_word, trigger);
  output   [3: 0]     index_value;
  input    [15: 0]    A_word;
  input               trigger;
  reg      [3: 0]     index_value;

  always @  (trigger)
    begin: search_for_1
      index_value = 0;
      for (index_value = 0; index_value < 15; index_value = index_value + 1)
        if (A_word[index_value] == 1) disable search_for_1;
    end
endmodule
```

# Example 5.35: Multi-Cycle Operations (4-Cycle Adder)

```
module add_4cycle (sum, data, clk, reset);
  output        [5: 0]      sum;
  input         [3: 0]      data;
  input                     clk, reset;
  reg           [5: 0]      sum;      // Redundant for some compilers

always @ (posedge clk) begin:  add_loop
    if (reset) disable add_loop;                          else sum <= data;
      @ (posedge clk) if (reset) disable add_loop;   else sum <= sum + data;
        @ (posedge clk) if (reset) disable add_loop;     else sum <= sum + data;
          @ (posedge clk) if (reset) disable add_loop;       else sum <= sum + data;
   end
endmodule
```

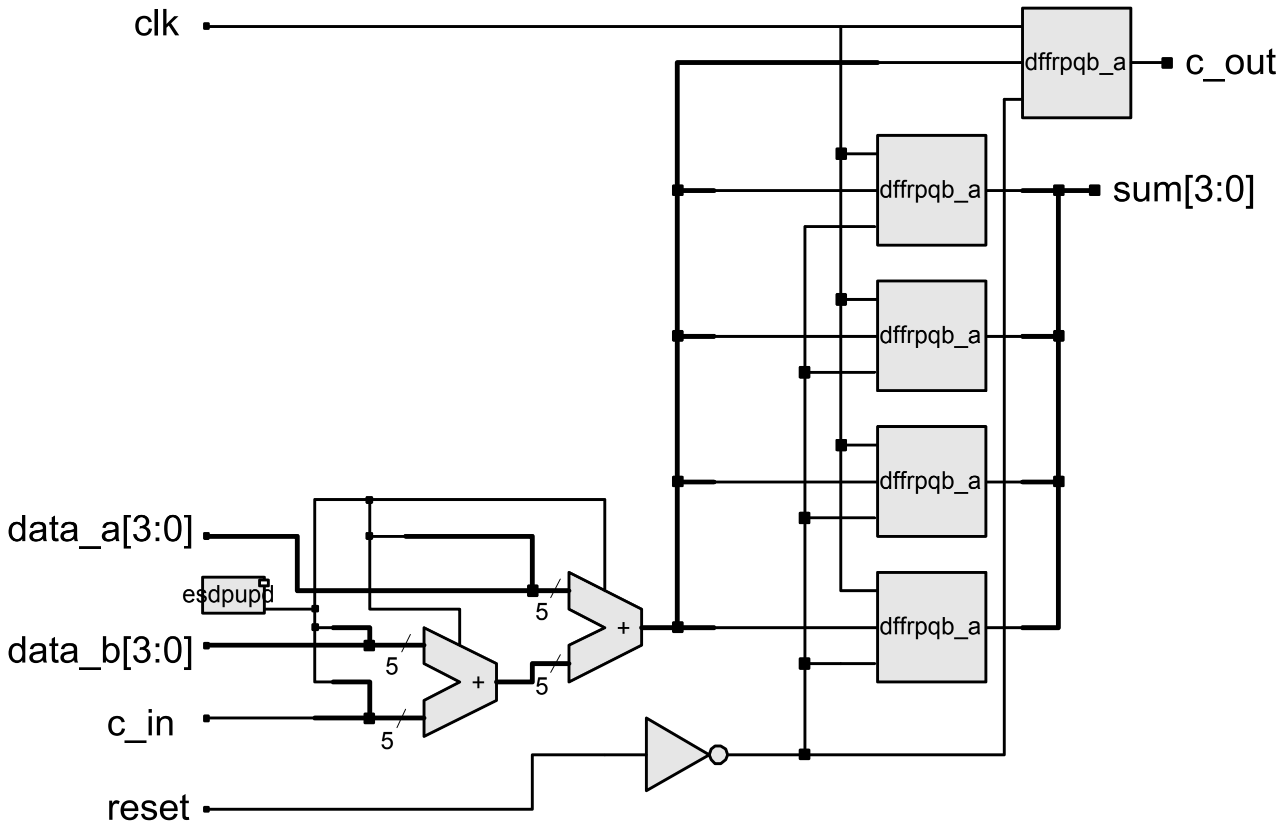


reset
esdpupd
dffspqb_a
dffspqb_a
dffrgpqb_a
sum[5:0]
mux_2a
data[3:0]
+

# Example 5.36: Task (Adder)

```
module adder_task (c_out, sum, clk, reset, c_in, data_a, data_b, clk);
  output      [3: 0]     sum;
  output                 c_out;
  input       [3: 0]     data_a, data_b;
  input                  clk, reset;
  input                  c_in;

  reg         [3: 0]     sum;// Redundant for some compilers
  reg                    c_out;
  reg         [3: 0]     acc;

  always @  (posedge clk or posedge reset)
    if (reset)     {c_out, sum} <= 0; else
      add_values (c_out, sum, data_a, data_b, c_in);
```

```
task add_values;
  output          c_out;
  output  [3: 0]  sum;
  input   [3: 0]  data_a, data_b;
  input           c_in;

  reg             sum;
  reg             c_out;

  begin
    {c_out, sum} <= data_a + (data_b + c_in);
  end
 endtask
endmodule
```

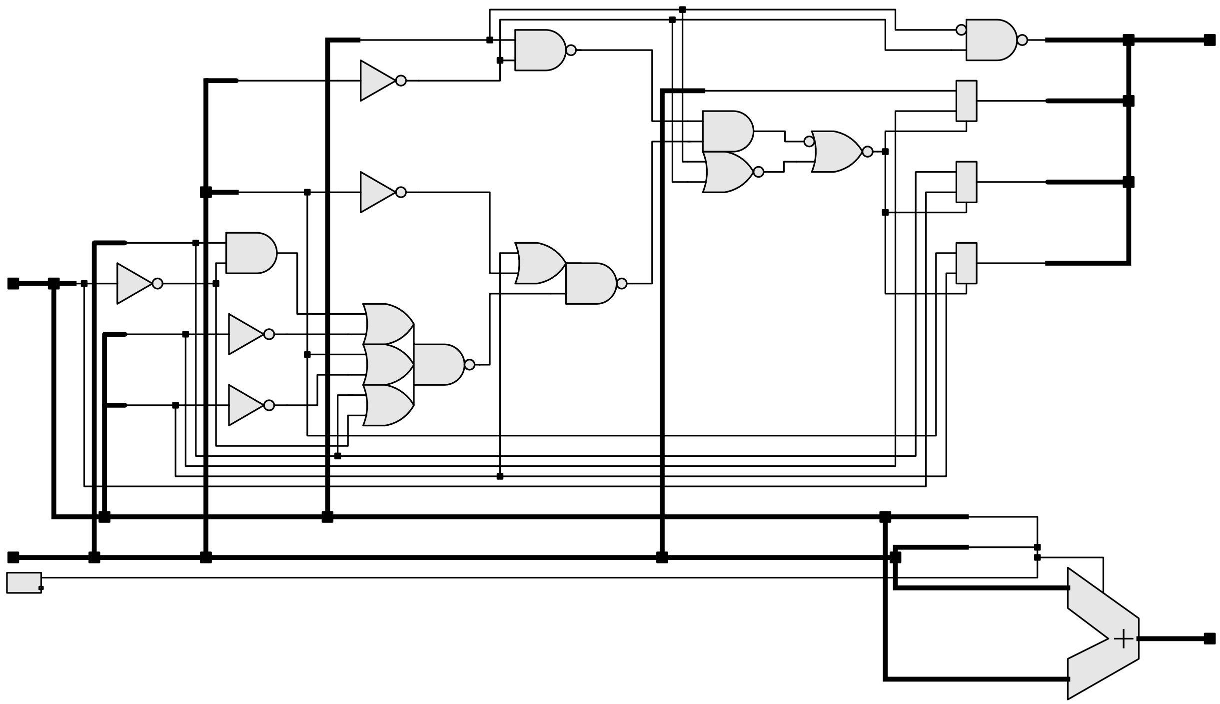

clk
dffrpqb_a
c_out
dffrpqb_a
sum[3:0]
dffrpqb_a
dffrpqb_a
data_a[3:0]
esdpupd
5
5
+
dffrpqb_a
data_b[3:0]
5
+
5
c_in
5
reset

# Example 5.37: Function (Word Aligner)

```
module word_aligner (word_out, word_in);
  output        [7: 0]      word_out;
  input         [7: 0]      word_in;

  assign word_out = aligned_word(word_in);

  function      [7: 0]      aligned_word;
    input       [7: 0]      word_in;
    begin
      aligned_word = word_in;
      if (aligned_word != 0)
        while (aligned_word[7] == 0) aligned_word = aligned_word << 1;
    end
  endfunction
endmodule
```

# Example 5.38: Arithmetic Unit

```
module arithmetic_unit (result_1, result_2, operand_1, operand_2);
  output      [4: 0] result_1;
  output      [3: 0] result_2;
  input       [3: 0] operand_1, operand_2;

  assign result_1 = sum_of_operands (operand_1, operand_2);
  assign result_2 = largest_operand (operand_1, operand_2);

  function [4: 0] sum_of_operands;
    input [3: 0] operand_1, operand_2;

    sum_of_operands = operand_1 + operand_2;
  endfunction

  function [3: 0] largest_operand;
    input [3: 0] operand_1, operand_2;

    largest_operand = (operand_1 >= operand_2) ? operand_1 : operand_2;
  endfunction
endmodule
```



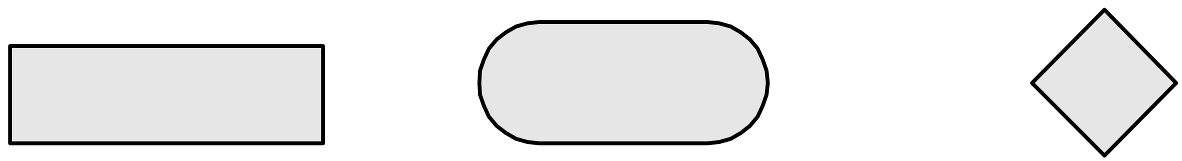
[1]

operand_2[3:0]

# Algorithmic State Machine (ASM) Chart

- STGs do not directly display the evolution of states resulting from an input
- ASM charts reveal the sequential steps of a machine's activity
- Focus on machine's activity, rather than contents of registers
- ASM chart elements
    1. state box
    2. decision box
    3. conditional box
- Clock governs transitions between states
- Linked ASM charts describe complex machines
- ASM charts represent Mealy and Moore machines

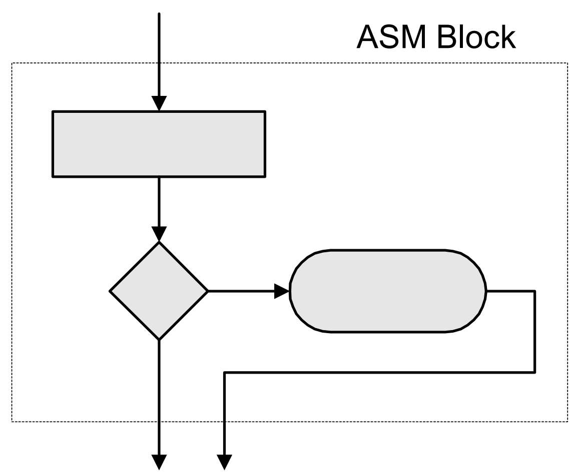
# ASM Charts (Cont.)



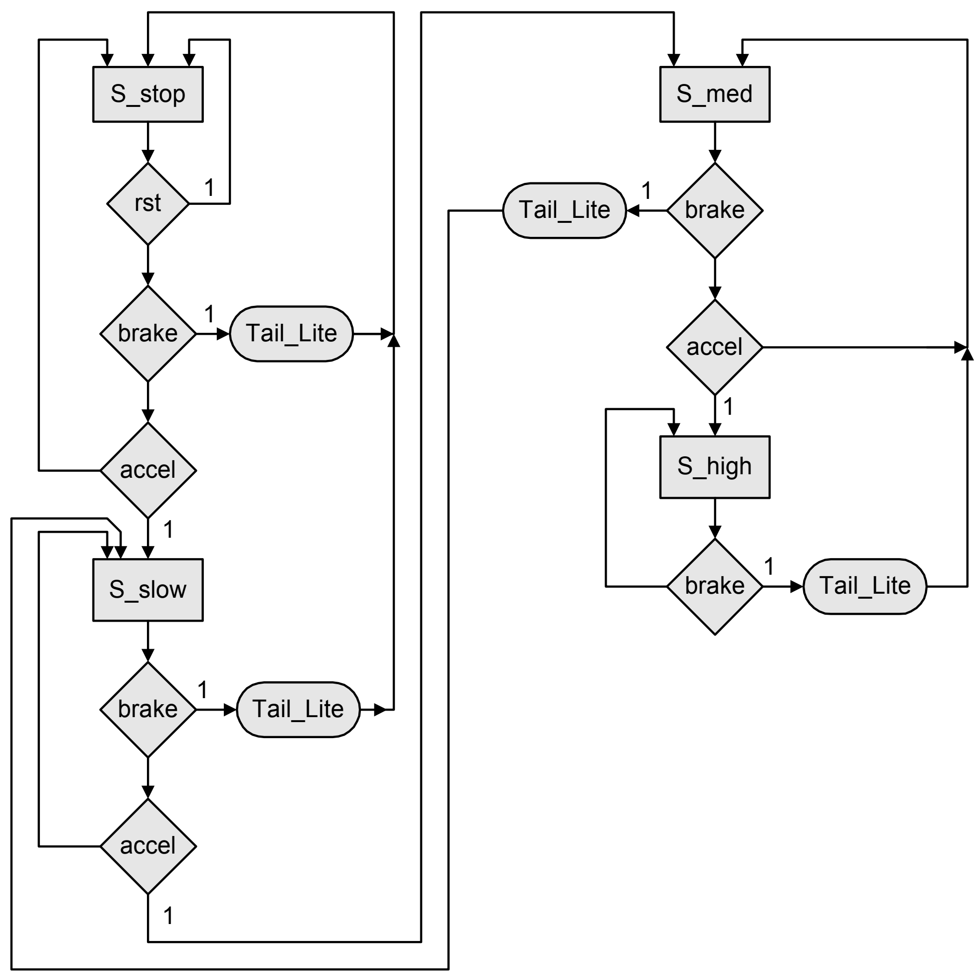
State Box

Conditional Output or Register Operation Box

Decision Box



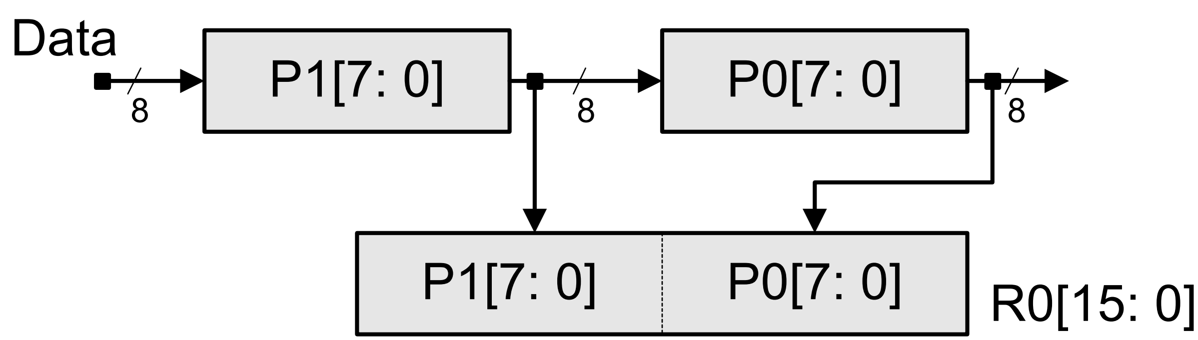
# Example 5.39 Tail Light Controller

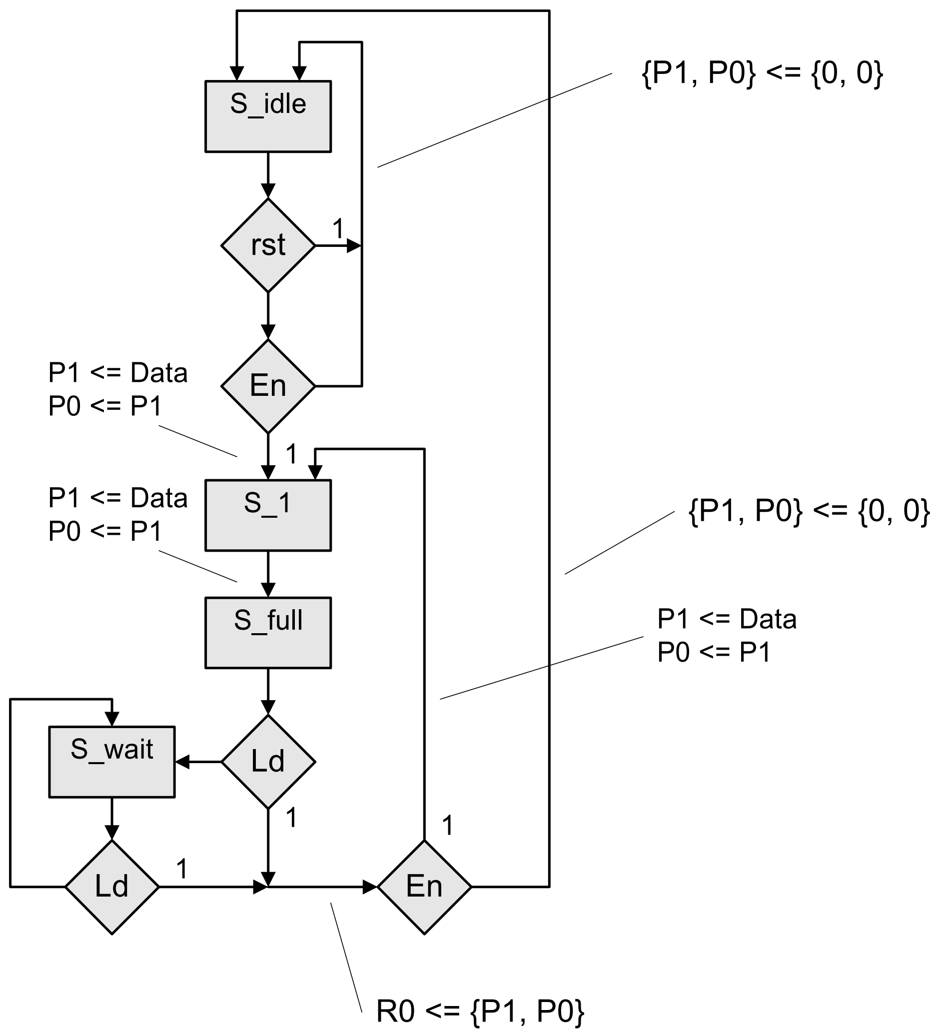


# ASMD Chart

- Form an ASMD (Algorithmic State Machine and datapath) chart by annotating each of its paths to indicate the concurrent register operations that occur in the associated datapath unit when the state of the controller makes a transition along the path
- Clarify a design of a sequential machine by separating the design of its datapath from the design of the controller
- ASMD chart maintains a clear relationship between a datapath and its controller
- Annotate path with concurrent register operations
- Outputs of the controller control the datapath

# Example 5.39 Two-Stage Pipeline



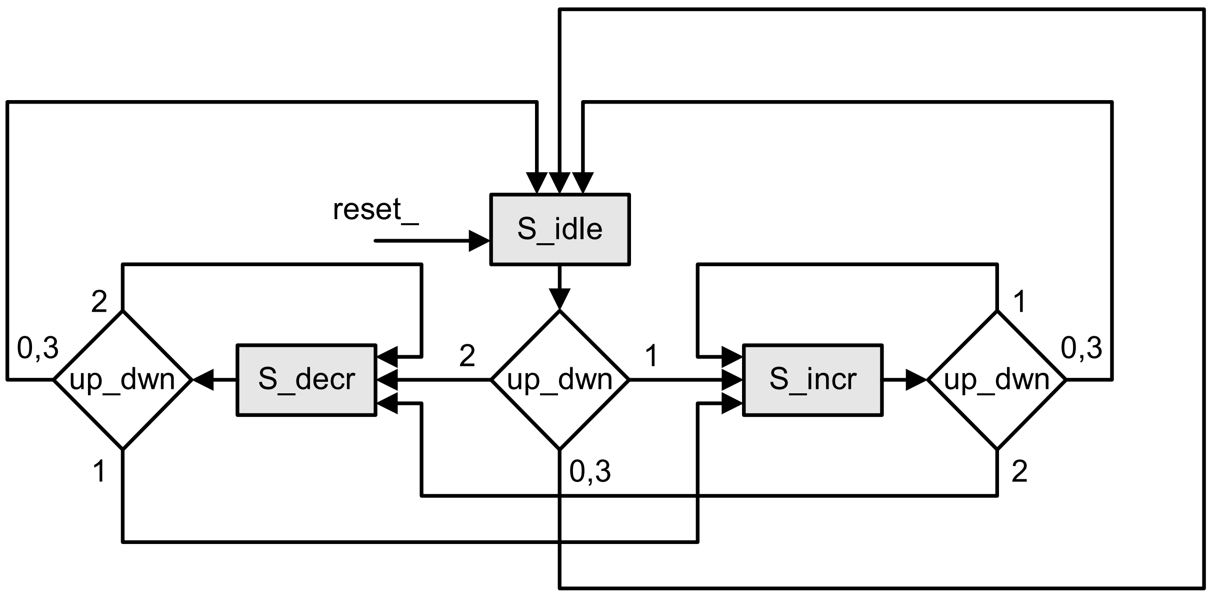


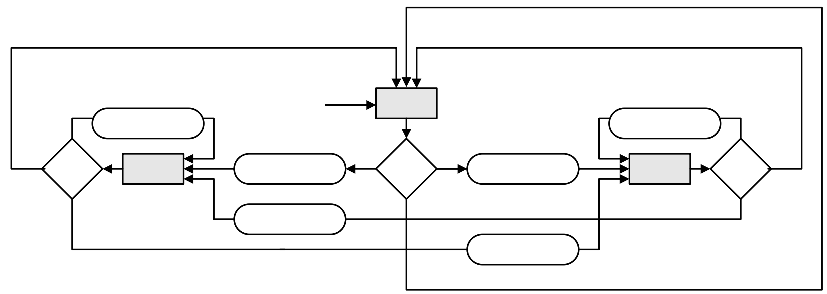

See Problem 24

# Datapath Controller Design

- Specify register operations for the datapath
- Define the ASM chart of the controller (PI and feedback from datapath)
- Annotate the arcs of the ASM chart with the datapath operations associated with the state transitions of the controller
- Annotate the state of the controller with unconditional output signals
- Include conditional boxes for the signals generated by the controller to control the datapath.
- Verify the controller
- Verify the datapath
- Verify the integrated units

# Example 5.40 Counters



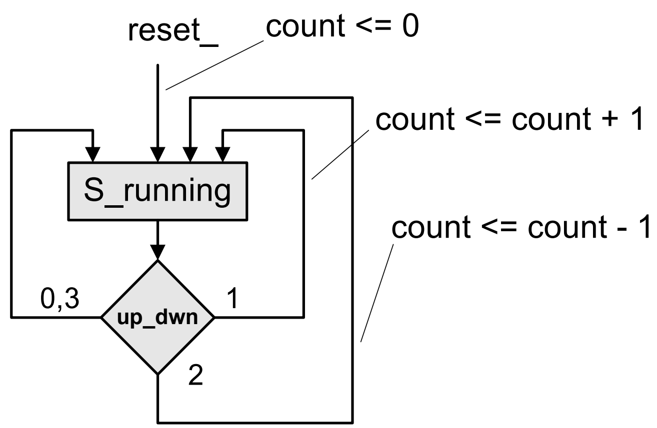


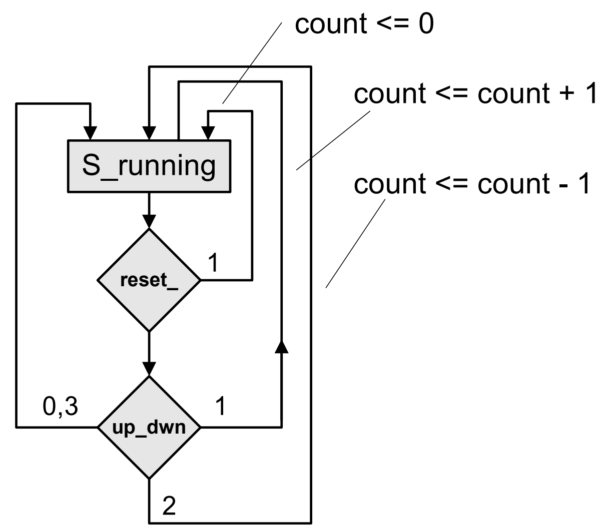
2

0,3

up_

1


reset_
count <= 0
count <= count + 1
count <= count - 1
S_running
0,3
up_dwn
1
2


count <= 0
count <= count + 1
count <= count - 1
S_running
reset_
1
0,3
up_dwn
1
2

```
module Up_Down_Implicit1 (count, up_dwn, clock, reset_);

  output  [2: 0]      count;
  input   [1: 0]      up_dwn;
  input           clock, reset_;

  reg [2: 0] count;

  always @  (negedge clock or negedge reset_)
    if (reset_ == 1)                    count <= 3'b0; else
      if (up_dwn == 2'b00 || up_dwn == 2'b11) count <= count; else
        if (up_dwn == 2'b01)                    count <= count + 1; else
          if (up_dwn == 2'b10)            count <= count –1;

endmodule
```

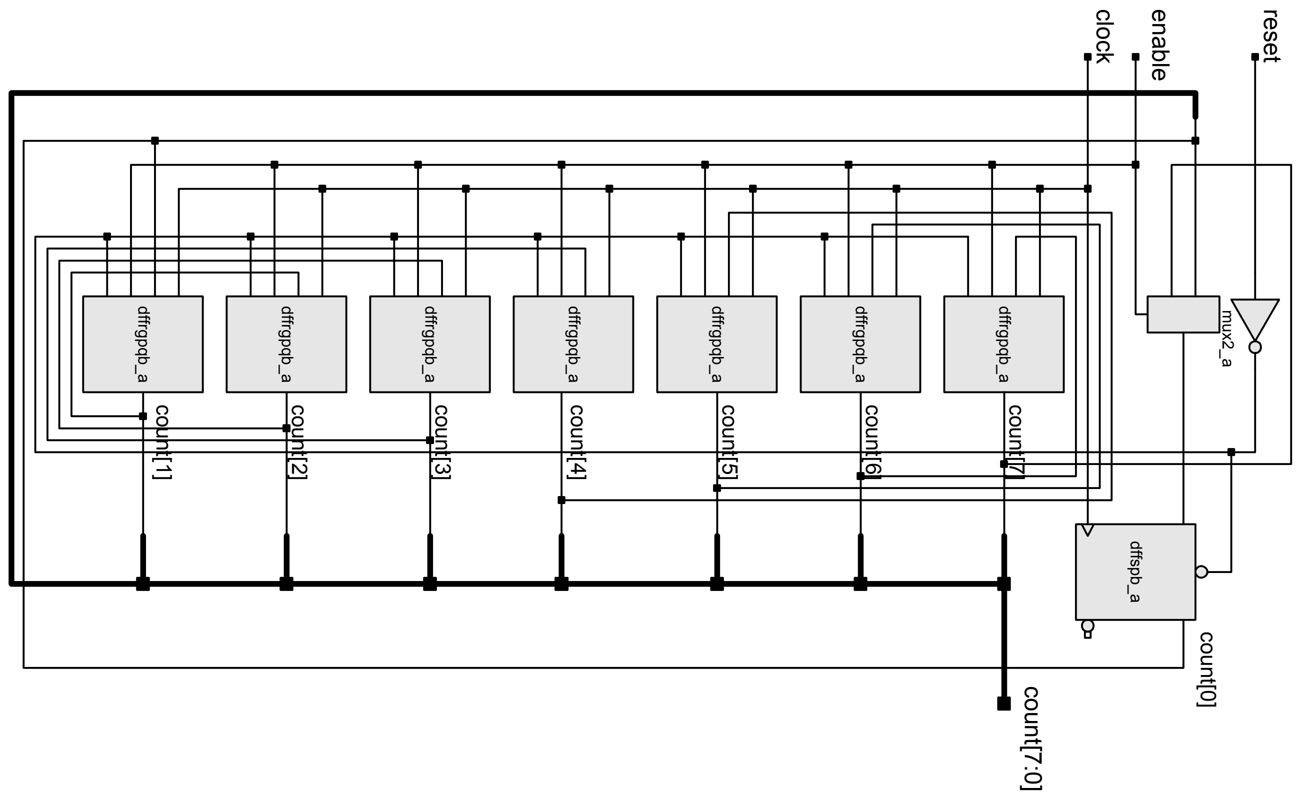
# Example 5.41 Ring Counter

count [7:0]

t

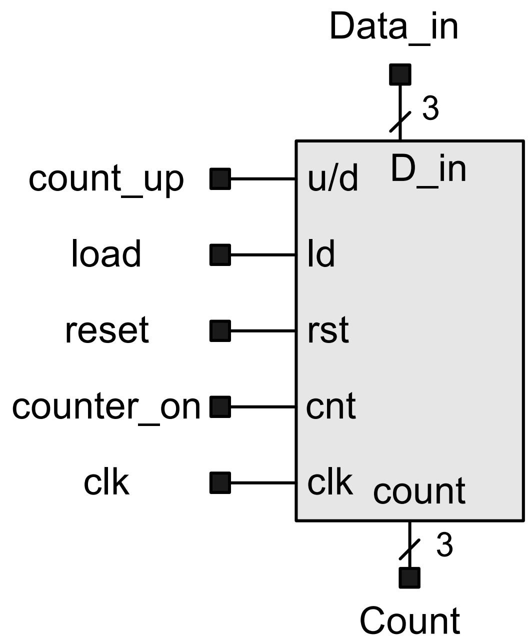
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

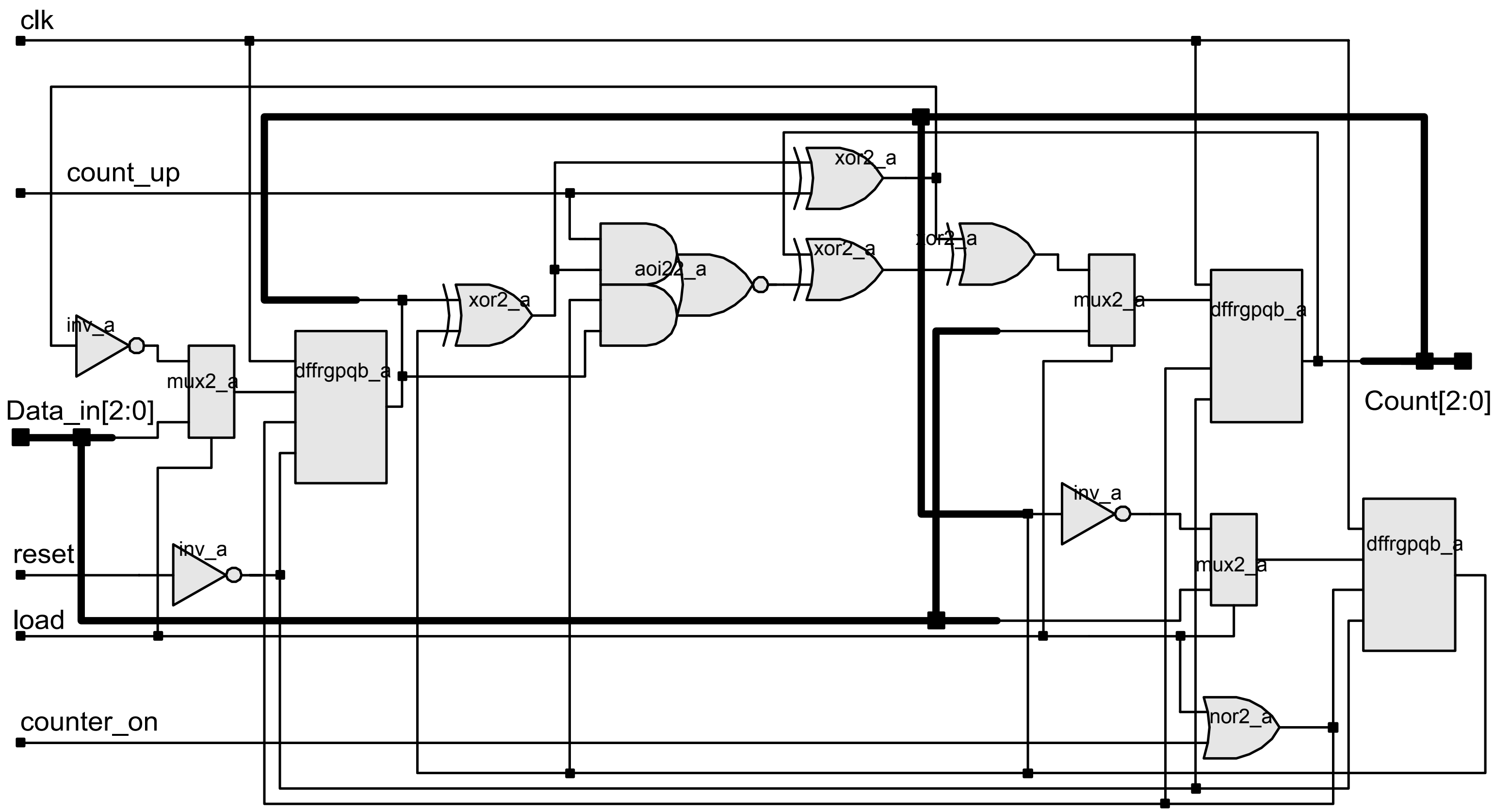
```
module ring_counter (count, enable, clock, reset);
 output          [7: 0]       count;
 input                        enable, reset, clock;
 reg             [7: 0]       count;

 always @  (posedge reset or posedge clock)
  if (reset == 1'b1)  count <= 8'b0000_0001; else
   if (enable == 1'b1) count <= {count[6: 0], count[7]};
    // Concatenation operator
endmodule
```


clock
enable
reset
dffrgpqb_a
dffrgpqb_a
dffrgpqb_a
dffrgpqb_a
dffrgpqb_a
dffrgpqb_a
dffrgpqb_a
mux2_a
dffspb_a
count[1]
count[2]
count[3]
count[4]
count[5]
count[6]
count[7]
count[0]
count[7:0]

# Example 5.423-Bit Up_Down Counter

```
module up_down_counter (Count, Data_in, load, count_up, counter_on, clk, reset);
  output        [2: 0]      Count;
  input                     load, count_up, counter_on, clk, reset,;
  input         [2: 0]      Data_in;
  reg           [2: 0]      Count;

  always @  (posedge reset or posedge clk)
    if (reset == 1'b1) Count = 3'b0; else
      if (load == 1'b1) Count = Data_in; else
        if (counter_on == 1'b1) begin
          if (count_up   == 1'b1) Count = Count +1;
            else Count = Count –1;
      end
endmodule
```


Data_in
3
D_in
count_up
u/d
load
ld
reset
rst
counter_on
cnt
clk
clk
count
3
Count

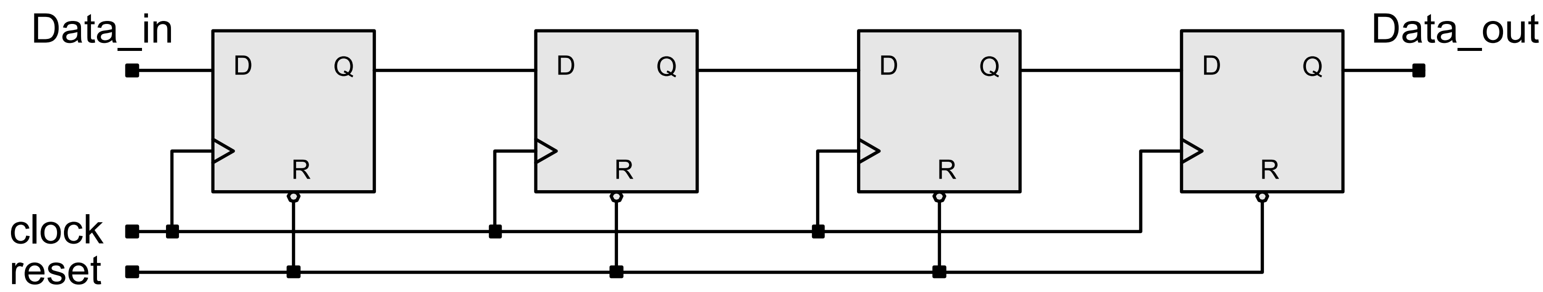



## See Appendix H for Flip-Flop types

# Example 5.43: Shift Register

```
module Shift_reg4 (Data_out, Data_in, clock, reset);
  output              Data_out;
  input               Data_in, clock, reset;
  reg     [3: 0]      Data_reg;

  assign  Data_out = Data_reg[0];

  always @  (negedge reset or posedge clock)
    begin
      if (reset == 1'b0)      Data_reg <= 4'b0;
      else                    Data_reg <= {Data_in, Data_reg[3:1]};
    end
endmodule
```

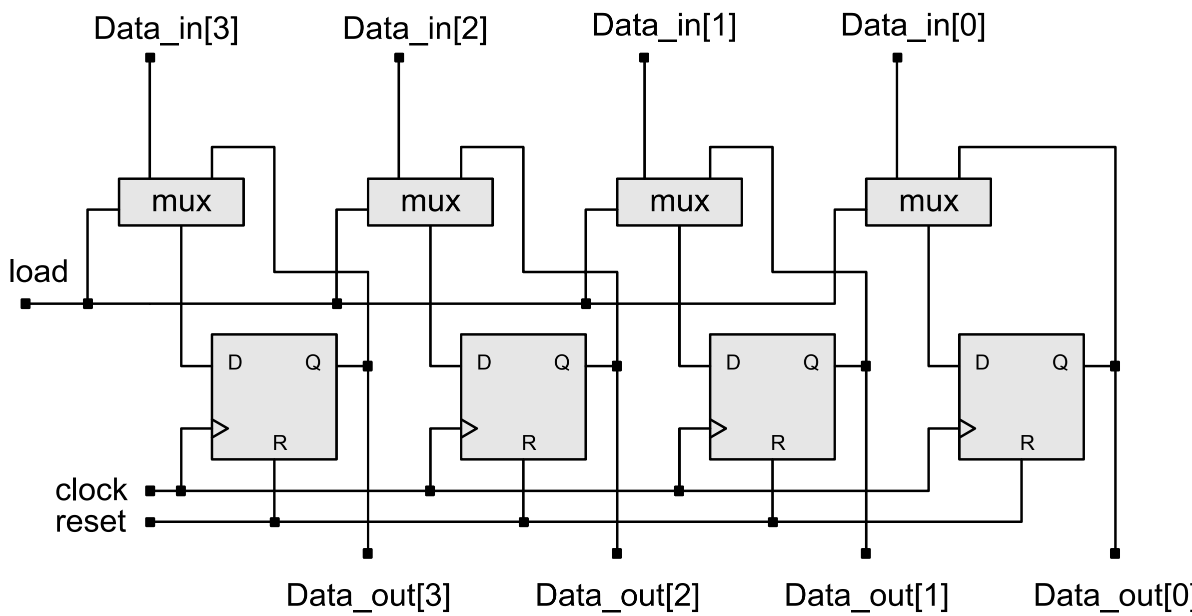
## Synthesis Result:



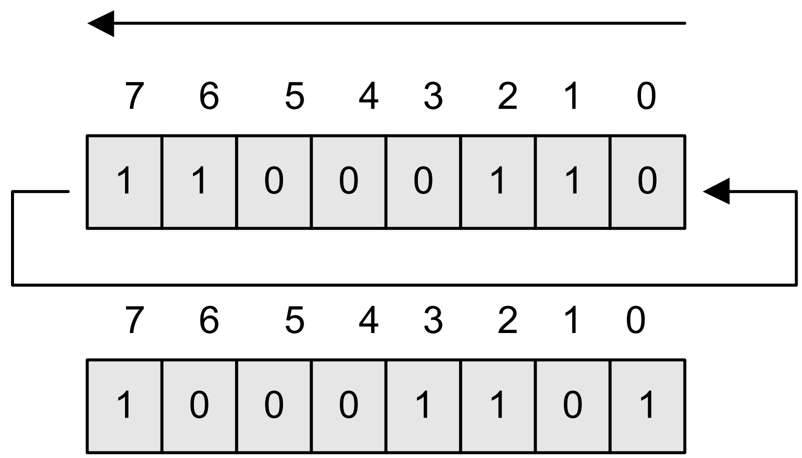
# Example 5.44 Parallel Load Shift Register

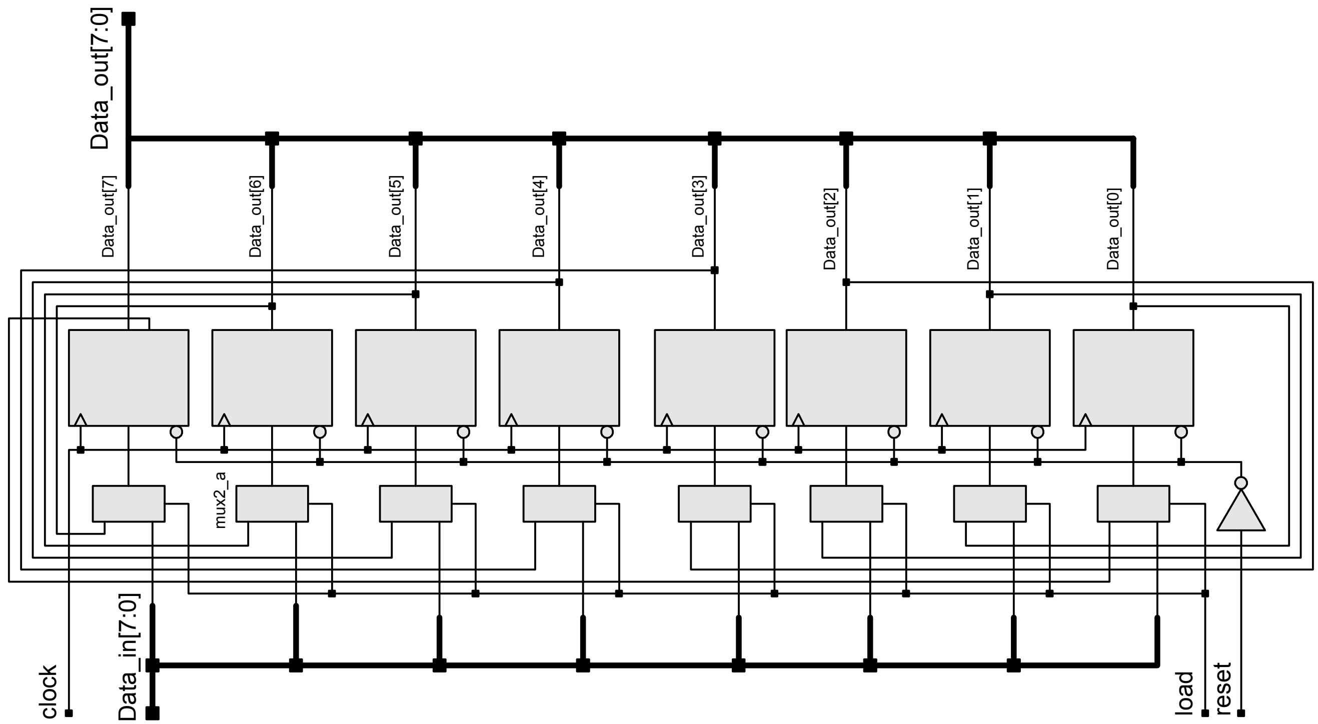
```
module Par_load_reg4 (Data_out, Data_in, load, clock, reset);
  input    [3: 0]     Data_in;
  input               load, clock, reset;
  output   [3: 0]     Data_out;      // Port size
  reg                 Data_out;      // Data type

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

## Synthesis Result



# Example 5.45: Barrel Shifter

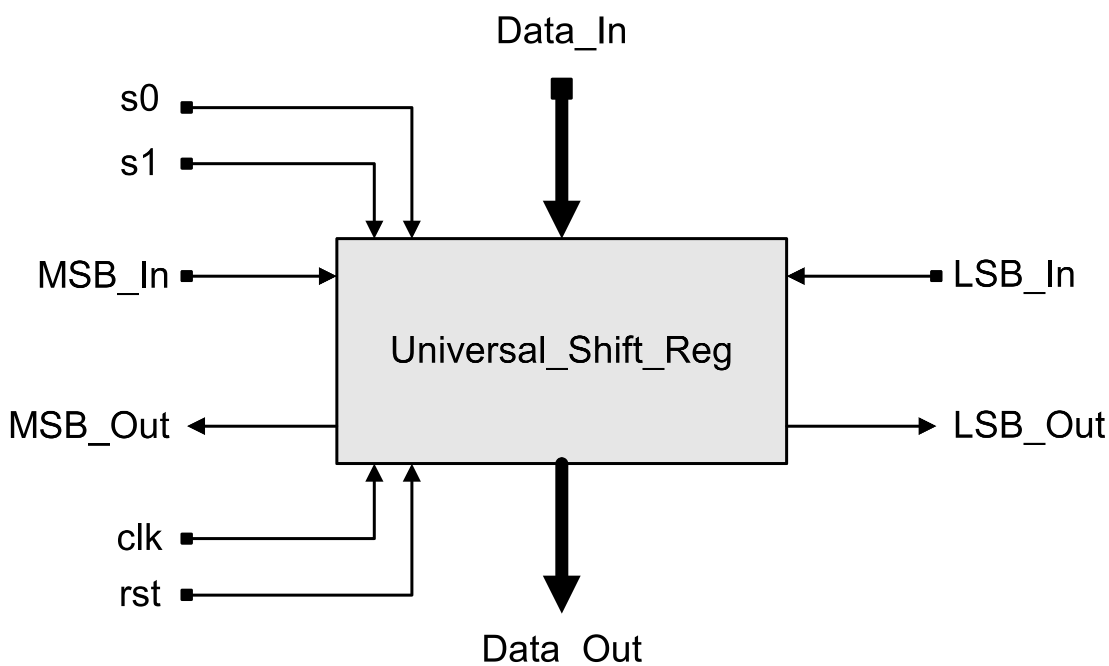




Data_out[7:0]
Data_out[7]
Data_out[6]
Data_out[5]
Data_out[4]
Data_out[3]
Data_out[2]
Data_out[1]
Data_out[0]
mux2_a
clock
Data_in[7:0]
load
reset

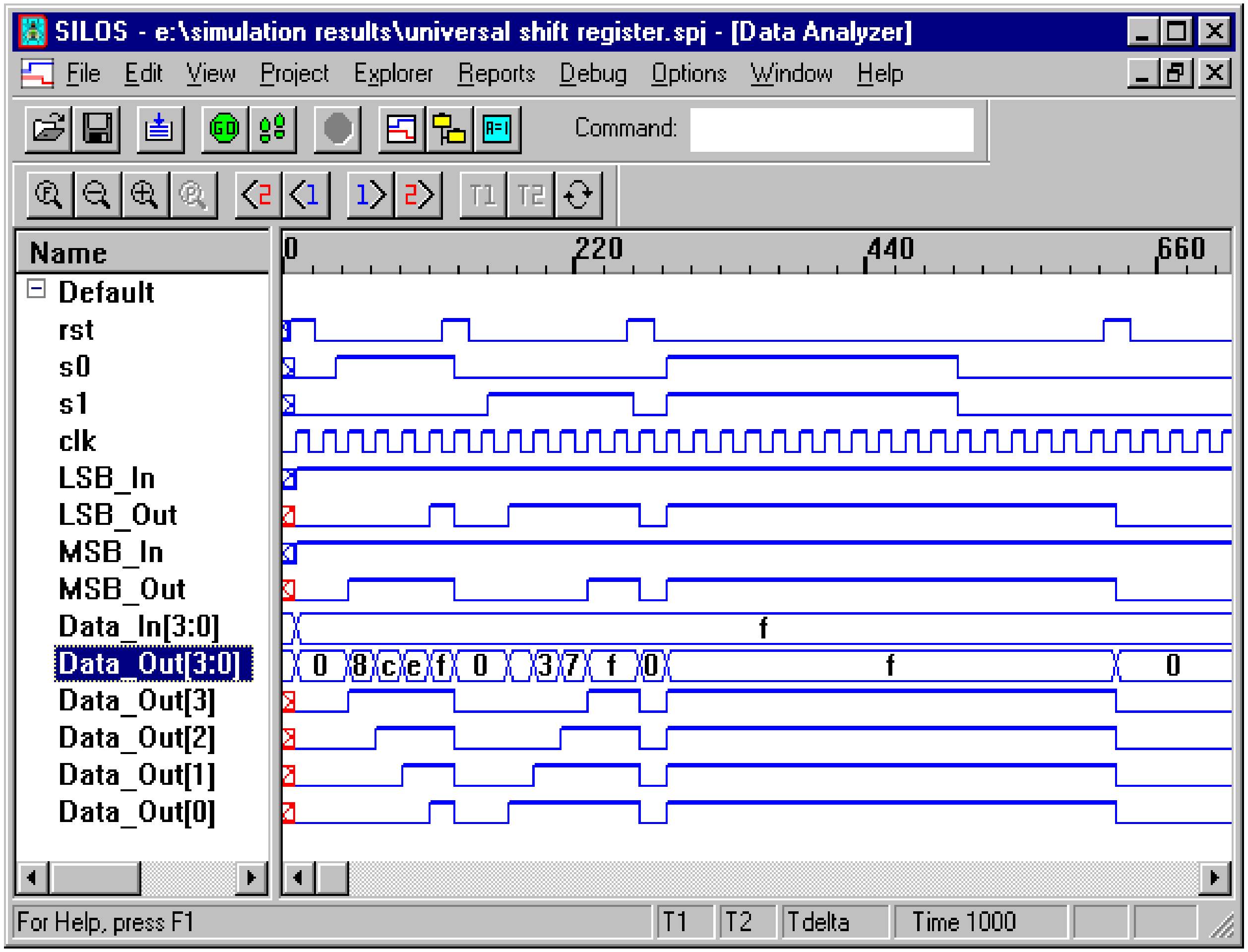
```
module barrel_shifter (Data_out, Data_in, load, clock, reset);
  output   [7: 0]      Data_out;
  input    [7: 0]      Data_in;
  input                load, clock, reset;
  reg      [7: 0]      Data_out;

  always @  (posedge reset or posedge clock)
    begin
      if (reset == 1'b1)         Data_out <= 8'b0;
      else if (load == 1'b1)     Data_out <= Data_in;
      else                       Data_out <= {Data_out[6: 0], Data_out[7]};
    end
endmodule
```

# Example 5.46: Universal Shift Register

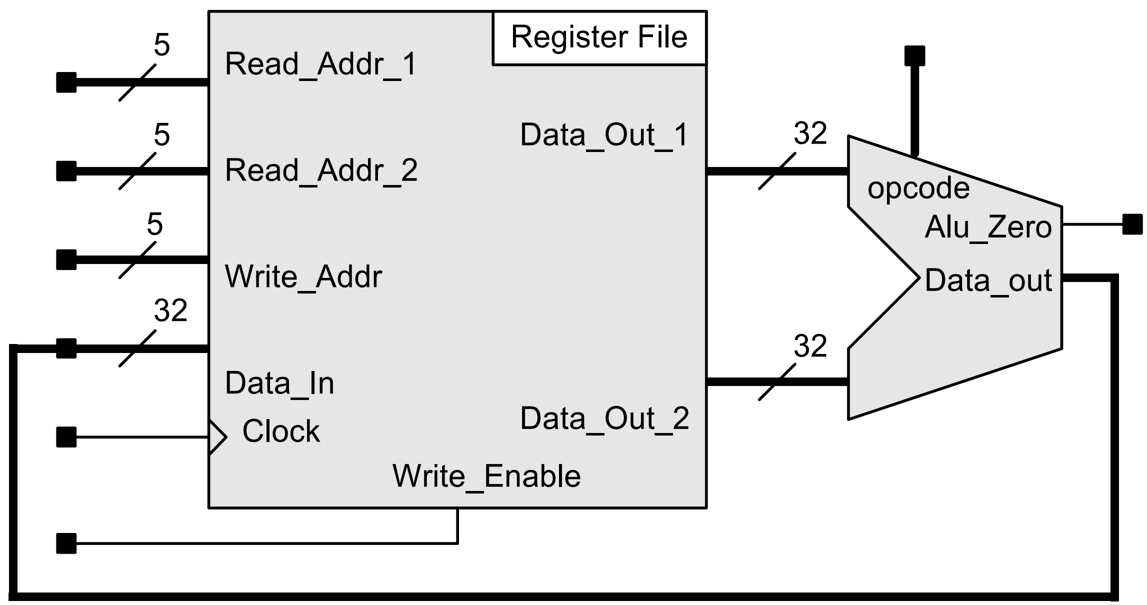


```
module Universal_Shift_Reg
  (Data_Out, MSB_Out, LSB_Out, Data_In, MSB_In, LSB_In, s1, s0, clk, rst);
 output   [3: 0]       Data_Out;
 output                MSB_Out, LSB_Out;
 input    [3: 0]       Data_In;
 input                 MSB_In, LSB_In;
 input                 s1, s0, clk, rst;
 reg      [3: 0]       Data_Out;

 assign MSB_Out = Data_Out[3];
 assign LSB_Out = Data_Out[0];

 always @ (posedge clk) begin
  if (rst) Data_Out <= 0;
  else  case ({s1, s0})
   0:      Data_Out <= Data_Out;                      // Hold
   1:      Data_Out <= {MSB_In, Data_Out[3:1]};  // Serial shift from MSB
   2:      Data_Out <= {Data_Out[2: 0], LSB_In};  // Serial shift from LSB
   3:      Data_Out <= Data_In;                       // Parallel Load
  endcase
 end
endmodule
```

## Simulation Results:



# Example 5.47: Register File

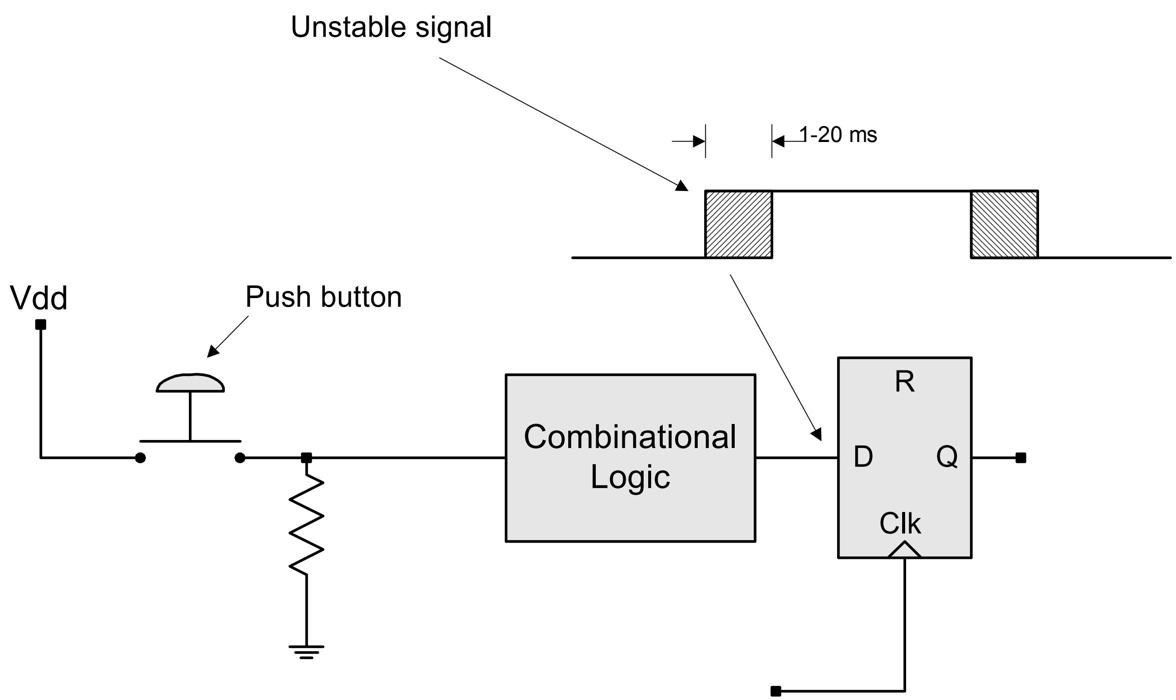



**module** Register_File (Data_Out_1, Data_Out_2, Data_in, Read_Addr_1, Read_Addr_2, Write_Addr, Write_Enable, Clock);
**output** [31: 0] Data_Out_1, Data_Out_2;
**input** [31: 0] Data_in;
**input** [4: 0] Read_Addr_1, Read_Addr_2, Write_Addr;

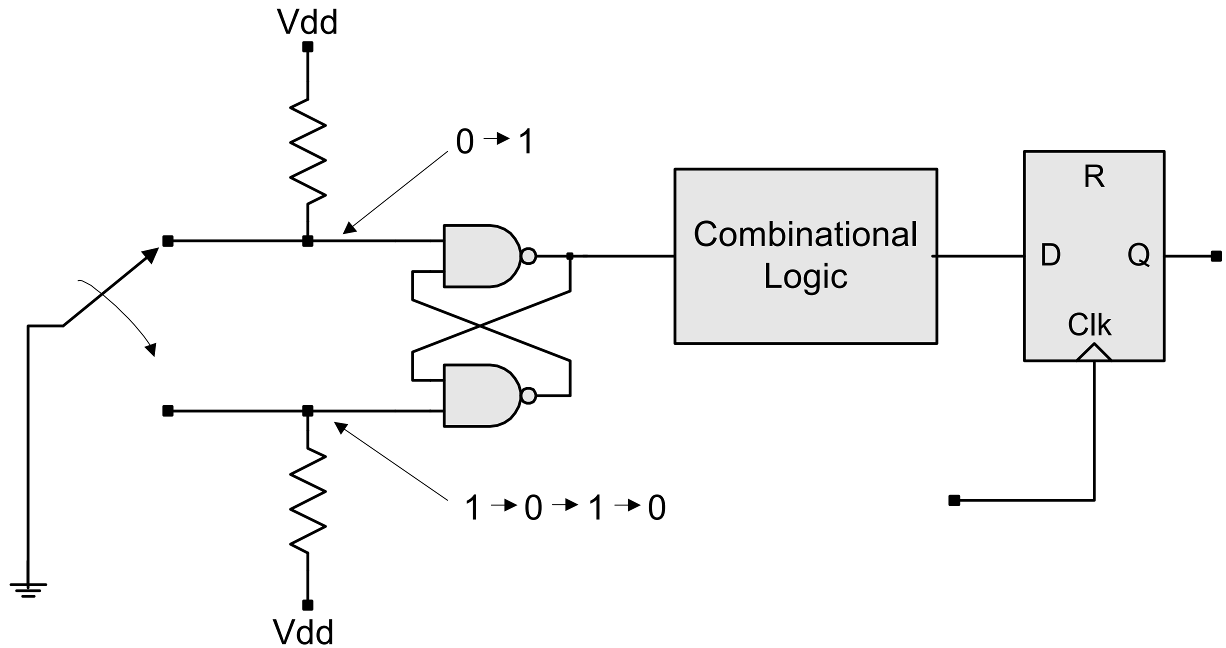
```
  input                 Write_Enable, Clock;
  reg       [31: 0]     Reg_File [31: 0];     // 32bit  x32 word memory declaration

  assign Data_Out_1 = Reg_File[Read_Addr_1];
  assign Data_Out_2 = Reg_File[Read_Addr_2];

  always @ (posedge Clock) begin
   if (Write_Enable) Reg_File [Write_Addr] <= Data_in;
  end
endmodule
```

# Metastability and Synchronizers

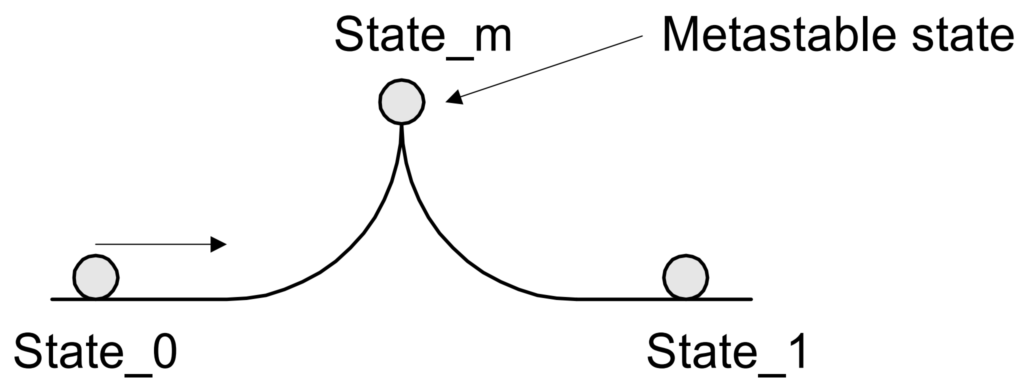
Push-button device with closure bounce:



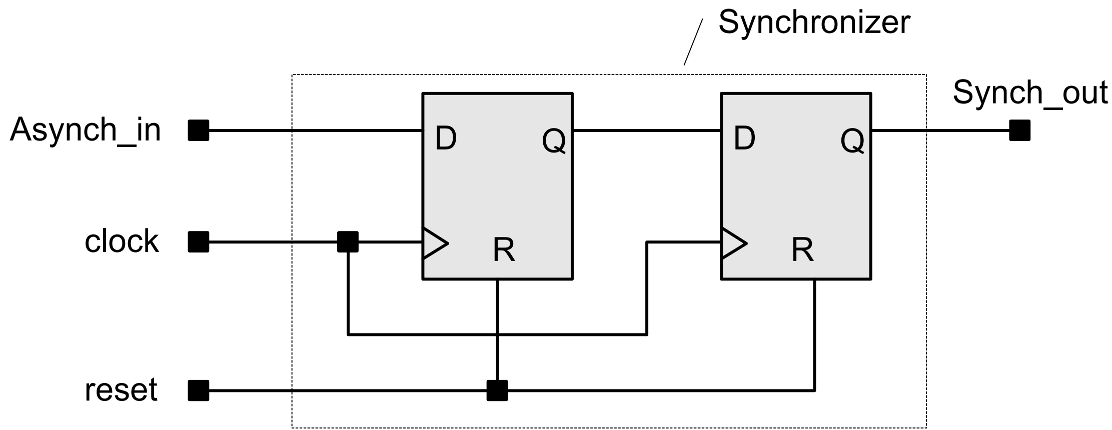
# Nand latch Circuit



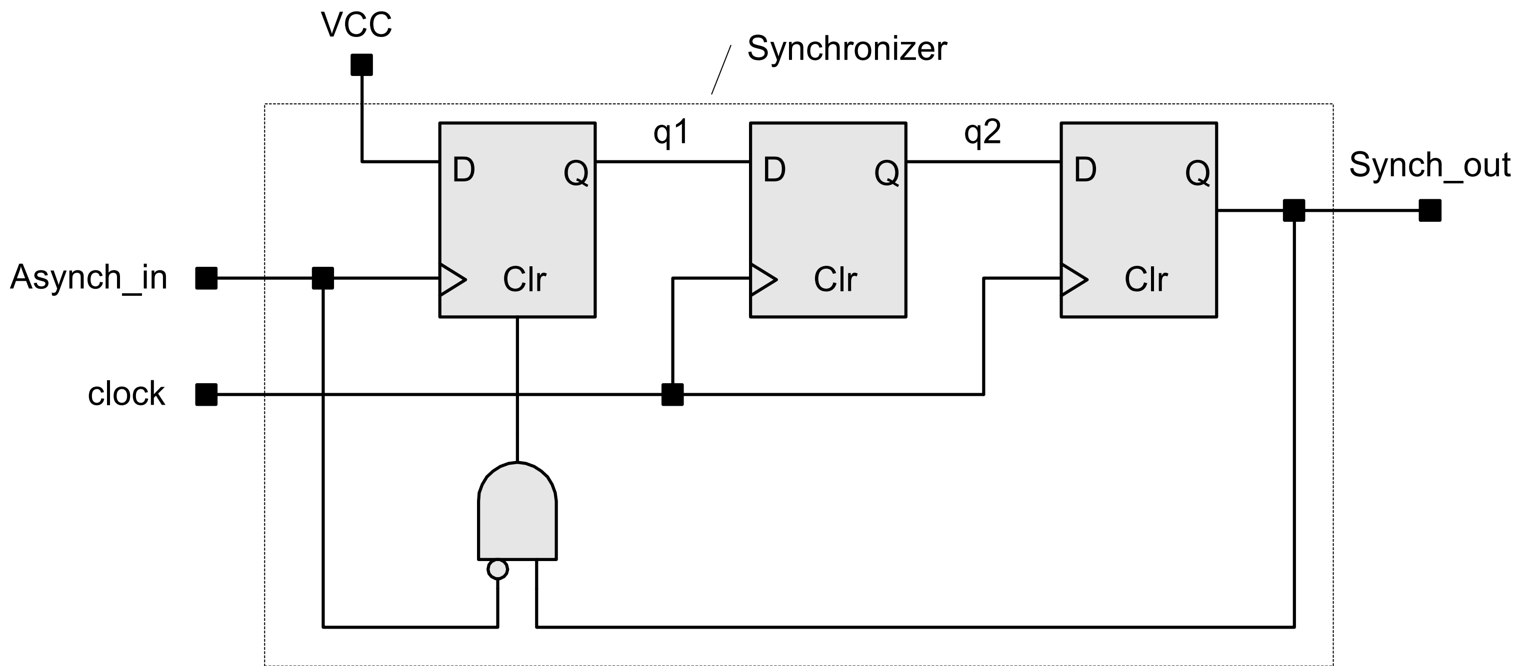
Metastability:



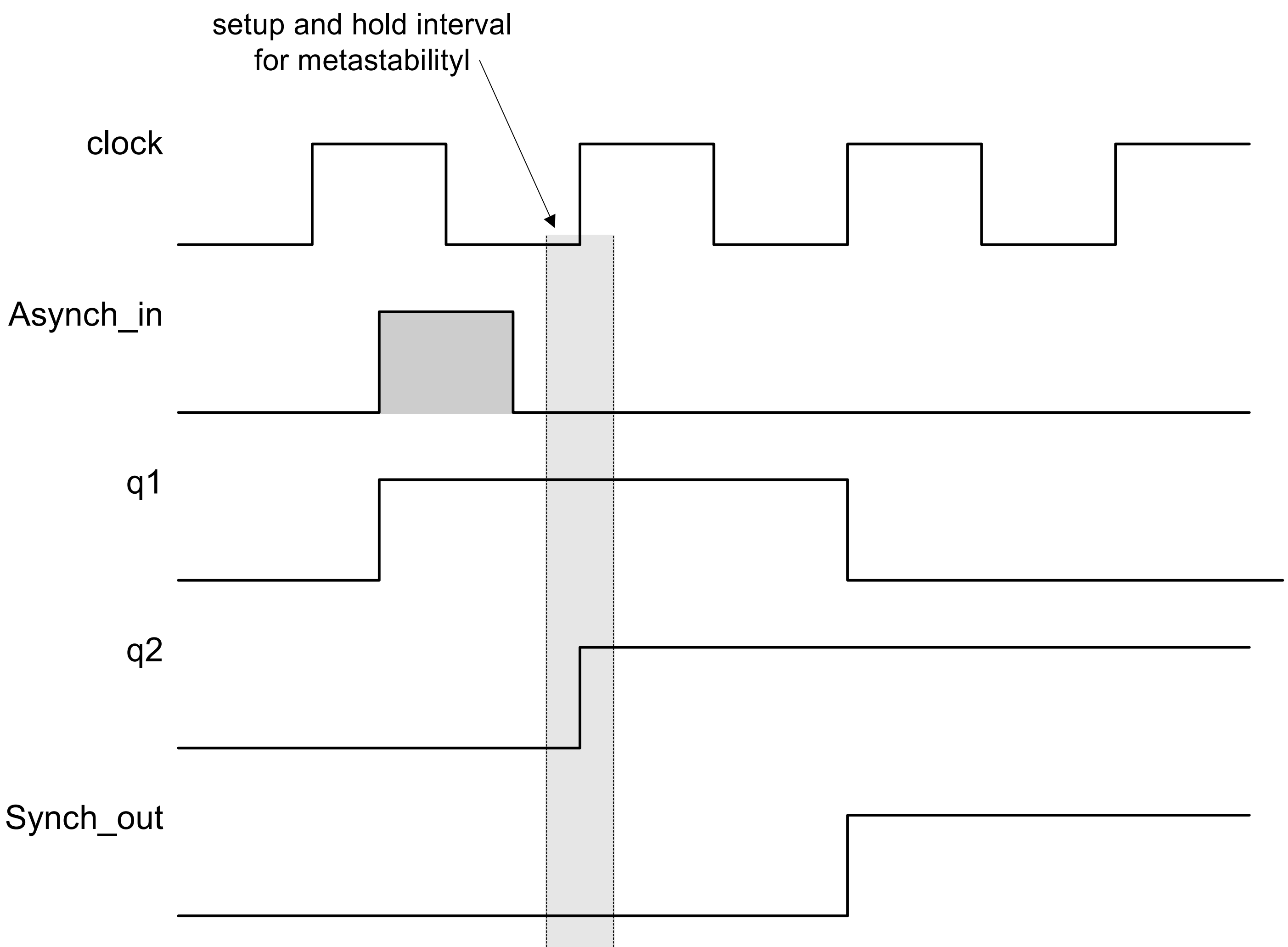
## Synchronizer for relatively long asynchronous input pulse:



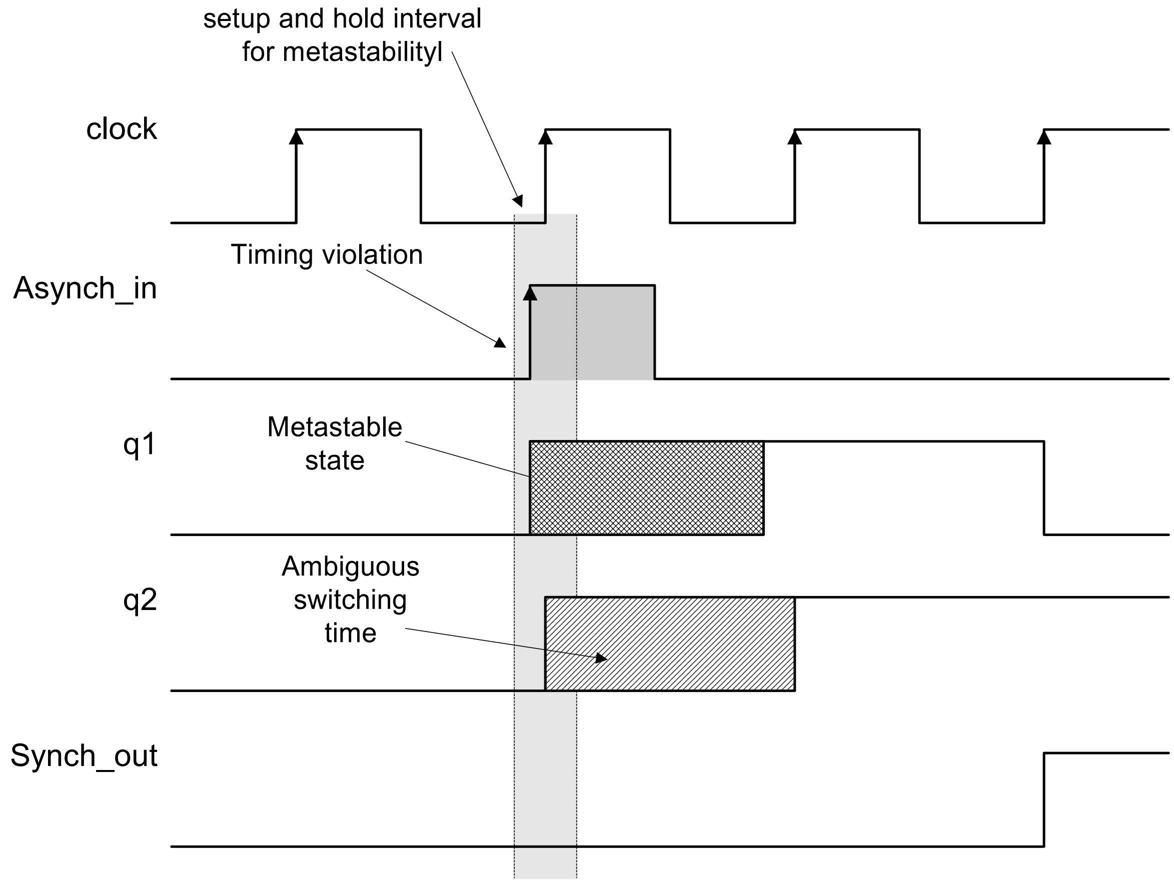
# Synchronizer for relatively short asynchronous input pulse:



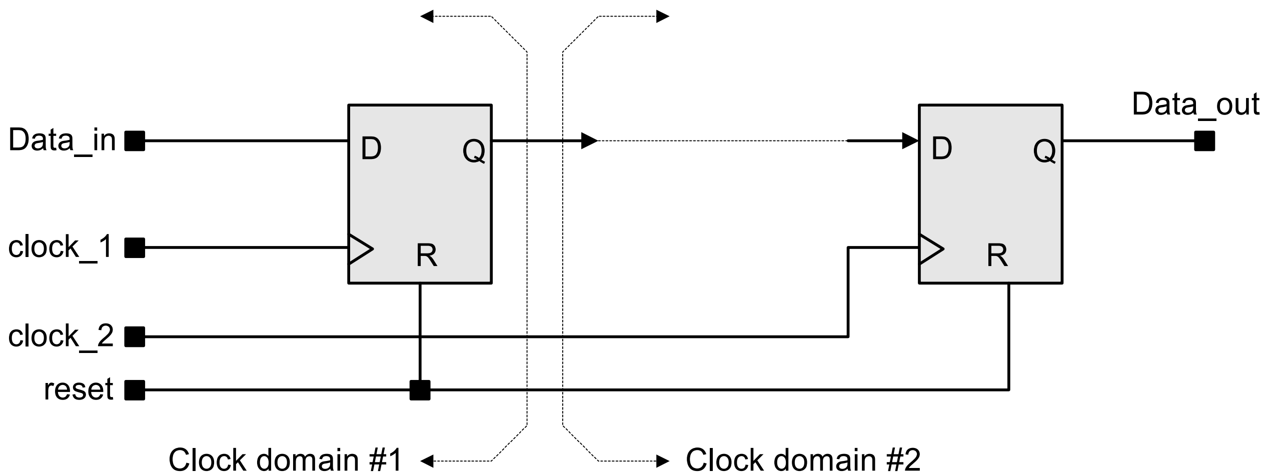
# Waveforms without metastabilty condition:



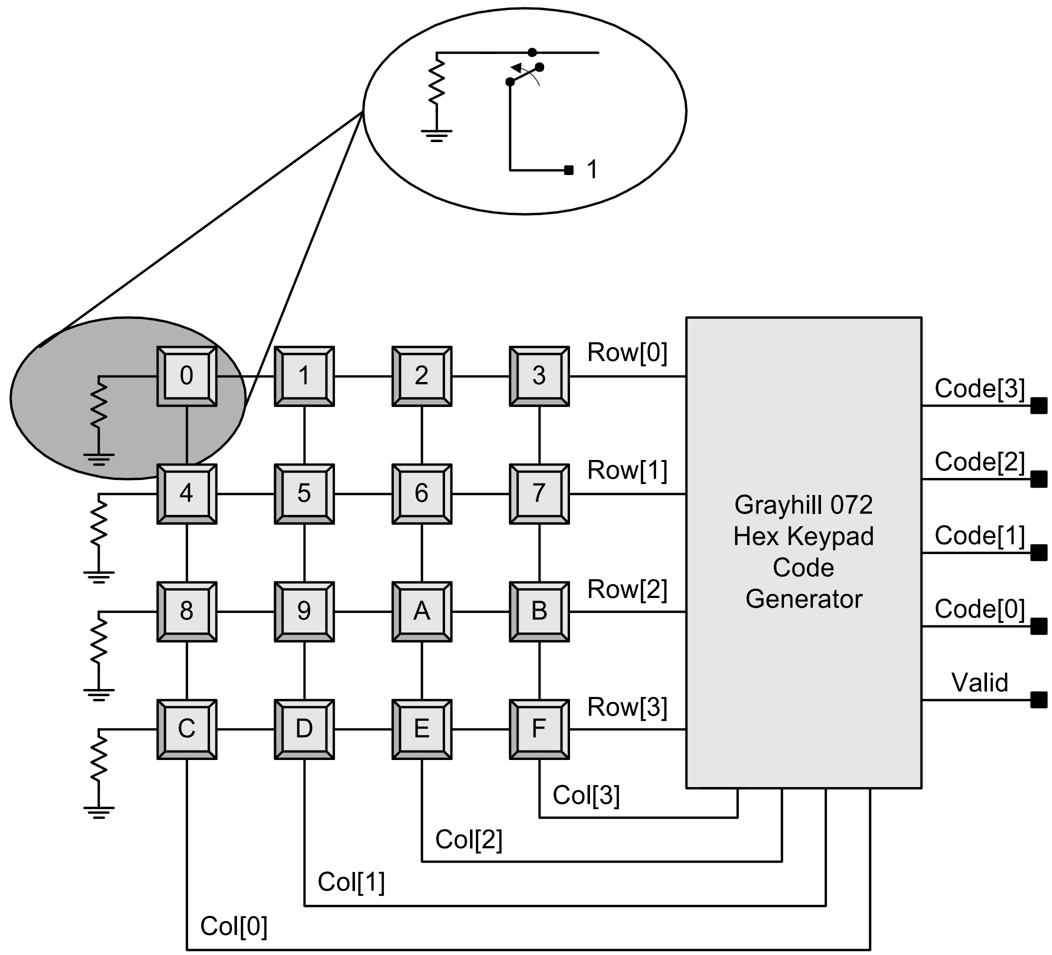
# Waveforms with metastability condition



## Synchronization across clock domains (more later)

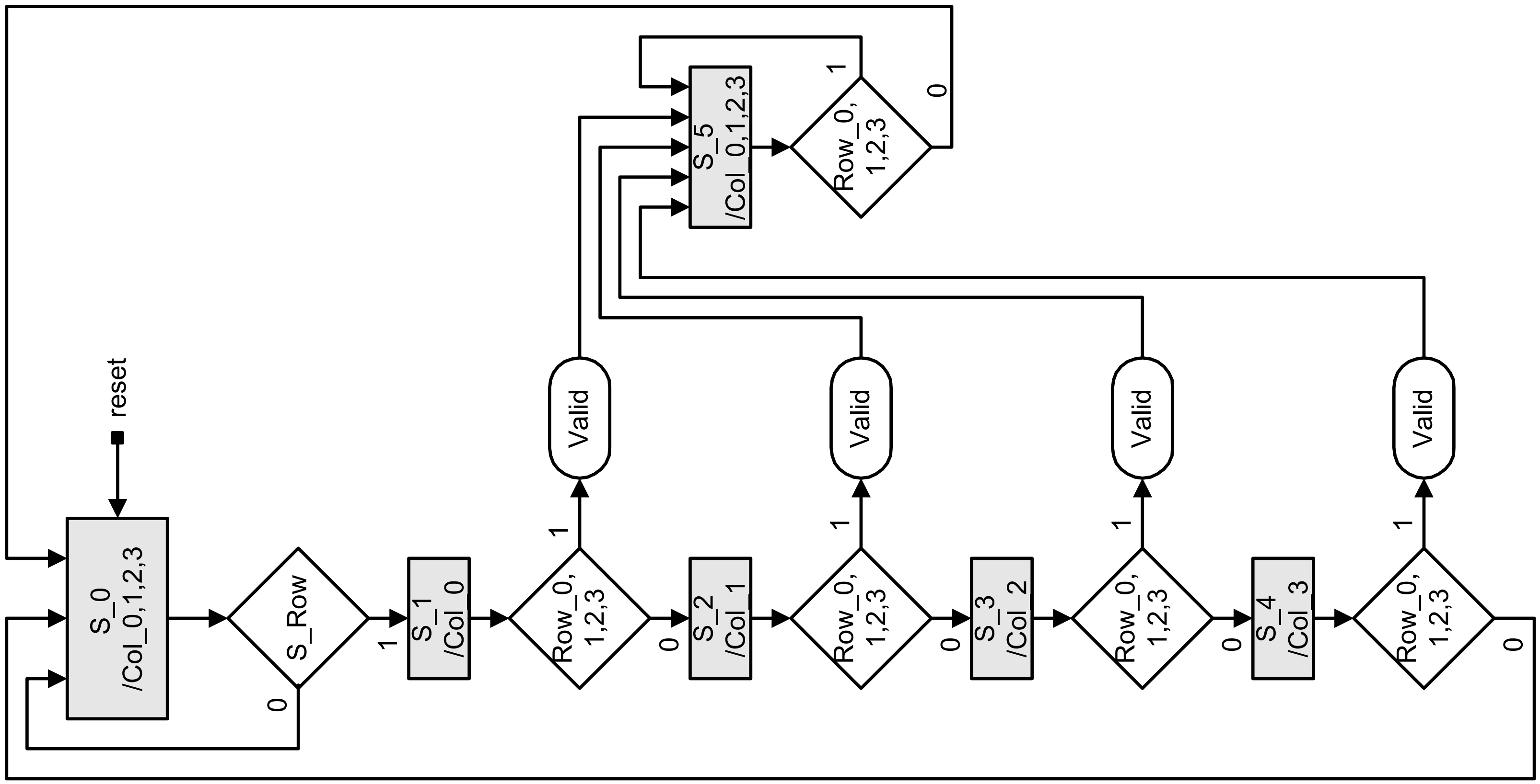


# Design Example: Keypad Scanner and Encoder (p 216)

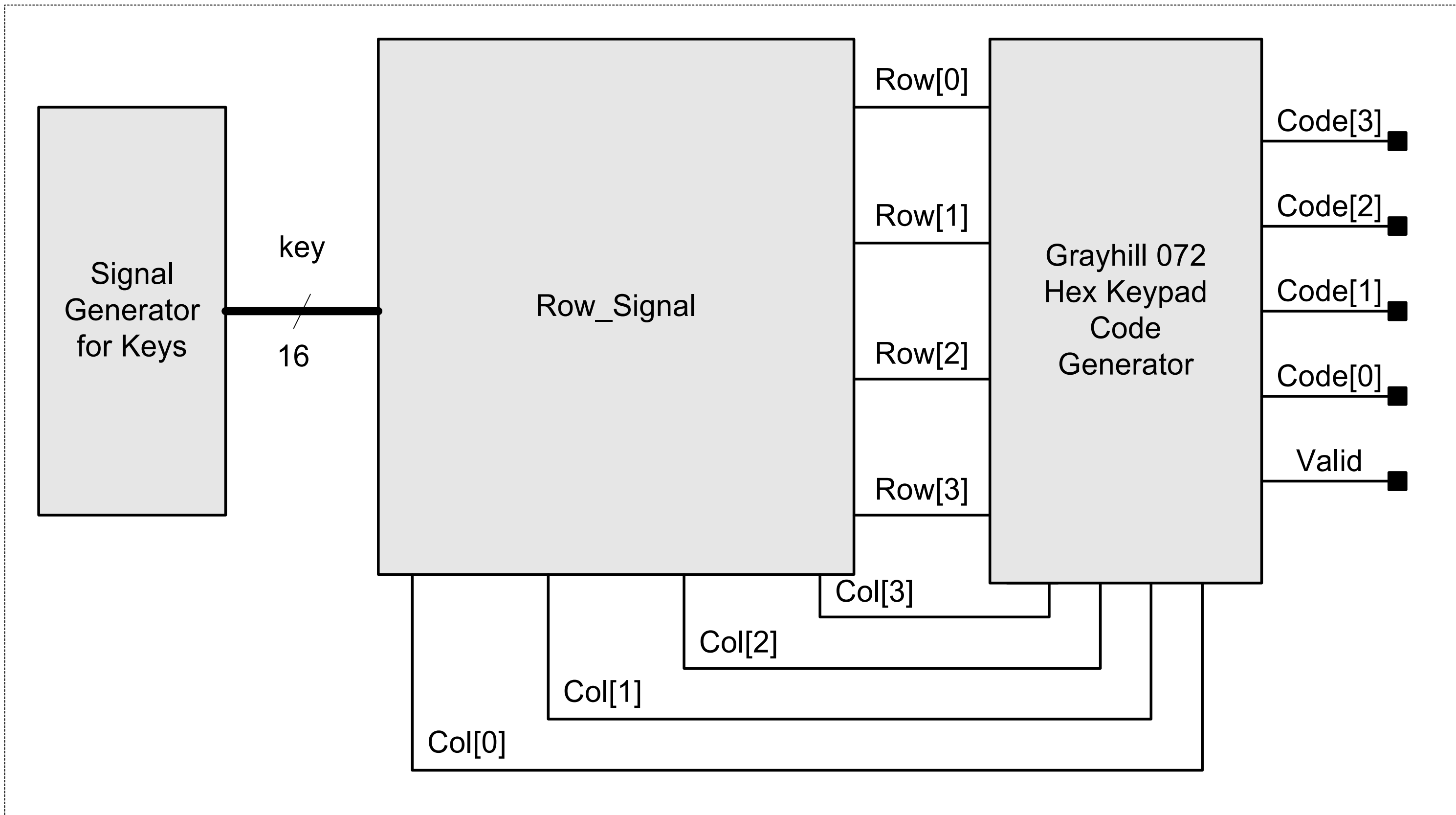


# Keypad Codes

| Key | Row[3:0] | Col[3:0] | Code |
|---|---|---|---|
| 0 | 0 0 0 1 | 0 0 0 1 | 0 0 0 0 |
| 1 | 0 0 0 1 | 0 0 1 0 | 0 0 0 1 |
| 2 | 0 0 0 1 | 0 1 0 0 | 0 0 1 0 |
| 3 | 0 0 0 1 | 1 0 0 0 | 0 0 1 1 |
| 4 | 0 0 1 0 | 0 0 0 1 | 0 1 0 0 |
| 5 | 0 0 1 0 | 0 0 1 0 | 0 1 0 1 |
| 6 | 0 0 1 0 | 0 1 0 0 | 0 1 1 0 |
| 7 | 0 0 1 0 | 1 0 0 0 | 0 1 1 1 |
| 8 | 0 1 0 0 | 0 0 0 1 | 1 0 0 0 |
| 9 | 0 1 0 0 | 0 0 1 0 | 1 0 0 1 |
| A | 0 1 0 0 | 0 1 0 0 | 1 0 1 0 |
| B | 0 1 0 0 | 1 0 0 0 | 1 0 1 1 |
| C | 1 0 0 0 | 0 0 0 1 | 1 1 0 0 |
| D | 1 0 0 0 | 0 0 1 0 | 1 1 0 1 |
| E | 1 0 0 0 | 0 1 0 0 | 1 1 1 0 |
| F | 1 0 0 0 | 1 0 0 0 | 1 1 1 1 |


reset
S_0
/Col_0,1,2,3
S_Row
0
1
S_1
/Col_0
Row_0,
1,2,3
1
Valid
0
S_2
/Col_1
Row_0,
1,2,3
1
Valid
0
S_3
/Col_2
Row_0,
1,2,3
1
Valid
0
S_4
/Col_3
Row_0,
1,2,3
1
Valid
0
S_5
/Col_0,1,2,3
Row_0,
1,2,3
1
0

Test bench for Hex_Keypad_Grayhill_072



```
// Decode the asserted Row and Col

//      Grayhill 072 Hex Keypad
//
//                  Co[0]     Col[1]    Col[2]    Col[3]
//      Row [0]     0         1         2         3
//      Row [1]     4         5         6         7
//      Row [2]     8         9         A         B
//      Row [3]     C         D         E         F

module Hex_Keypad_Grayhill_072 (Code, Col, Valid, Row, S_Row, clock, reset);
  output   [3: 0]    Code;
  output             Valid;
  output   [3: 0]    Col;
  input    [3: 0]    Row;
  input              S_Row;
  input              clock, reset;
  reg      [3: 0]    Col, Code;
  reg                state;
  reg      [5: 0]    state, next_state;
```

```
// One-hot state codes
parameter   S_0 = 6'b000001, S_1 = 6'b000010, S_2 = 6'b000100;
parameter   S_3 = 6'b001000, S_4 = 6'b010000, S_5 = 6'b100000;

assign Valid = ((state == S_1) || (state == S_2)
               || (state == S_3) || (state == S_4)) && Row;

// Does not matter if the row signal is not the debounced version.
// Assumed to settle before it is used at the clock edge

always @ (Row or Col)
  case ({Row, Col})
    8'b0001_0001: Code = 0;
    8'b0001_0010: Code = 1;
    8'b0001_0100: Code = 2;
    8'b0001_1000: Code = 3;

    8'b0010_0001: Code = 4;
    8'b0010_0010: Code = 5;
    8'b0010_0100: Code = 6;
    8'b0010_1000: Code = 7;

    8'b0100_0001: Code = 8;
    8'b0100_0010: Code = 9;
    8'b0100_0100: Code = 10;          // A
```

```
    8'b0100_1000: Code = 11;       // B

    8'b1000_0001: Code = 12;       // C
    8'b1000_0010: Code = 13;       // D
    8'b1000_0100: Code = 14;       // E
    8'b1000_1000: Code = 15;       // F

   default:       Code = 0;        // Arbitrary choice
  endcase

always @ (posedge clock or posedge reset)
  if (reset) state <= S_0; else state <= next_state;

always @ (state or S_Row or Row) // Next-state logic
 begin next_state = state; Col = 0;
  case (state)
   // Assert all rows
   S_0:  begin Col = 15; if (S_Row) next_state = S_1; end
   // Assert col 0
   S_1:  begin Col = 1; if (Row) next_state = S_5; else next_state = S_2; end
   // Assert col 1
   S_2:  begin Col = 2; if (Row) next_state = S_5; else next_state = S_3; end
   // Assert col 2
   S_3:  begin Col = 4; if (Row) next_state = S_5; else next_state = S_4; end
   // Assert col 3
```

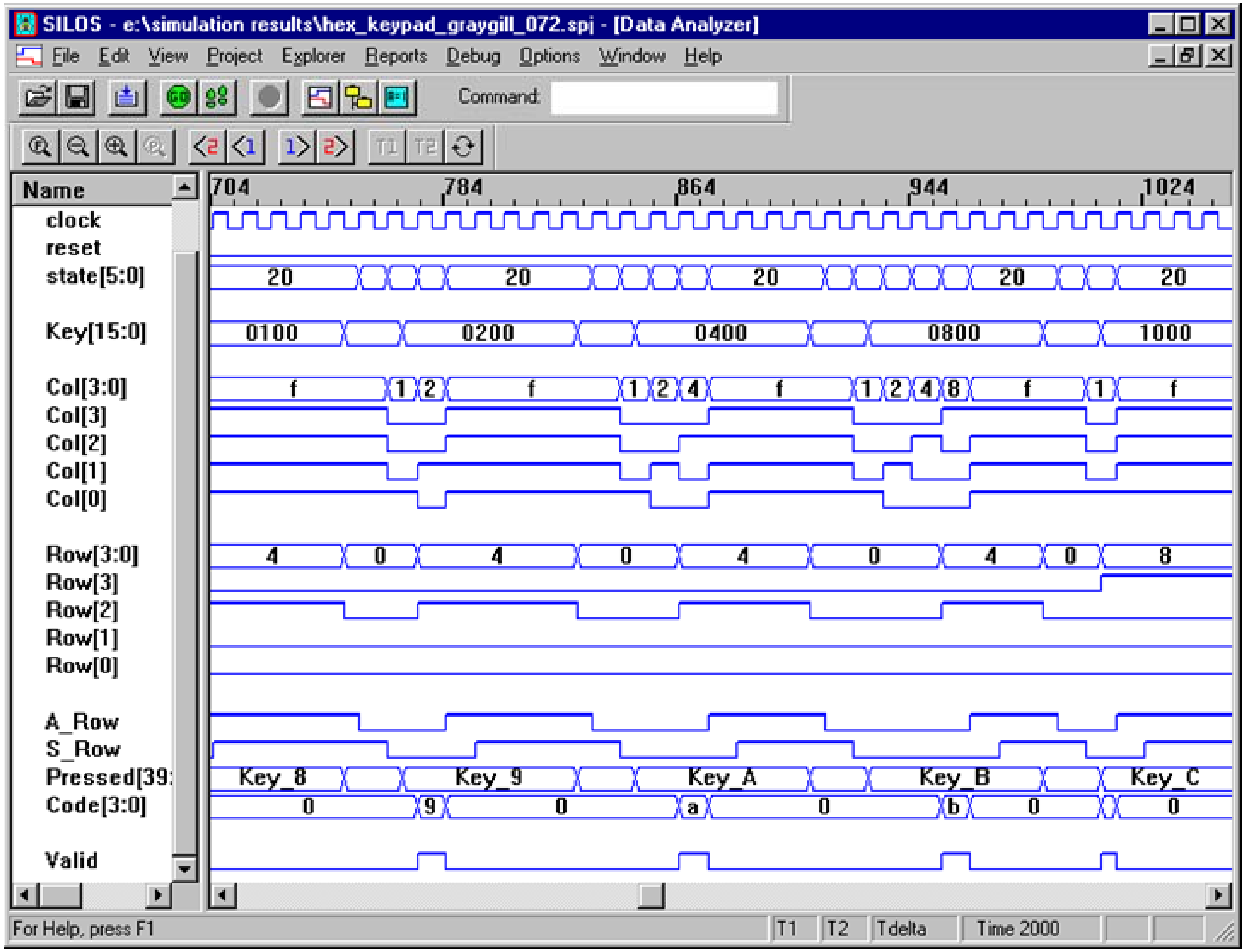
```
      S_4:   begin Col = 8; if (Row) next_state = S_5; else next_state = S_0; end
      // Assert all rows
      S_5:   begin Col = 15; if (Row == 0) next_state = S_0; end
    endcase
end
endmodule

module Synchronizer (S_Row, Row, clock, reset);
  output                S_Row;
  input    [3: 0]       Row;
  input                 clock, reset;
  reg                   A_Row, S_Row;

  // Two stage pipeline synchronizer
  always @ (posedge clock or posedge reset) begin
    if (reset)  begin   A_Row <= 0;
                        S_Row <= 0;
    end
    else begin          A_Row <= (Row[0] || Row[1] || Row[2] || Row[3]);
                        S_Row <= A_Row;
    end
 end
endmodule
```

```
module Row_Signal (Row, Key, Col);     // Scans for row of the asserted key
  output   [3: 0]       Row;
  input    [15: 0]      Key;
  input    [3: 0]       Col;
  reg      [3: 0]       Row;

  always  @ (Key or Col) begin              // Combinational logic for  key assertion
    Row[0] = Key[0]   && Col[0  ||  Key[1]   && Col[1] || Key[2]   && Col[2] || Key[3]   &&
Col[3];
    Row[1] = Key[4]   && Col[0] ||  Key[5]   && Col[1] || Key[6]   && Col[2] || Key[7]   &&
Col[3];
    Row[2] = Key[8]   && Col[0] ||  Key[9]   && Col[1] || Key[10] && Col[2] || Key[11] &&
Col[3];
    Row[3] = Key[12] && Col[0] ||  Key[13] && Col[1] || Key[14] && Col[2] || Key[15] &&
Col[3];
  end
endmodule

//////////////////////////////  Test Bench ////////////////////////////////
module test_Hex_Keypad_Grayhill_072 ();
  wire      [3: 0]       Code;
  wire                   Valid;
  wire      [3: 0]       Col;
  wire      [3: 0]       Row;
  reg                    clock, reset;
```

```
reg         [15: 0]     Key;
integer          j, k;
reg[39: 0]     Pressed;
parameter    [39: 0] Key_0 = "Key_0";
parameter    [39: 0] Key_1 = "Key_1";
parameter    [39: 0] Key_2 = "Key_2";
parameter    [39: 0] Key_3 = "Key_3";
parameter    [39: 0] Key_4 = "Key_4";
parameter    [39: 0] Key_5 = "Key_5";
parameter    [39: 0] Key_6 = "Key_6";
parameter    [39: 0] Key_7 = "Key_7";
parameter    [39: 0] Key_8 = "Key_8";
parameter    [39: 0] Key_9 = "Key_9";
parameter    [39: 0] Key_A = "Key_A";
parameter    [39: 0] Key_B = "Key_B";
parameter    [39: 0] Key_C = "Key_C";
parameter    [39: 0] Key_D = "Key_D";
parameter    [39: 0] Key_E = "Key_E";
parameter    [39: 0] Key_F = "Key_F";
parameter    [39: 0] None = "None";

always @ (Key) begin  // "one-hot" code for pressed key
  case (Key)
    16'h0000:   Pressed = None;
    16'h0001:   Pressed = Key_0;
```

```
    16'h0002:  Pressed = Key_1;
    16'h0004:  Pressed = Key_2;
    16'h0008:  Pressed = Key_3;

    16'h0010:  Pressed = Key_4;
    16'h0020:  Pressed = Key_5;
    16'h0040:  Pressed = Key_6;
    16'h0080:  Pressed = Key_7;

    16'h0100:  Pressed = Key_8;
    16'h0200:  Pressed = Key_9;
    16'h0400:  Pressed = Key_A;
    16'h0800:  Pressed = Key_B;

    16'h1000:  Pressed = Key_C;
    16'h2000:  Pressed = Key_D;
    16'h4000:  Pressed = Key_E;
    16'h8000:  Pressed = Key_F;

    default: Pressed = None;
  endcase
end
```

```
Hex_Keypad_Grayhill_072 M1(Code, Col, Valid, Row, S_Row, clock);
  Row_Signal M2      (Row, Key, Col);
  Synchronizer M3    (S_Row, Row, clock, reset);

    initial #2000 $finish;
    initial begin clock = 0; forever #5 clock = ~clock; end
    initial begin reset = 1; #10 reset = 0; end
    initial
      begin for (k = 0; k <= 1; k = k+1)
        begin Key = 0; #20 for (j = 0; j <= 16; j = j+1)
          begin
            #20 Key[j] = 1; #60 Key = 0; end end end
 endmodule
```


SILOS - e:\simulation results\hex_keypad_graygill_072.spj - [Data Analyzer]
File Edit View Project Explorer Reports Debug Options Window Help
Command:
Name
704 784 864 944 1024
clock
reset
state[5:0]
Key[15:0]
Col[3:0]
Col[3]
Col[2]
Col[1]
Col[0]
Row[3:0]
Row[3]
Row[2]
Row[1]
Row[0]
A_Row
S_Row
Pressed[39:
Code[3:0]
Valid
For Help, press F1
T1
T2
Tdelta
Time 2000