Chapter 5

Logic Design with Behavioral Models of Combinational and Sequential Logic

Contents

- Verilog models based on Boolean equations
- More general and abstract style of modeling for combinational and sequential logic
- More general sequential machines consisting of a datapath and a control unit designed by ASM and ASMD
- ASM **:Algorithmic State Machine**
- ASMD: **Algorithmic State Machine and Datapath**

5.1 Behavioral Modeling

- Behavioral modeling is the predominant descriptive style used by industry, enabling the design of massive chips
- Behavioral modeling describes

  the functionality of a design —that is:

  what the designed circuit will do, not how to build it in hardware
- Propagation delays are not included in the behavioral model of the circuit, are considered by the synthesis tool

Behavioral modeling

- (1) rapidly create a behavioral prototype of a design (without binding it to hardware detail)
- (2) verify its functionality
- (3) use a synthesis tool to optimize and map the design into a selected physical technology, subject to constraints on timing and/or area
- Focusing the designer's **attention on the functionality** that is to be implemented ,rather than on individual logic gates and their interconnections
- Provides the freedom to explore alternatives and tradeoffs before committing a design to production

Behavioral Modeling

- A much easier way to write testbenches
- Also good for more abstract models of circuits
  - **Easier to write**
  - **Simulates faster**
  - **More flexible**
- Verilog succeeded in part because it allowed both the model and the testbench to be described together

5.2 Data Types for Behavioral Modeling

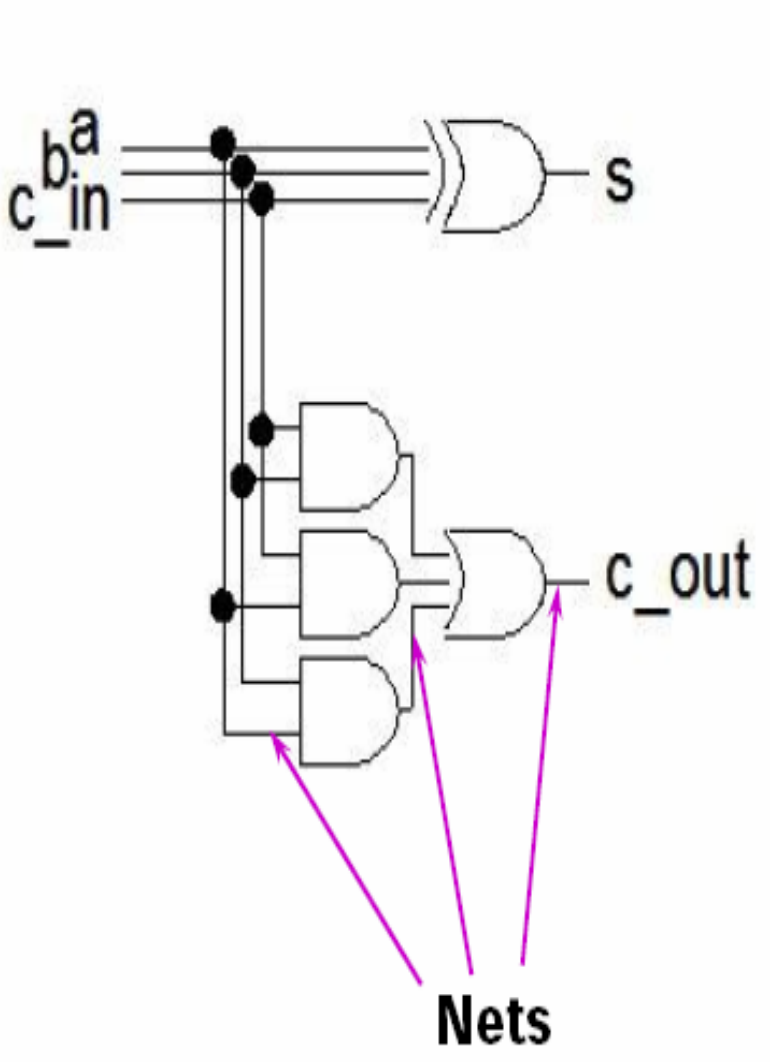
- Two main data types: nets and registers
- *Net variables* act like wires in a physical circuits and establish connectivity between designed objects
- *Register variables* act like variables in ordinary procedural languages-they store information while the program executes
- **Use mainly the net type *wire* and the register types *reg* and *integer* in synthesis**
- A ***wire*** and a ***reg*** have a default size of 1 bit
- An ***integer*** is automatically fixed at the word length ,at least 32 bit

Nets

- **Physical connections** between structural entities.
- As the driver changes its value, Verilog automatic propagates the value onto a net.

- **Default value is z if no drivers are connected to net**

  An undriven wire defaults to a value of Z (high impedance).



Registers

- Registers represent abstract storage elements.
- A register holds its value until a new value is assigned to it.
- Registers are used extensively in behavior modeling and in applying stimuli.
- **Default value is X.**

Type of Registers

- reg
  - Unsigned integer variable of varying bit width
- integer
  - Signed integer variable, 32-bit wide. Arithmetic operations produce 2's complement results.
- real
  - Signed floating point variable, double precision
- time
  - Unsigned integer variable, 64-bit wide.
- Do not confuse register data type with structural storage element (e.g. D-type FF)

Choosing the Correct Data Types

- An **input** or **inout** port must be a net.
- An **output** port can be a register or net data type.
- A signal assigned a value in a procedural block must be a register data type.

Common Mistakes in Choosing Data Types

- Make a procedural assignment to a net

  wire [7:0] databus;

  always @(read or addr) databus=read ? mem[addr] : 'bz;

  Illegal left-hand-side assignment

- Connect a register to an instance output

  reg myreg;

  and (myreg, net1, net2);

  Illegal output port specification

- Declare a module **input** port as a register

  input myinput;

  reg myinput;

  Incompatible declaration

Assignments

- Drive values into nets and registers.
- There are two basic forms of asignment
  - Continuous assignment, assigns values to net
  - Procedural assignment, assigns values to register
- Basic form
  - <left_hand_side>=<right_hand_side>

Continuous Assignment

- **Another way to describe combinational function**
- Convenient for logical or datapath specifications
- wire [8:0] sum;

  wire [7:0] a, b;

  wire carryin;

  assign sum = a + b + carryin;

Define bus widths

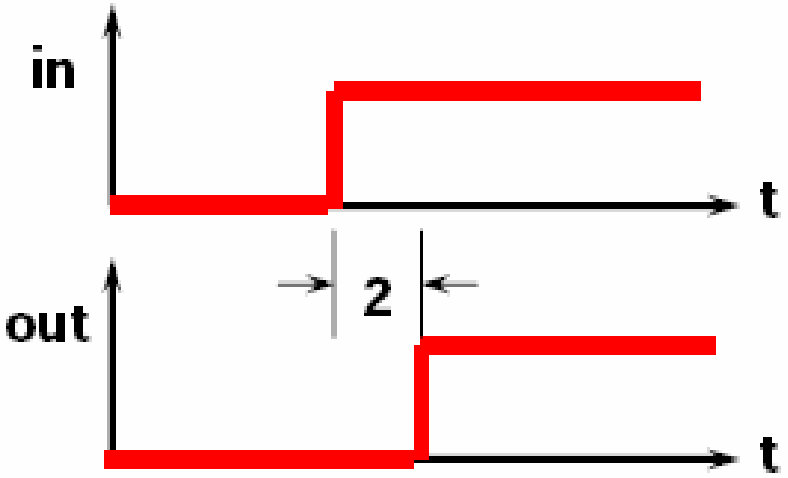
Continuous assignment: permanently sets the value of sum to be a+b+carryin

Recomputed when a, b, or carryin changes

Continuous Assignment

- Any changes in the RHS of the continuous assignment are evaluated and the LHS is updated
- Continuous assignment provides a way to model combinational logic.

```
assign #2 out=in;
```



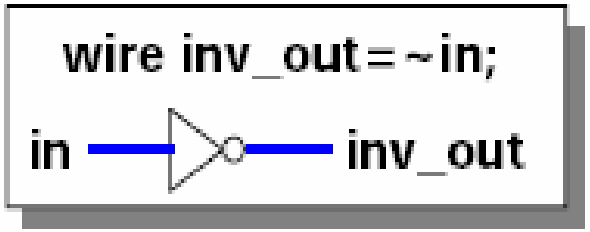
Continuous Assignment

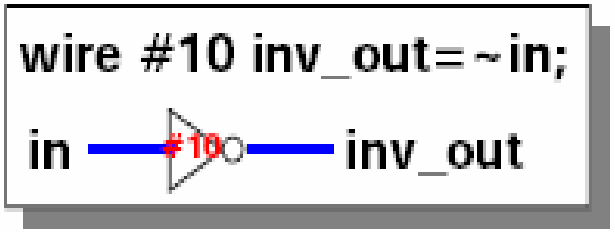
- Right hand side can be
  - Expression
    - assign and_out = a & b;
  - Value
    - assign c = 1;
  - Other net
    - assign d = e;
- Type of continuous assignment declaration
  - The net declaration assignment
  - The continuous assignment statement

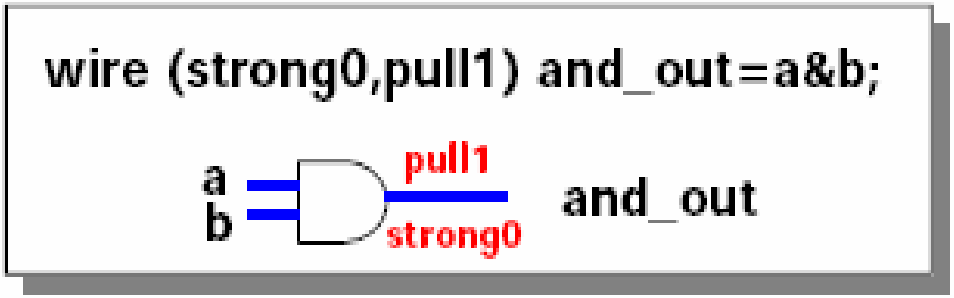
Net Declaration Assignment

The net declaration assignment

- <net_type><drive_strength>?<range>?<delay>?<list_of_assignment>;





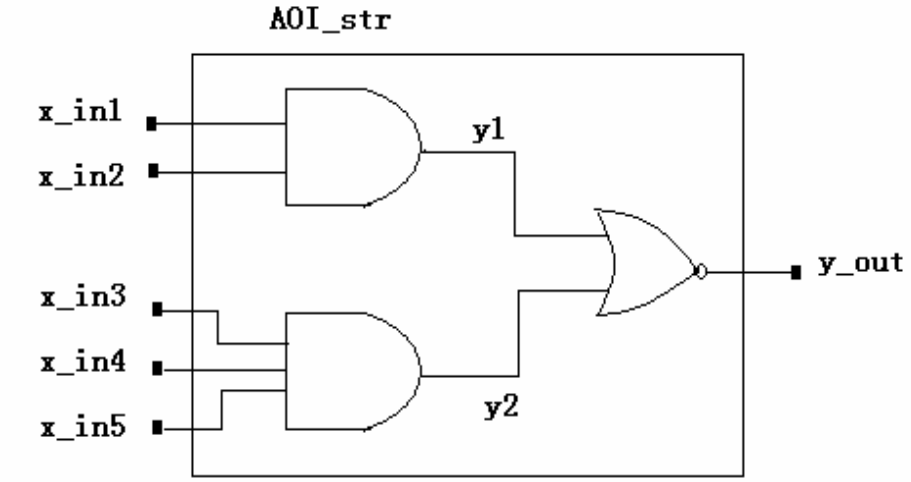


5.3 Boolean-Equation-Based Behavioral Models of Combinational Logic

- A Boolean equation describes combinational logic by an expression of operations on variables
- Its counterpart in Verilog is the continuous assignment statement
- **A continuous assignment describe implicit combinational logic**
- A continuous assignment is more compact and understandable than a schematic or a netlist of primitives

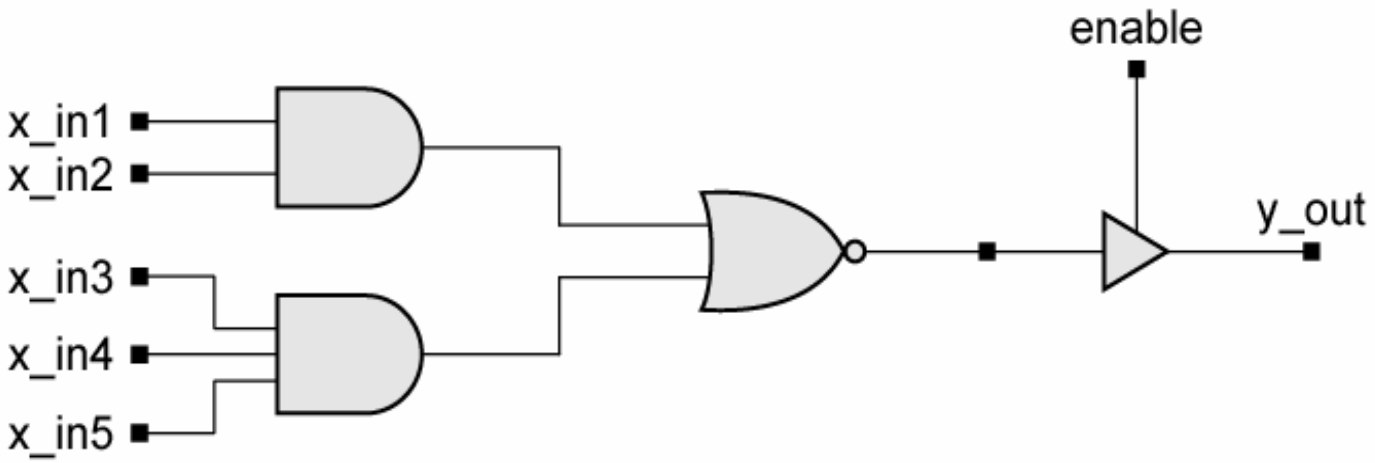
Example 5.1 A five-input AOI described statement

```
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
```



Ex 5.2 A five-input AOI with *enable* and have a three-state output

```
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
```



Continuous Assignment Statement

- assign <drive_strength>?<delay>?<list_of_assignment>;
- The continuous assignment statement drive a net that has been previously declared.

```
module MUX2_to_1(out,i0,i1,sel);
output out;
input i0,i1,sel;

assign out = (sel) ? i1 : i0;

endmodule
```

Conditional Operation ? :

- Used with various relational operators **provide convenient mechanisms for description of fairly complex logical functions**
- Example:
- ‘timescale 1ns/100ps

```
Module quad_mux2_1
        (input[3:0] i0,i1,input s,output[3:0] y);
 assign y = s ? i1 : i0 ;
 endmodule
```

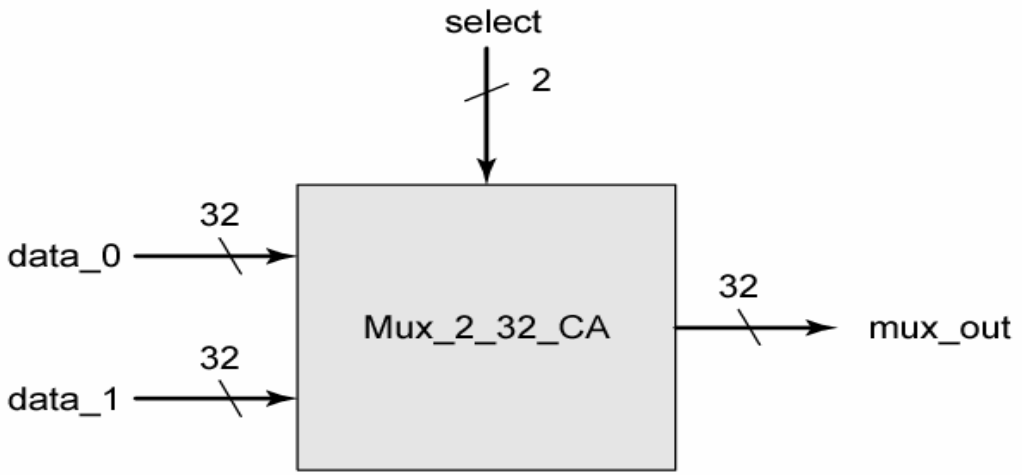
Example 5.3

- Continuous assignments can also be written implicitly and efficiently (without the keyword assign) as part of the declaration of a wire.

```
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 A two-channel mux with 32-bit datapaths

```
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
```



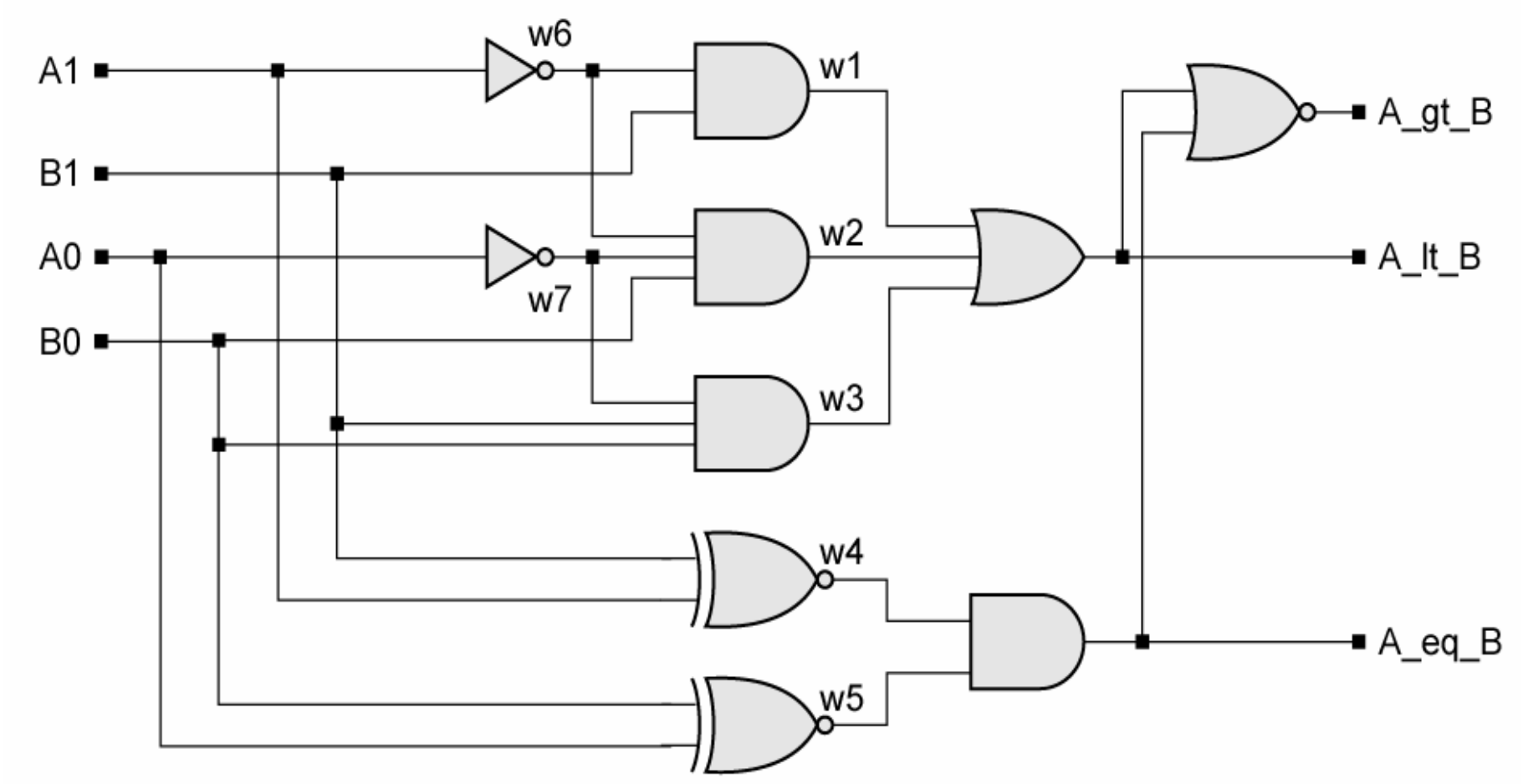
5.4 Propagation Delay and Continuous Assignments

- Propagation (inertial) delay can be associated with a continuous assignment
- So that its implicit logic has the same functionality and timing characteristics as its gate-level counterpart

Ex 5.5 an AOI gate structure with unit propagation delays on its implicit gates

```
module AOI_5_CA3 (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_in4;
  wire #1 y_out=~(y1|y2);   // complement the result of
                                  //   bitwise OR operation
endmodule
```

Ex. schematic of a 2-bit binary comparator(Fig 4.10)

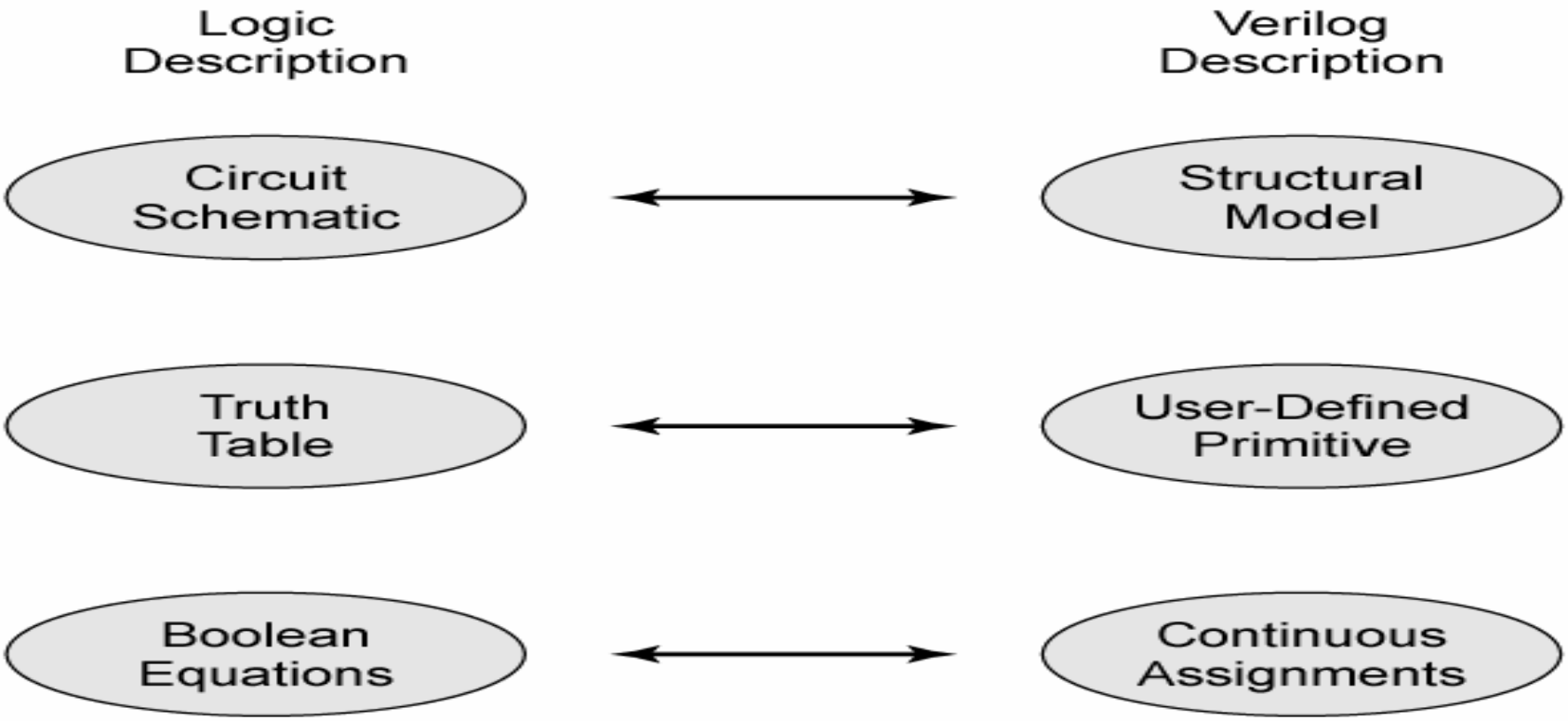


Ex 5.6 a 2-bit comparator described by three continuous assignments

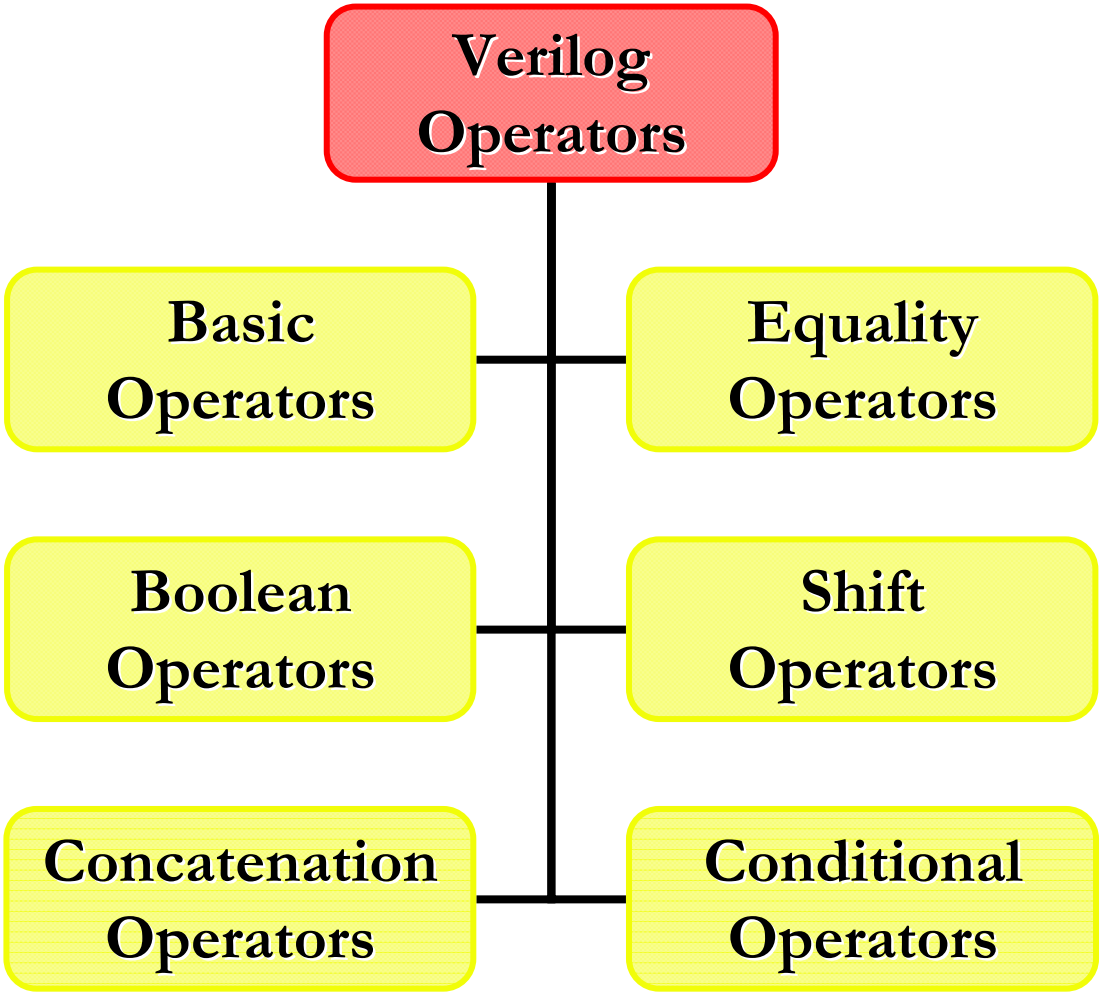
```
module compare_2_CA0
        (A_lt_B,A_gt_b,A_eq_B,A1,A0,B1,B0);
  input   A1,A0,B1,B0;
  output  A_lt_B,A_gt_B,A_eq_B;
  assign  A_lt_B=(~A1)&B1|(~A1)&(~A0)&B0|(~A0)&B1&B0;
  assign  A_gt_B=A1&(~B1)|A0&(~B1)&(B0)|A1&A0&(~B0);
  assign  A_eq_B=(~A1)&(~A0)&(~B1)&(~B0)|(~A1)&A0&
           (~B1)&B0|A1&A0&&B1&B0|A1&(~A0)&B1&(~B0);
endmodule
```

counterparts of three common descriptions of comb. logic

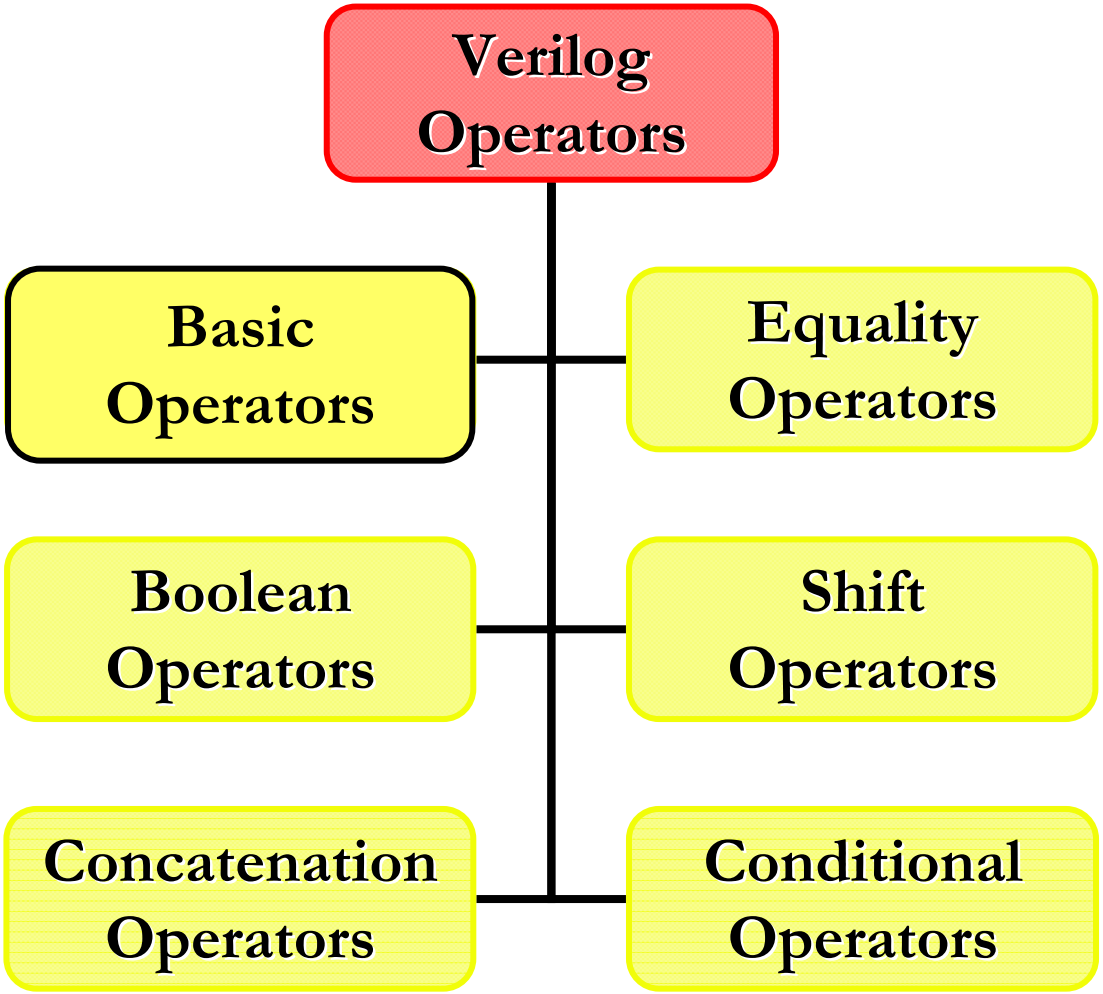
- All three descriptions describe level-sensitive behavior—variables are updated immediately when an input changes



Verilog Operators



Basic Operators



Basic Operators

Basic	OPERATION	DESCRIPTION	RESULT
Arithmetic	+ – * / **	Basic arithmetic	Multi-bit
Relational	> >= < <=	compare	One-bit

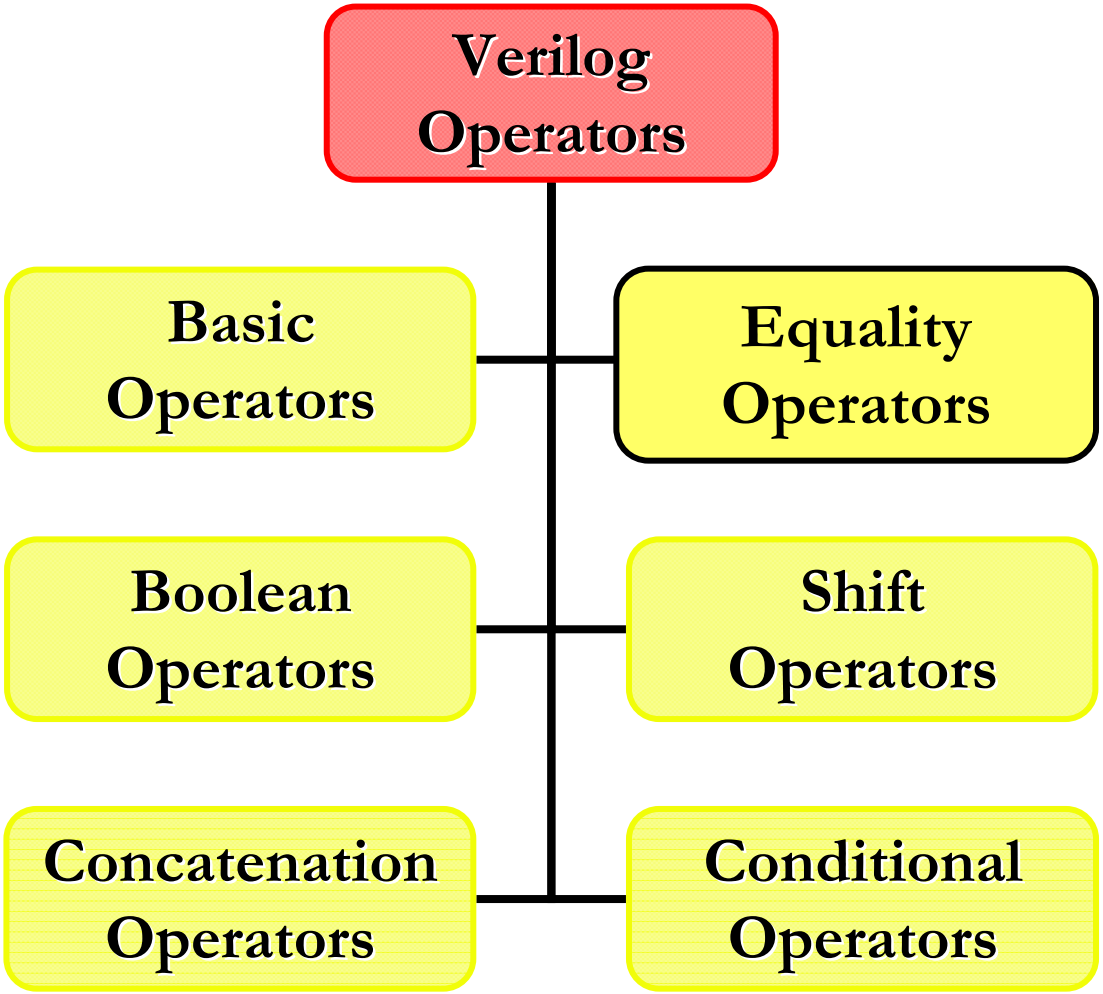
- **Arithmetic Operations in Verilog take bit, vector, integer and real operands.**
- **Basic operators of Verilog are +, -, *, / and **.**
- **An X or a Z value in a bit of either of the operands of a multiplication causes the entire result of the multiply operation to become X.**
- **Unary plus (+) and minus (–) are allowed in Verilog. These operators take precedence over other arithmetic operators.**
- **If any of the operands of a relational operator contain an X or a Z, then the result becomes X.**

Basic Operators

Example	Results in
25 * 8'b6	150
25 + 8'b7	32
25 / 8'b6	4
22 % 7	1
8'b10110011 > 8'b0011	1
4'b1011 < 10	0
4'b1Z10 < 4'b1100	x
4'b1x10 < 4'b1100	x
4'b1x10 <= 4'b1x10	x

- **Examples of Basic Operations**

Equality Operators



Equality Operators

Equality	OPERATION	DESCRIPTION	RESULT
Logical	== !=	Equality not including Z, X	One-bit
Case	=== !==	Equality including Z, X	One-bit

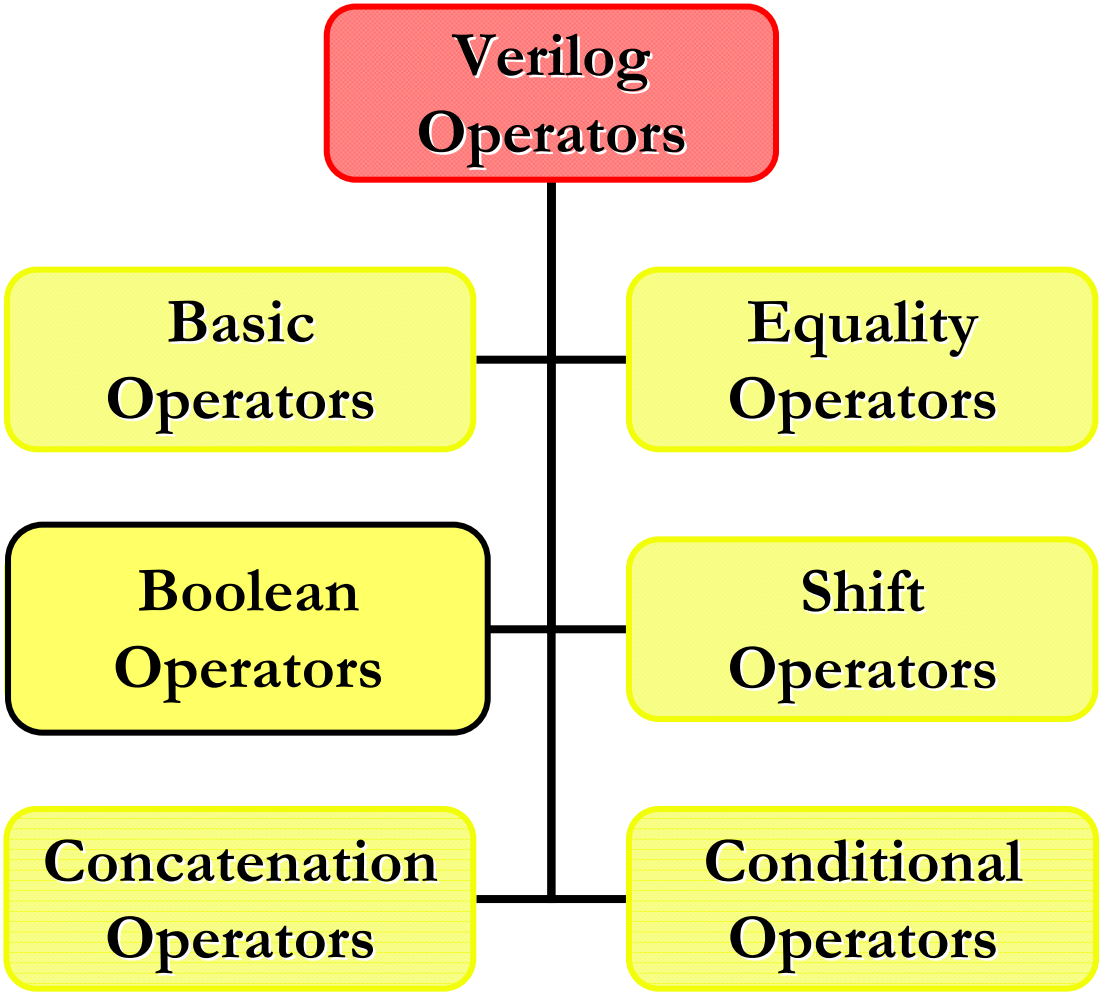
- **Equality operators are categorized into two groups:**
- **Logical: Compare their operands for equality (==) or inequality (!=)**
  - **Return a one-bit result, 0, 1, or x**
  - **An X ambiguity arises when an X or a Z occurs in one of the operands.**
- **Case: (=== , !==) , Consider X and Z values comparing their operands. The result is always 0 or 1.**

Equality Operators

Example			Results in
8'b10110011	==	8'b10110011	1
8'b1011	==	8'b00001011	1
4'b1100	==	4'b1Z10	0
4'b1100	!=	8'b100X	1
8'b1011	!==	8'b00001011	0
8'b101X	===	8'b101X	1

- **Examples of Equality Operations**

Boolean Operators



Boolean Operators

Boolean	OPERATION	DESCRIPTION	RESULT
Logical	&& || !	Simple logic	One-bit
Bit-wise	~ & | ^ ^~ ~^	Vector logic operation	One-bit
Reduction	& ~& |~| ^ ^~ ~^	Perform operation on all bits	One-bit

If an X or a Z appears in an operand of a logical operator, an X will result.

The complement operator ~ results in 1 and 0 for 0 and 1 inputs, and X for X and Z inputs.

Boolean Operators

& / ^	| / ^~	0		1		X		Z	
0		0	0	0	1	0	x	0	x
		0	1	1	0	x	x	x	x
1		0	1	1	1	x	1	x	1
		1	0	0	1	x	x	x	x
X		0	x	x	1	x	x	x	x
		x	x	x	x	x	x	x	x
Z		0	x	x	1	x	x	x	x
		1	x	x	x	x	x	x	x

- **Bit-by-bit Bitwise and Reduction Operators**

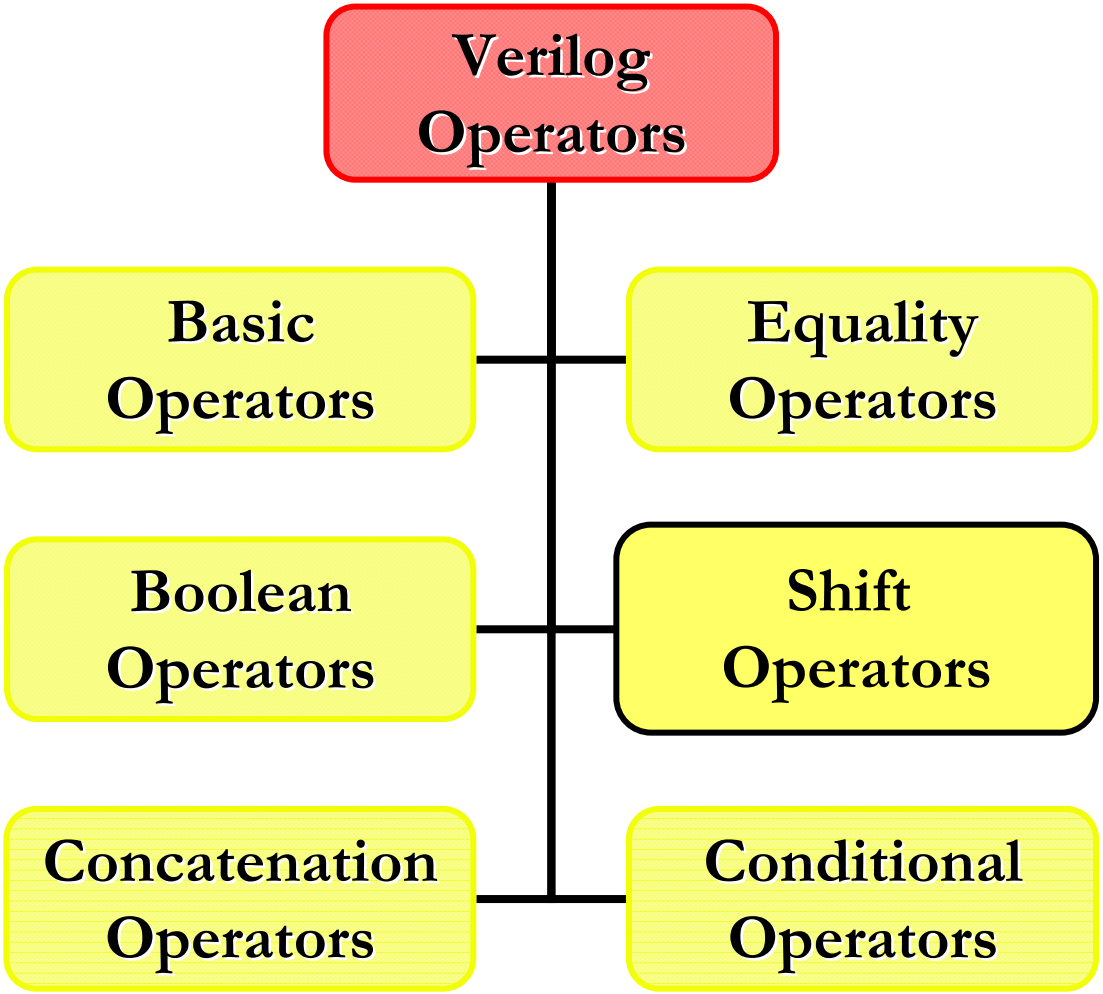
Boolean Operators

- Complement reduction operations (~&, ~|, and ~^) perform reduction first and then complement the result.

Example	Results in
8'b01101110 && 4'b0	0
8'b01101110 || 4'b0	1
8'b01101110 && 8'b10010001	1
! (8'b10010001)	1
8'b01101110 & 8'bxxzz1100	8'b0xx01100
8'b01101110 | 8'bxxzz1100	8'bx11x1110
~& (4'b0xz1)	1
~| (4'b0xz1)	0

- Logical, Bit-wise, and Reduction

Shift Operators



Shift Operators

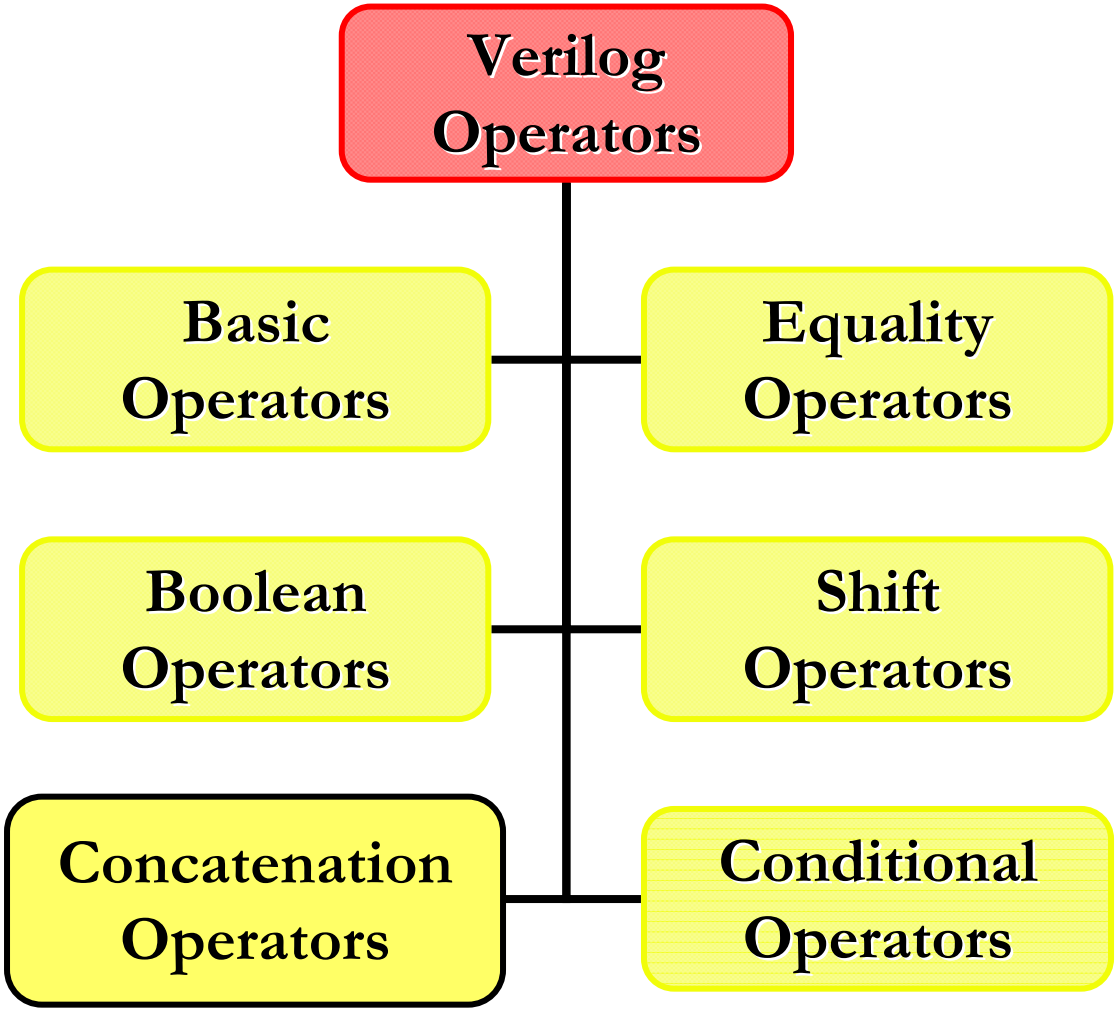
Shift	OPERATION	DESCRIPTION	RESULT
Logical right	>>n	Zero-fill Shift n places	Multi-bit
Logical left	<<n	Zero-fill Shift n places	Multi-bit
Arithmetic right	>>>n		Multi-bit
Arithmetic left	<<<n		Multi-bit

- **Logical shift operators (>> and << for shift right and left) fill the vacated bit positions with zeros.**
- **Fill values for arithmetic shift operators depend on the type of their results being signed or unsigned .**

Shift Operators

Example	Results in
8'b0110_0111 << 3	8'b0011_1000
8'b0110_0111 << 1'bz	8'bxxxx_xxxx
Signed_LHS = 8'b1100_0000>>>2	8'b1111_0000

Concatenation Operators



Concatenation Operators

Concat	OPERATION	DESCRIPTION	RESULT
Concatenation	{ }	Join bits	Multi-bit
Replication	{{ }}	Join & replicate	Multi-bit

- **The notation used for this operator is a pair of curly brackets ({...}) enclosing all scalars and vectors that are being concatenated.**
- **If a is a 4-bit reg and *aa* is a 6-bit reg, the following assignment places 1101 in a and 001001 in aa:**

```
{a, aa} = 10'b1101_001001
```

Concatenation Operators

- If the a and aa registers have the values assigned to them above, and aaa is a 16-bit reg data type, then the assignment,

  ```
  aaa = {aa, {2{a}}, 2'b11}
  ```

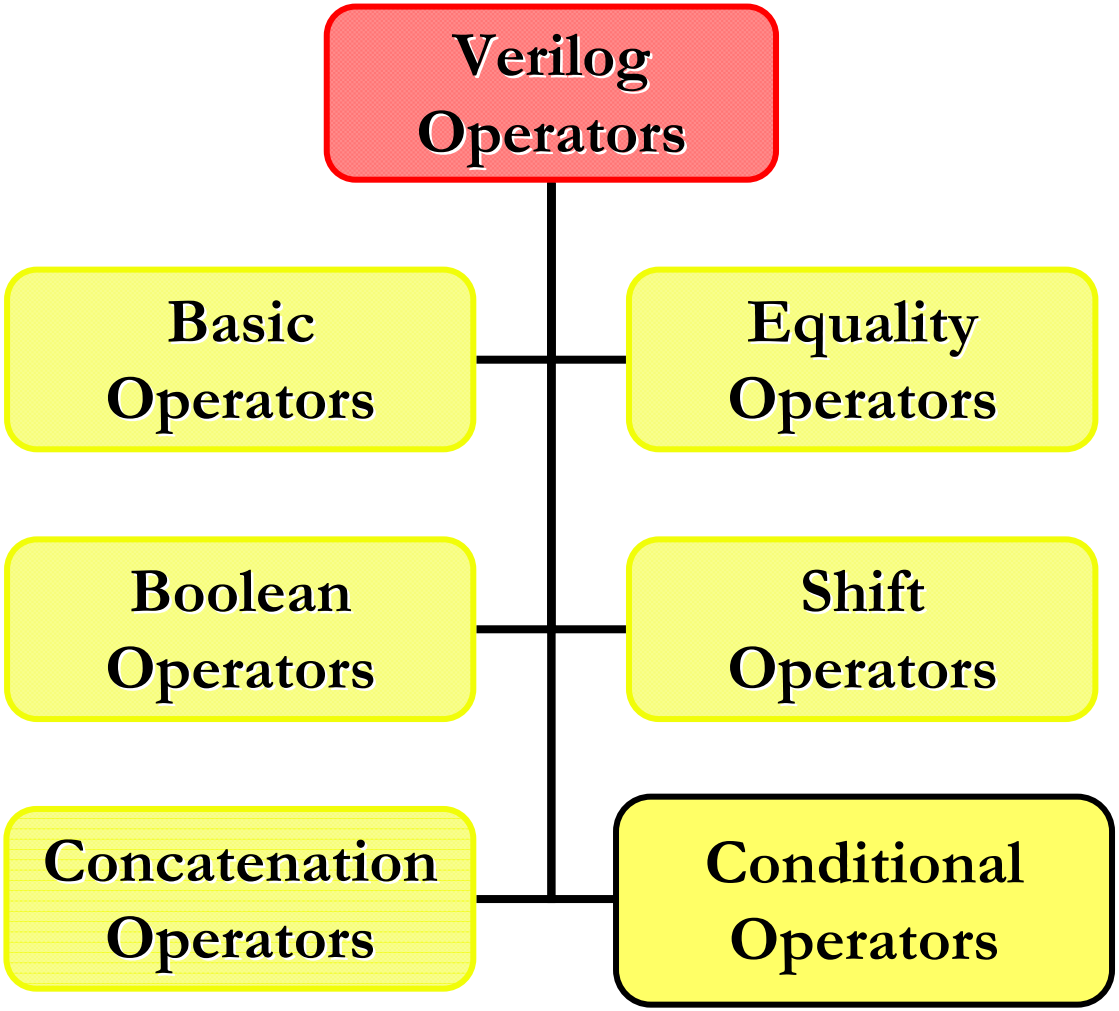
  puts 001001_1101_1101_11 in aaa.

- `{a, 2{b,c}, 3{d}}` is equivalent to:

  {a, b, c, b, c, d, d, d}

- `{2'b00, 3{2'01}, 2'b11}` results in: 10'b0001010111

Conditional Operators



Conditional Operators

- expression1 ? expression2 : expression3

If *expression1* is true, then *expression2* is selected as the result of the operation; otherwise *expression3* is selected.

```
…
assign a = (b == c)? 1 : 0;
…
```

- If *expression1* is X or Z, both expressions 2 and 3 will be evaluated, and the result becomes the bit-by-bit combination of these two expressions.

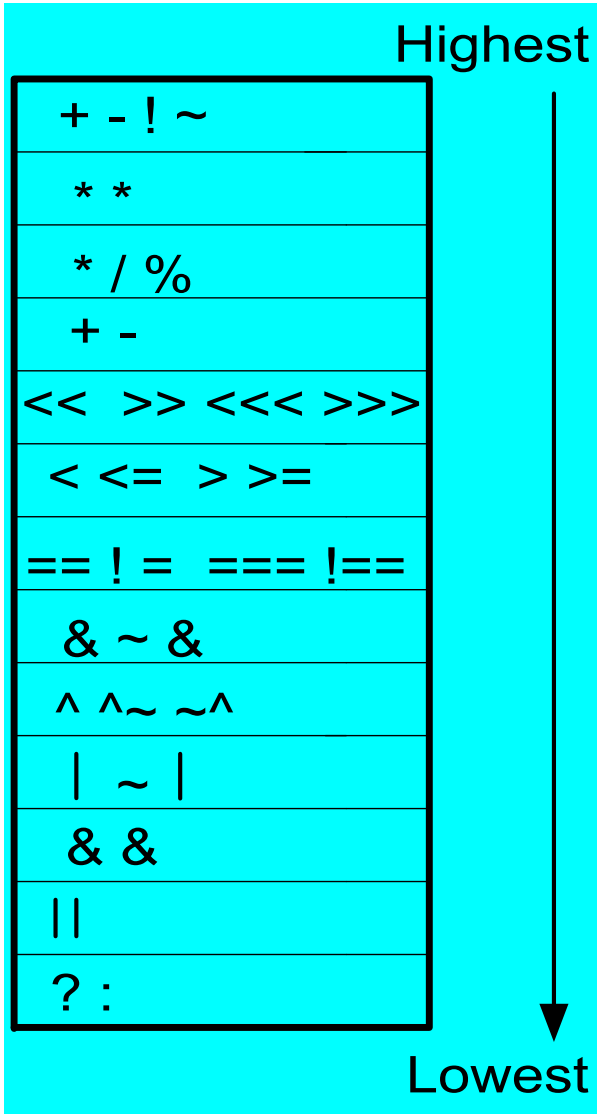
Conditional Operators

Example	Results in
1 ? 4'b1100 : 4'b|ZX0	4'b1100
0 ? 4'b1100 : 4'b1ZX0	4'b1ZX0
X ? 4'b1100 : 4'b1ZX0	4'b1XX0

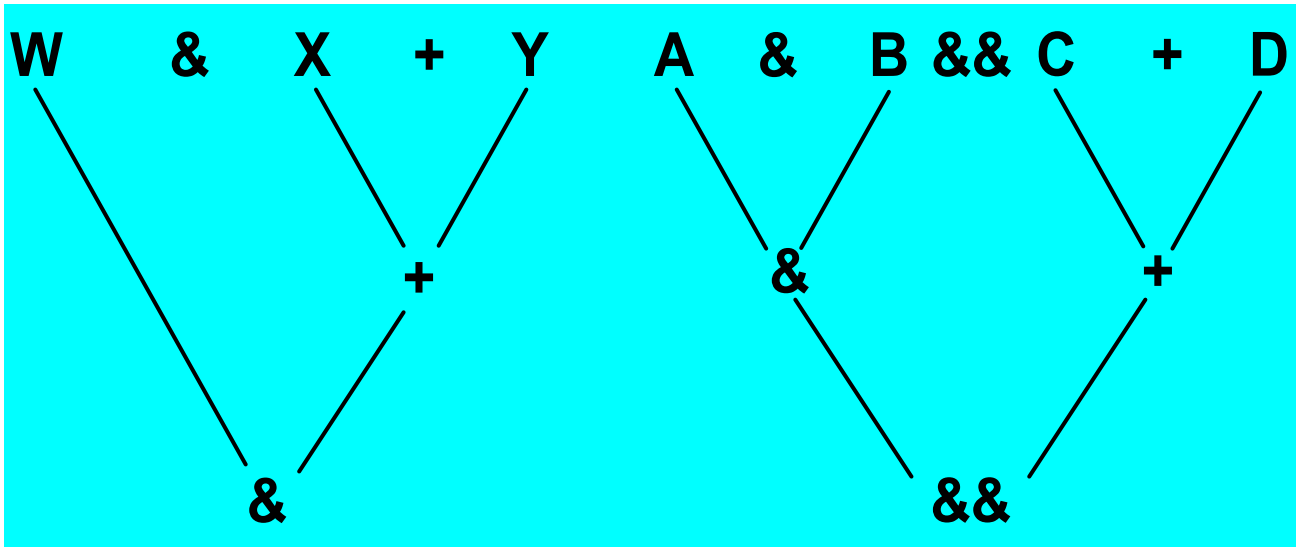
- **Conditional Operators**

Precedence of Operators

- **Operator Precedence**



Precedence of Operators



- **Precedence Examples**

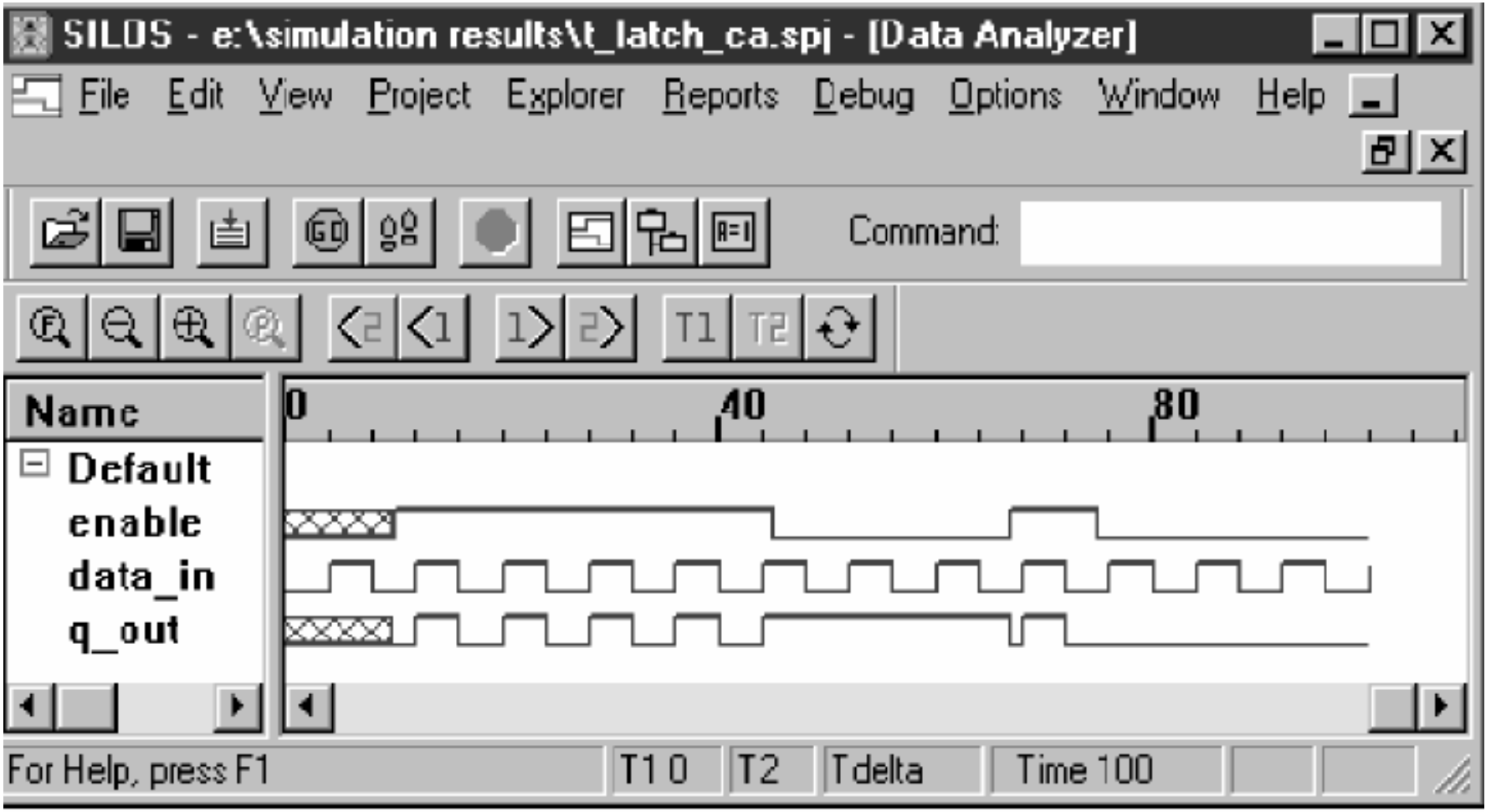
5.5 Latches and Level Sensitive circuits in Verilog

- A set of continuous-assignment statements has implicit feedback if a variable in one of the RHS expressions is also the target of an assignment
- Synthesis tool do not accommodate this form of feedback
- They support the feedback that is implied by a continuous assignment in which the RHS uses a conditional operator

Example 5.7
A Transparent Latch

```
module  Latch_CA (q_out, data_in, enable);
   output      q_out;
   input       data_in, enable;
   assign      q_out=enable? data_in : q_out;
endmodule
```

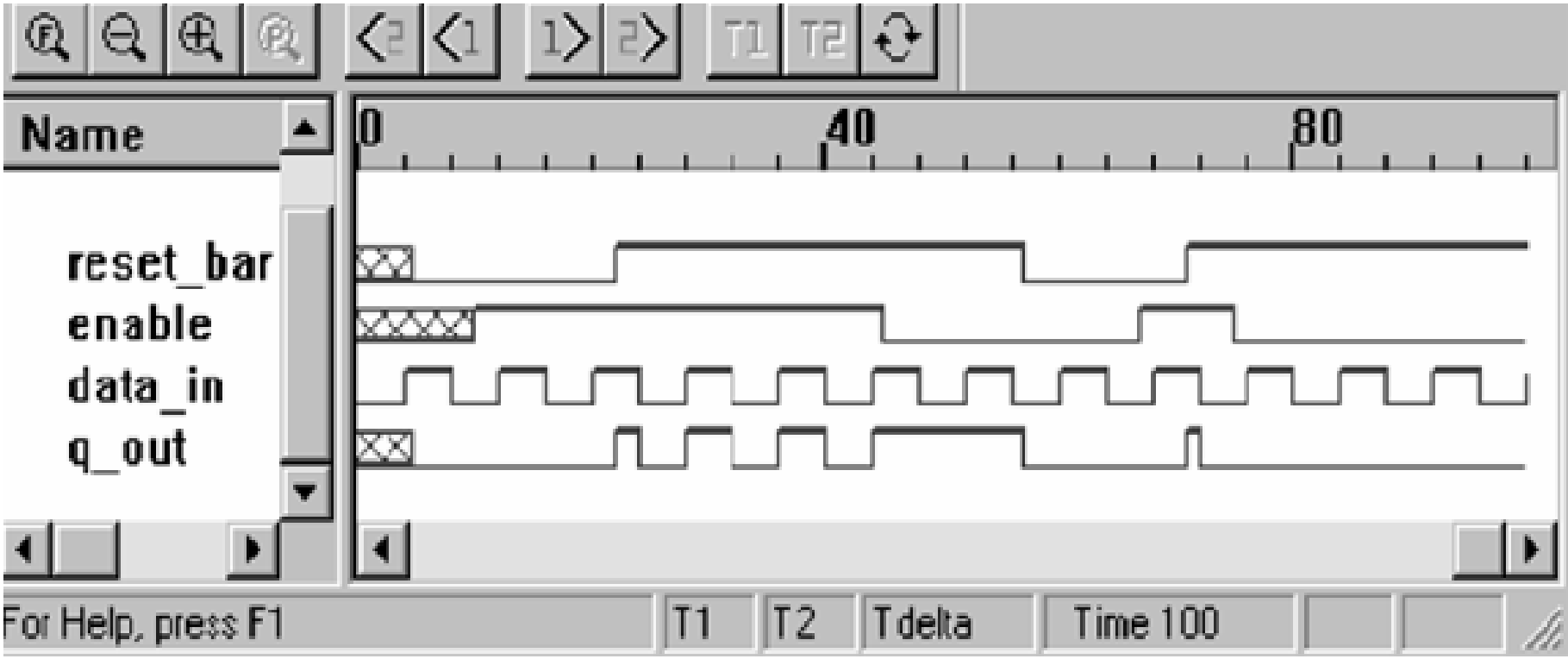
The waveforms produced by simulation of Latch_CA



Example 5.8 a latch with a nested conditional operator

```
module Latch_Rbar_CA (q_out, data_in, enable, reset);
   output    q_out;
   input     data_in, enable, reset;
   assign    q_out= !reset ? 0 : enable ? Data_in : q_out;
endmodule
```

The simulation of the latch model Latch_Rbar_CA



Use Continuous assignments

- Convenient for modeling small Boolean expressions, three-state behavior, and transparent latches
- Easy to make mistake when design large Boolean equation models
- Boolean expressions might obscure the functionality of the design

5.6 Cyclic Behavioral Models of Flip-Flops and Latches

- Many digital systems are triggered by a clock
- Verilog uses a cyclic behavior to model edge-sensitive functionality
- **Cyclic behaviors execute procedural statements** to generate the values of variables
- Cyclic behaviors are used to model both level-sensitive and edge-sensitive behavior

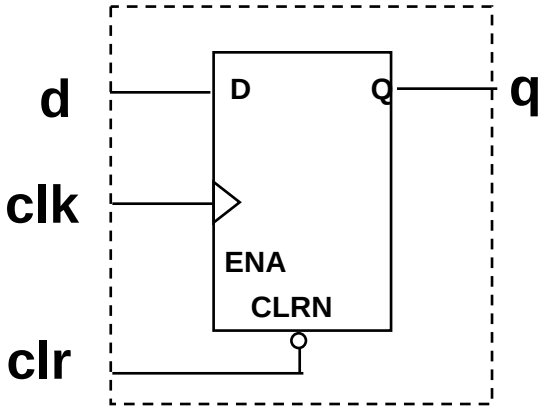
Example 5.9

```
module df_behav (q, q_bar, data, set, reset, clk);
  input       data, set, clk, reset;
  output      q, q_out;
  reg         q;
  assign q_bar=~q;
  always@ (posedge clk)  // flip-flop with synchronous set/reset
  begin
    if (reset==0) q<=0;
      else if (set==0) q<=1;
              else q<=data;
  end
endmodule
```

Clocked Process

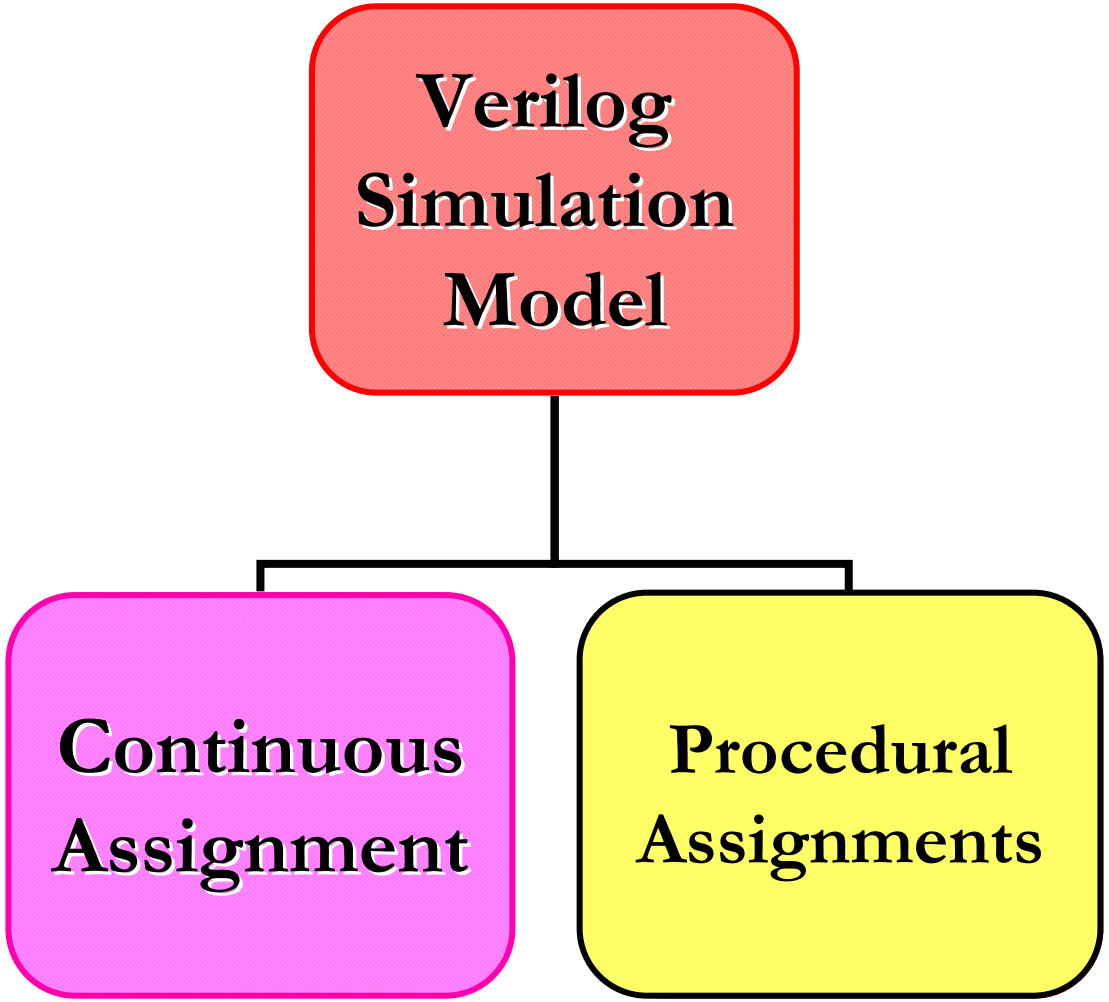
- **Clocked Process**
  - Sensitive to a clock or/and control signals
- **Example**

always @(posedge clk **or negedge** clr)



sensitivity list does not include the **d** *input, only the clock or/and control signals*

Procedural Assignments



Procedural Assignments

- **Procedural assignments in Verilog take place in the initial and always procedural constructs, which are regarded as procedural bodies.**

Initial and Always Blocks

- Basic components for behavioral modeling

```
initial
  begin
    … imperative statements …
  end
```

Runs when simulation starts

Terminates when control reaches the end

Good for providing stimulus

```
always
  begin
    … imperative statements …
  end
```

Runs when simulation starts

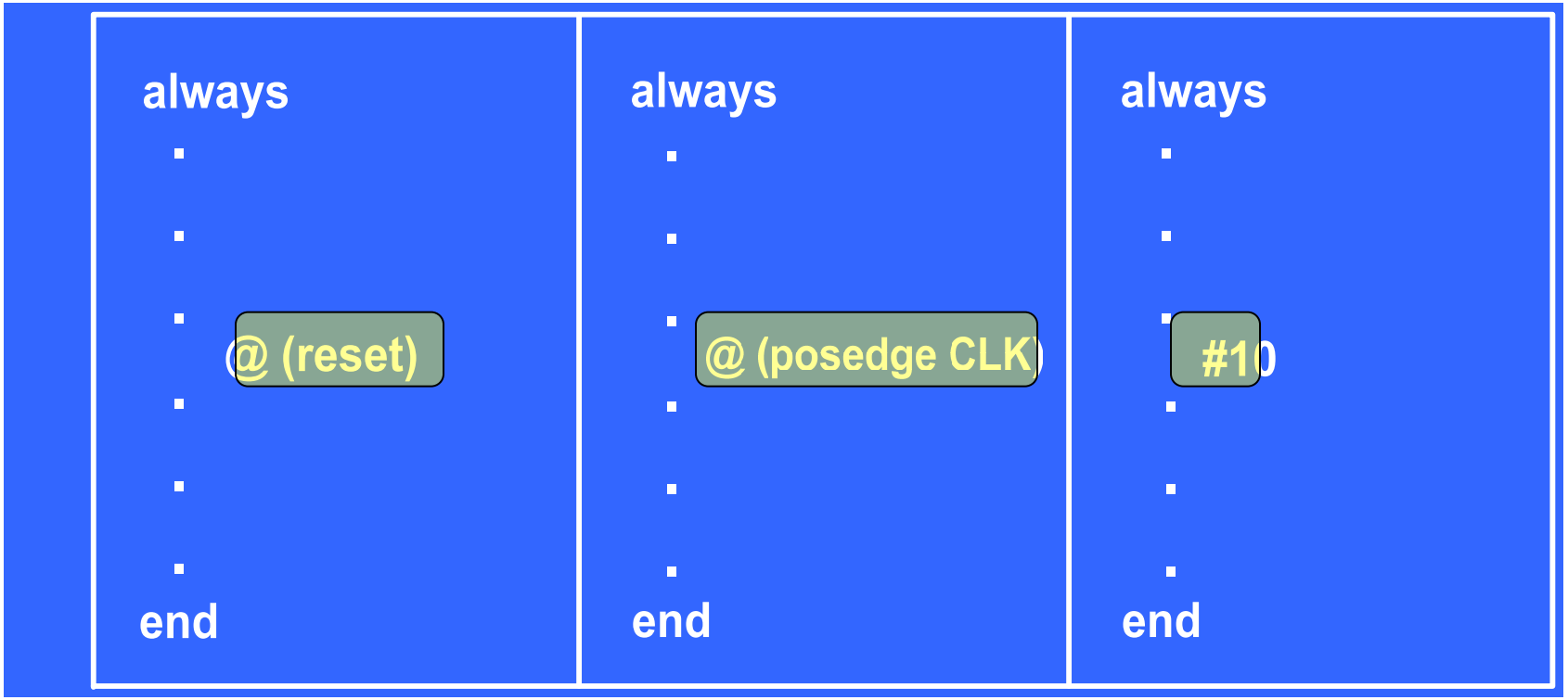
Restarts when control reaches the end

Good for modeling/specifying hardware

Procedural Flow Control

- **Statements in a procedural body are executed when program flow reaches them.**
- **Flow control statements are classified as delay control and event control.**
- **An event or delay control statement in a procedural body causes program flow to be put on hold temporarily.**

Procedural Flow Control



- **Procedural Flow Control**

Timing Control

- **DelayControl**
- **EventControl**
- **WaitControl**
- **Example**
  - #10
  - #(Period/2)
  - #(1.2:3.5:7.1)
  - @Trigger
  - @(a or b or c)
  - @(posedge clock or negedge reset)
  - wait (!Reset)

Wait : Timing Control

- Wait — waits for a level on a line
  - How is this different from an "@" ?
- Semantics
  - wait (expression) statement;
    - e.g. wait (a == 35) q = q + 4;
  - if the expression is FALSE, the process is stopped
    - when *a* becomes 35, it resumes with q = q + 4
  - if the expression is TRUE, the process is not stopped
    - it continues executing
- Partial comparison to @ and #
  - @ and # always "block" the process from continuing
  - wait blocks only if the condition is FALSE

Operator: @

- **@ denotes event control:**

  meaning that the procedural statements that follow the event-control expression do not execute **until an activating event occurs**
- **When such an event occurs, the statements execute in sequence, top to bottom**
- When the last statement completes execution, the computational activity returns to the location of the keyword ***always***, then the cycle repeats

Initial and Always

- **Run until they encounter a delay**

```
initial
begin
  #10 a = 1; b = 0;
  #10 a = 0; b = 1;
end
```

- **Wait for an event**

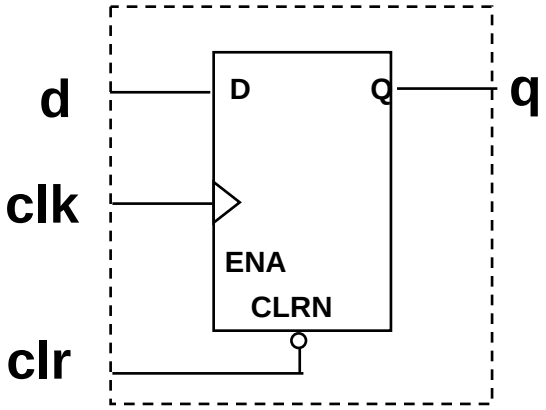
```
always @(posedge clk) q = d;

always
 begin wait(i); a = 0; wait(~i); a = 1; end
```

5.7 Cyclic Behavior and Edge Detection

- **delay control(# operator)**
- **event-control (@ operator)**
- **rising edge(*posedge*)**
- **falling edge(*negedge*)**
- A simulator monitors the variables in an event-control expression
- When the expression changes value, the associated procedural statements execute if the enabling change took place

Example 5.10

```
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

- Verilog language allows a mixture of level-sensitive and edge-qualified variables in an event-control expression
- Synthesis tools do not support such models of behavior
- That description must be entirely edge-sensitive or entirely level-sensitive

Example 5.11

```
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
```

5.8 A Comparison of styles for Behavioral Modeling

- Modeling styles based on :
- 1. Continuous-assignment models
- 2. Dataflow/RTL models
- 3. Algorithm-based models

5.8.1 Continuous-Assignment Models

- Continuous assignments
- Execute concurrently
  - –with each other assignments
  - –with gate-level primitives ,and
  - –with all of the behaviors in a description

Example 5.12

```
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
```

Note that the gate-level implementation is not apparent.

Example 5.13
2-bit comparator

```
module
      compare_2_CA2(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);
  assign           A_gt_B=(A>B);
  assign           A_eq_B=(A==B);
endmodule
```

Example 5.14
A 32-bit comparator

```
module compare_32_CA (A_lt_B,A_gt_B,A_eq_B,A,B);
  parameter                word_size=32;
  input [wordsize-1:0]     A,B;
  output                   A_lt_B, A_gt_B, A_eq_B;
  assign                   A_lt_B=(A<B);    //note:list of multiple
                                            // assignment
  assign                   A_gt_B=(A>B);
  assign                   A_eq_B=(A==B);
endmodule
```

5.8.2 Dataflow/RTL Models

- Describe **concurrent operations on signals**
- Usually in a synchronous machine
- Where computations are initiated at the active edges of a clock ,and are completed in time to be stored in a register at the next active edge

Dataflow : Key Points

- **Dataflow modeling enables the designer to focus on where the state is in the design and how the data flows between these state elements without becoming bogged down in gate-level details**
- **– Continuous assignments are used to connect combinational logic to nets and ports**
- **– A wide variety of operators are available**

- **But: Arithmetic: - * / % **…..**
- **Avoid these operators since they usually synthesize poorly**

Example 5.15
2-bit comparator of RTL model

```
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
```

the order of three procedural assignments which the assignments are listed does not affect the outcome.

Example 5.16
a 4-bit serial shift register

```
module shiftreg_PA(E ,A, B, C, D, 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;
   B=C;
   C=D;
   D=E;
 end
end
endmodule
```

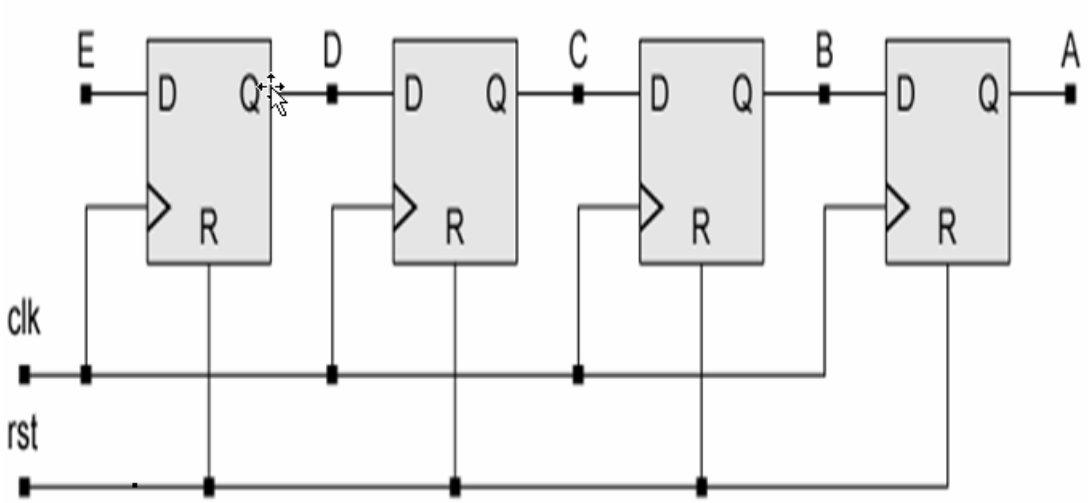
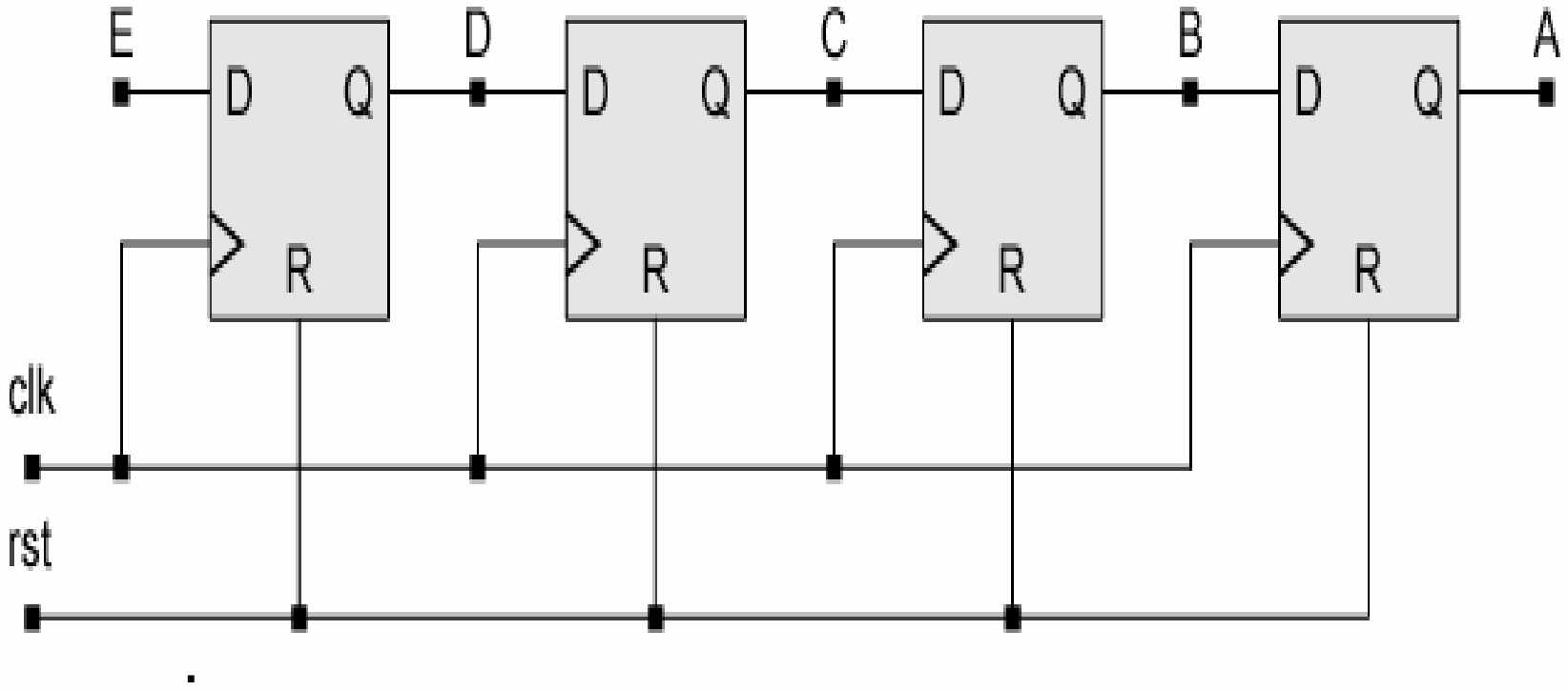
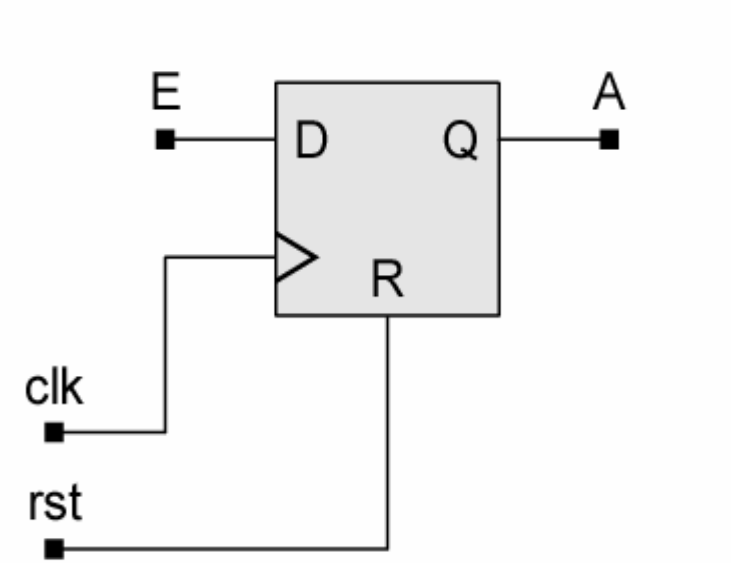


Fig 5-7
The shift register by a synchronous cyclic behavior with a list of procedural assignments



When the order of the procedural assignments in the model is reversed

```
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
```



Two types of Procedural Assignments

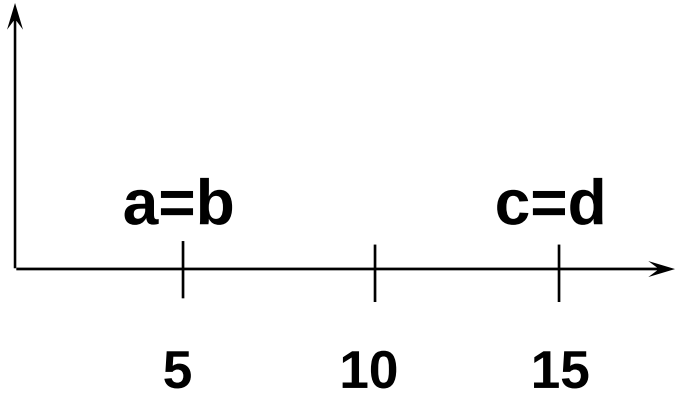
- Blocking Assignment (=) : executed in the order they are specified in a sequential block

- Nonblocking Assignment (<=) : allow scheduling of assignments without blocking execution of the statements that follow in a sequential block
  - Recommended: **Use Nonblocking assignments for clocked processes when writing synthesizable code.**

Blocking vs. Nonblocking Assignments

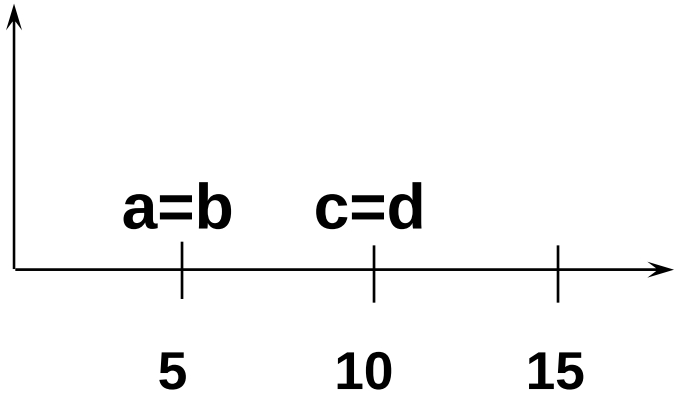
Blocking (=)

```
initial
   begin
         #5      a = b;
         #10     c = d;
   end
```



Nonblocking (<=)

```
initial
   begin
         #5      a <= b;
         #10     c <= d;
   end
```

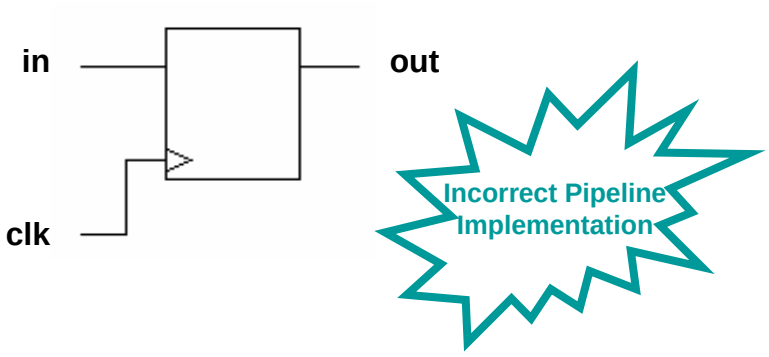


Blocking vs. Nonblocking Assignments

Blocking (=)

```
always @(posedge clk)
   begin
         a = in;
         b = a;
         out = b;
   end
```

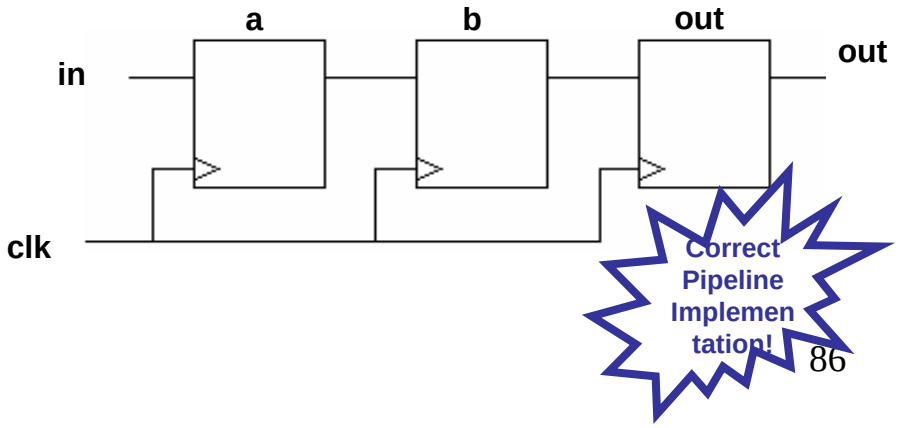
Synthesized Result:



Nonblocking (<=)

```
always @ (posedge clk)
   begin
         a <= in;
         b <= a;
         out <= b;
   end
```

Synthesized Result:



Non-blocking Can Behave Oddly

- A sequence of non-blocking assignments don't communicate

```
a = 1;
b = a;
c = b;
```

```
a <= 1;
b <= a;
c <= b;
```

Blocking assignment:

a = b = c = 1

Non-blocking assignment:

a = 1

b = old value of a

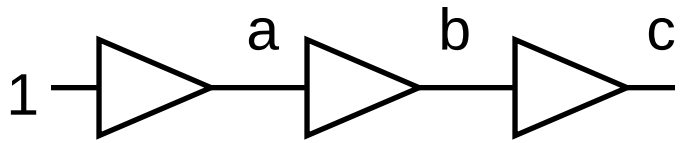
c = old value of b

Non-blocking Looks Like Latches

- RHS of blocking taken from wires
- RHS of non-blocking taken from latches

```
a = 1;
b = a;
c = b;
```

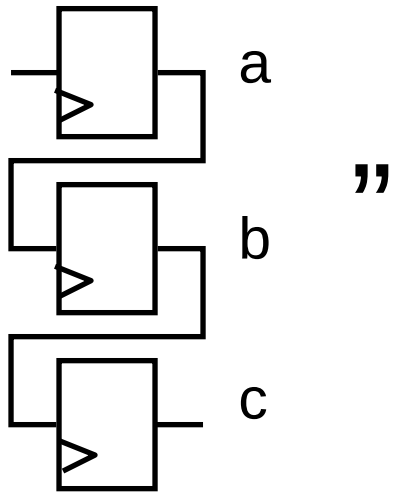
“



”

```
a <= 1;
b <= a;
c <= b;
```

“

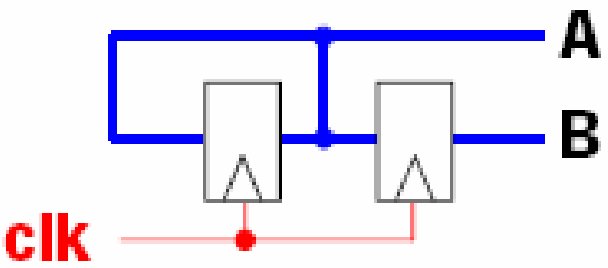


”

Blocking & Non-blocking Assignments

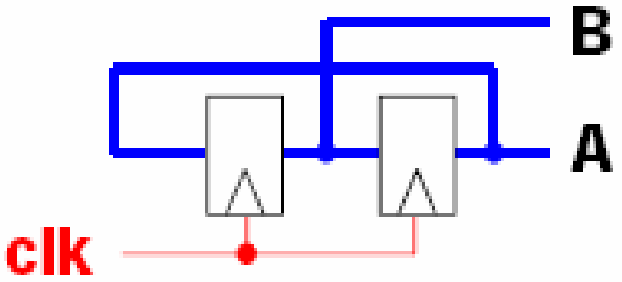
Bad: Circuit from blocking assignment.

```
always @(posedge clk)
begin
    b=a;
    a=b;
end
```



Good: Circuit from nonblocking assignment.

```
always @(posedge clk)
begin
    b<=a;
    a<=b;
end
```



Ex. A Flawed Shift Register

- This doesn't work as you'd expect:

```
reg d1, d2, d3, d4;
always @(posedge clk) d2 = d1;
always @(posedge clk) d3 = d2;
always @(posedge clk) d4 = d3;
```

- **These run in some order, but you don't know which**

Non-blocking Assignments

- This version does work:

```
reg d1, d2, d3, d4;

always @(posedge clk) d2 <= d1;
always @(posedge clk) d3 <= d2;
always @(posedge clk) d4 <= d3;
```

Non-blocking rule:

RHS evaluated when assignment runs

LHS updated only after all events for the current instant have run

Example 5.17 4-bit serial shift register

```
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
    D<=E;          // D<=E;
    C<=D;          // C<=D;
    B<=C;          // B<=C;
    A<=B;          // A<=B;
  end
 end
endmodule
```

Strongly Recommended

- When modeling logic that includes edge-driven register transfers
- **The edge-sensitive operations be described by non-blocking assignments**
- **The combinational logic be described with blocked assignments**
- This practice will prevent race conditions between combinational logic and register operations

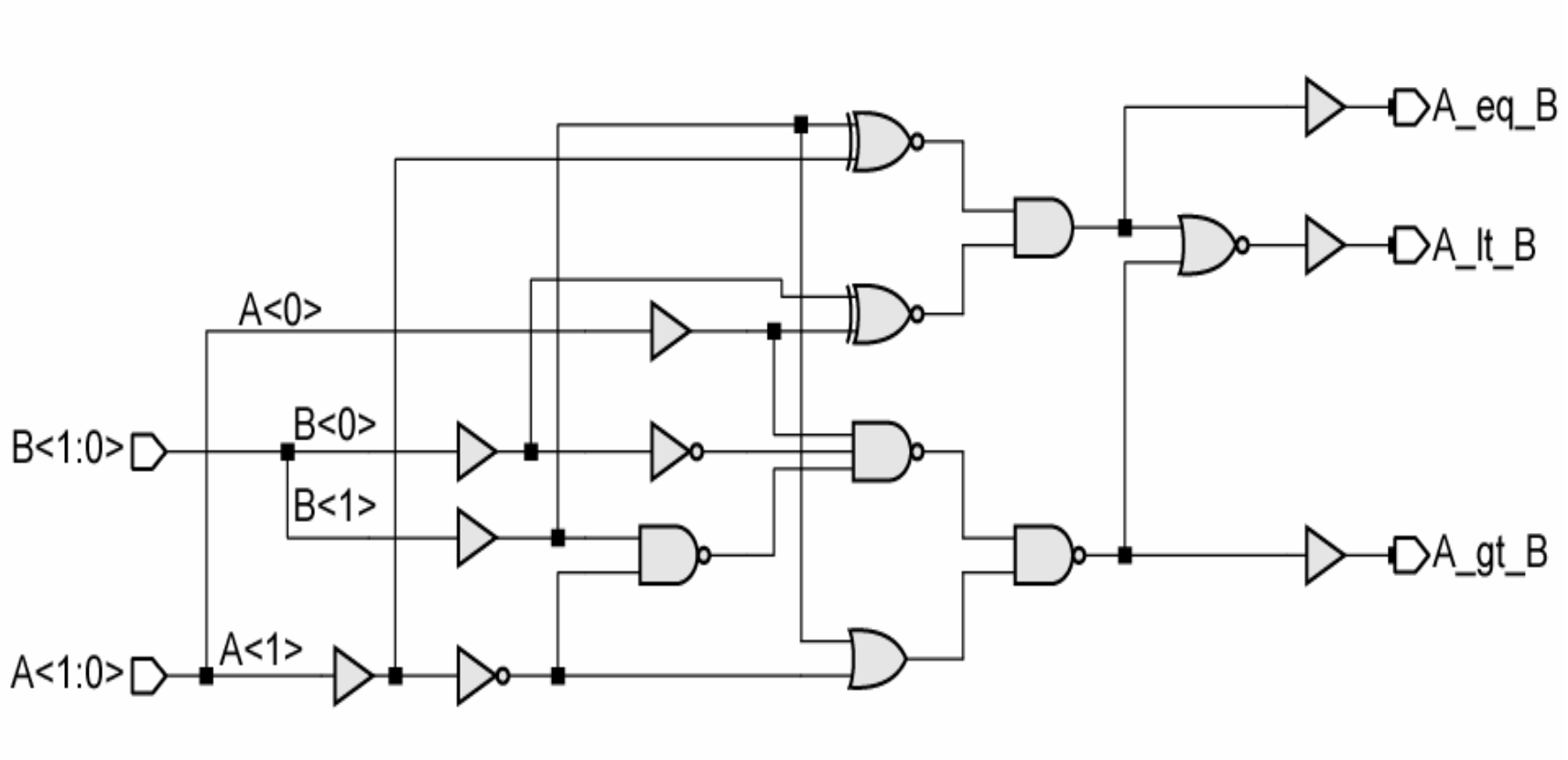
5.8.3 Algorithm-Based Models

- Algorithm prescribes a sequence of procedural assignments within a cyclic behavior
- The algorithm described by the model **does not have explicit binding to hardware,** and it **does not have an implied architecture of registers, datapaths, and computational resources**.
- **Not all algorithms can be implemented in hardware**
- This descriptive style is abstract, and eliminates the need for an a priori architecture
- Description can be readable and understandable

Example 5.18
a 2-bit comparator

```
module compare_2_algo(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;
  reg             A_lt_B, A_gt_B, A_eq_B;
 always@ (A or B)      //Level-sensitive behavior
   begin
    A_lt_B=0;              // prevent the synthesis of unwanted latches
    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
```

Synthesis result :the gate-level schematic obtained by synthesizing compare_2_algo



5.8.4 Port Names: A Matter of style

- **Signals ,modules, functions and tasks** should be given names that describe their use the encapsulated functionality
- **The ports will be ordered in the following sequence:**
- (datapath bidirectional signals,
  bidirectional control signals,
  datapath outputs, control outputs,
  datapath inputs, control inputs,
  synchronizing signals)

Ports

- **Port List:**
  - A listing of the port names
  - Example: `module mult_acc (out, ina, inb, clk, clr);`
- **Port Types:**
  - `input` --> input port
  - `output` --> output port
  - `inout` --> bidirectional port
- **Port Declarations:**
  - `<port_type> <port_name>;`
  - Example:

```
input [7:0] ina, inb;
input clk, clr;
output [15:0] out;
```

5.8.5 Simulation with Behavioral Models

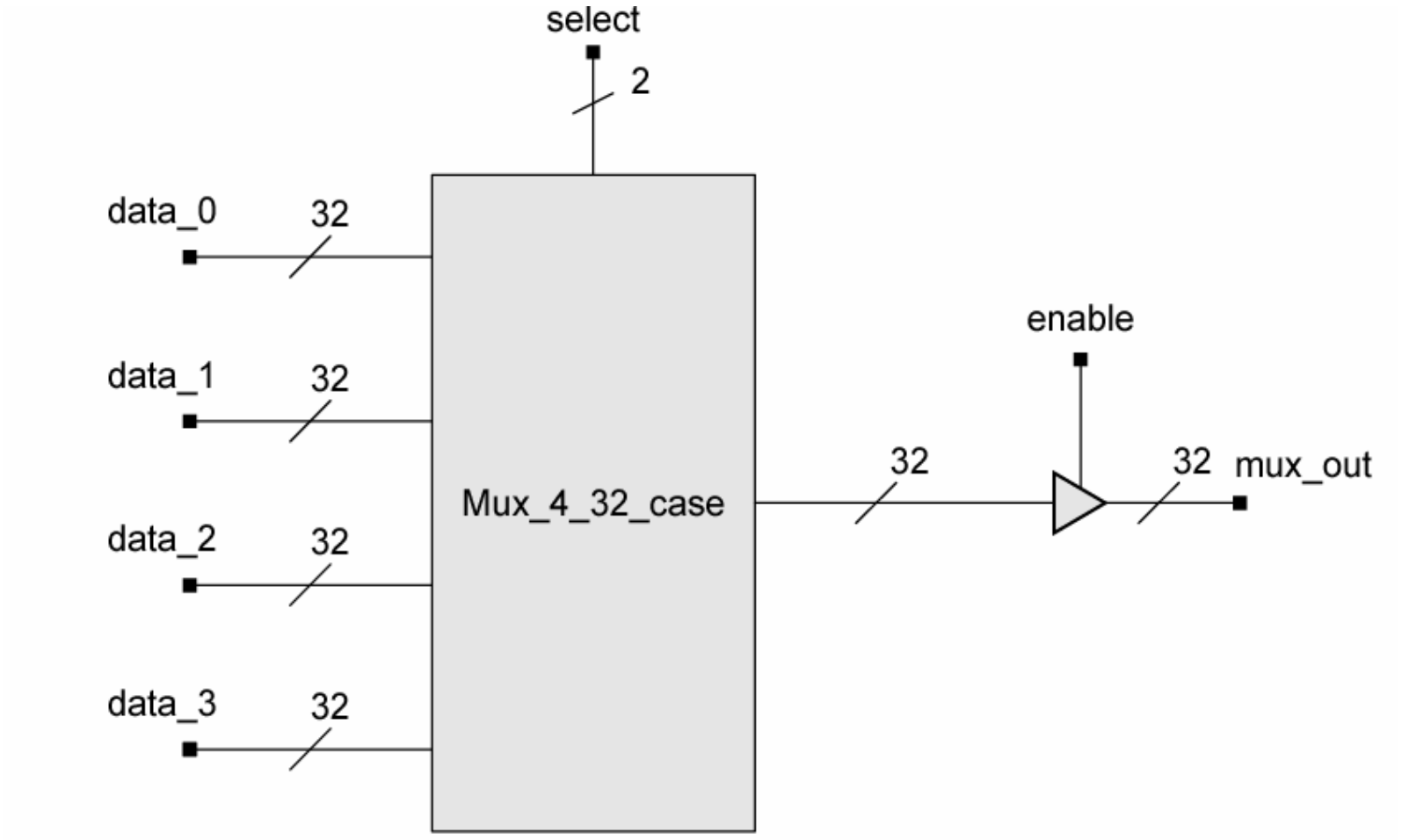
- An event at the input of a primitive causes the simulator to schedule an updating event for its output
- An event in the RHS expression of a continuous-assignment statement causes the scheduling of an event for the assignment's target variable
- Cyclic behaviors can suspend themselves

  prevent endless execution or multiple behaviors at the same time

Procedural Control

- The associated statements execute sequentially, in the same time step, until the simulator encounters either
- (1) a delay control operator (#)
- (2) an event control operator (@)
- (3) a wait construct
- (4) the last statement of the behavior

5.9 Behavioral Models of Multiplexers, Encoders, and Decoders



Ex 5.19: a 4-channel,32-bit multiplexer

```
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
```

case statement

- Similar to its counterpart in other languages (eg., the switch statement in C)
- Searches from top to bottom to find a match
- The case statement executes the first match found
- And does not consider any remaining possibilities

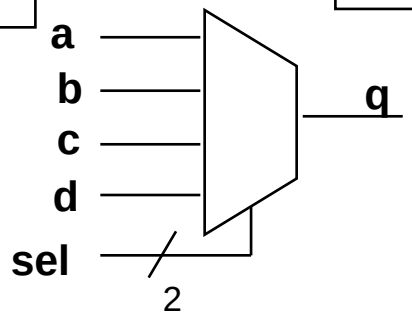
Case Statement

- Format:

```
case (expression)
    <condition1> :
            sequence of statement(s)
    <condition2> :
            sequence of statement(s)
                    .
                    .
     default :
            sequence of statement(s)
 endcase
```

- Example:

```
always @(sel or a or b or c or d)
    begin
            case (sel)
                    2'b00 :
                            q = a;
                    2'b01 :
                            q = b;
                    2'b10 :
                            q = c;
                    default :
                            q = d;
            endcase
    end
```



Efficient Case Statement

```
reg [4:0] state;
parameter     s0 = 5'b00000,        s1 = 5'b10001,
              s2 = 5'b10010,        s3 = 5'b10100,
              s4 = 5'b11000;
always @ (posedge clk or negedge reset) begin
  if (reset == 0) state = s0;
  else
     case (state)
       s0:        if (in == 1) state = s1;
                  else state = s0;
       s1:        state = s2;
       s2:        state = s3;
       s3:        state = s4;
       s4:        if (in == 1) state = s1;
                  else state = s0;
       default:   state = s0;
     endcase
end
```

- Logic Elements: 10
- All cases covered
- Wherever mutual exclusivity exists, a **case** structure *usually* synthesizes most efficiently
- A **case** structure where **all cases are covered will typically synthesize most efficiently**

Two Other Forms of Case Statements

- **casez**
  - Treats all **'z'** values in the case conditions as don't cares, instead of logic values
  - All **'z'** values can also be represented by **'?'**
- **casex**
  - Treats all **'x'** and **'z'** values in the case conditions as don't cares, instead of logic values

```
casez (encoder)
        4'b1???  :  high_lvl = 3;
        4'b01??  :  high_lvl = 2;
        4'b001?  :  high_lvl = 1;
        4'b0001  :  high_lvl = 0;
        default  :  high_lvl = 0;
endcase
```

- if **encoder = 4'b1zzz**, then **high_lvl = 3**

```
casex (encoder)
        4'b1xxx  :  high_lvl = 3;
        4'b01xx  :  high_lvl = 2;
        4'b001x  :  high_lvl = 1;
        4'b0001  :  high_lvl = 0;
        default  :  high_lvl = 0;
endcase
```

- if **encoder = 4'b1xzx**, then **high_lvl = 3**

Example 5.20 use nested conditional statements (if) to model a multiplexer

```
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
```

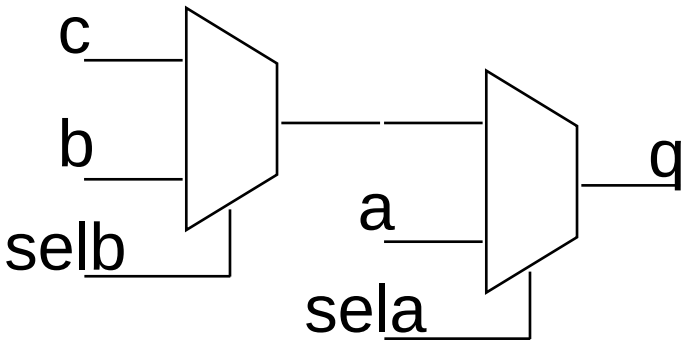
If-Else Statements

- Format:

```
if  (<condition1>)
        sequence of statement(s)
else
 if  (<condition2>)
        sequence of statement(s)
                .
                .
else
        sequence of statement(s)
```

- Example:

```
always @(sela or selb or a or b or c)
        begin
                if (sela)
                        q = a;
                else
                  if (selb)
                        q = b;
                  else
                        q = c;
        end
```



Ex 5.21 use *?* *:* operator to model a multiplexer

```
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;
  reg    [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'bz;
endmodule
```

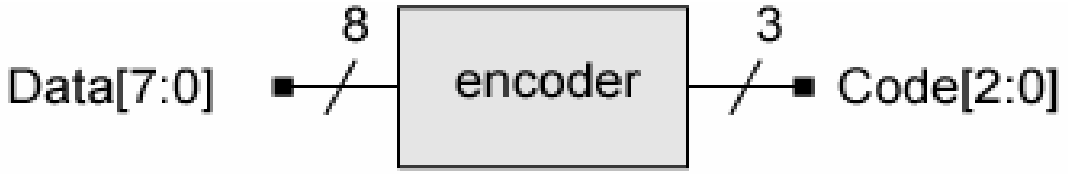
Ex 5.22 an 8:3 encoder

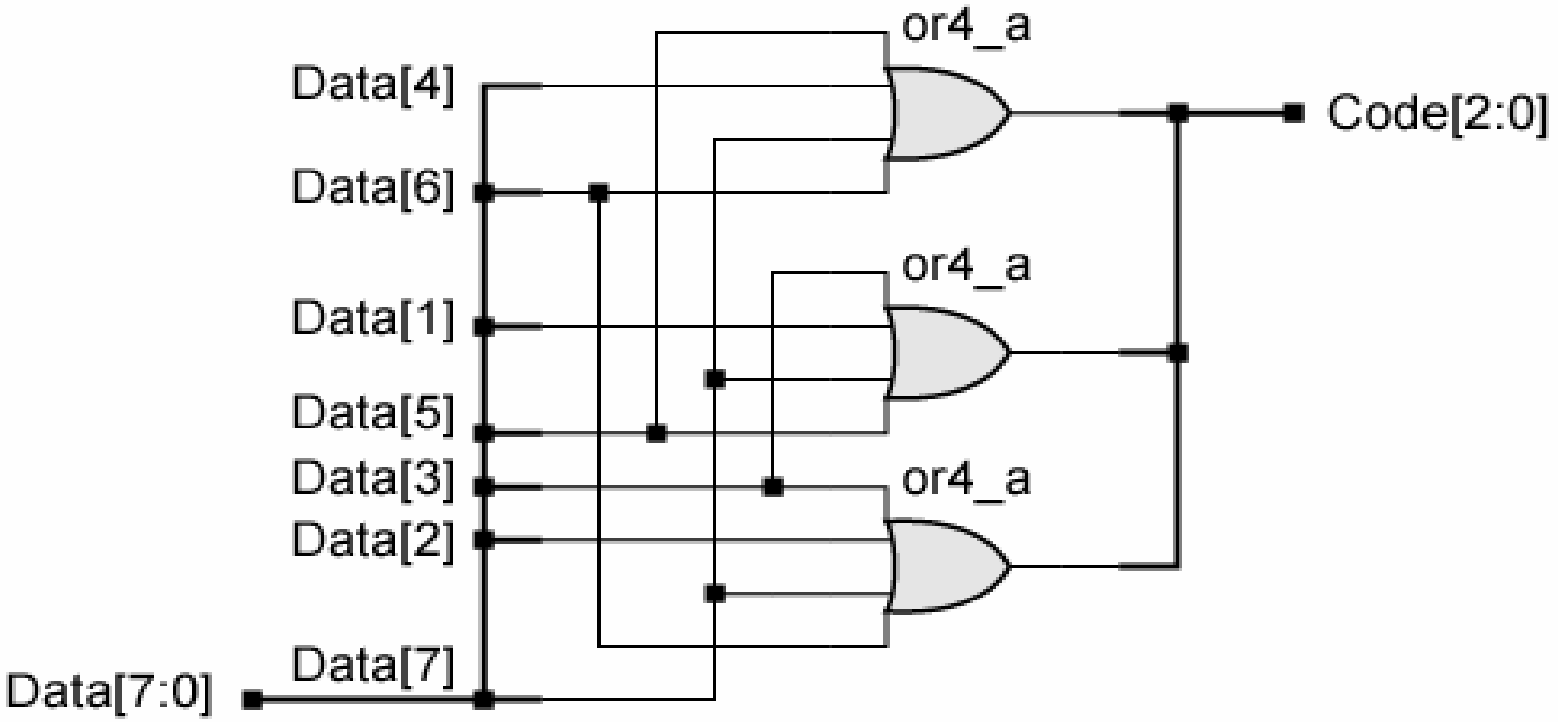
```
module encoder (Code, Data);
    output      [2:0] Code;
    input       [7:0] Data;
    reg [2:0] Code;
    always @ (Data)
     begin
       if (Data==8'b00000001) Code=0; else
       if (Data==8'b00000010) Code=1; else
       if (Data==8'b00000100) Code=2; else
       if (Data==8'b00001000) Code=3; 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
endmodule
```

Alternative description of an 8:3 encoder

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

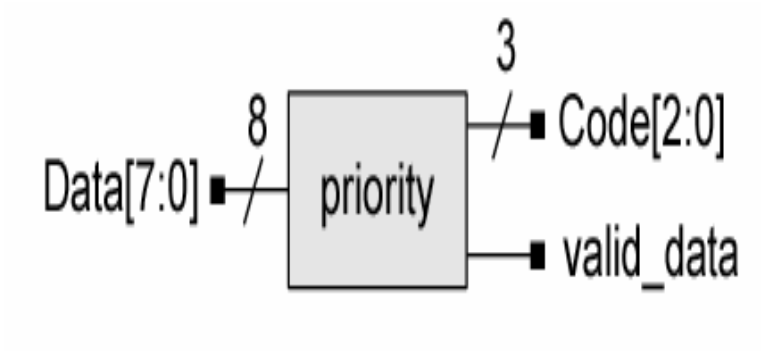
The 8:3 encoder's the synthesis result





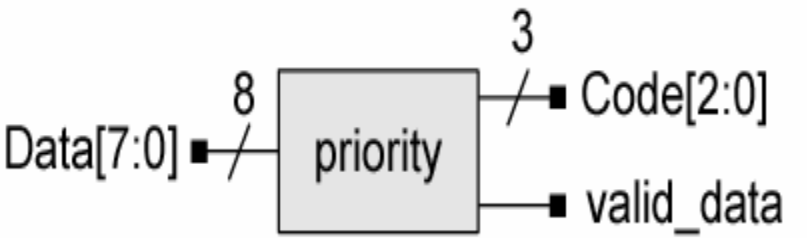
Ex 5.23 an 8:3 priority encoder

```
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;
 // The reduction operators (&  ~&  |  ~|  ^  ~^  ^~)
//  reduce a vector to a scalar value
    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
endmodule
```

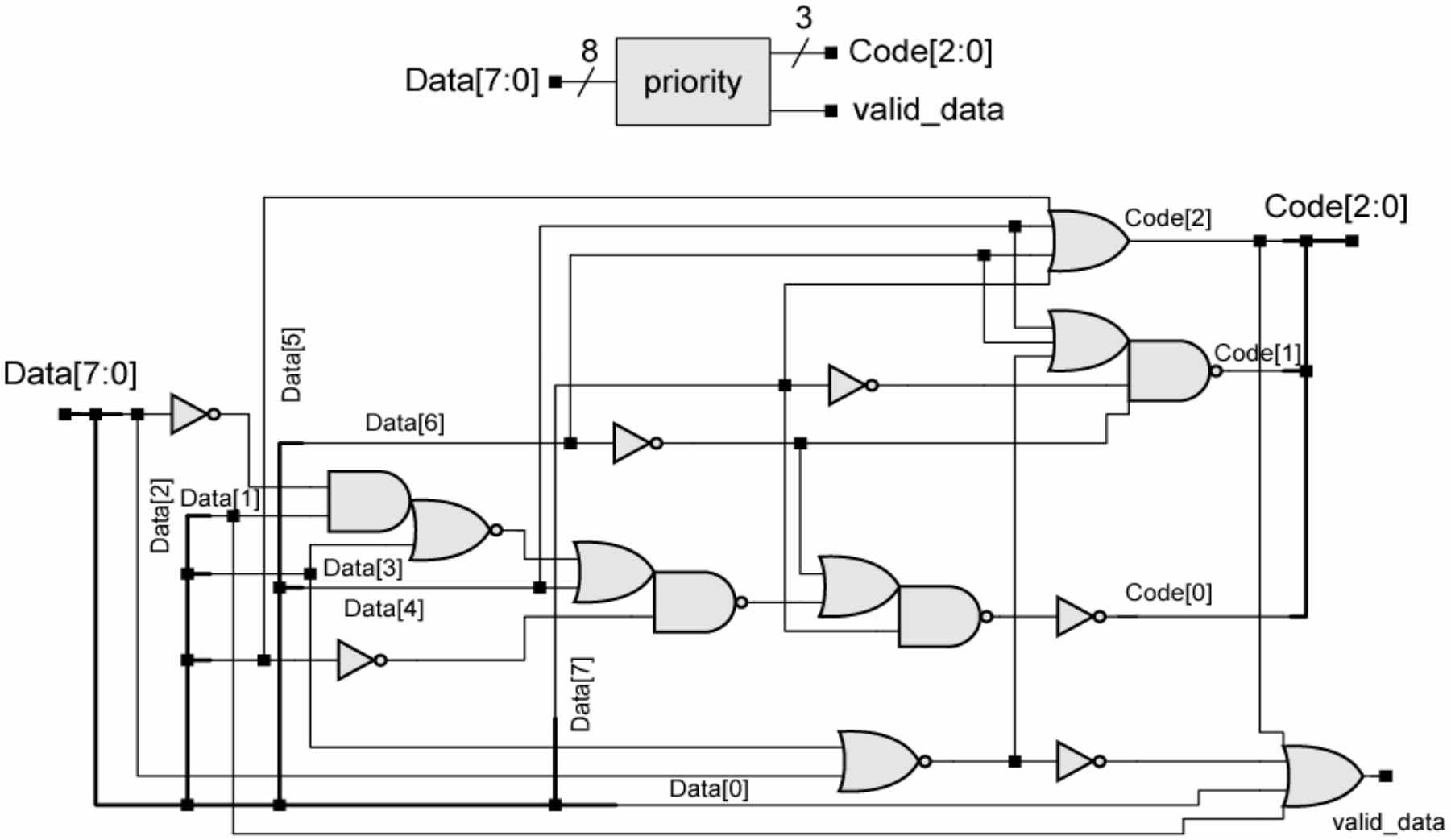


Ex 5.23 an 8:3 priority encoder (Alternative description)

```
/*// is give 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
*/
```



synthesis circuit



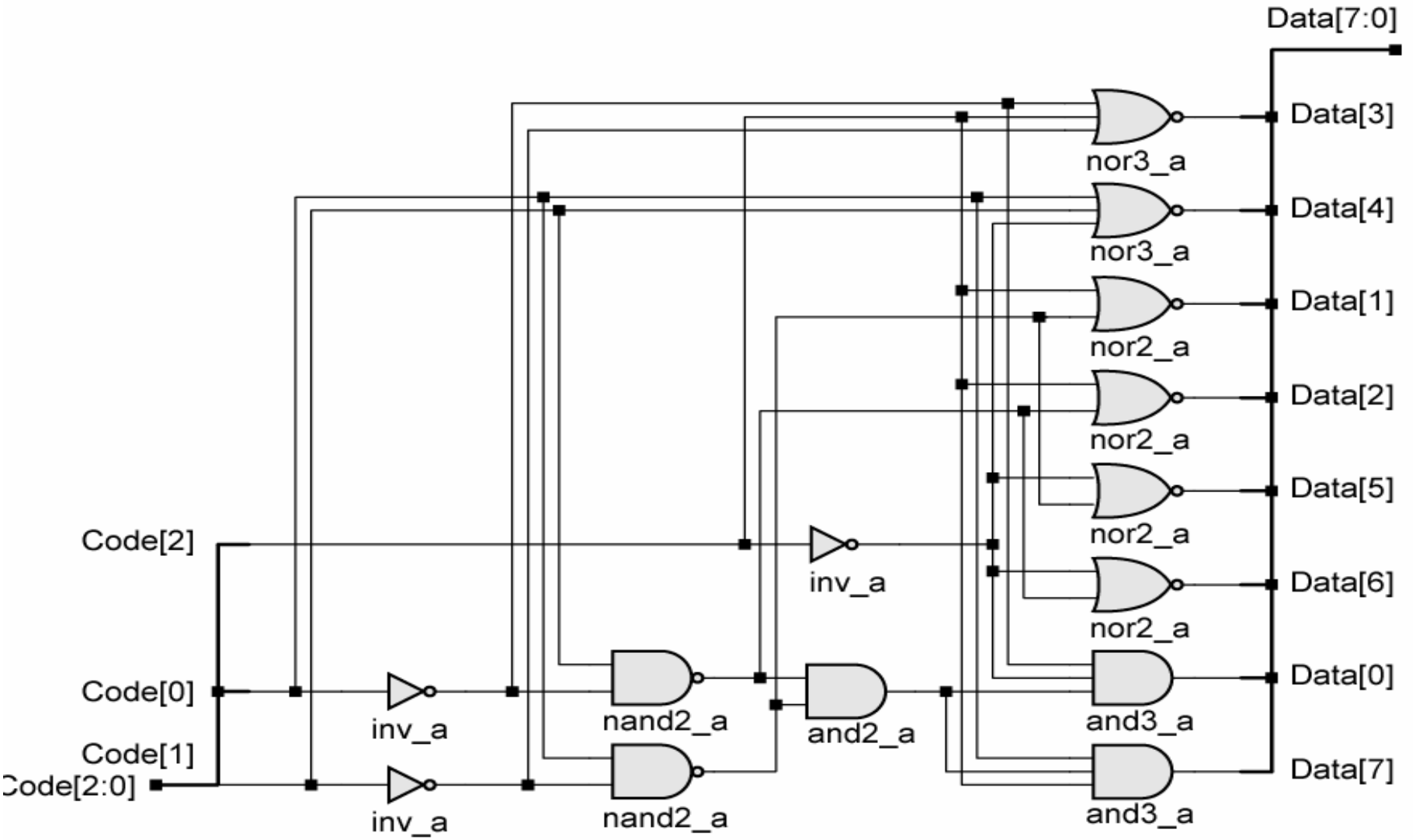
Ex 5.24 a 3:8 decoder

```
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
endmodule
```

Example 5.24 a 3:8 decoder

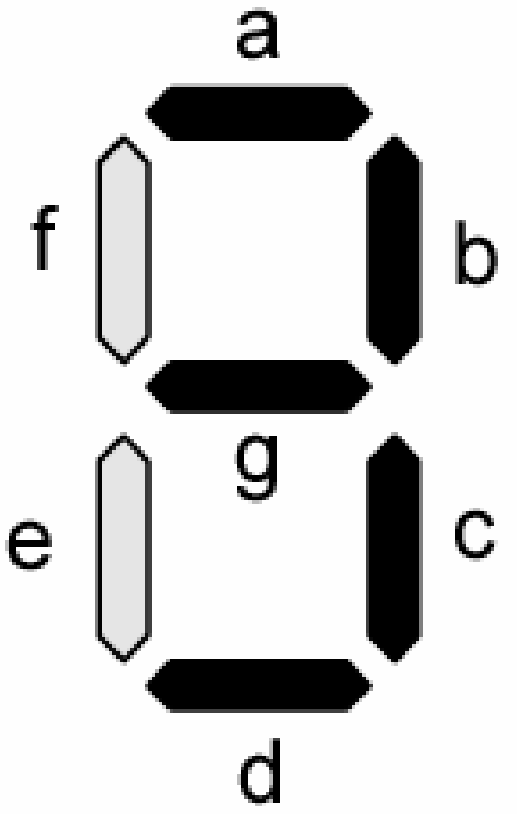
```
/* Alternative description is given below
 always @ (Code)
   case (Code)
        0: Data=8'b00000001;
        1: Data=8'b00000010;
        2: Data=8'b00000100;
        3: Data=8'b00001000;
        4: Data=8'b00010000;
        5: Data=8'b00100000;
        6: Data=8'b01000000;
        7: Data=8'b10000000;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

Synthesis circuit



Ex 5.25 The seven-segment light-emitting diode (LED) display

- Module **Seven_Seg_Display** accepts 4-bit words representing BCD digits and displays their decimal value
- Display has active-low illumination outputs
- It can be implemented with combinational logic
- The default assignment blanks the display for all unused codes

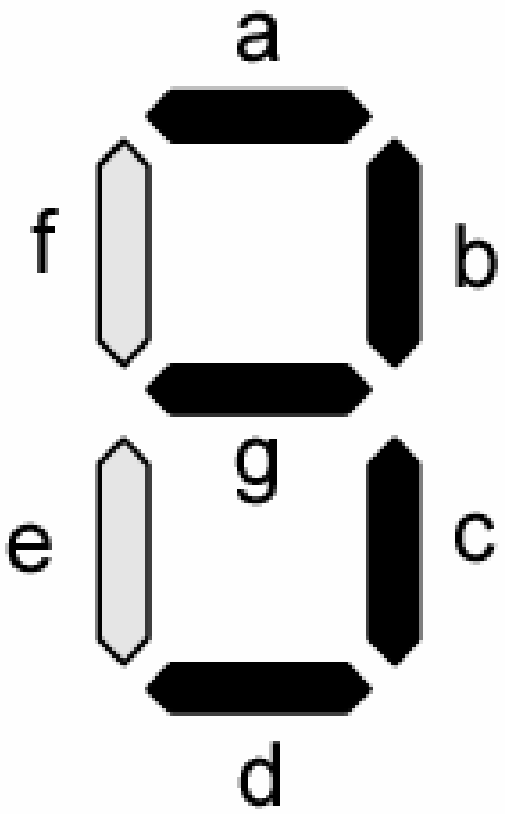


Example 5.25
The seven-segment light-emitting diode

```
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
```

Example 5.25 – cont.
The seven-segment light-emitting diode

```
always @ (BCD)
    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
```



Parameter

- Parameter - **assigning a value to a symbolic name**
- **Parameters are a means of giving names to constant values**
- **The values of parameters can be overridden when a design is compiled** (but not during simulation), thus enabling parameterization of bus widths etc.

```
parameter size = 8;
reg [size-1:0] a, b;
```

Parameters

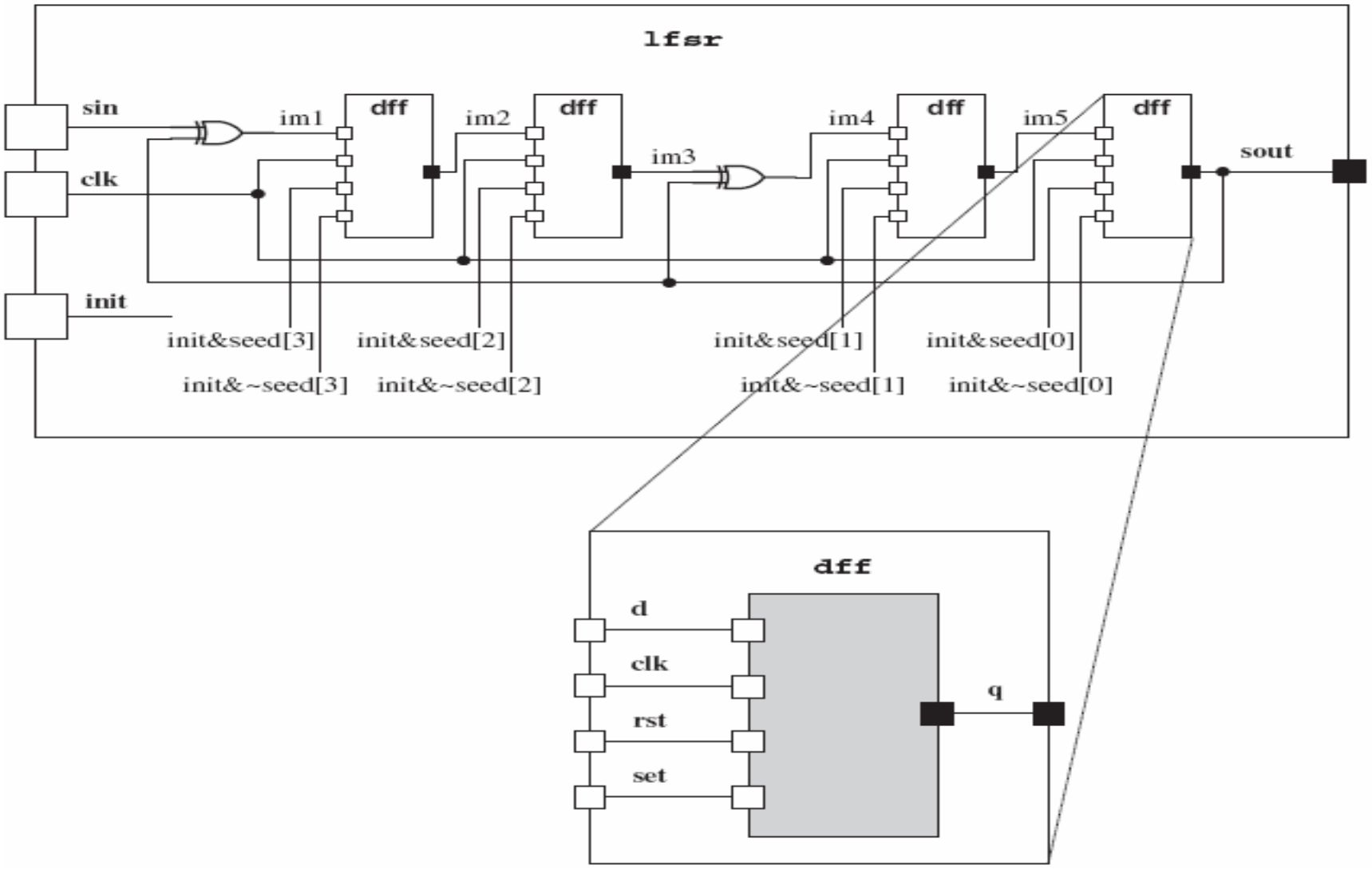
- **Parameter - assigning a value to a symbolic name**
- **Parameters are a means of giving names to constant values.**
  - **Parameters cannot be changed at runtime.**
  - **Parameters can be declared as signed, real, integer, time or realtime.**

Example	Explanation
`parameter p1=5, p2=6;`	32 bit parameters
`parameter [4:0] p1=5, p2=6;`	5 bit parameters
`parameter integer p1=5;`	32 bit parameters
`parameter signed [4:0] p1=5;`	5 bit signed parameters

5.10 Dataflow Models of a Linear-Feedback Shift Register

- Example 5.26
- **Linear-feedback shift registers (LFSRs)**
- A linear feedback shift register can be formed by performing exclusive-OR on the outputs of two or more of the flip-flops together and feeding those outputs back into the input of one of the flip-flops
- LFSRs are frequently **used as pseudorandom pattern generators to generate a random number of 1s and 0s**

Linear-feedback shift registers (LFSRs)



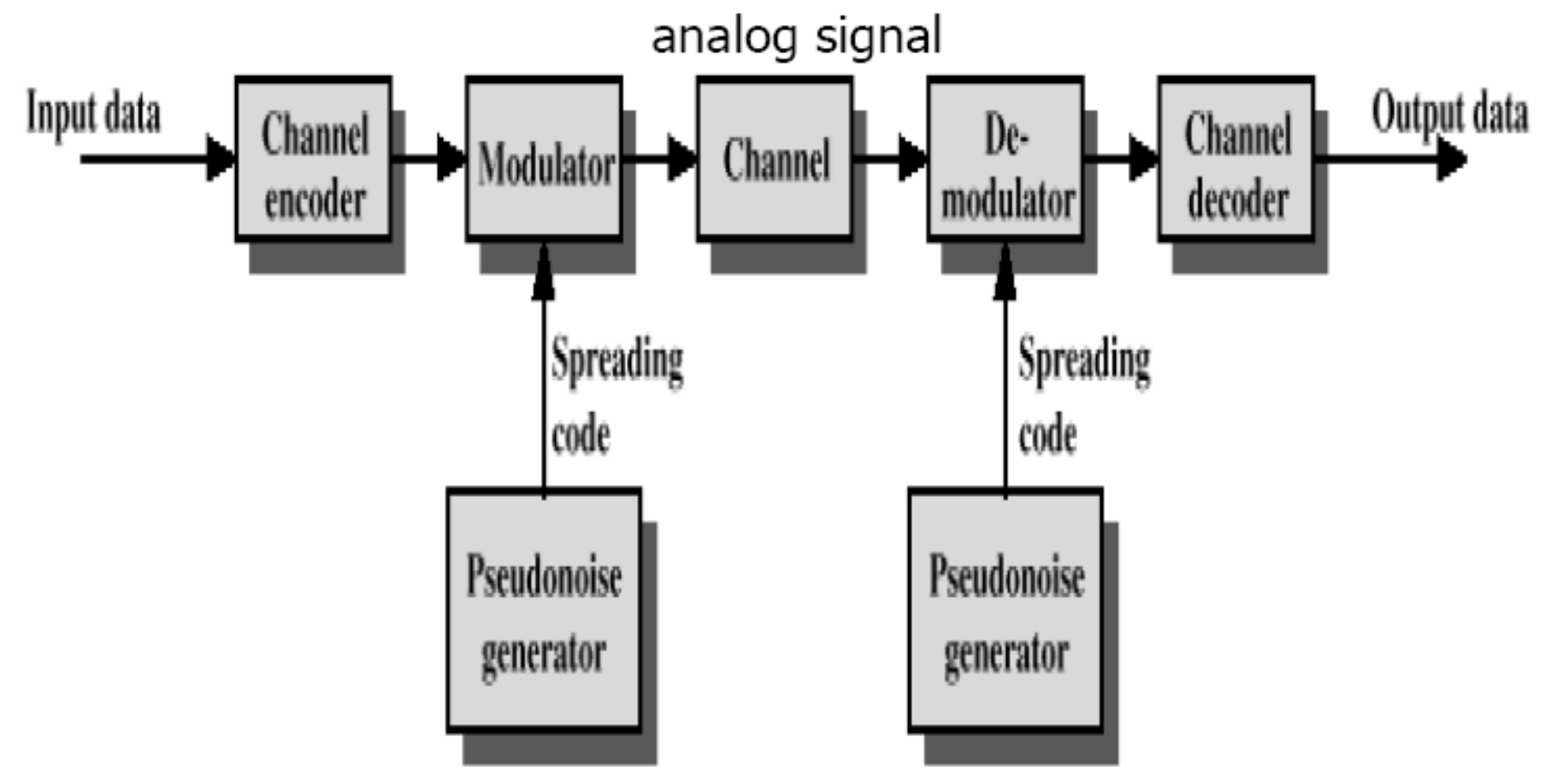
Linear Feedback Shift Registers

- **Properties**
  - LFSRs are well-suited to hardware implementation;
  - Can produce sequences of large period
  - Can produce sequences with good statistical properties
  - Because of the structure, can be analyzed using algebra
- **Definition**
  - LFSR of length N consists of N stages numbered 0, 1, …, N− 1, each capable of storing one bit and having one input and one output, and clock which controls the movement of data
  - content of stage 0 is output and forms part of the output sequence
  - the content of stage i is moved to stage i − 1 for each i, $1 \leq i \leq N - 1$
  - new content of stage N− 1 is feedback bit s_j calculated by adding together modulo 2 previous contents of fixed subset of stages

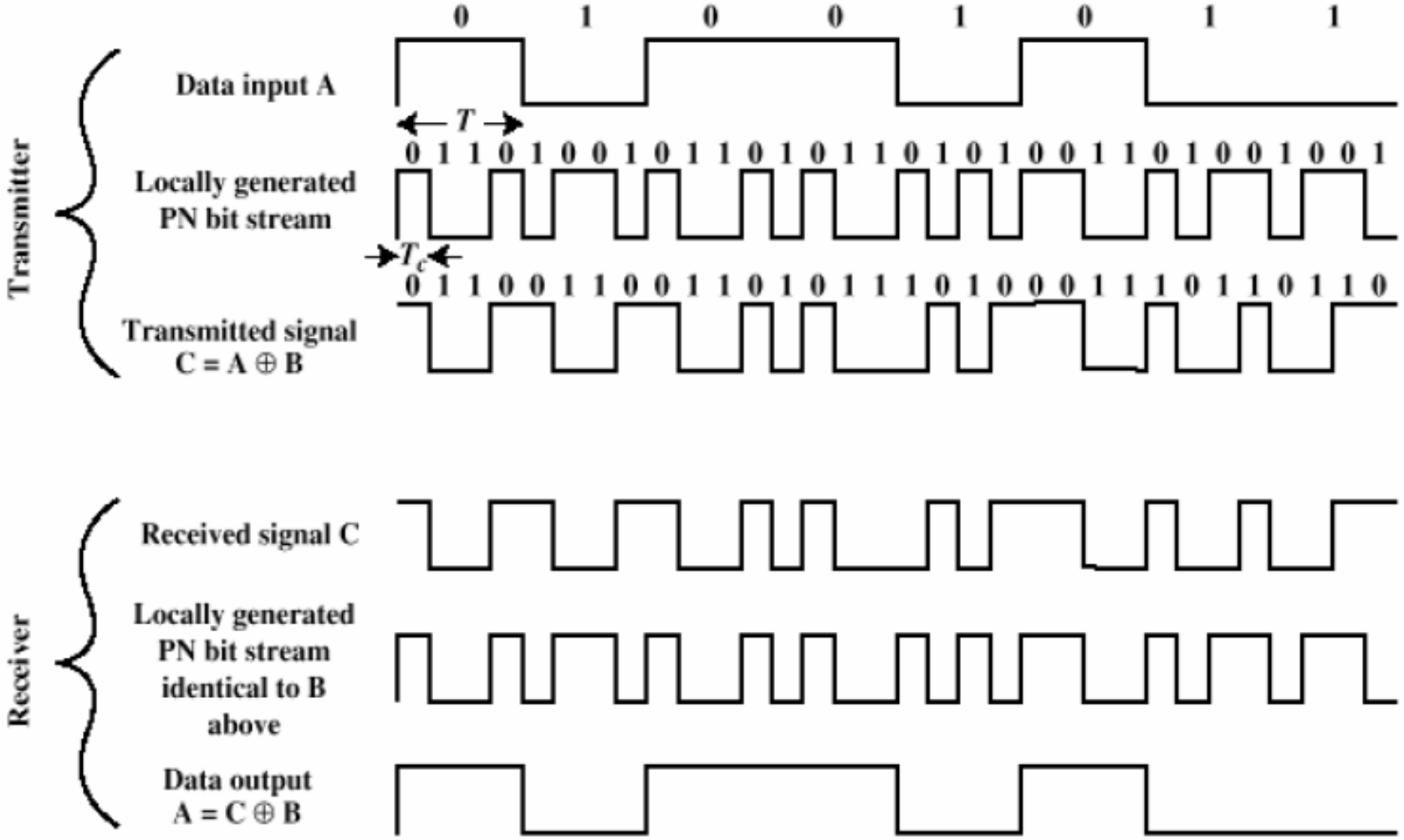
LFSR Applications

- Pattern Generators
- Counters
- Built-in Self-Test (BIST)
- Encryption
- Compression
- Checksums
- Pseudo-Random Bit Sequences (PRBS)

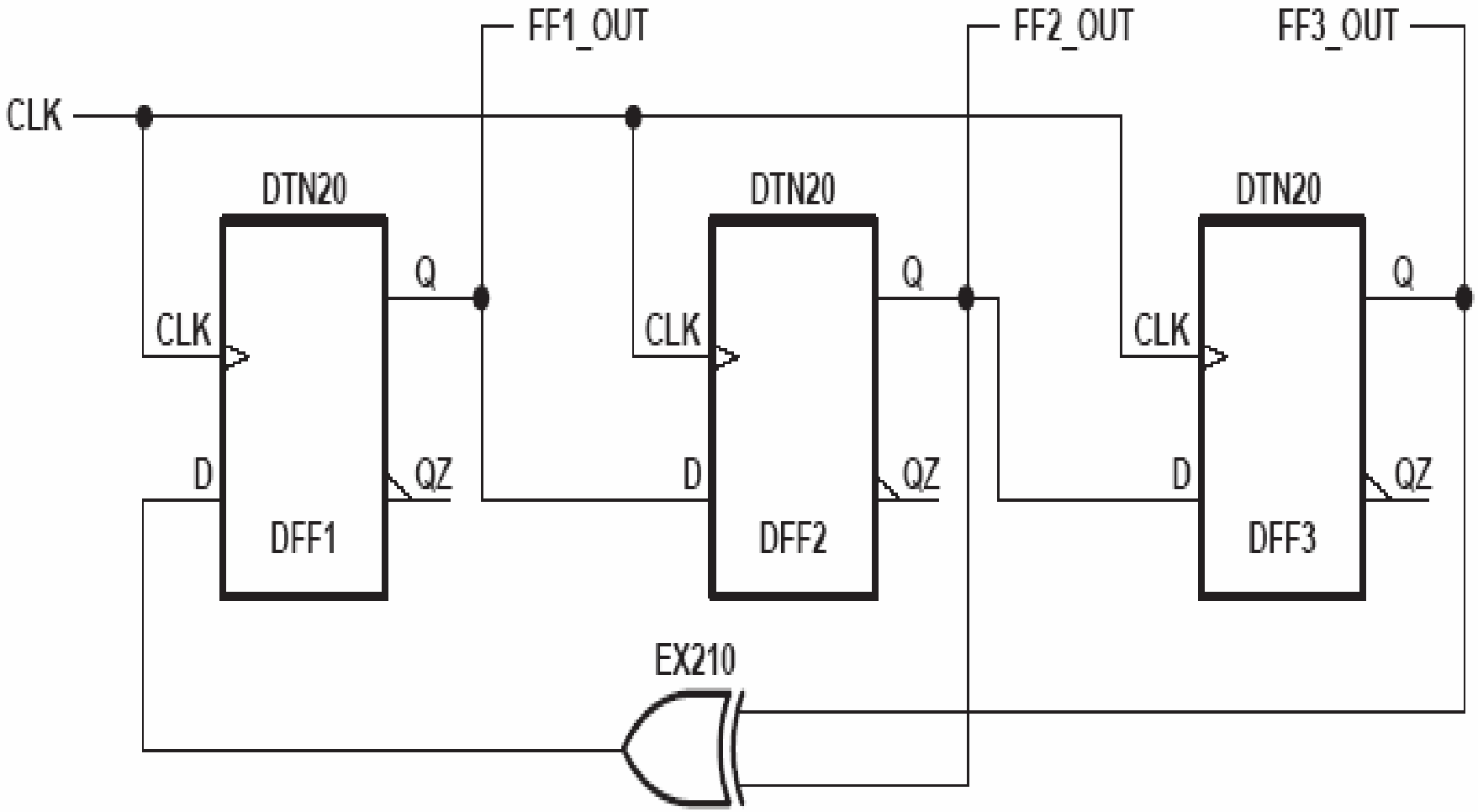
Spread Spectrum



Spread Spectrum



Linear Feedback Shift Register



Maximal-Length LFSRs

- A maximal-length LFSR produces the maximum number of PRPG patterns possible and has a pattern count equal to **2**n – 1**
- It produces patterns that have an approximately equal number of 1s and 0s
- **No way to predict mathematically if an LFSR will be maximal length, Peterson and Weldon have compiled tables of maximal-length LFSRs to which designers may refer**

Table . Pattern-Generator Seed Values

CLOCK PULSE	FF1_OUT	FF2_OUT	FF3_OUT	COMMENTS
1	1	1	1	Seed value
2	0	1	1	
3	0	0	1	
4	1	0	0	
5	0	1	0	
6	1	0	1	
7	1	1	0	
8	1	1	1	Starts repeat

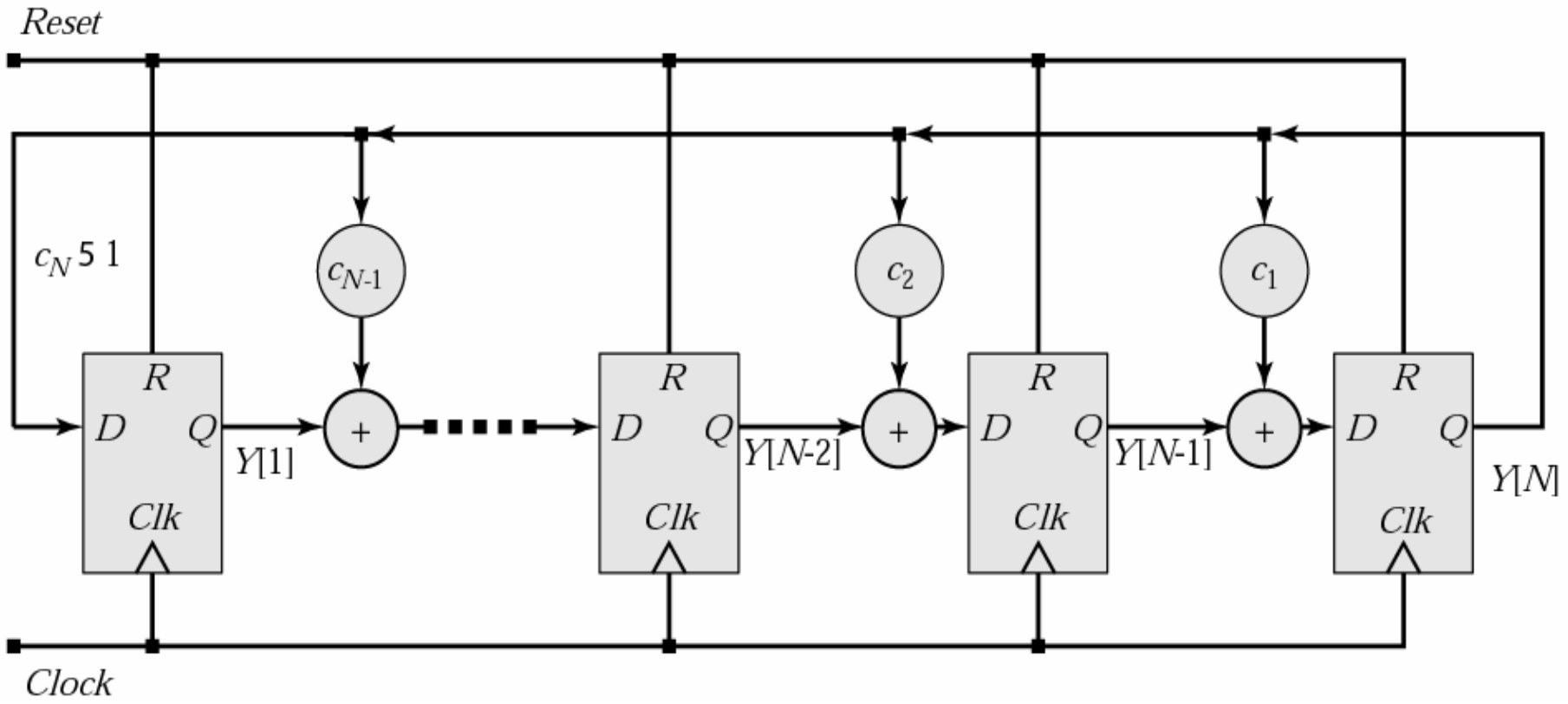
In a practical ASIC design

- The user would create an LFSR that is much bigger than three bits to get a large number of pseudorandom patterns before the patterns repeated
- **There are some practical restrictions to the length of the LFSR.**
- A 32-bit maximal-length LFSR would create over 4 billion patterns that, at a 16-MHz clock rate, would take almost 5 minutes to generate the whole pattern set

Pseudo Random Pattern Generation (PRPG))

- Linear feedback shift registers make extremely good pseudorandom pattern generators
- When the outputs of the flip-flops are loaded with a seed value (anything except all 0s, which would cause the LFSR to produce all 0 patterns) and when the LFSR is clocked, it will generate a pseudorandom pattern of 1s and 0s
- Note that the only signal necessary to generate the test patterns is the clock

LFSR with modulo-2 addition



Features of LFSR

- **Binary tap coefficients $C_1,\ldots,C_N$** that determine whether Y(N) is fed back to a given stage of the register
- The vector of tap coefficients determines the coefficients of the characteristic polynomial of the LFSR
- The characteristic polynomial determines the period of the register

$$p(x) = \sum_{i=0}^{m} f_i x^i = x_m + f_{m-1} x^{m-1} + \ldots + f_1 x + 1$$

The characteristic polynomial

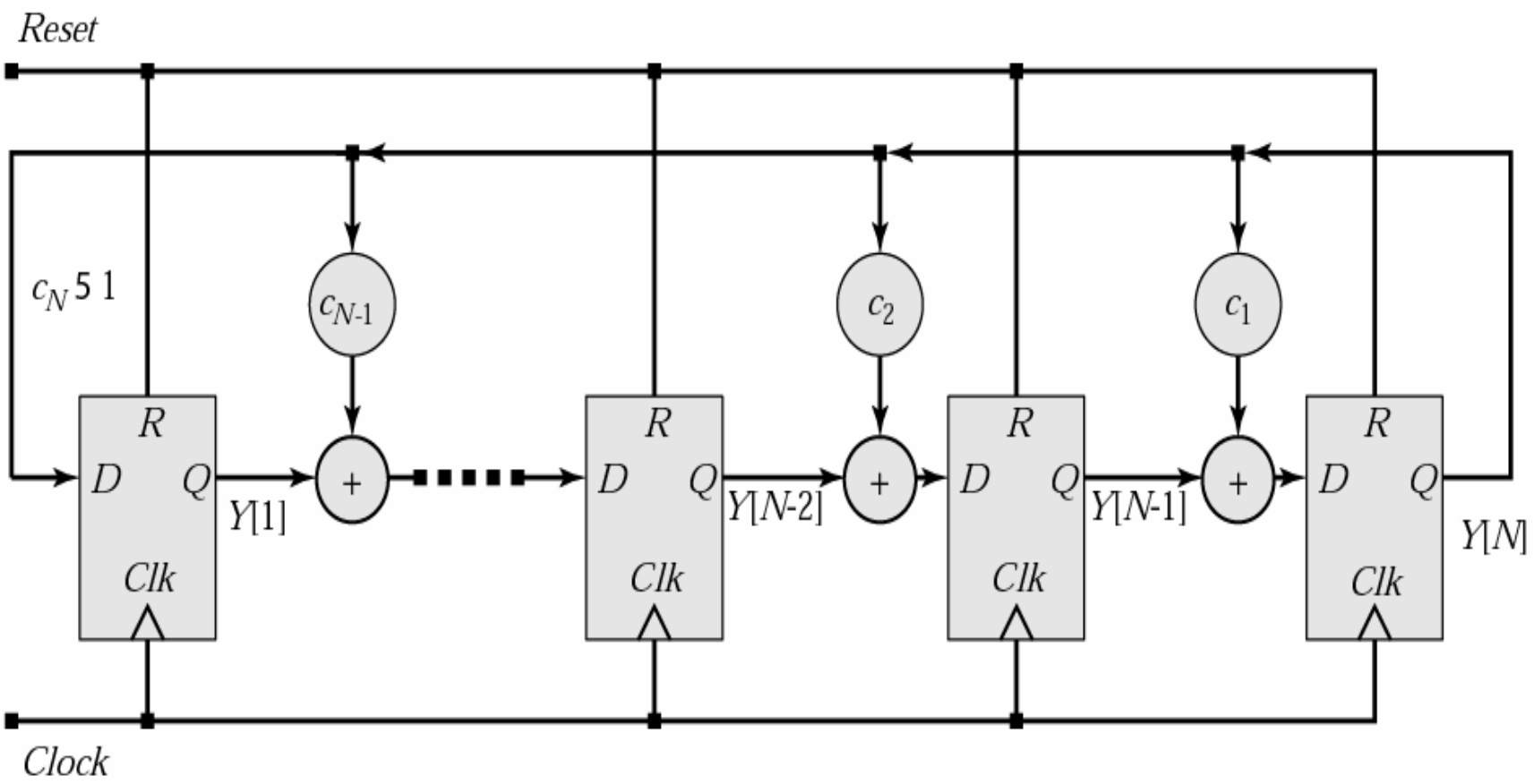
- The characteristic polynomial determines the period of the register

- *If* $p(x)= x**63+x+1$
- 2**63-1 ≈9.22337*10**18
- If clk=50Mhz, T=20ns
- The period of the register > 5800 years

Example 5.26

- The Verilog code below describes an eight-cell autonomous LFSR with a synchronous (edge-sensitive) cyclic behavior using an RTL style of design.
- The movement of data through the register under simulation is shown in binary and hexadecimal format in Figure 5-16 for the initial state and three cycles of the clock.

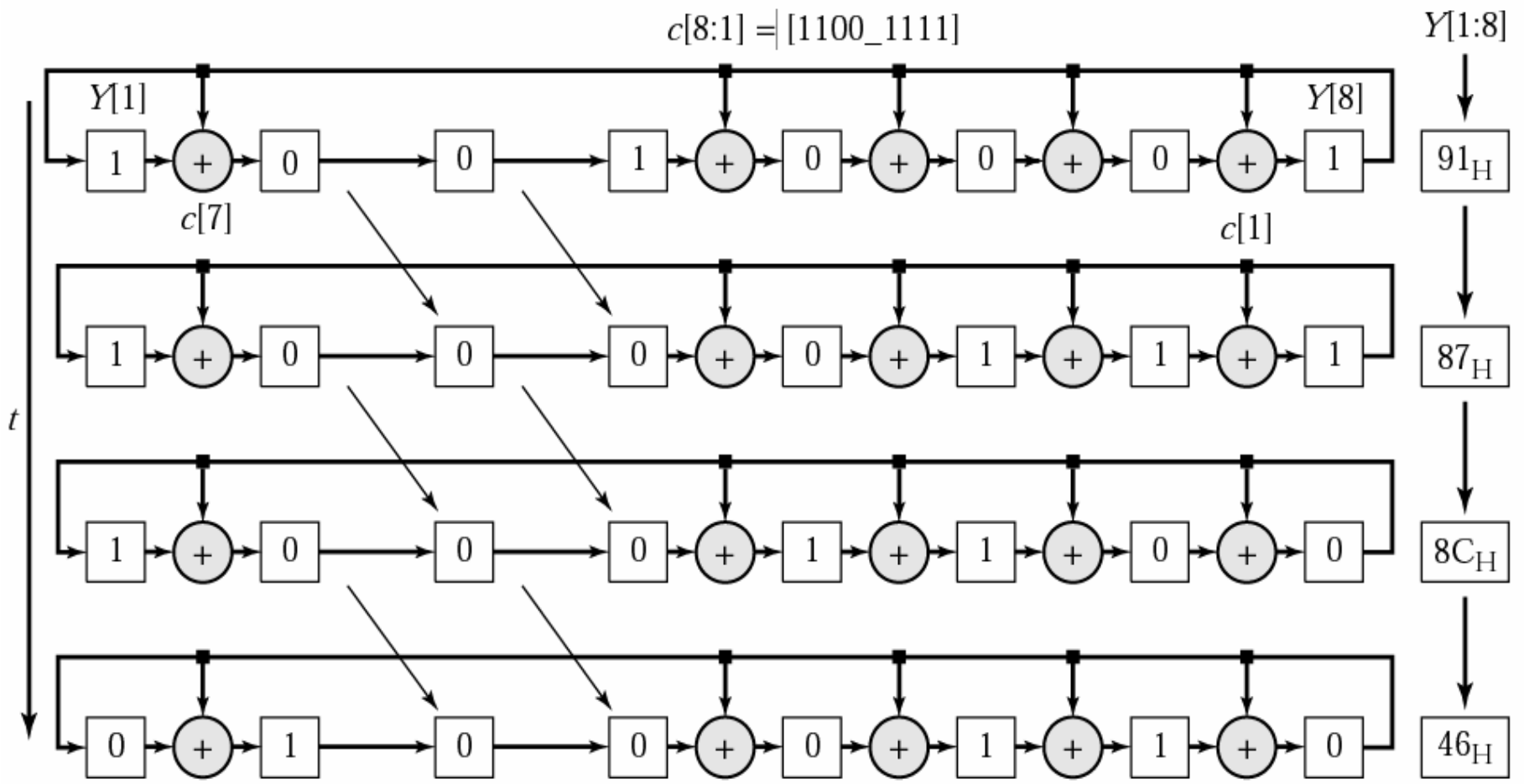
LFSR with modulo-2 addition



Example 5.26

```
module Auto_LFSR_RTL (Y,Clock,Reset);
parameter           Length=8;
parameter           initial_state=8'b1001_0001;  // 91h
parameter [1:Length]  Tap_Coefficient=8'b1100_1111;
input  Clock,Reset;
output    [1:Length]  Y;
 reg [1:Length]   Y;
always @ (posedge Clock)
  if (Reset==0) Y<= initial_state; // active-low reset to initial state
    else begin
        Y[1]<=Y[8];
        Y[2]<=Tap_Coefficiet[7] ? Y[1]^Y[8] : Y[1];
        Y[3]<=Tap_Coefficiet[6] ? Y[2]^Y[8] : Y[2];
        Y[4]<=Tap_Coefficiet[5] ? Y[3]^Y[8] : Y[3];
        Y[5]<=Tap_Coefficiet[4] ? Y[4]^Y[8] : Y[4];
        Y[6]<=Tap_Coefficiet[3] ? Y[5]^Y[8] : Y[5];
        Y[7]<=Tap_Coefficiet[2] ? Y[6]^Y[8] : Y[6];
        Y[8]<=Tap_Coefficiet[1] ? Y[7]^Y[8] : Y[7];
     end
 endmodule
```

data movement in an LFSR with modulo-2 addition



5.11 Modeling Digital Machines with Repetitive Algorithms

- An algorithm for modeling the behavior of a digital machine may **execute some or all of its steps repeatedly in a given machine cycle**, depending on whether the steps executed unconditionally or not
- For example, an algorithm that sequentially shifts the bits of an LFSR can be described by a for loop in Verilog

Example 5.27

```
 module Auto_LFSR_ALGO (Y,Clock,Reset);
parameter          Length=8;
parameter    initial_state=8'b1001_0001;
parameter [1:Length] Tap_Coefficient=8'b1100_1111;
input          Clock,Reset;
output       [1:Length]  Y;
 reg [1:Length]          Y;
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
```

Loop Statements

- **forever** loop - executes continually
- **repeat** loop - executes a fixed number of times
- **while** loop - executes if expression is true
- **for** loop - executes once at the start of the loop and then executes if expression is true

⇨Loop statements - used for repetitive operations

Loop Statement

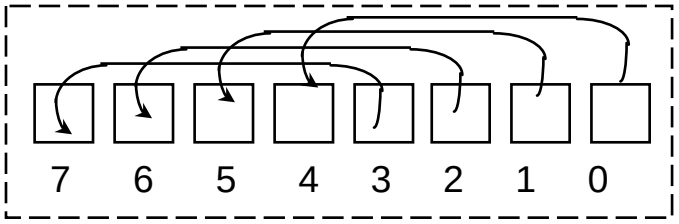
A ***for*** **loop** has the form:

```
For (initial_statement;
        control_expression;
            index_statement )
 statement_for_execution;
```

For Loop

- **for** loop - executes once at the start of the loop and then executes if expression is true

4-bit Left Shifter




```
integer  i;  // declare the index for the FOR LOOP

always @(inp or cnt)
   begin
            result[7:4] = 0;
            result[3:0] = inp;
            if (cnt == 1)
               begin
                        for (i = 4; i <= 7; i = i + 1)
                                 begin
                                    result[i] = result[i-4];
                                 end
                        result[3:0] = 0;
               end
```

Forever and Repeat Loops

- **forever** loop - executes continually

```
initial
   begin
         clk = 0;
         forever  #25  clk = ~clk;
   end
```

Clock with period of 50 time units

- **repeat** loop - executes a fixed number of times

```
if (rotate = = 1)
    repeat (8)
         begin
            tmp = data[15];
            data = {data << 1, temp};
         end
```

Repeats a rotate operation 8 times

while loop

A Verilog ***while*** loop has the form:

- **while (Expression)**
- **Statement**
- A loop statement that repeats a statement or block of statements as long as a controlling expression remains true (i.e. non-zero).

While Loop

- **while** loop - executes if expression is true

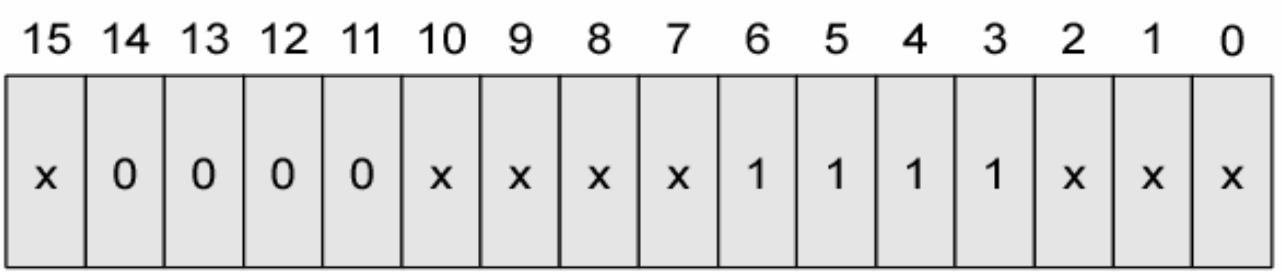
```
initial
   begin
          count = 0;
          while (count < 101)
             begin
                   $display ("Count = %d", count);
                   count = count + 1;
             end
   end
```

Counts from 0 to 100
Exits loop at count 101

Ex 5.29

The for loop is used to assign values to bits within a register after it has been initialized to x

```
reg [15:0] demo_register;
 integer K ;
    …
    for (K=4;K>0;K=K-1)
       begin
          demo_register [K+10]=0;
          demo_register [K+2]=1;
       end
    …
```



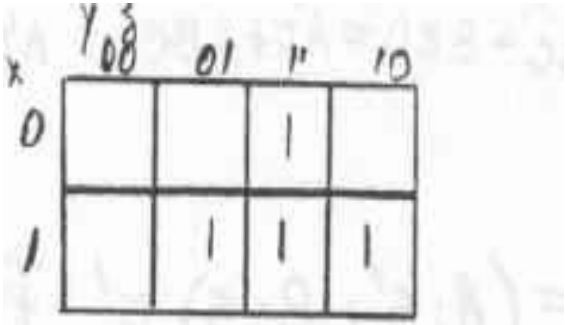
Ex 5.30 A majority circuit

- A majority circuit is a combinational circuit whose output *is equal* to 1 if the input variables have more 1's than 0's.

  The output is 0 otherwise.

- Ex. Design a 3-input majority circuit.

x	y	z	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



$F = xy + xz + yz$

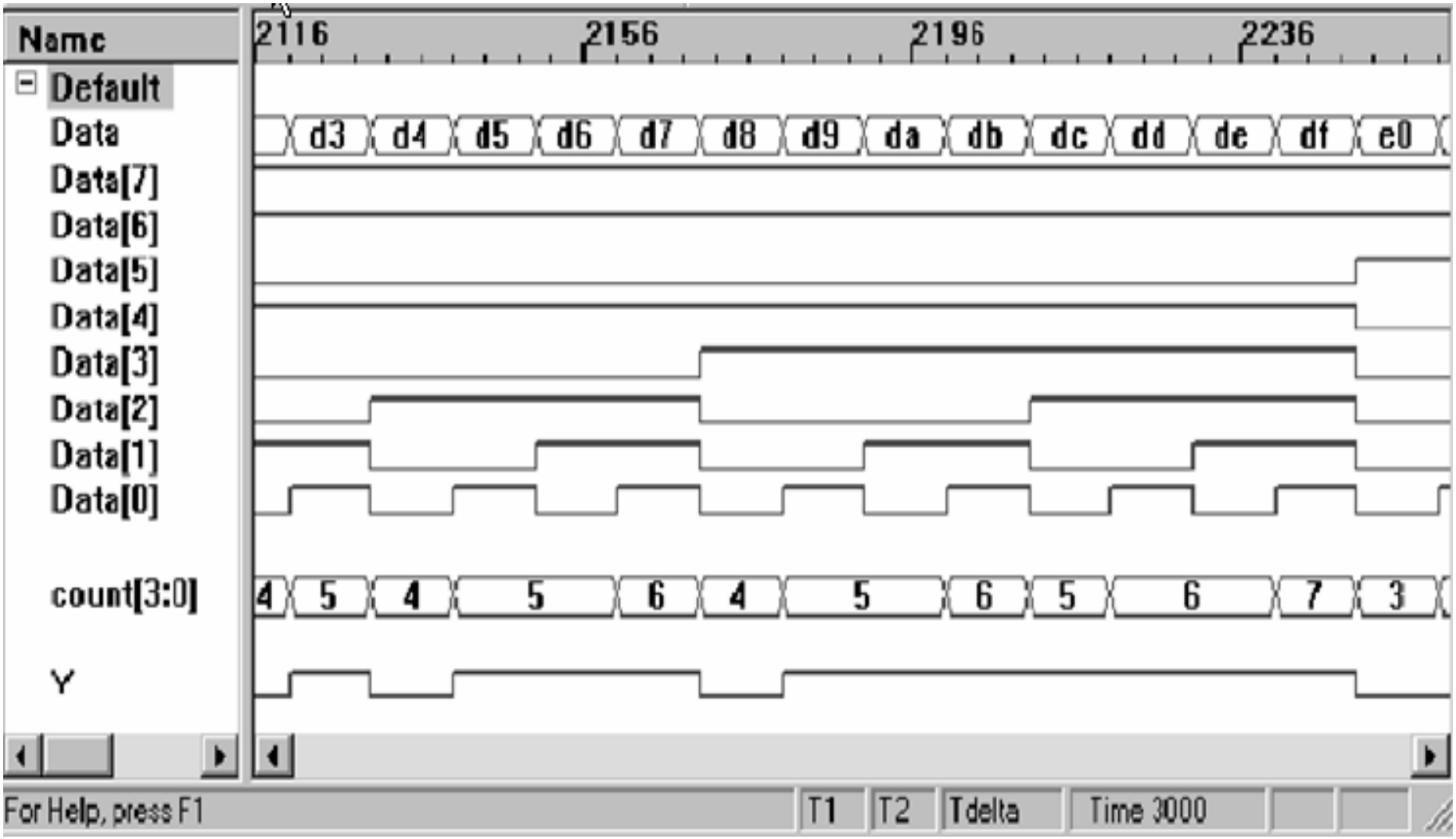
Example 5.30

```
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
```

Ex 5.30- Majority

```
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
```

Simulation result



5.11.1 Intellectual Property Reuse and Parameterized Models

- Models have increased value if they are written to be extendable to more than one application
- Use parameters to specify：

  bus widths, word length,

  and other details that customize a model to an application

Ex 5.31 an eight-cell autonomous LFSR

```
module Auto_LFSR_Param (Y, Clock, Reset);
   parameter              Length=8;
   parameter              initial_state=8'b1001_0001;
   parameter [1:Length]   Tap_Coefficient=8'b1100_1111;
   input            Clock, Reset;
   output   [1:Length]  Y;
   reg [1:Length]       Y;
   integer              k;
  always @ (posedge Clock)
    if (Reset==0) 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

```
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
```

An alternative description of Ex 5.32

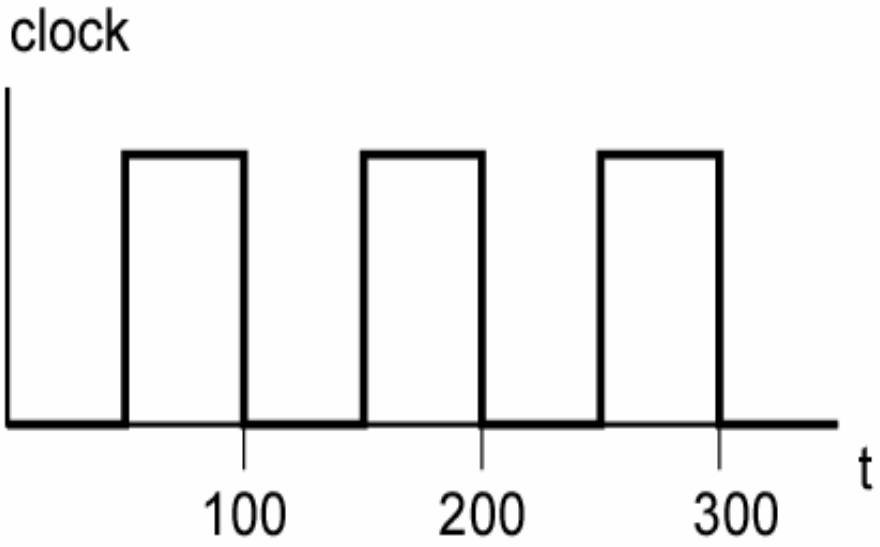
```
begin: count_of_1s // count_of_1s declares a named
                   // block of statements
  reg [7:0] temp_reg;
     count=0;
     temp_reg=reg_a;   // load a data word
    while (temp_reg)
       begin
        count=count+temp_reg[0];
        temp_reg = temp_reg>>1;
     end
end
```

5.11.2 Clock Generators

- **In testbenches to provide a clock signal** for testing the model of a synchronous circuit
- A flexible clock generator will be parameterized for a variety of applications
- The ***forever*** loop causes unconditional repetitive execution of statements, **subject to the *disable* statement**
- The ***disable*** statement is used to prematurely terminate a named block of procedural statement
- **Some EDA synthesis tools will synthesize only the *for* loop**

Example 5.33

```
parameter half_ccle = 50;
parameter stop_time = 350;
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;
```



Distinction of *always* and *forever*

- ***always*** and ***forever*** are not the same construct
- ***1. always*** construct declares a concurrent behavior;

  forever loop is a **computational activity** flow and is used only within a behavior
- ***2. forever*** loops can be nested; cyclic and single-pass behaviors may not be nested
- ***3. forever*** loop executes only when it is reached within a sequential activity flow

  always behavior becomes active and can execute at the beginning of simulation

Example 5.34 finds the location of the first 1 in a 16-bit

```
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
```

5.12 Machines with Multicycle Operations

- Some digital machines have repetitive operations distributed over multiple clock cycles
- This activity can be modeled in Verilog by a synchronous cyclic behavior that has nested edge-sensitive event-control expressions

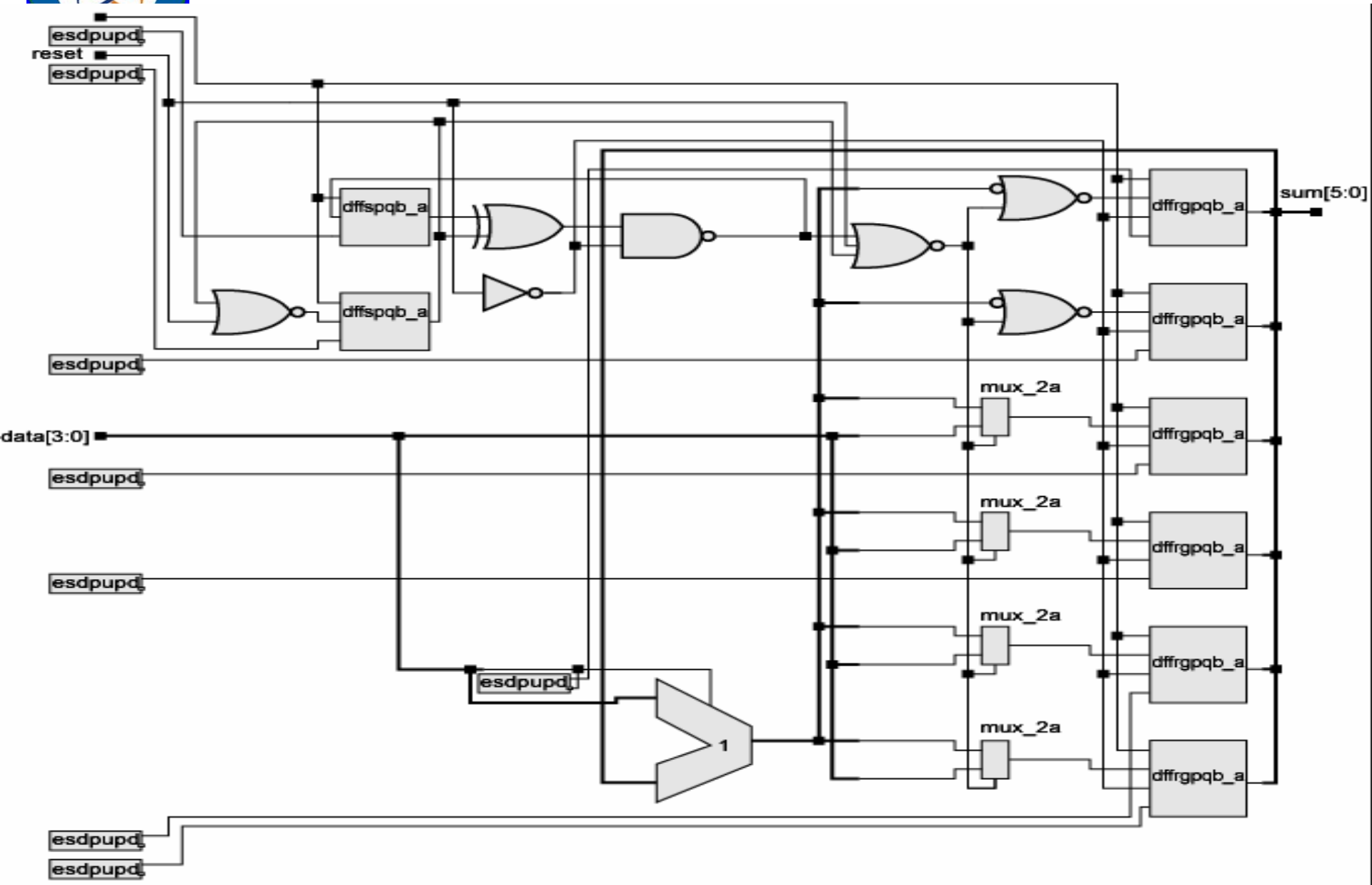
Ex 5.35 4-cycle adder

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

Disable：

Causes the execution of an active task or named block to terminate before all its statements have been executed.

synthesis circuit



5.13 Design documentation with Functions and Tasks

- Two types of subprograms can improve the clarity of a description: task, function
- **Tasks create a hierarchical organization** of the procedural statements within a Verilog behavior
- **Functions substitute for an expression**
- Tasks and functions let designers manage a smaller segment of code and hide the details of an implementation from the outside world
- Function and task **improve the readability, portability and maintainability of a model**

5.13.1 Tasks

- Two kinds of type: User Defined **Task** , System Task
- Tasks are declared within a module, and they may be **referenced only from within a cyclic or single-pass behavior**.
- A task can have parameters passed to it, and the results of executing the task can be passed back to the environment
- **A task can call itself**
- Original standard(1995) does not support recursion
- More general than functions and may contain timing controls
- Tasks to be synthesized may contain event-control operators but not delay-control operators

Tasks

- ***Syntax:***
- ***task TaskName;***
  [Declarations...]
  Statement
  endtask
- Declarations of any number or combination of the following:
- ***parameter, input, output, inout, reg, integer, event,*** (***real, time, realtime***)
- **All declarations of variables are local to the task**
- When a task is invoked, **its formal and actual arguments are associated in the order** in which the task's ports have been declared

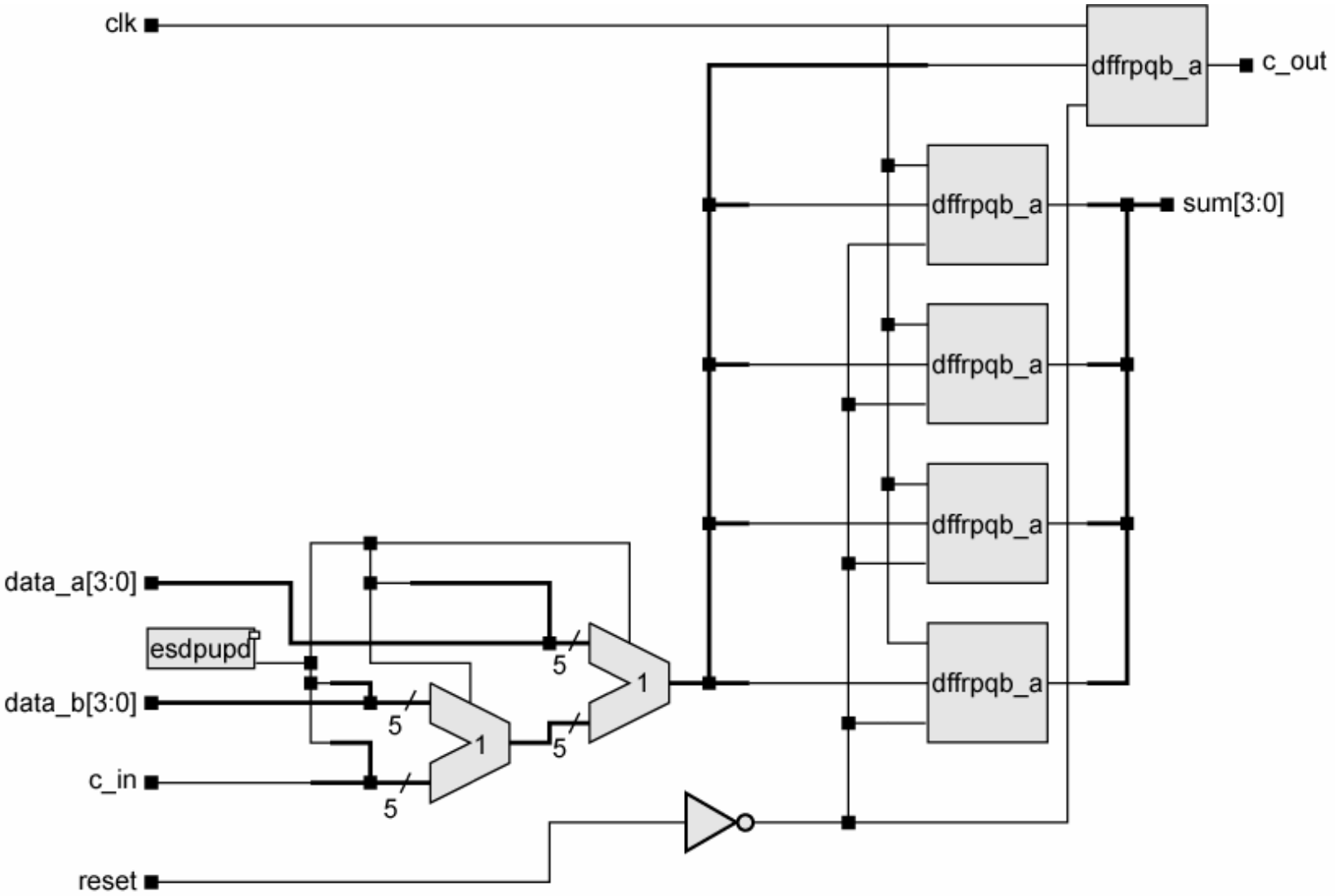
Task Enable (or 'call')

- When a task is enabled, values are passed into the task using the input and inout arguments
- When the task completes, values are passed out from the task using the task's output and inout arguments.
- Syntax
- **TaskName[(Expression,...)];**
- For synthesis, a task may not contain timing controls.
- Task enables are synthesized as combinational logic.

Ex 5.36 a user-defined task that adds two 4-bit words and a carry bit

```
module adder_task (c_out,sum,c_in,data_a,data_b,clk,reset)
    output   [3:0]  sum;
    output          c_out;
    input    [3:0]  data_a,data_b;
    input           clk, reset;
    input           c_in;
    reg             sum;
    reg             c_out;
    always @ (posedge clk or posedge reset)
      if (reset)       {c_out,sum}<=0; else   add_value (c_out,sum,data_a,data_b,c_in);
     task add_value;
    output    [3:0]   sum;
    output     c_out;
    input     [3:0]   data_a,data_b;
    input      c_in;
     begin
       {c_out,sum}<=data_a+(data_b+c_in);
     end
   endtask
endmodule
```

synthesis circuit



5.13.2 Functions

- A function is also implemented by an expression and returns a value at the location of the function's identifier
- **Functions may implement only combinational behavior**, not contain timing controls(no #,@)
- The definition of a function implicitly defines **an internal register variable with the same name, range, and type as the function itself**

Function

- **function [Range Or Type] FunctionName;**

  Declarations...

  Statement

  endfunction

- Calls a function, which returns a value for use in an expression.
- **FunctionName (Expression,...);**
- Each call to a function is synthesized as a separate block of combinational logic

Rules

- A function must have at least one input argument.
- It may **not have any outputs or inouts**.
- Functions may **not contain timing controls** (delays, event controls or waits).
- A function returns a value by assigning the function name, as if it were a register.
- Functions may not enable tasks.
- Functions may not be disabled.

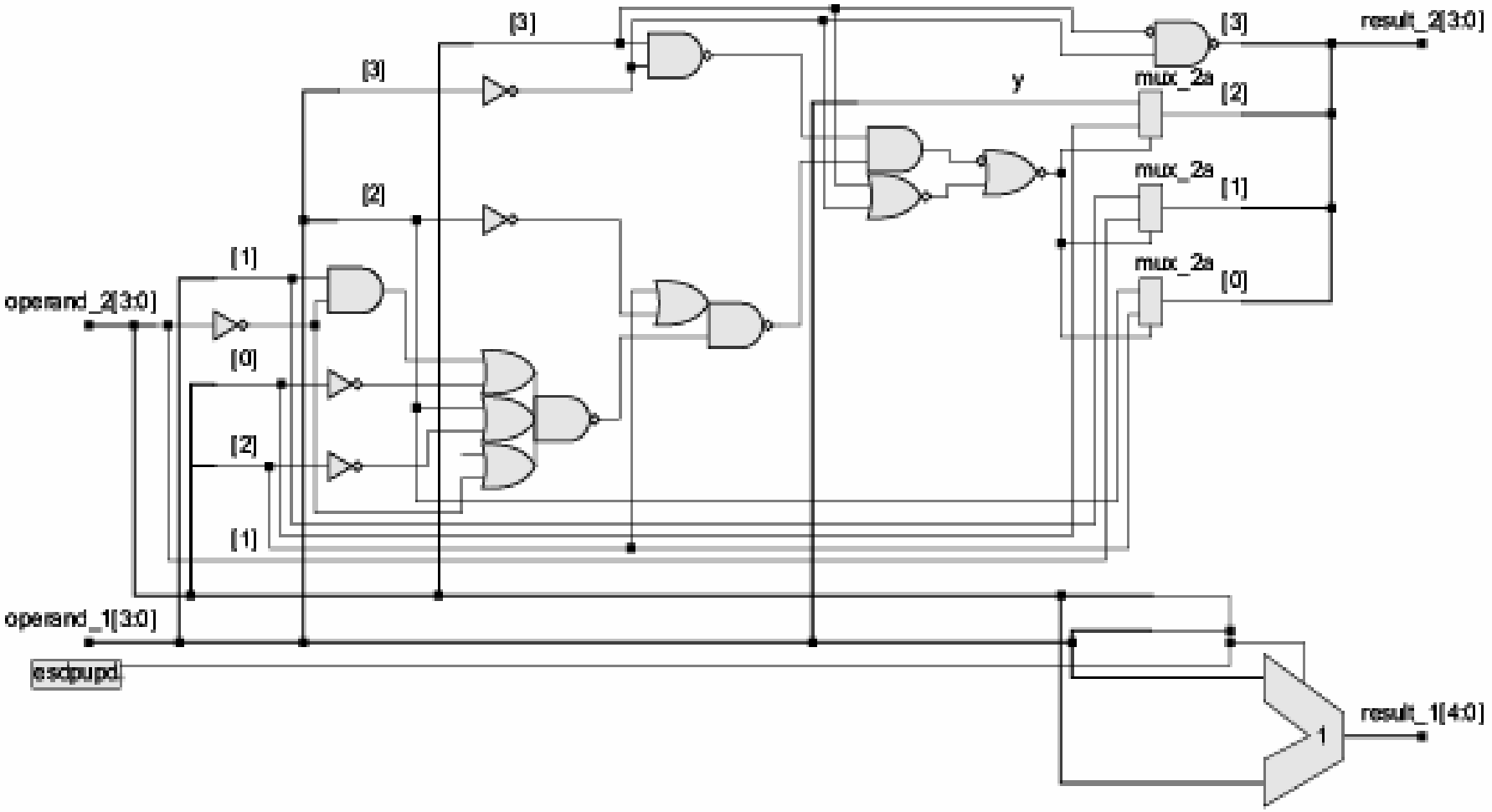
Ex 5.37 shift a word to the left until the most significant bit is '1'

```
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;
     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

```
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;
                                  //implicitly defines an internal register variable
     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)? oprand_1:oprand_2;
   endfunction
endmodule
```

Figure 5-22 synthesis circuit



Compare of Task ,Function

- Input,Output(Function:a input,no inout)
- Enabel(call):always(task),assign
- F:no Timing & Event Control
- T:no Timing Control
- Call another Task or Function :

  Func. Not call Task
- Return value: Func. Return Value

5.14 Algorithmic State Machine Charts for Behavioral Modeling

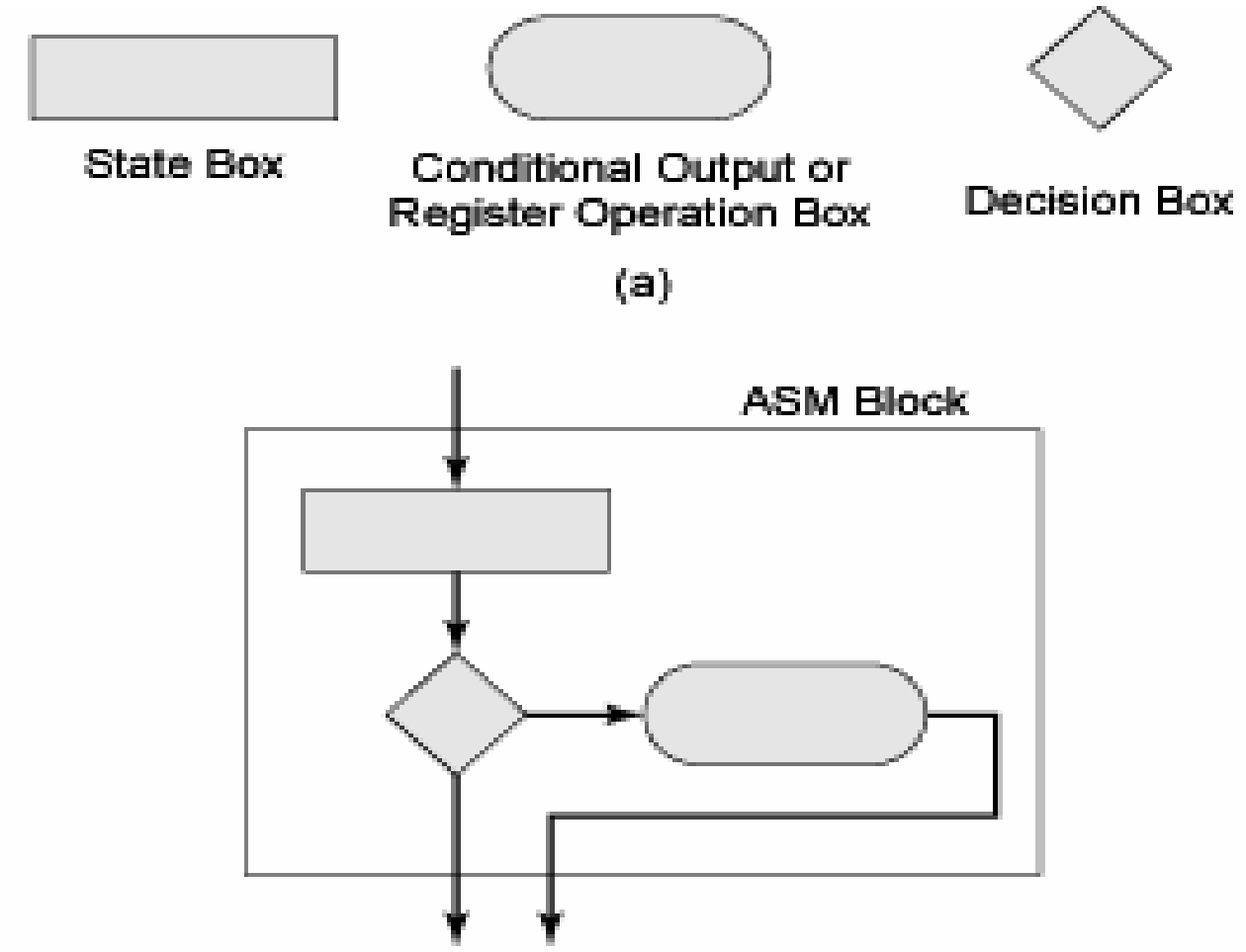
- A machine's activity consists of a synchronous sequence of operations on the registers of its datapaths, usually under the direction of a controlling state machine
- State-transition Graphs (STGs) indicate the transitions that result from inputs when the state machine is in a particular
- An ASM chart focuses on the activity of the machine, rather than on the contents of all the storage elements.

Algorithmic State Machine charts

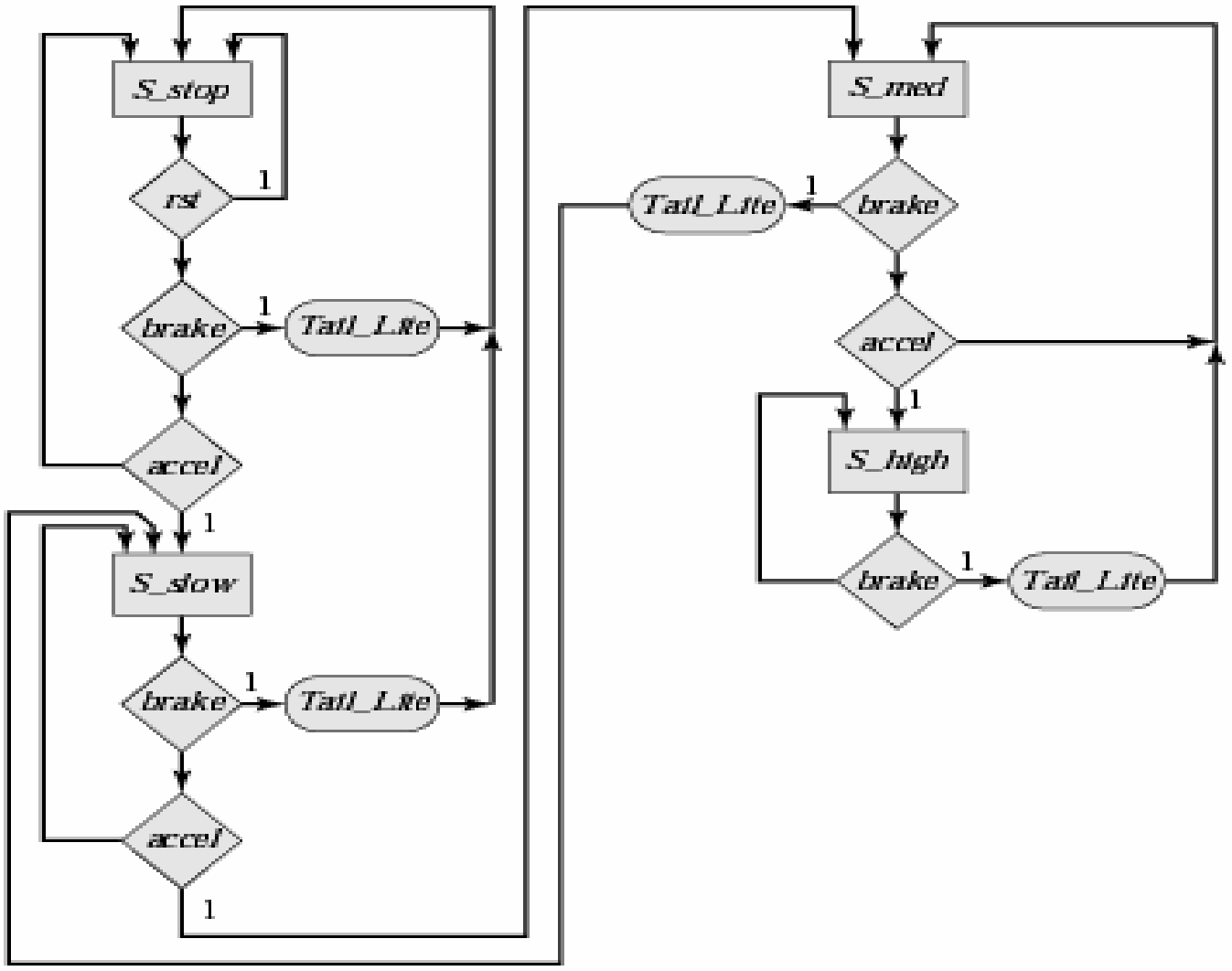
- **ASM are an abstraction of the functionality of a sequential machine**
- Similar to software flowcharts ,display the time sequence of computational activity as well as the sequential steps that occur under the influence of the machine's inputs
- Focus on the activity of the machine
- Helpful in describing the behavior of sequential machines and in designing a state machine to control a datapath

Construct

- An ASM chart has three basic elements:
- **state box**
- **conditional box**
- **decision box**



an ASM chart for a vehicle speed controller

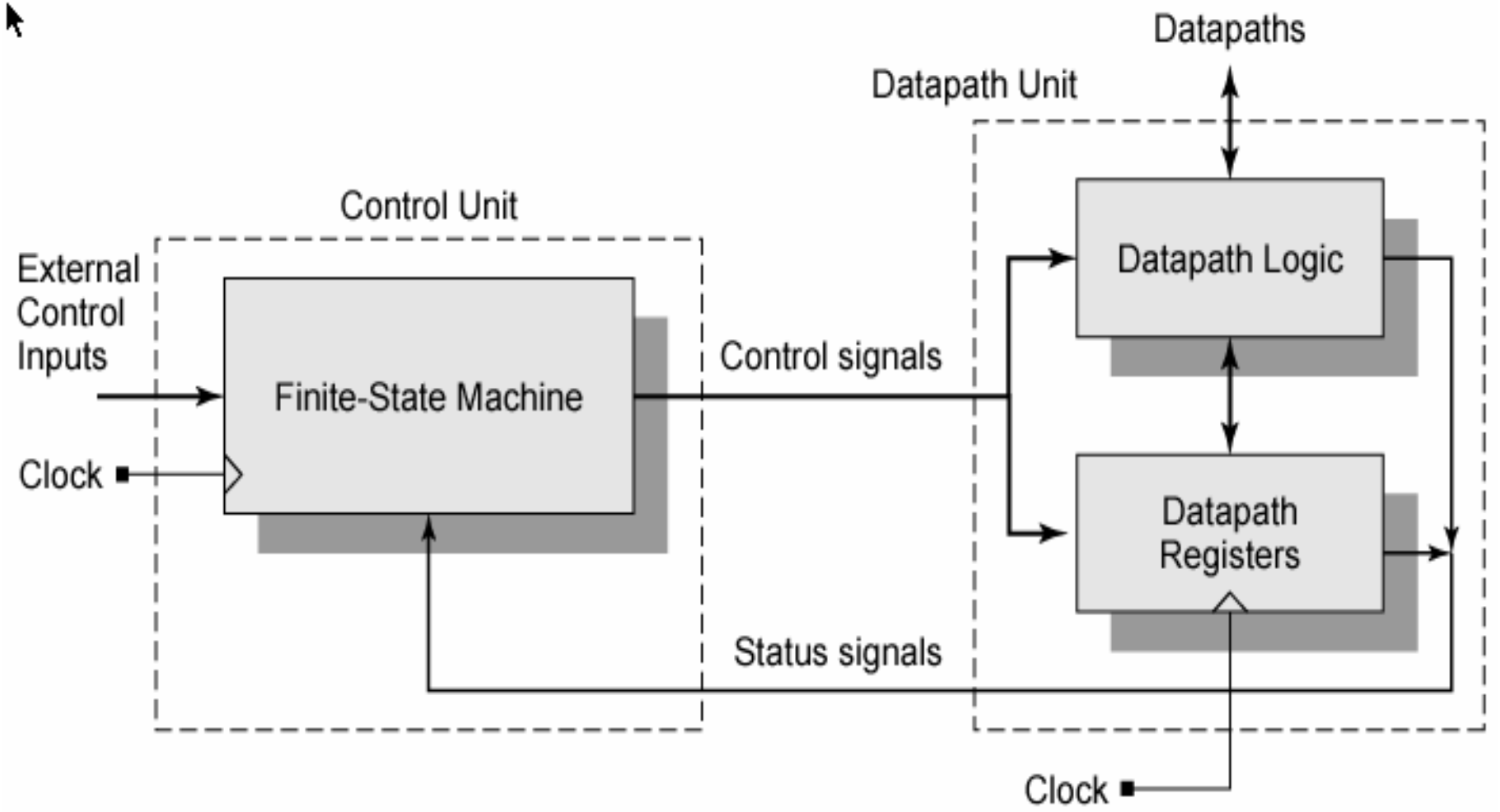


5.15 ASMD Charts

- A state machine will control register operations on a datapath in a sequential machine which has been partitioned into a controller and a datapath
- ASM charts that have been linked to a datapath in this manner are called Algorithmic State Machine and Datapath (ASMD) charts.
- The controller is described by an ASM chart modify the chart to link it to the datapath that is controlled by the machine

Datapath & Controller:

State-machine controller for a datapath



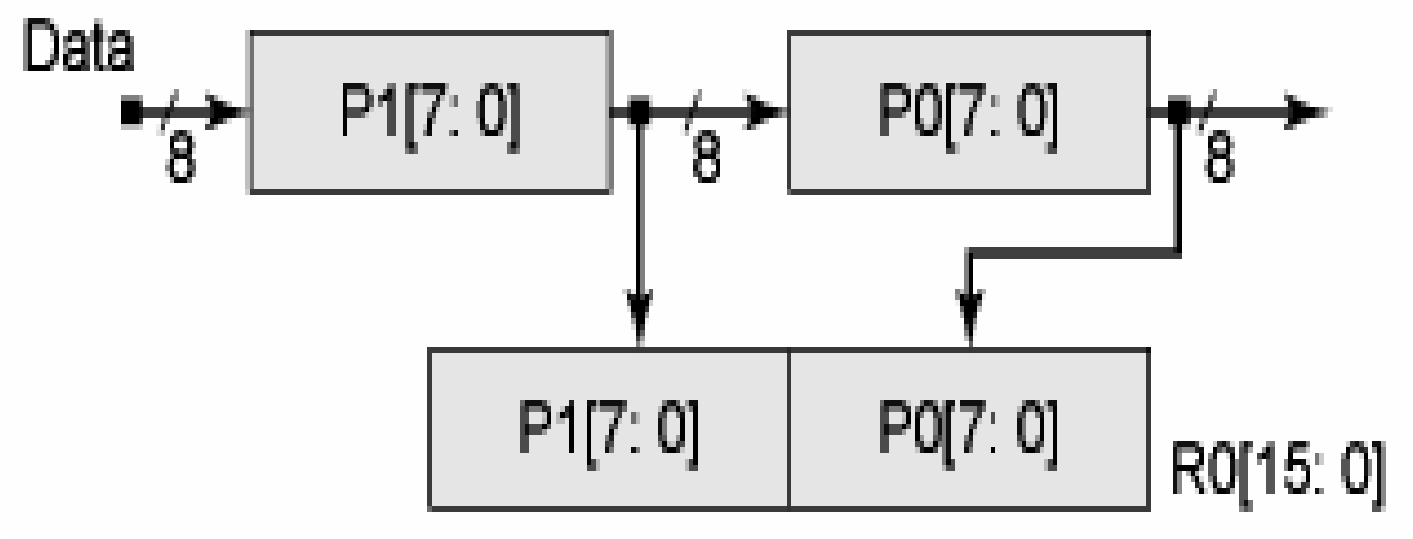
Datapath & Controller

- The datapath portion of the circuit is the portion that transforms the input signals into the corresponding output signals.
- The controller is the portion that governs the flow of the input signals through the various components comprising the datapath.

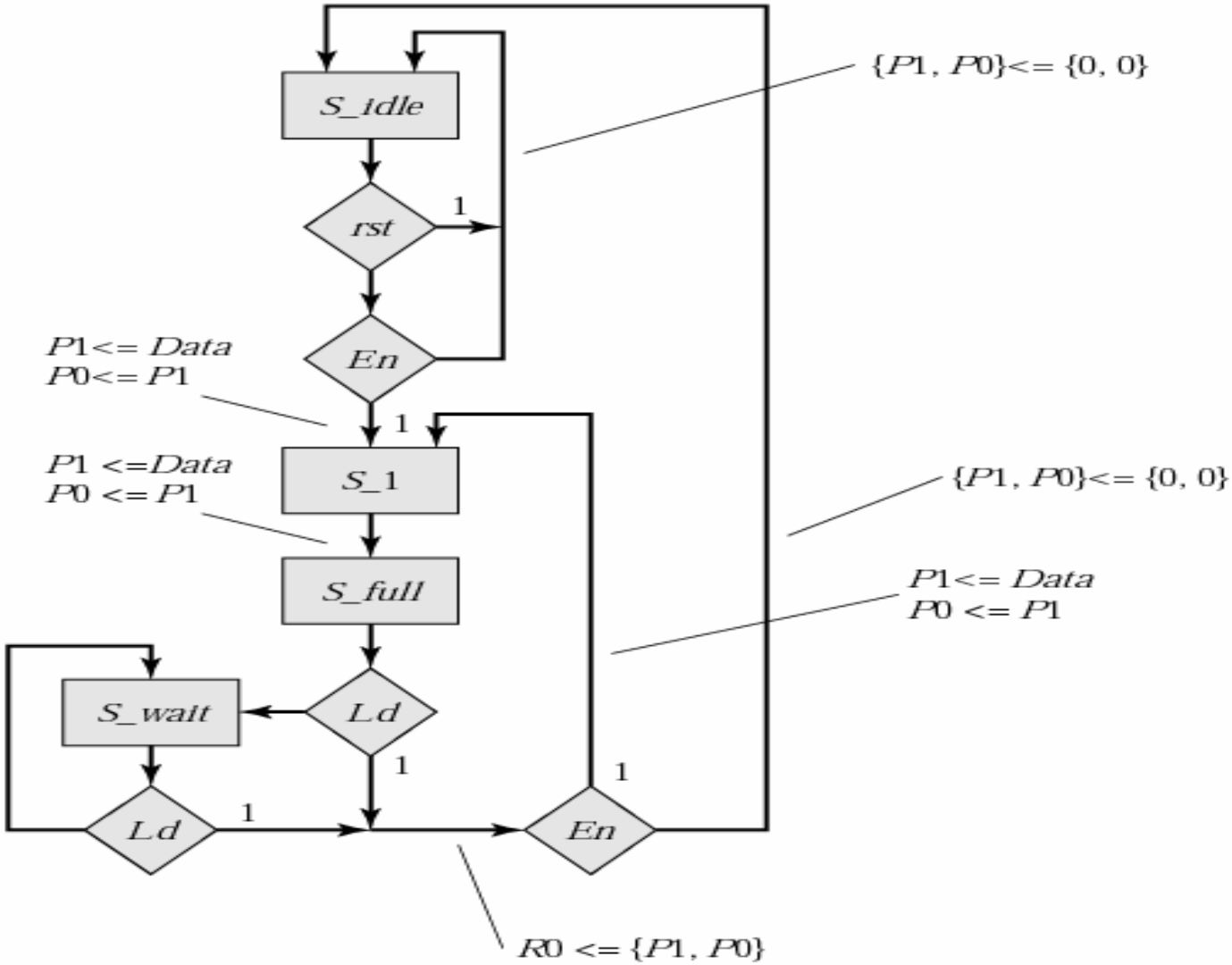
Example 5.39

- The architecture and ASMD chart describe the behavior of pipe_2stage, a two-stage pipeline that acts as **a 2:1 decimator** with a parallel input and output
- Decimators are used in digital signal processors to move data **from a high-clock-rate datapath to a lower-clock-rate** datapath
- They are also used **to convert data from a parallel format to a serial format**

A two-stage pipeline architecture



Ex 5.39 ASMD chart of a two-stage pipeline architecture



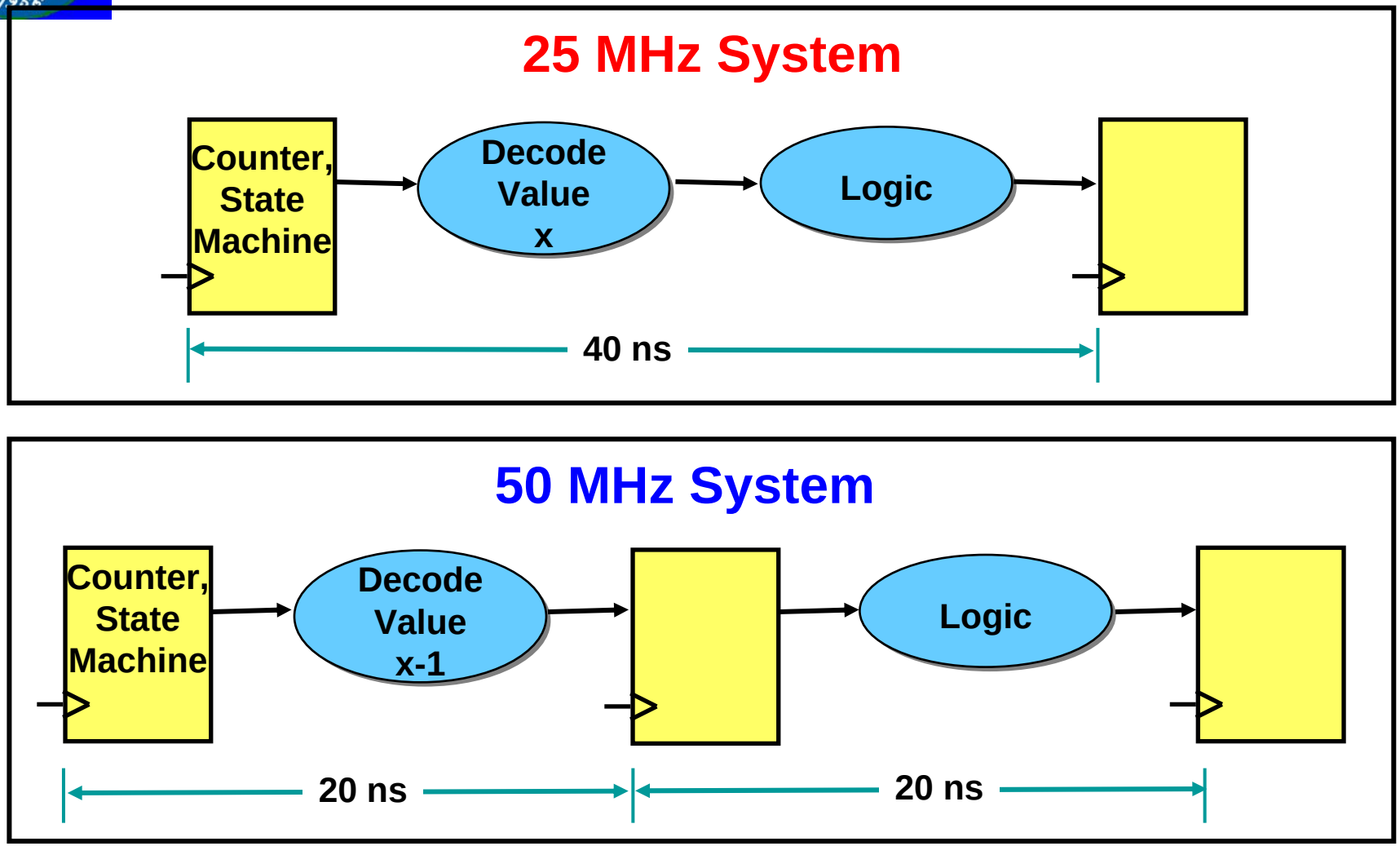
ASMD chart

- synchronous reset to S_idle
- With En asserted, the machine transitions from S_idle to S_1 ,accompanied by concurrent register operations that load the MSByte of the pipe with Data and move the content of P1 to the LSByte (P0)
- At the next clock the state goes to S_full, and now the pipe is full
- **The data rate at R0 is one-half the rate at which data is supplied to the unit from an external datapath**

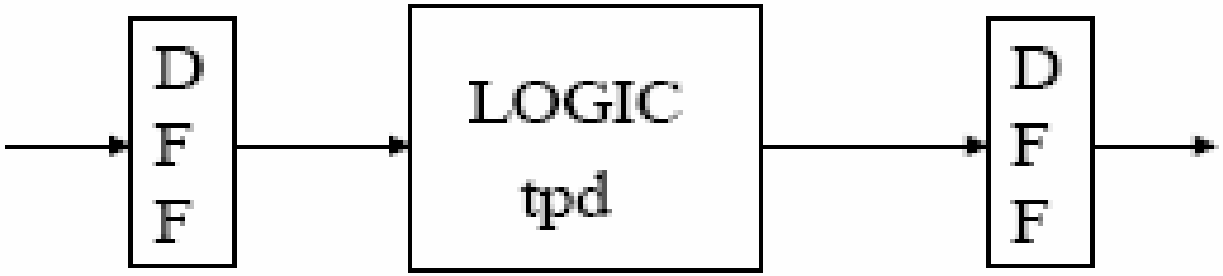
Pipelining

- Purposefully **Inserting Register(s) into Middle of Combinatorial Data (Critical) Path**
- Increases Clocking Speed
- Adds Levels of Latency
  - More Clock Cycles Needed to Obtain Output
- Some Tools Perform Automatic Pipelining
  - Same Advantages/Disadvantages As Automatic Fan-Out

Adding Single Pipeline Stage



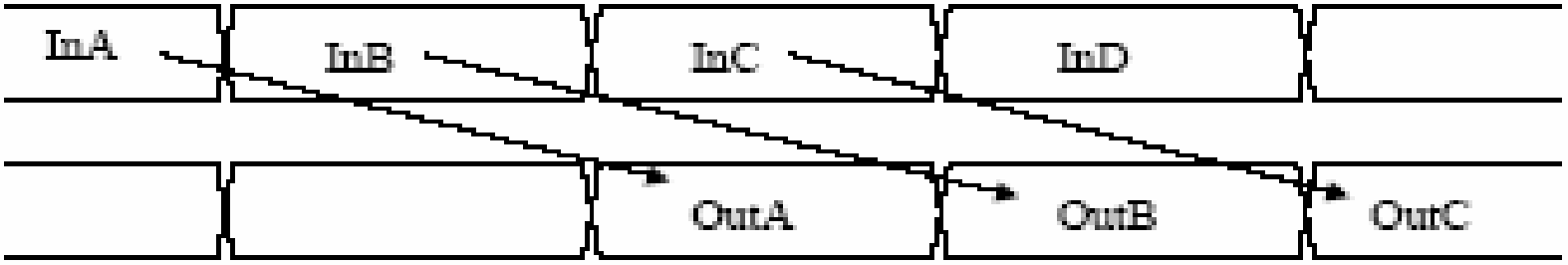
Registered Datapath



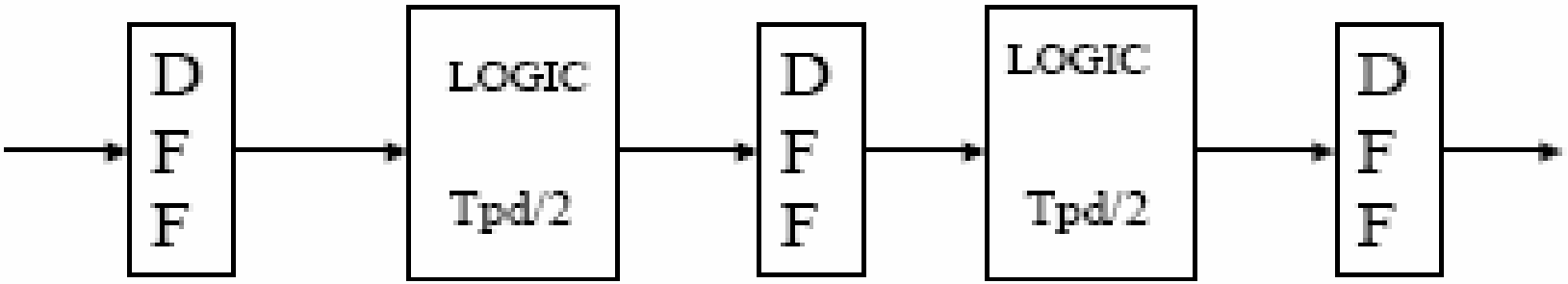
DFFs are rising edge triggered

Clk Freq = 1/ (Tclk2q + Tpd + Tsu)

Latency = 1 clk

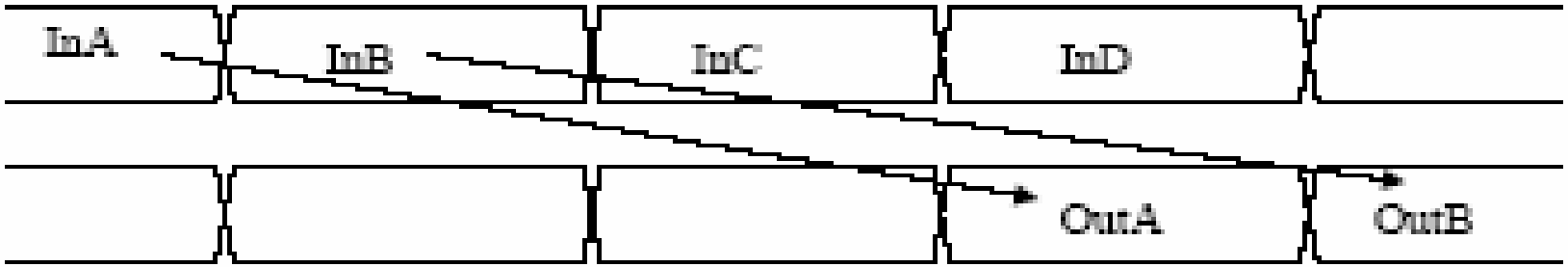


Add a pipeline stage



Clk Freq = 1/ (Tclk2q + Tpd/2 + Tsu)

Latency = 2 clks



The design of a datapath controller

1. Begins with an understanding of the sequential register operations that must execute on a given datapath architecture
2. Defines an ASM chart describing a state machine that is controlled by primary input signals and/or status signals from the datapath
3. Forms an ASMD chart by annotation the arcs of the ASM chart with the datapath operations associated with the state transition of the controller the design of a datapath controller
4. Annotates the state of the controller with unconditional output signals
5. Includes conditional boxes for the signals that are generated by the controller to control the datapath

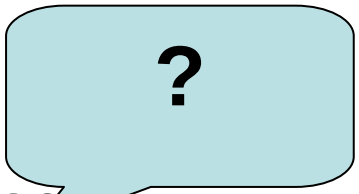
5.16 Behavioral Models of Counters, Shift Registers, and Register Files

- The storage elements of counters and registers usually have the same synchronizing and control signals
- A counter generates a sequence of related binary words
- A register stores data that can be retrieved and/or overwritten
- Register files are a collection of registers that share the same synchronizing and control signals
- **Behavioral descriptions of many counters, shift registers and register files are routinely synthesized by modern synthesis tools**

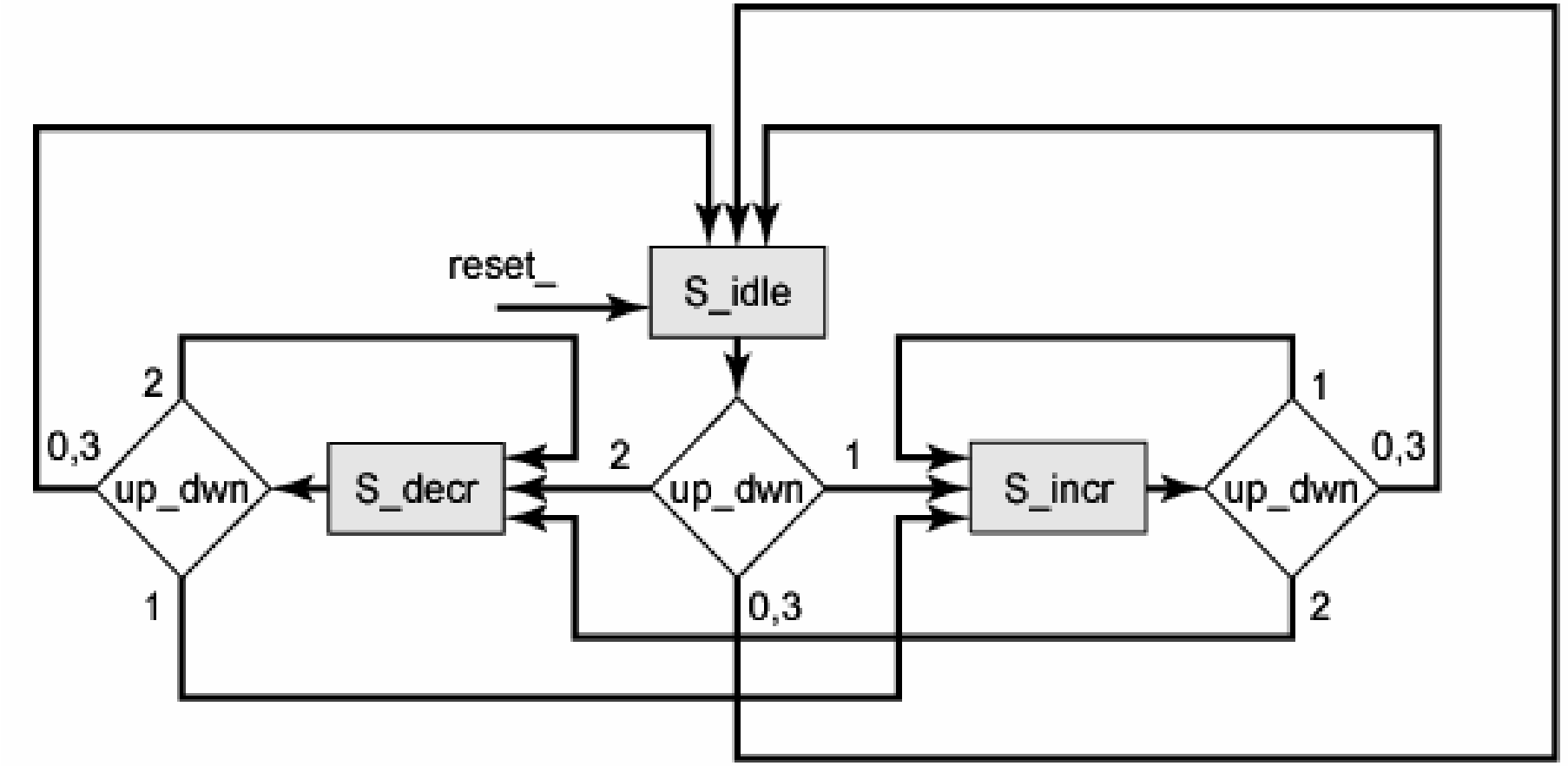
5.16.1 Counters

- Example 5.40

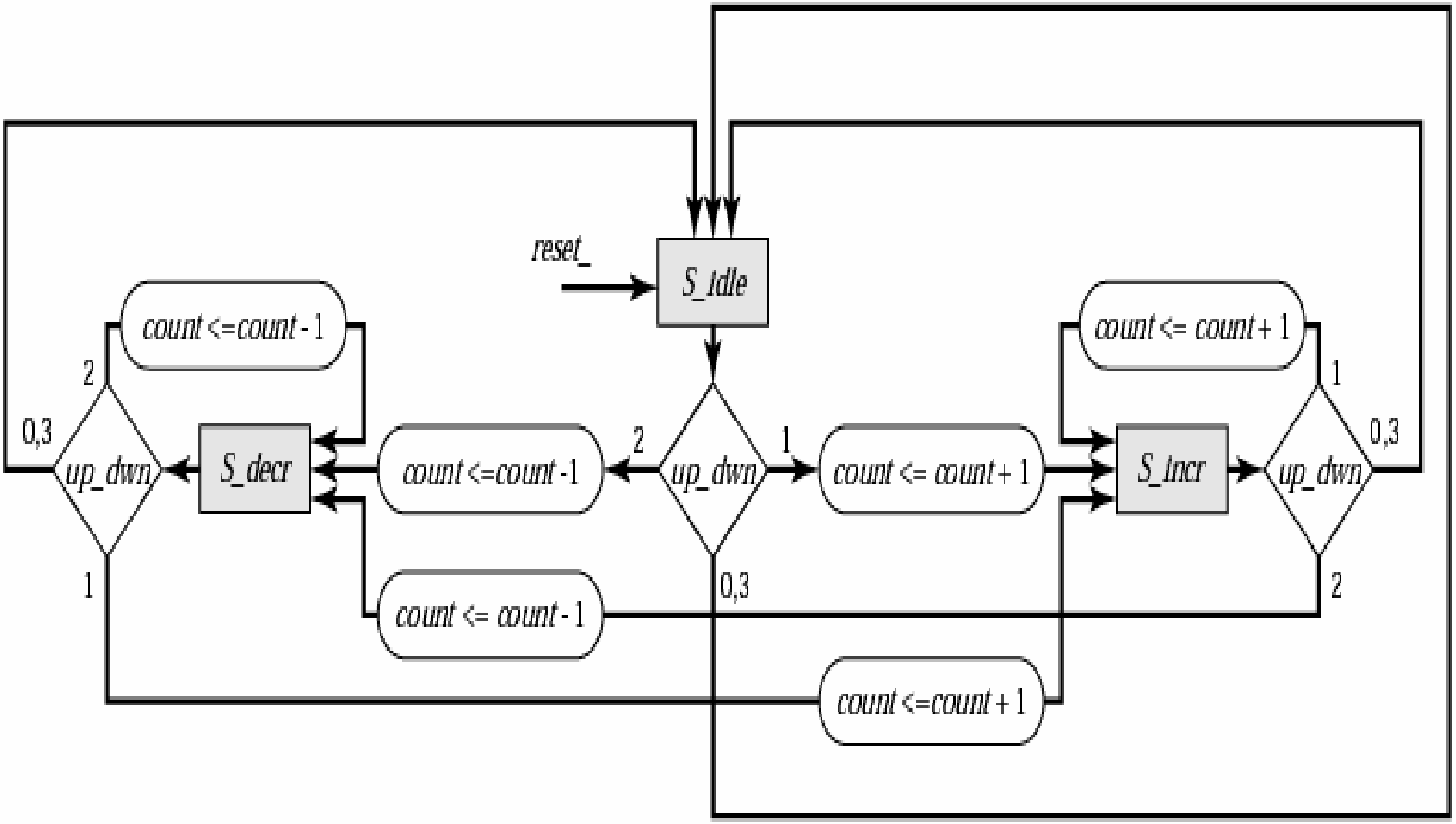
```
module Up_Down_Impicit1(count, up_dwn, clock, reset_);
    output [2:0] count;
    input   [1:0] up_down;
    input   clock,reset_;
    reg     [2:0] count;
    always @ (negedge clock or negedge reset_)
      if (reset_==0)     count<=3'b0;
      else if (up_dwn==2'b00||up_dwn==2'b11)  count<=count;
        else if (up_dwn==2'b01)   count<=count+1;
          else if (up_dwn==2'b10)
                    count<=count-1;
 endmodule
```



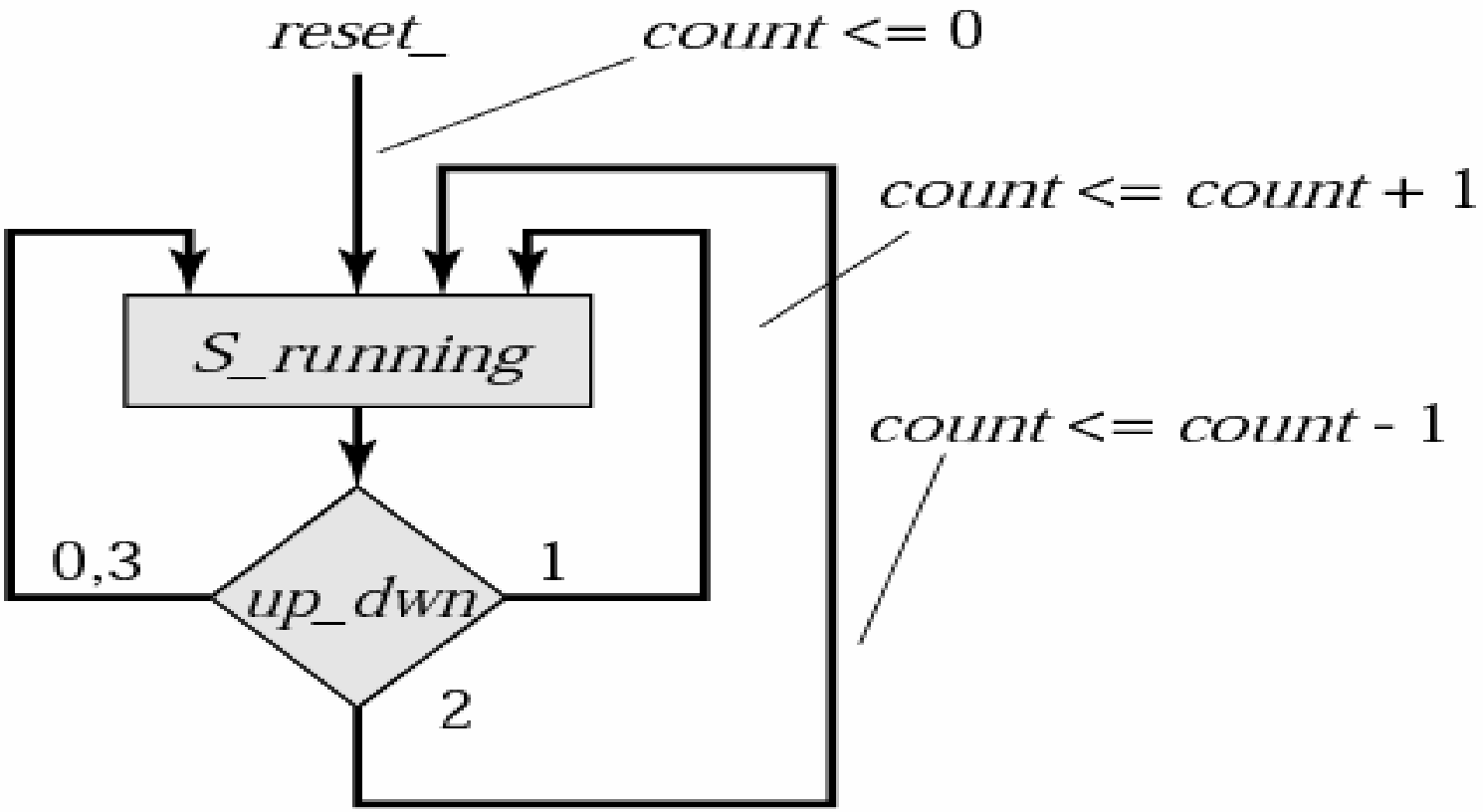
ASM chart without conditional output boxes



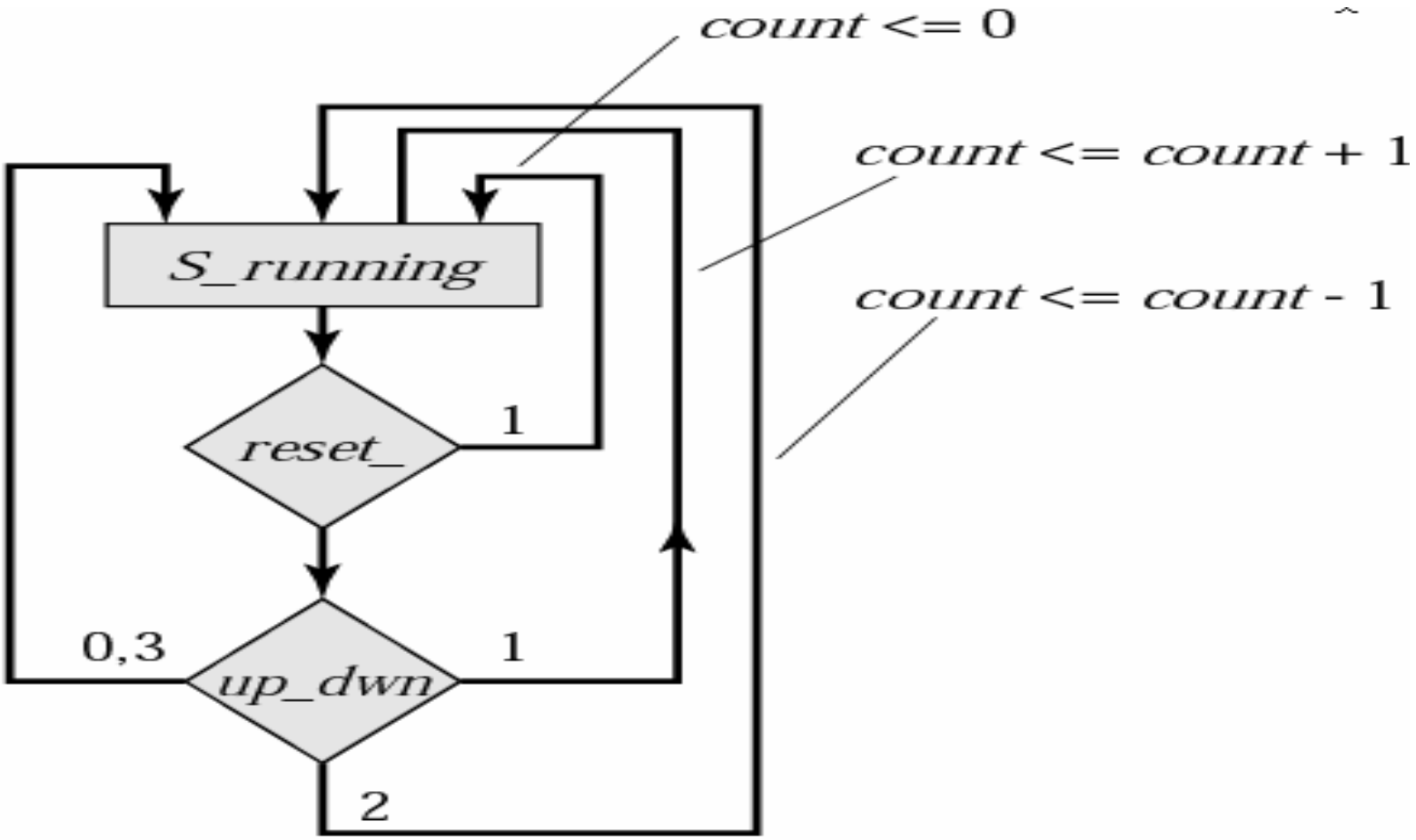
ASM chart with conditional output boxes for register operations



a simplified ASMD chart for a 4-bit binary counter with asynchronous active-low reset



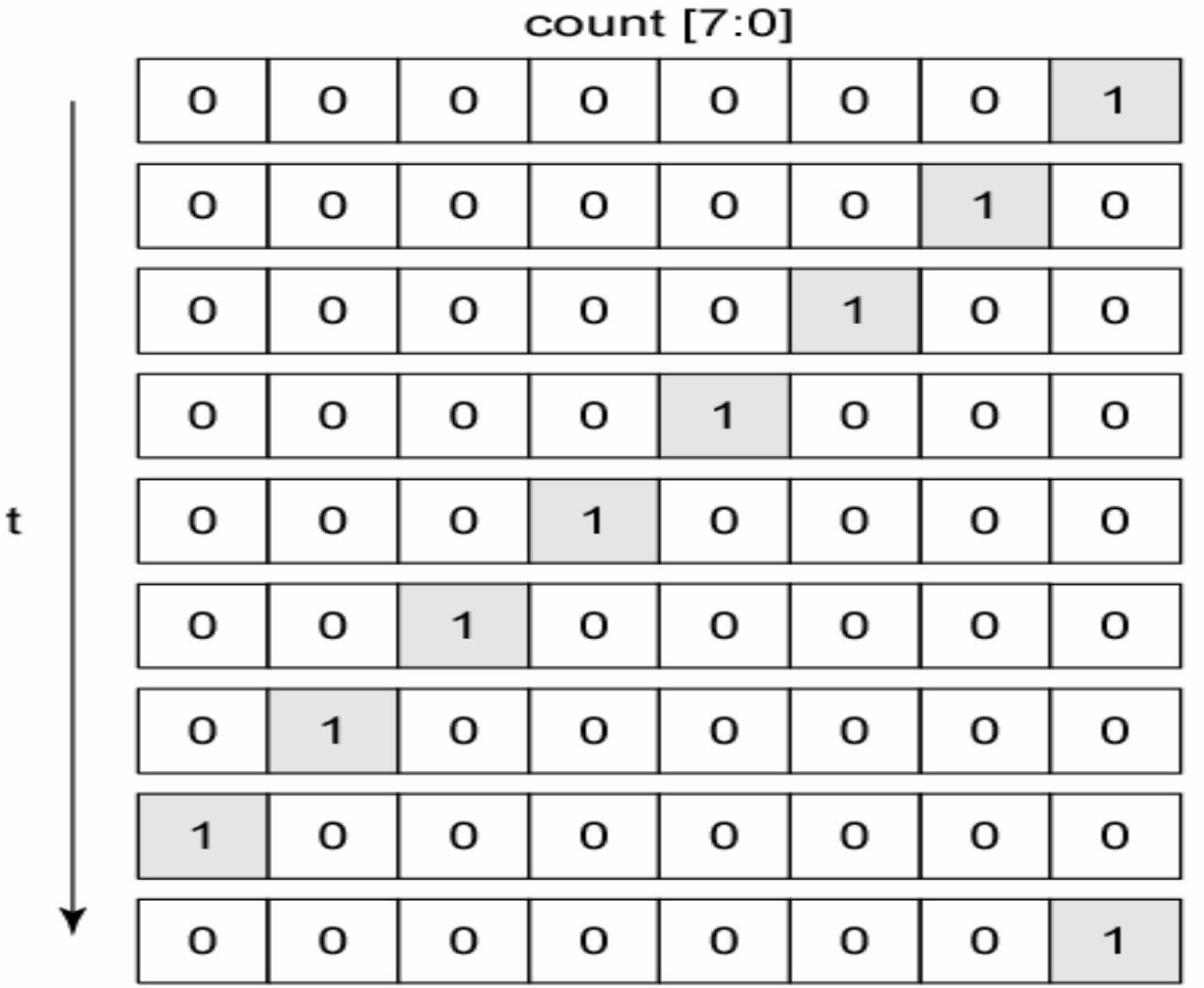
a simplified ASMD chart for a 4-bit binary counter with synchronous active-low reset



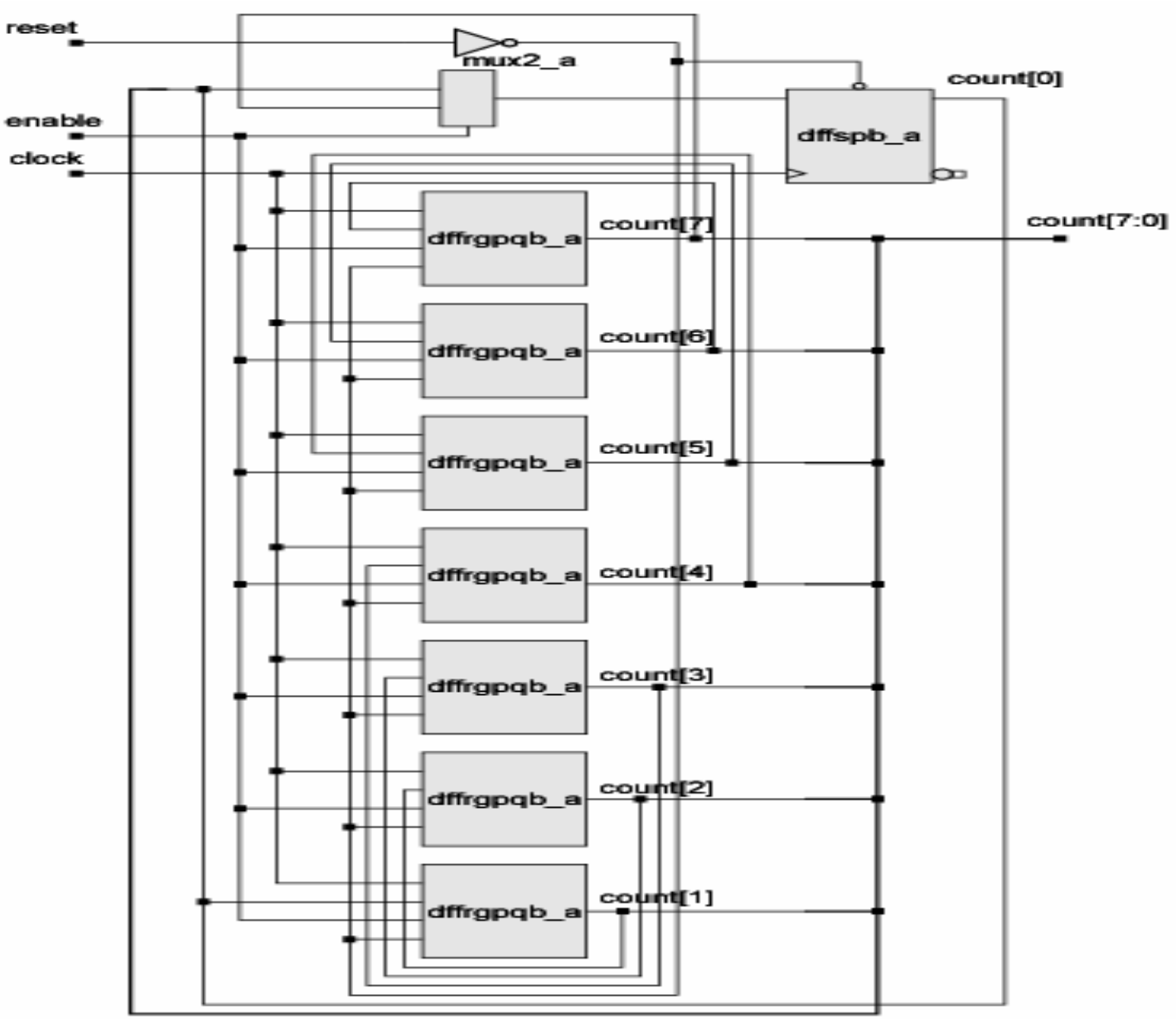
Example 5.41 a ring counter

```
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
```

Data movement in a 8-bit ring counter



ring counter



Example 5.42 3-bit up-down counter

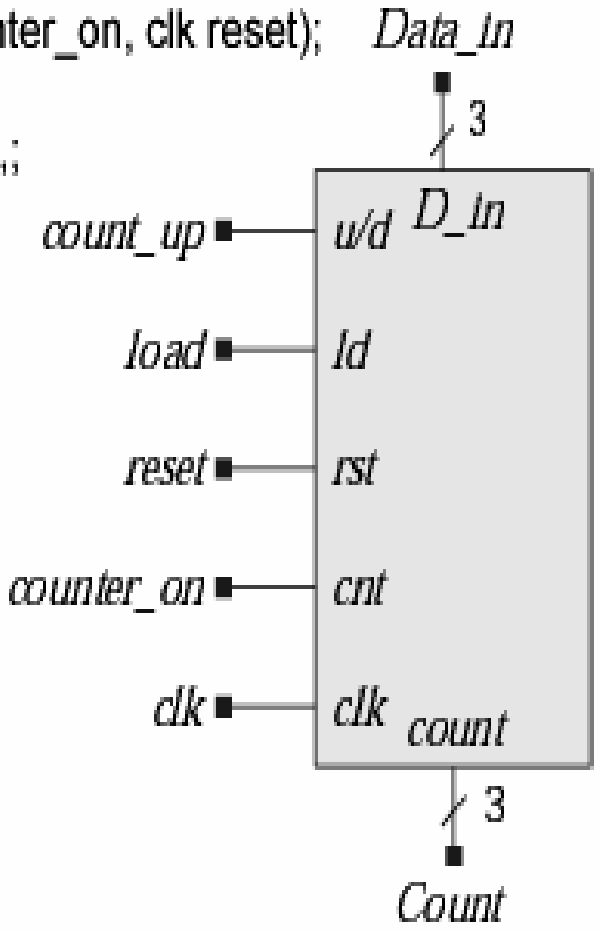
```
module up_down_counter
   (Count,Data_in,load,count_up,counter_on,clk,reset);
   output [2:0] Count;
   input          load, count_up, count_on, clk, reset;
   input   [2:0] Data_in;
   reg      [2:0] Count;
   always @ (posedge reset or posedge clk)
     if (reset==1'b1) Count=3'b0;
       else  if (load==1'b1) Count=Data_in;
         else  if (counter_on==1'b1)
            begin
             if (counter_up==1'b1)  Count=Count+1;
              else Count=Count-1;
            end
endmodule
```

Features

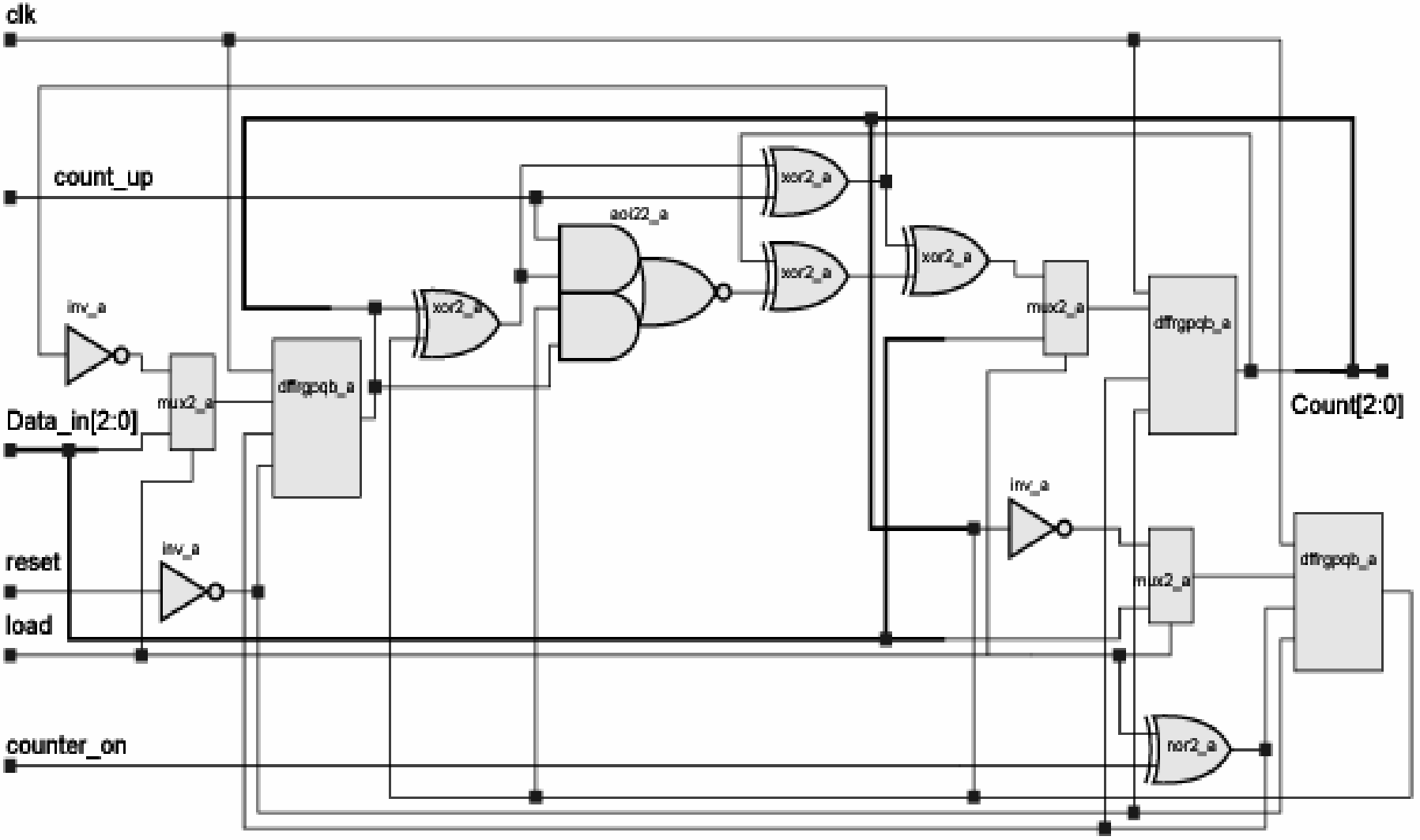
- Two additional features:
- a signal, counter_on, which enables the counter
- a signal, load, which loads an initial count from an external datapath

3-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
```



Synthesis circuit

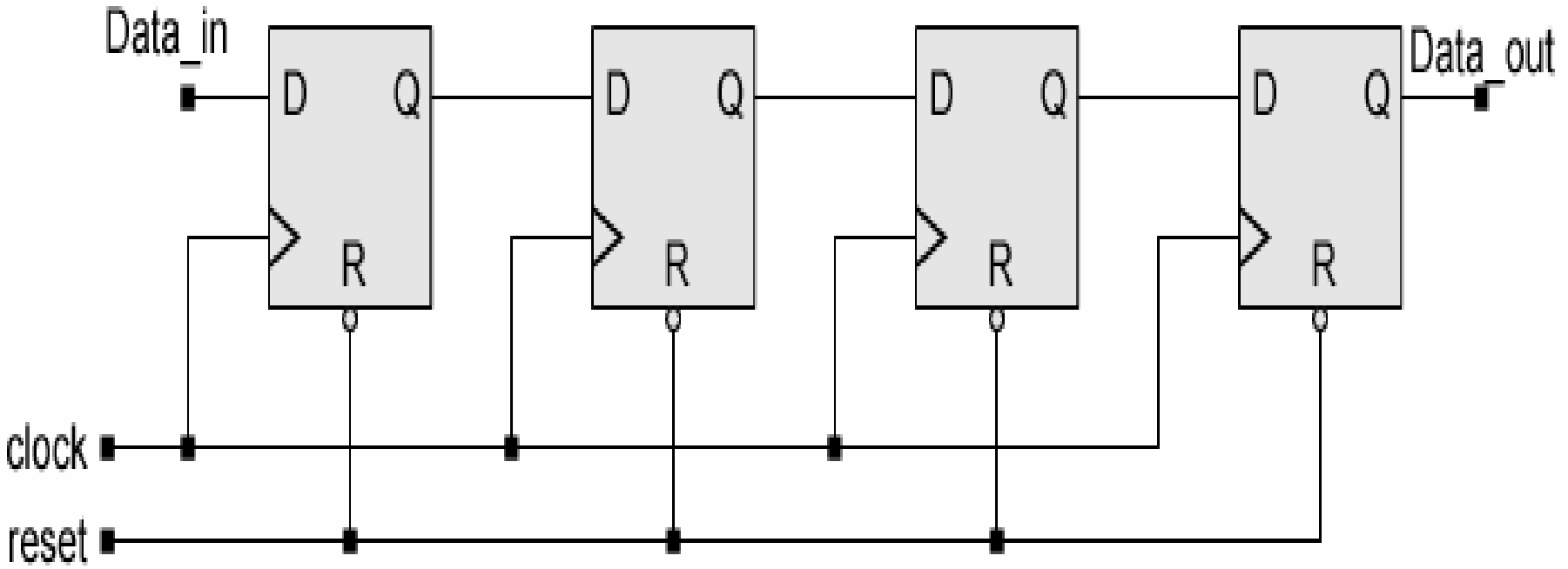


5.16.2 Shift Registers

- Example 5.43 an internal 4-bit 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
```

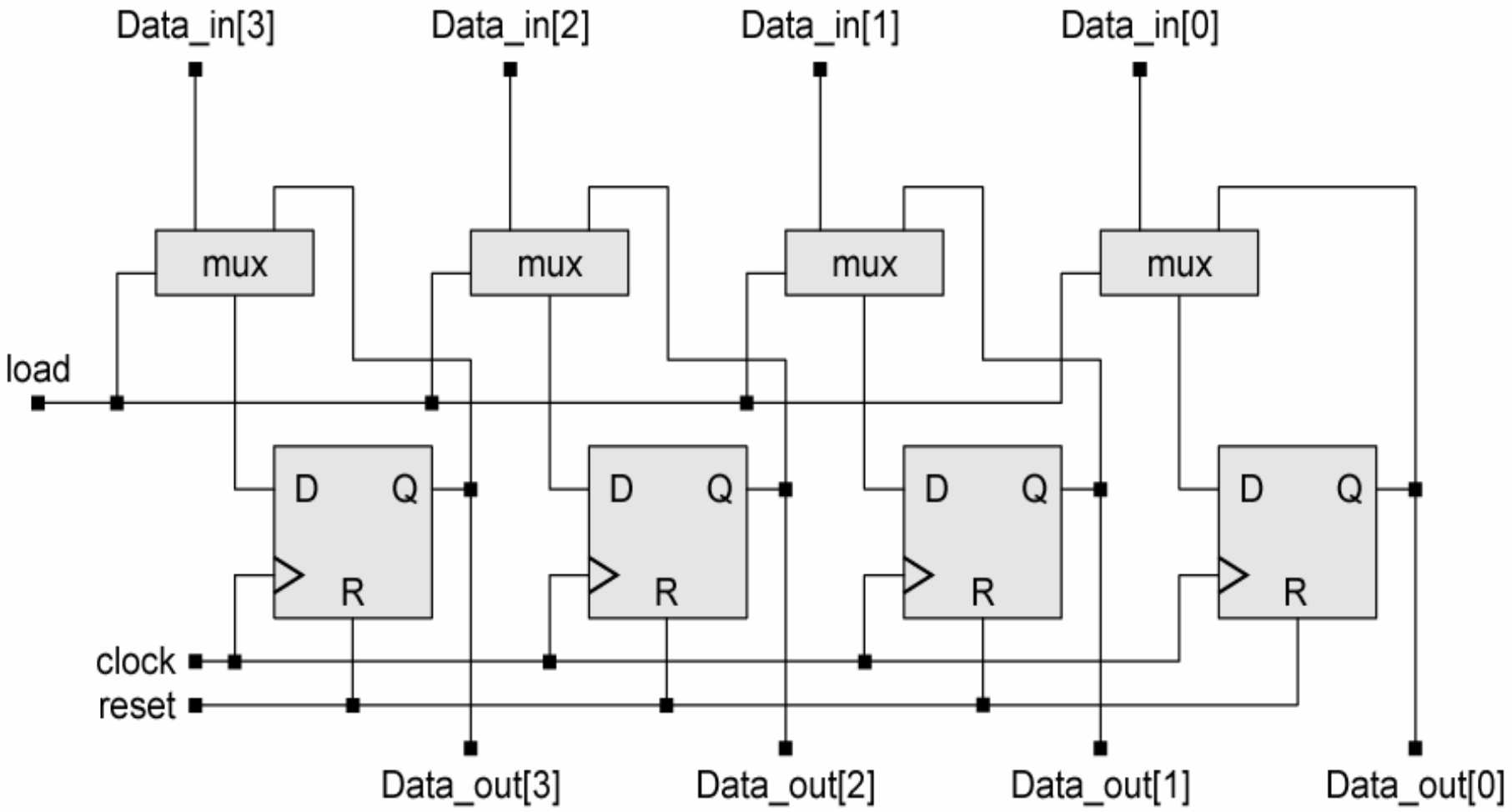
synthesis circuit



Example 5.44 a register with reset and parallel load

```
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 circuit

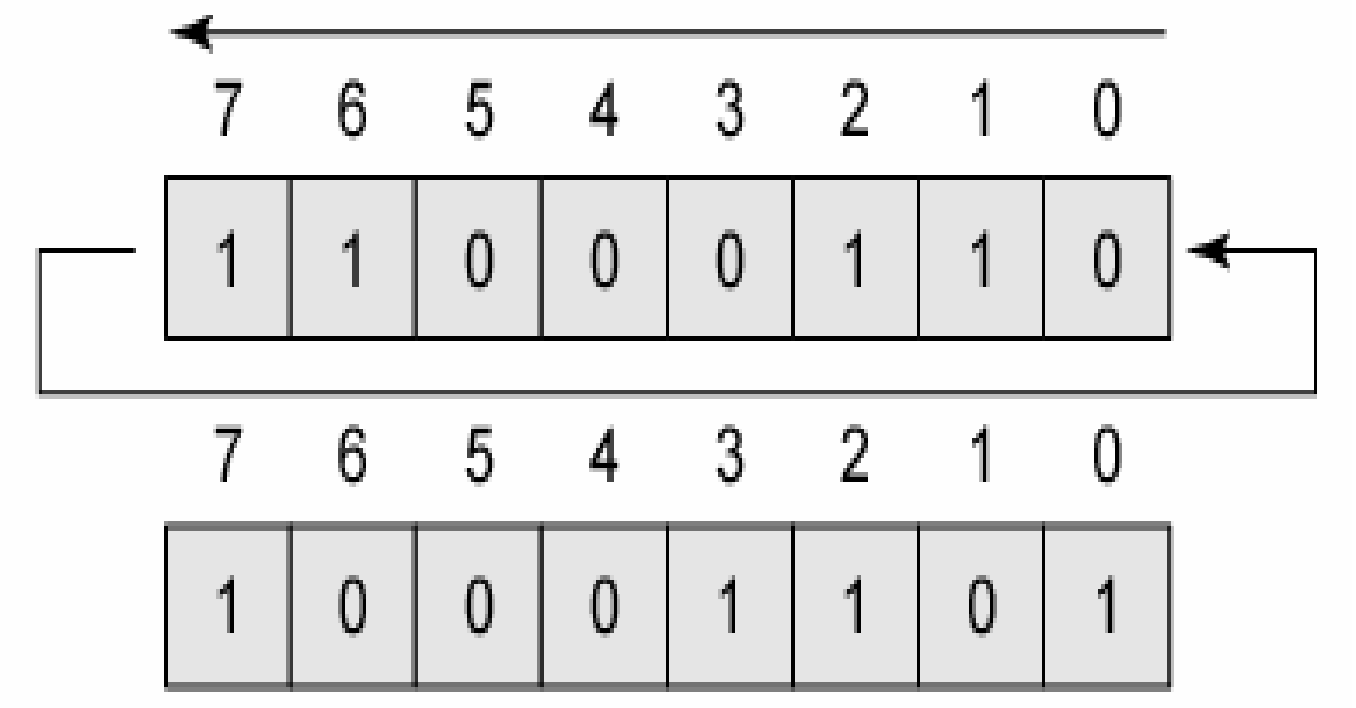


Example 5.45 barrel shifter

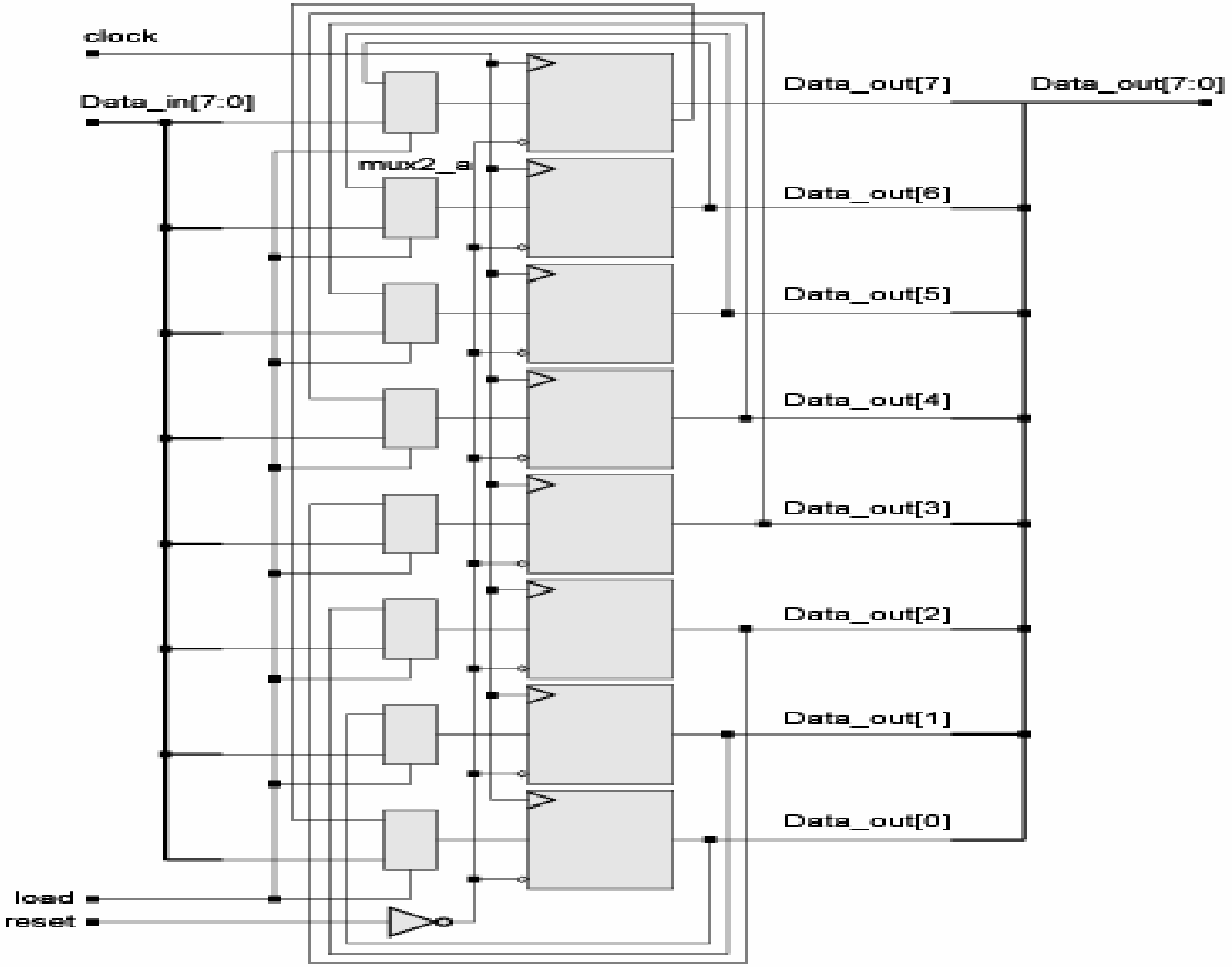
- Shifting a word to the right divides the word by 2; shifting a word to the left multiplies the word by 2

```
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
```

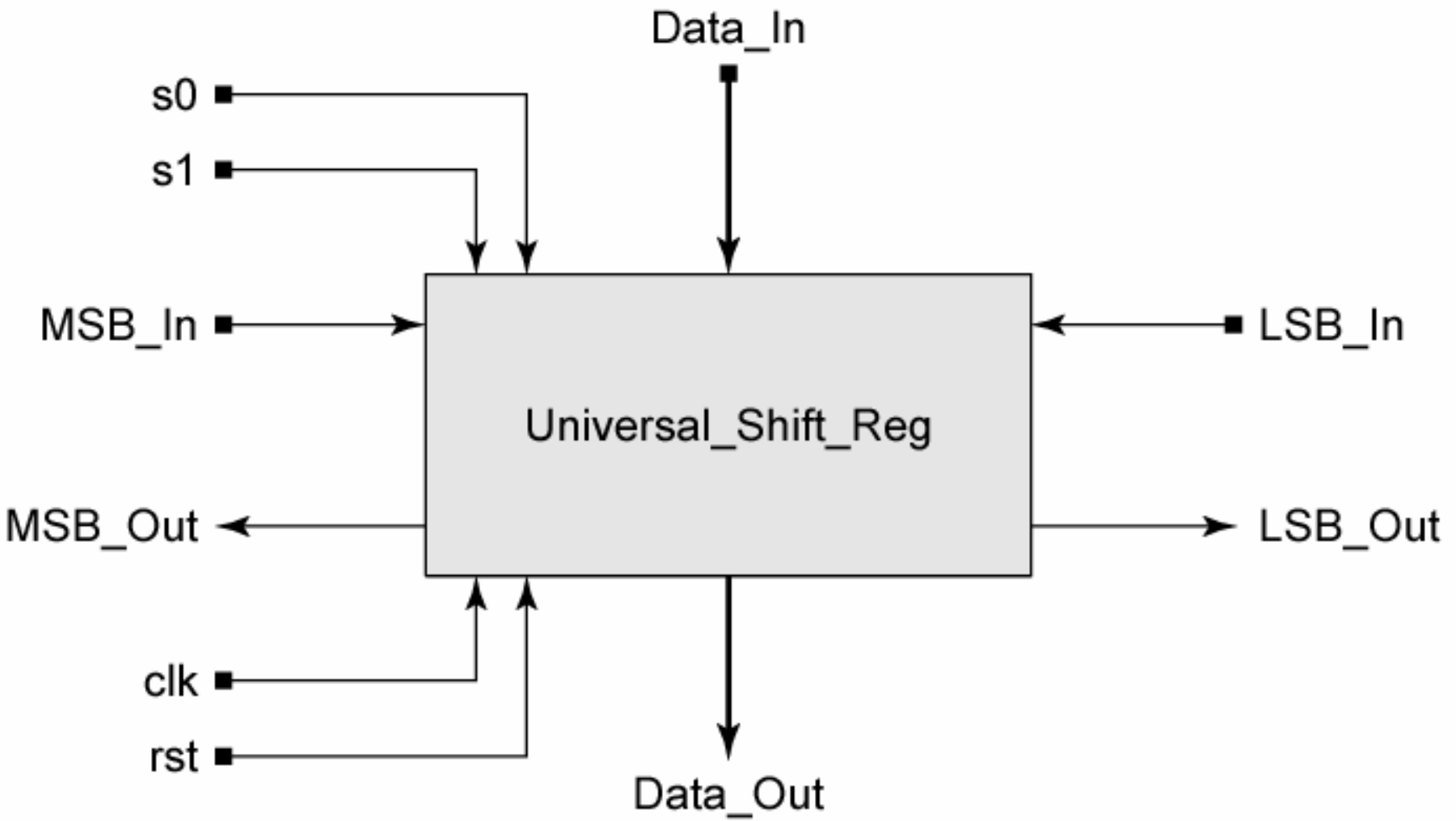
Data movement



synthesis circuit



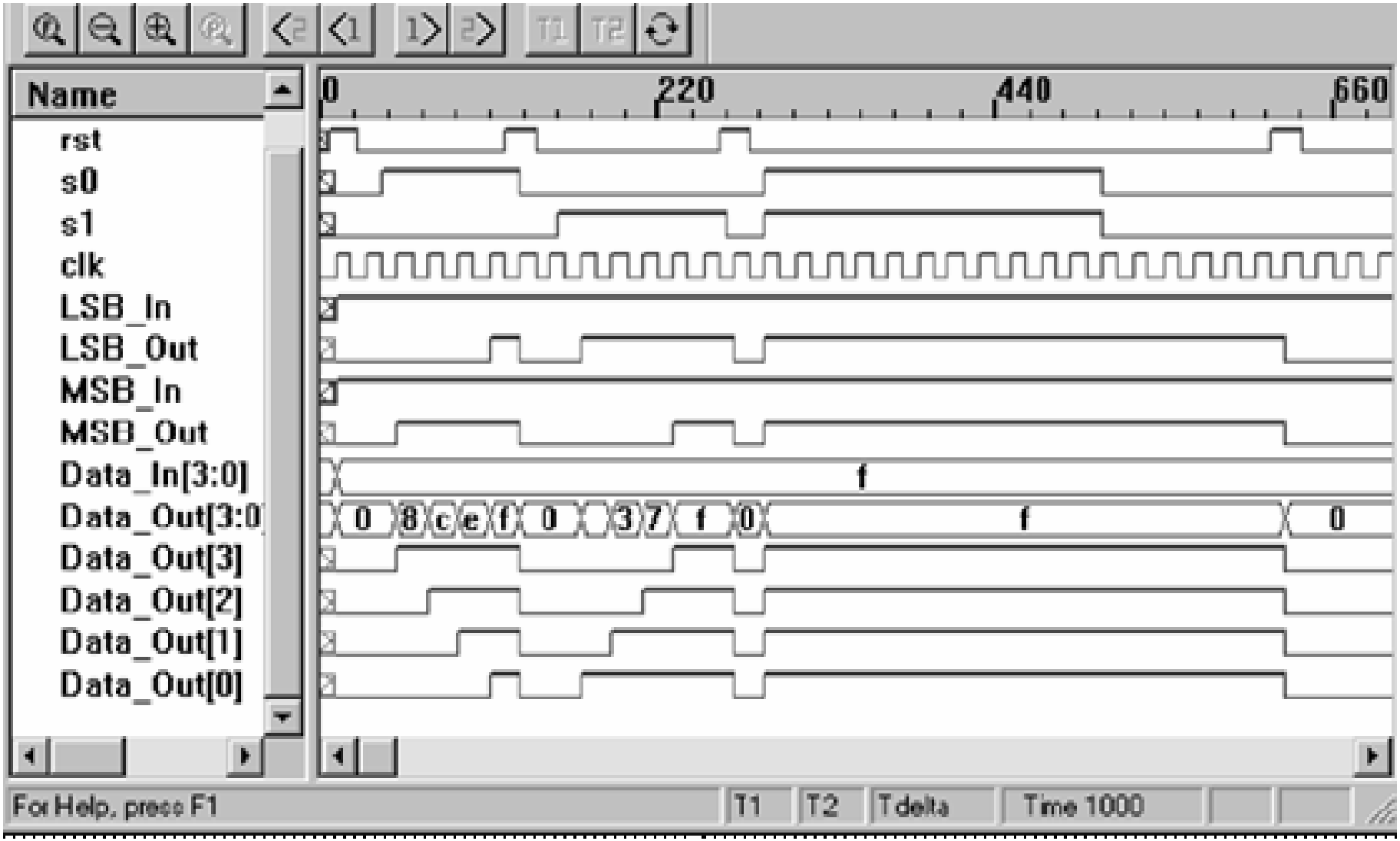
Example 5.46
a 4-bit universal shift register



Ex 5.46 a 4-bit 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          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[2:1],LSB_In};
                                 // Serial shift from LSB
     3: Data_Out<=Data_In;  // Parallel Load
     endcase
   end
endmodule
```

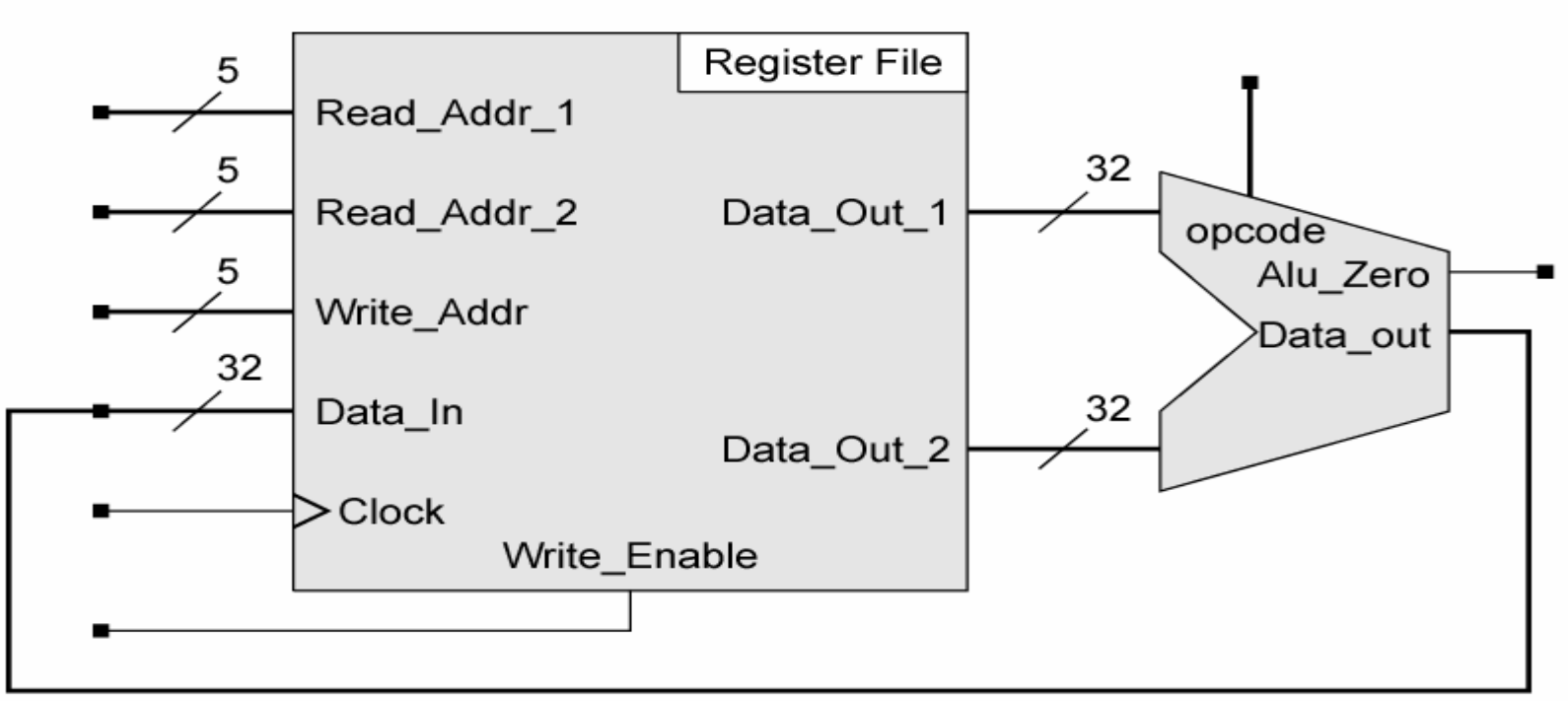
Simulation result



5.16.3 Register Files and Arrays of Registers (Memories)

- A register file consists of a small number of registers and is integrated with additional logic
- Usually implemented by D-type flip-flops, register files are not used for mass storage because they occupy significantly more silicon area than compiled memory
- A common application combines a register file in tandem with an arithmetic and logic unit (ALU), as shown in Figure 5-34

A register file in tandem with an Arithmetic and Logic Unit (ALU)



A register file in tandem with an ALU

- The dual-channel outputs of the register file form the datapaths to the ALU, and the output of the ALU is stored in the register file at a designated location
- A host processor provides the addresses for the operations and controls the sequence of reading and writing to prevent a simultaneous read and write affecting the same location

Ex 5.47 single-input, dual-output 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×32 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
```

5.17 Switch Debounce, Metastability, and Synchronizers for Asynchronous Signals

- A hardware latch may enter the metastable

1. if a pulse at one of its inputs is too short
2. if both inputs are asserted either simultaneously or within a sufficiently small interval of each other
3. If the data are unstable around the edge of the enable input

A D-type Flip-Flop may enter metastable

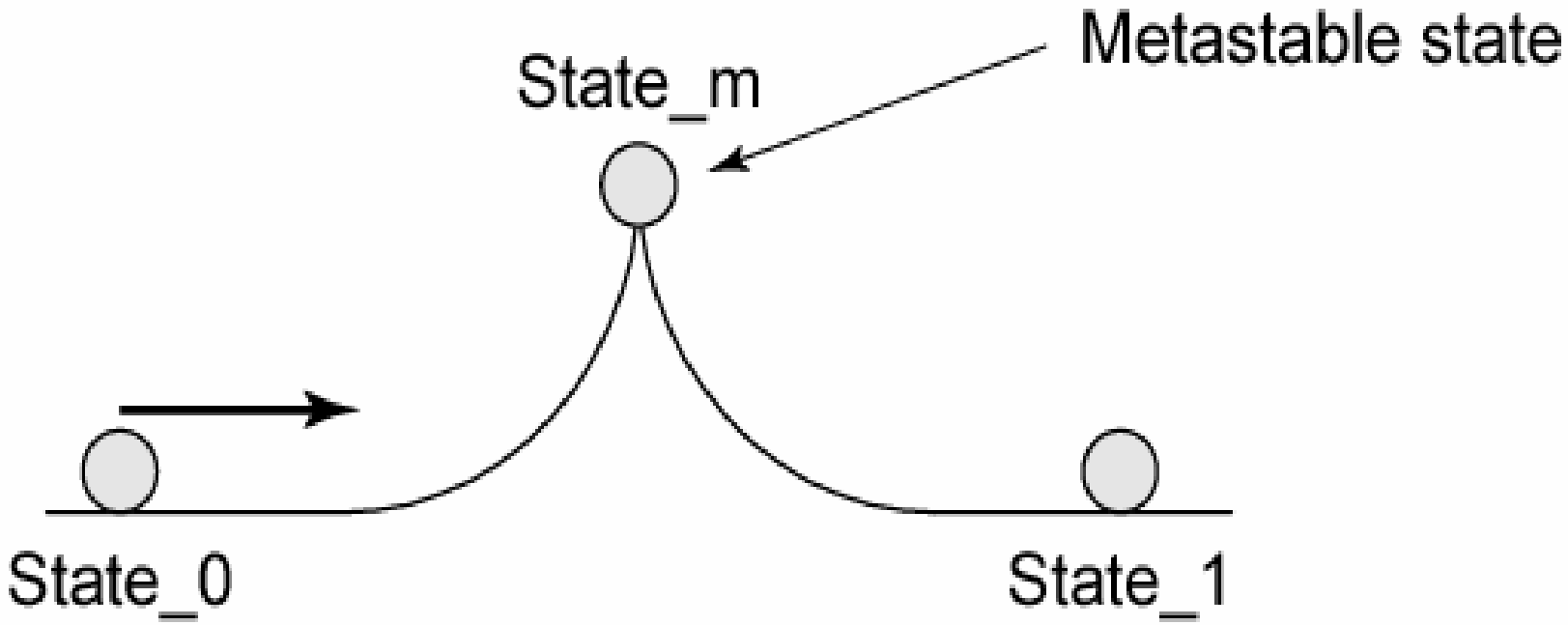
A DFF is metastable:

- if the data are unstable in the setup interval preceding the clock edge
- if the clock pulse is too narrow

A signal is asynchronous:

- If it is not controlled by a clock
- If it is synchronized by a clock in a different domain
- In both cases, a signal transition can occur in a random manner with respect to the active edge of the clock that is controlling sequential devices

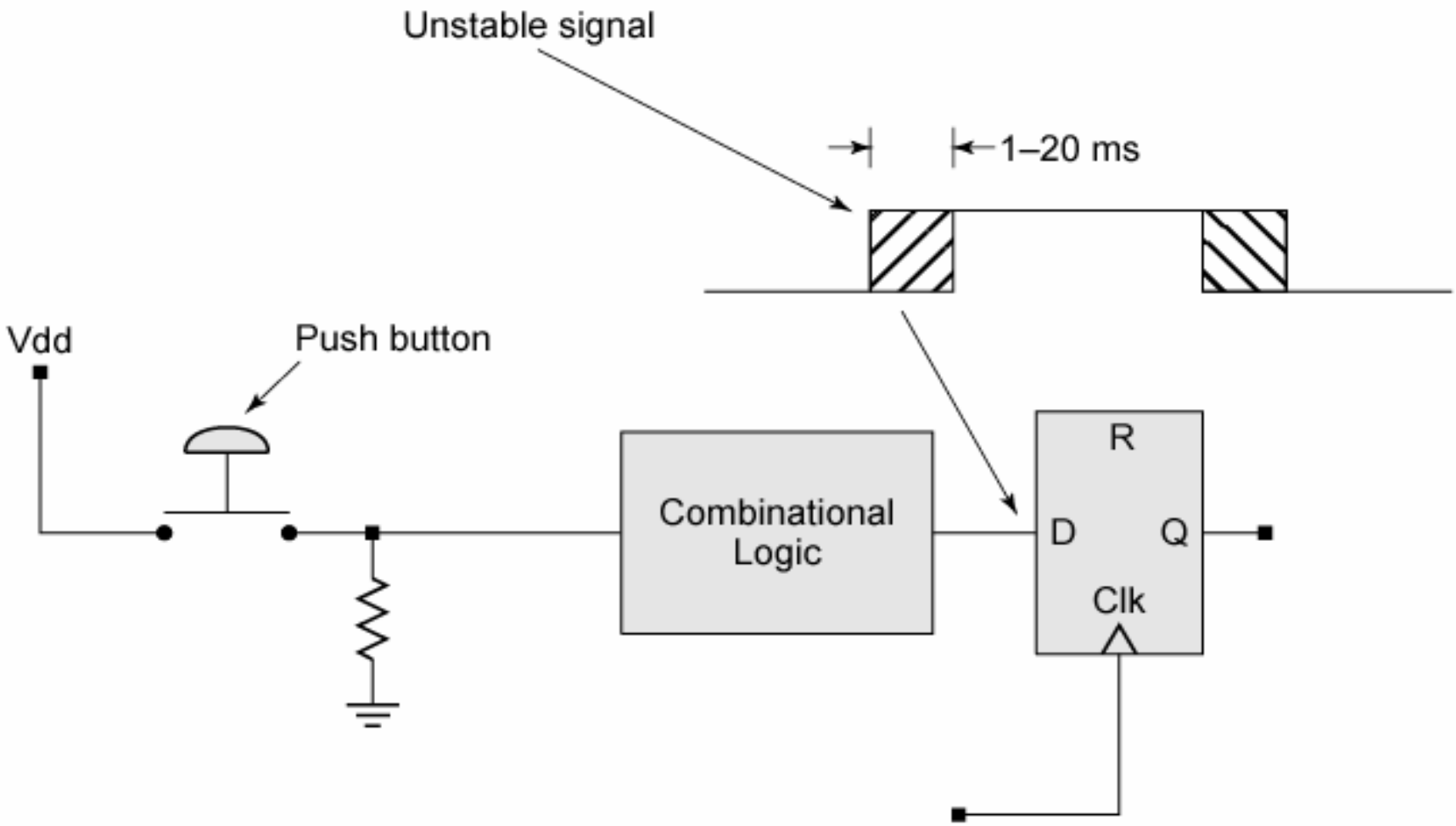
An illustration of how metastability can happen in a physical system



Switch bounce

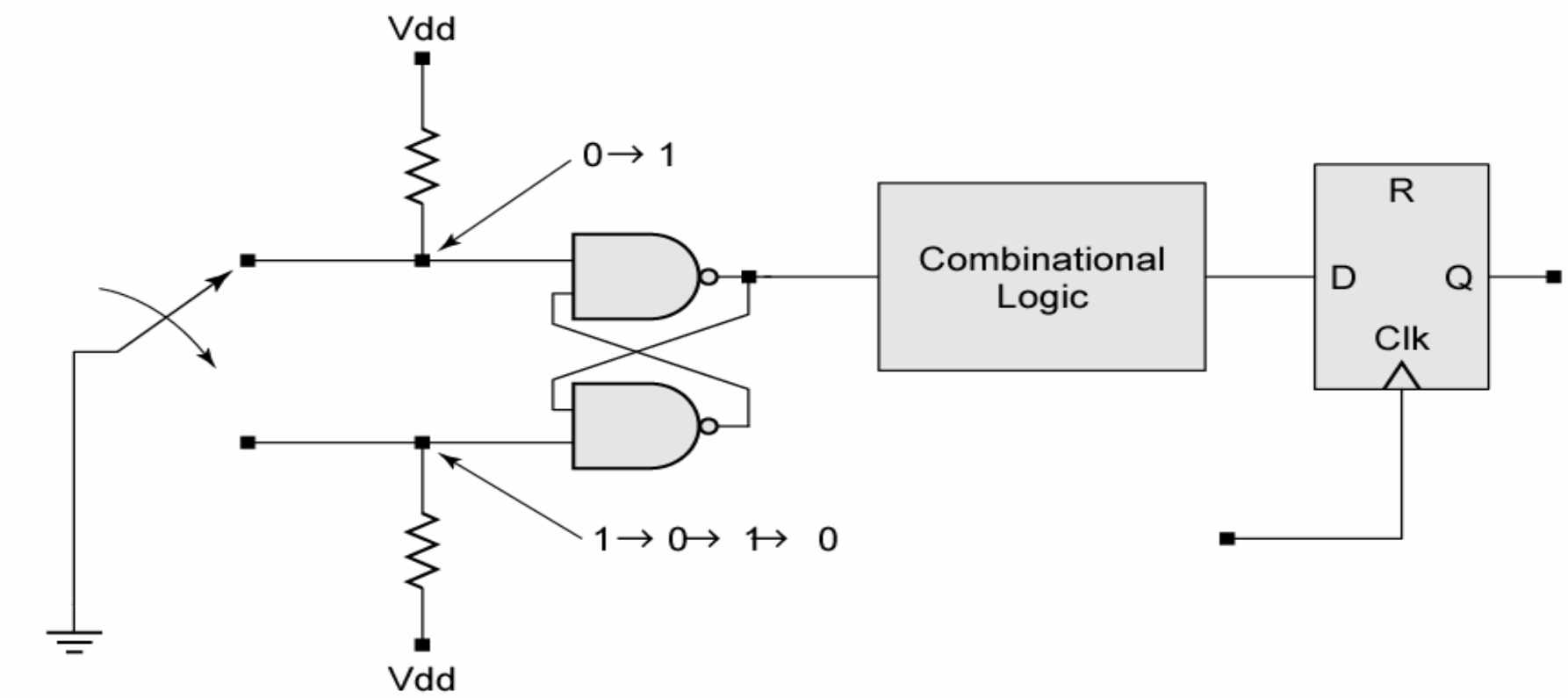
- If a mechanical switch generates an input that drives a DFF, **the input signal could oscillate during the setup interval and enter a metastable state**
- The mechanical contact will vibrate momentarily, **for a few milliseconds**, creating an unstable signal on the line
- There are various ways to deal with switch bounce, depending on the application
- For example, the push button switches have a resistor-capacitor (RC) lowpass filter and a buffer placed between the switch and the chip

a push-button input device with closure bounce



A NAND latch configuration for eliminating the effects of switch closure bounce

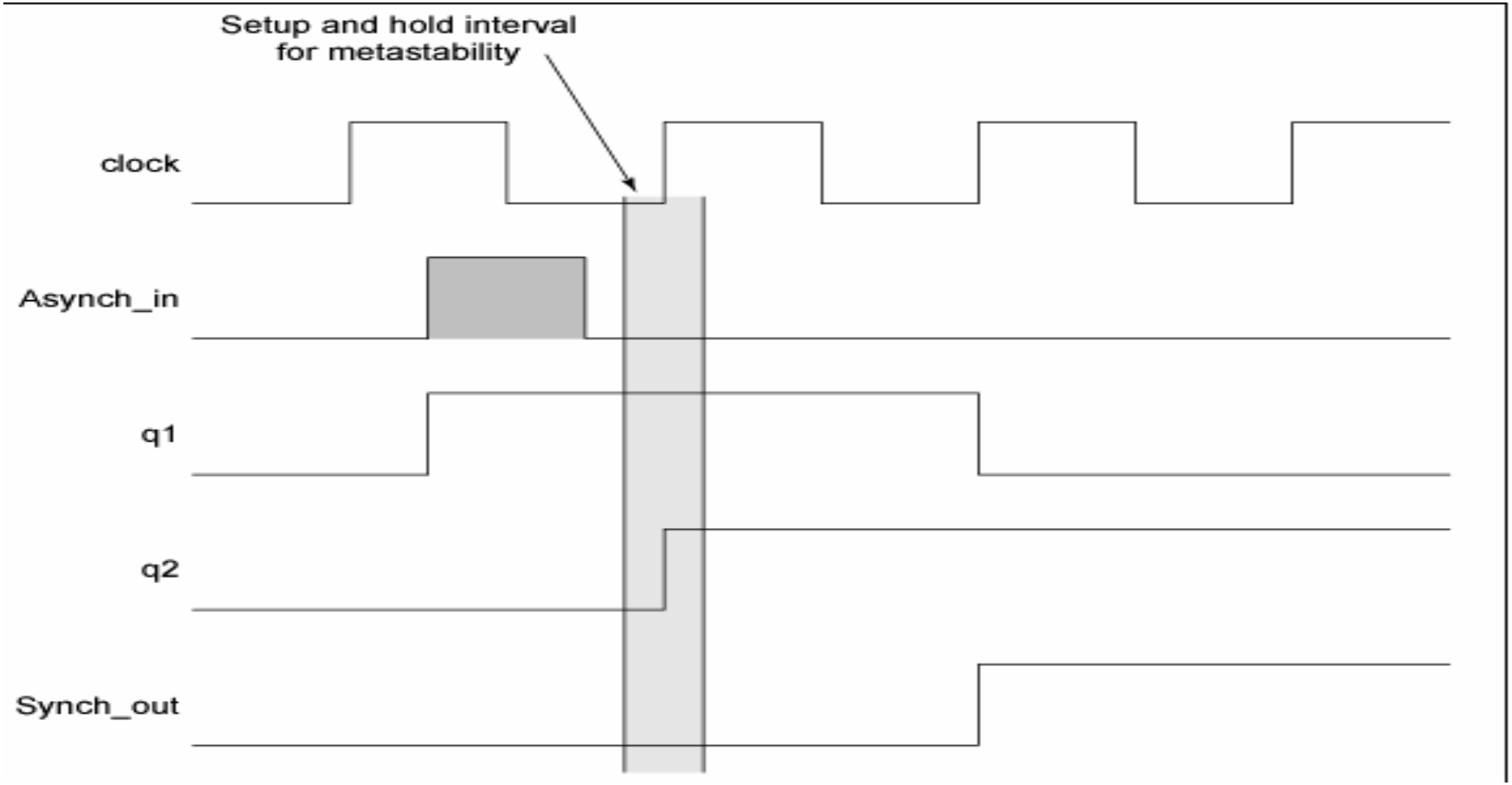
(a single pole-double throw switch)



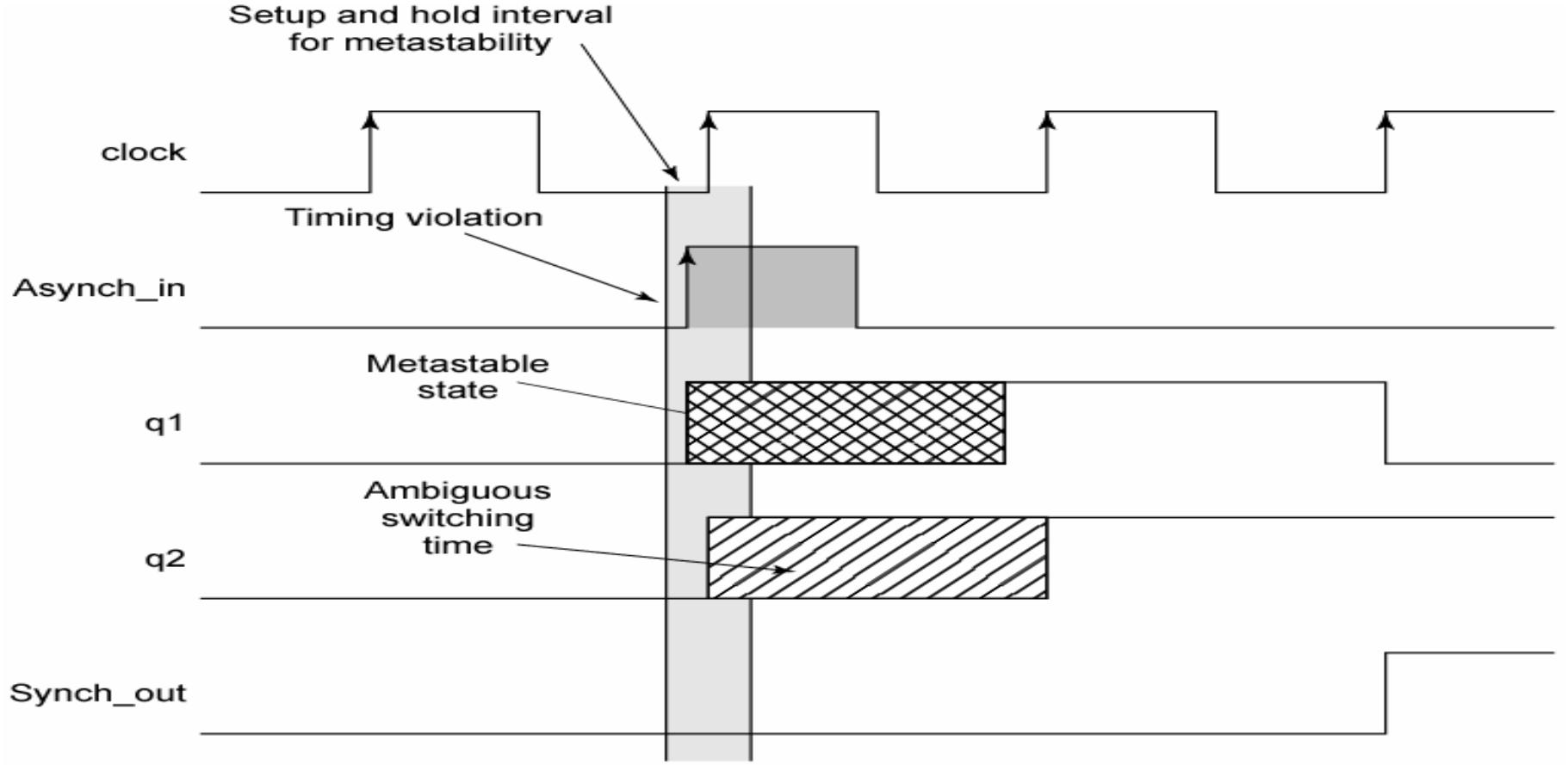
Metastability

- **It cannot be prevented** ,Its effect can be reduced
- The mean time between failures with an asynchronous input is exponentially related to the length of time available for recovery from the metastable condition
- High-speed digital circuits **rely on synchronizers to create a time buffer** for recovering from a metastable event
- **First rule is that an asynchronous signal should never be synchronized by more than one synchronizer**

waveforms in the circuit of the asynchronous pulse does not cause a metastable condition



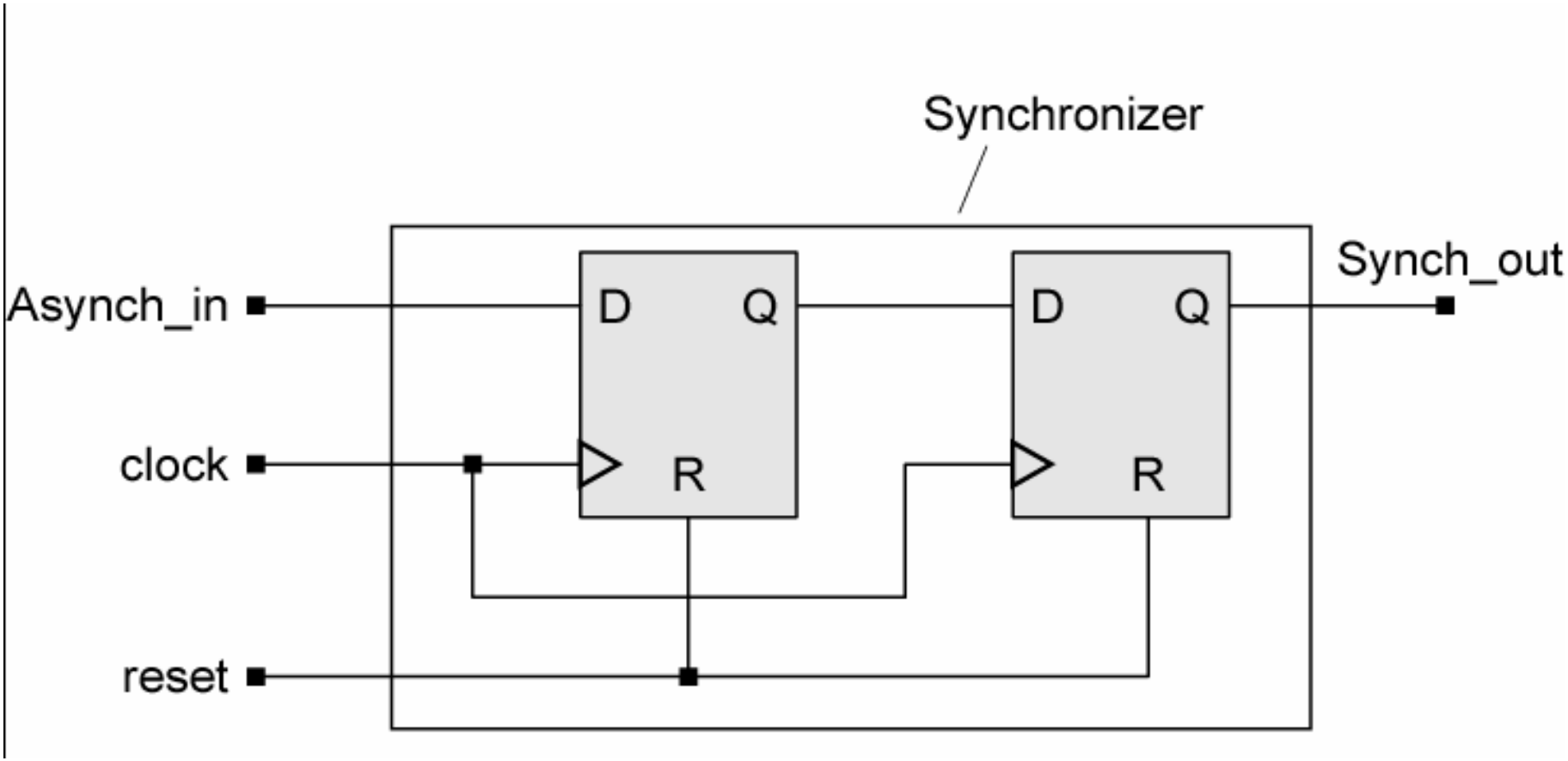
waveforms of the circuit in when the asynchronous input signal causes a metastable condition



Two basic types of synchronizers

- There are two basic types of synchronizer circuits
- Depending on whether the asynchronous input pulse has a width that is larger or smaller than the period of the clock

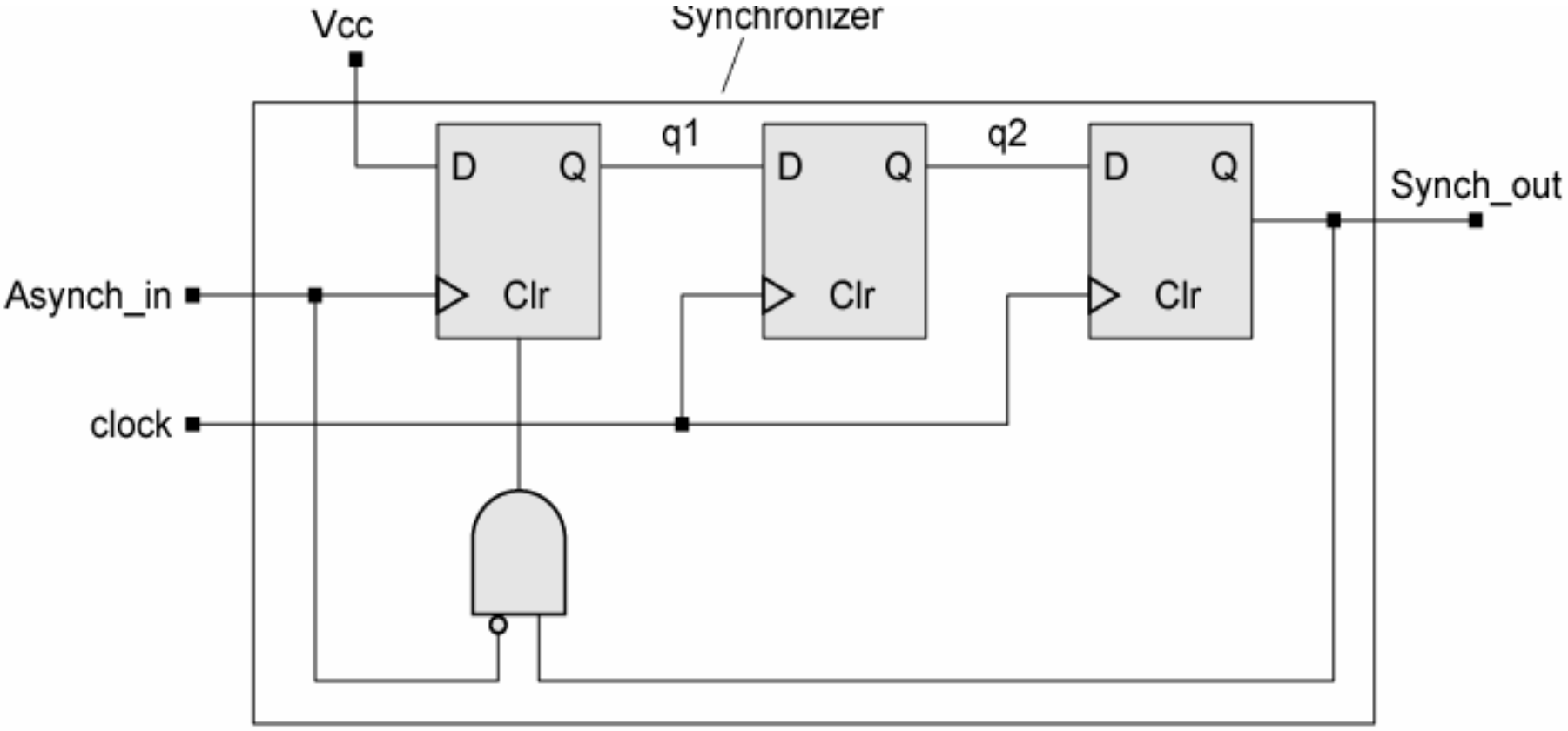
The width of the asynchronous input pulse is greater than the period of the clock



The width of the asynchronous input pulse is greater than the period of the clock

- 1、if the asynchronous input signal reaches a stable condition outside the setup interval, it will be clocked through with a latency of two cycles
- 2、if Asynch_in is unstable during the setup interval (due to bounce or to a late-arriving input) there are two possibilities
- If the unstable input is sampled as a 0, but ultimately settles to a 1, the 1 will appear at the output with a latency of three cycles
- If the signal settles to 0, it will have arrived with a latency of two cycles

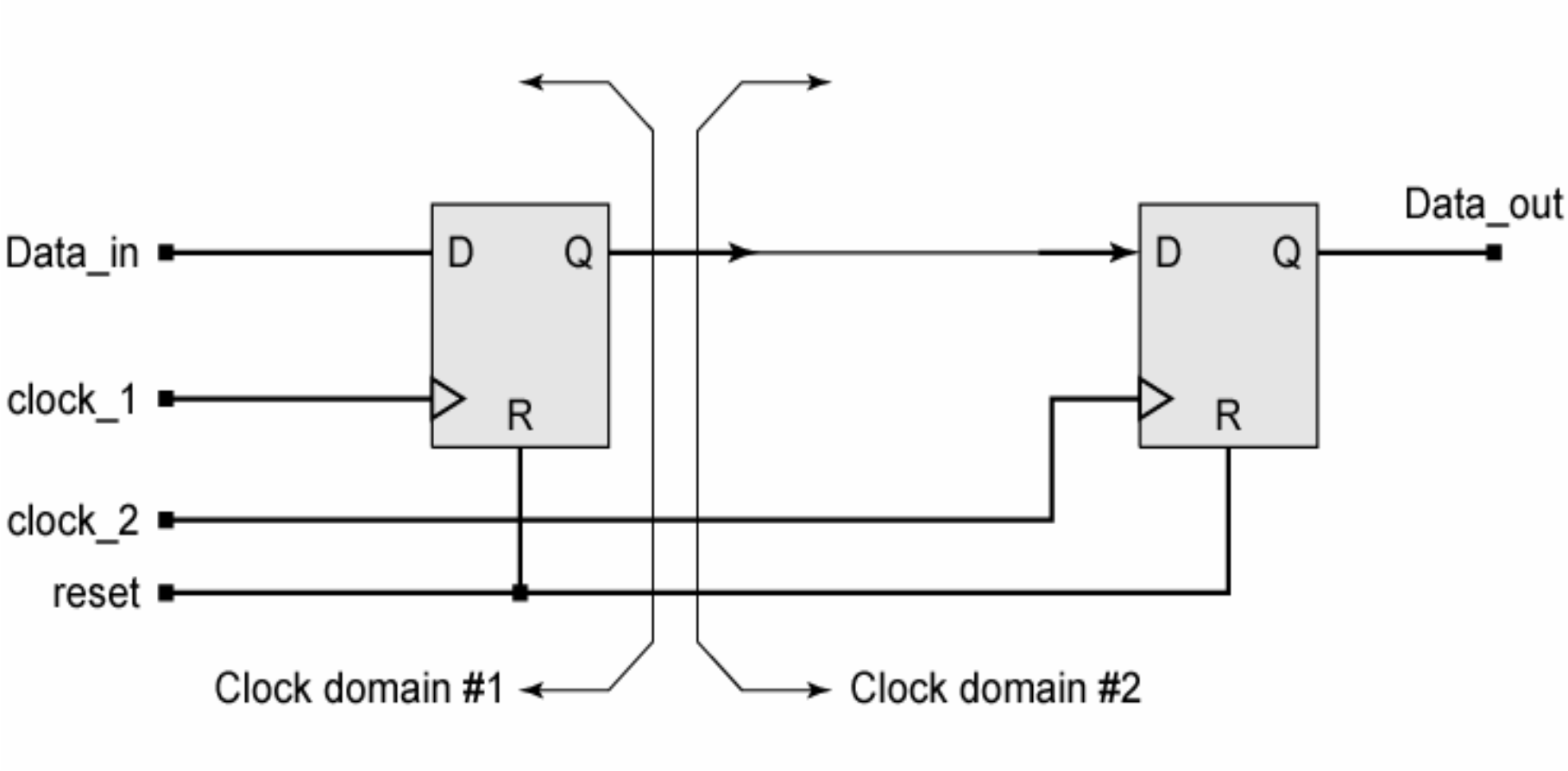
The width of the asynchronous input pulse is less than the period of the clock



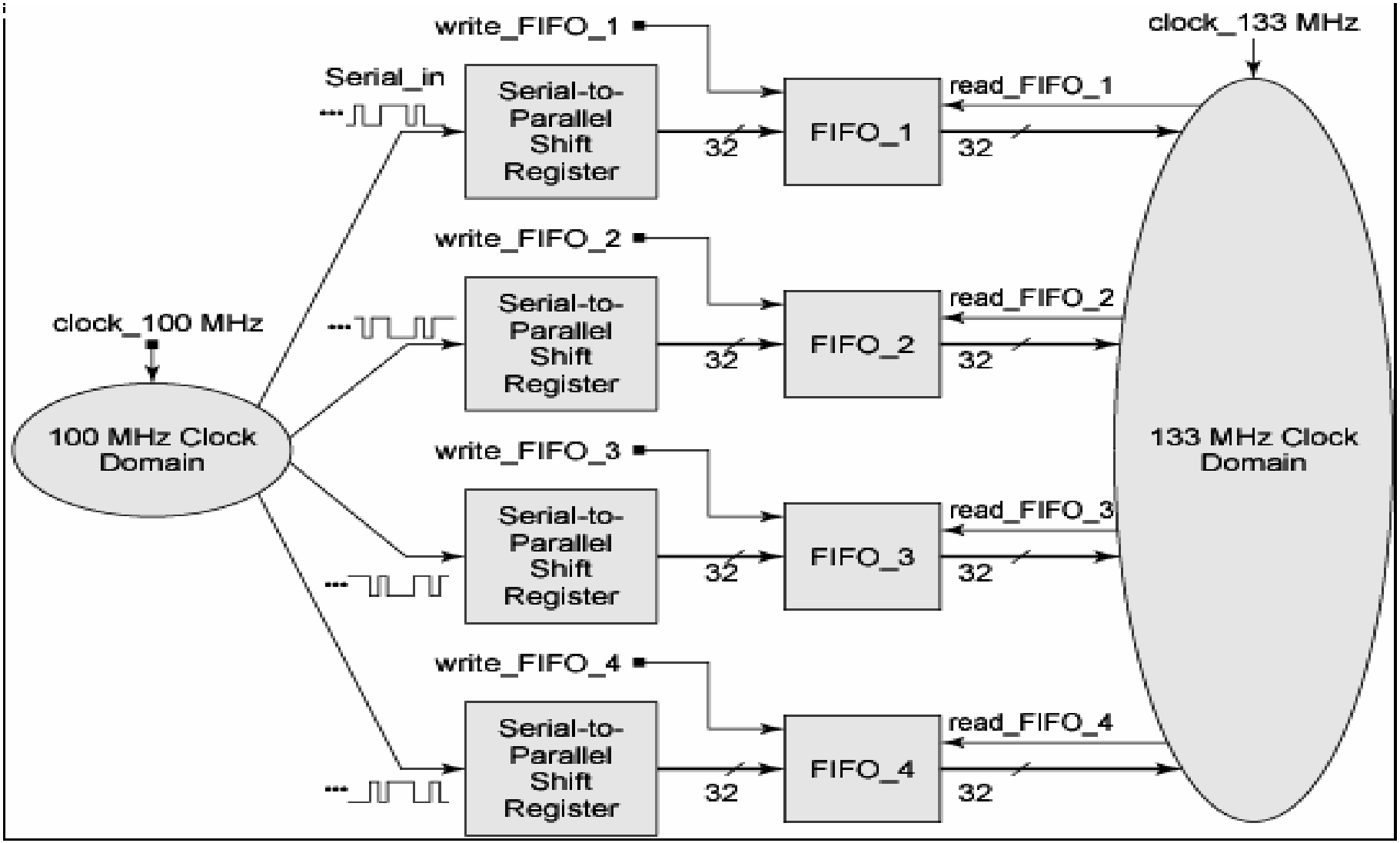
The width of the asynchronous input pulse is less than the period of the clock

- The first flip-flop has Vcc connected to its data input and has Asych_in connected to its clock input
- A short pulse at Asynch_in will drive q1 to 1; this value will propagate to Synch_out after the next two clock edges
- When Synch_out becomes 1, the signal at Clr is asserted, assuming that Asynch_in returns to 0 before Synch_out becomes 1
- The flip-flop driving q2 guards against metastability in the first stage of the chain

Synchronizers are also used when a signal must cross a boundary between two clock domains



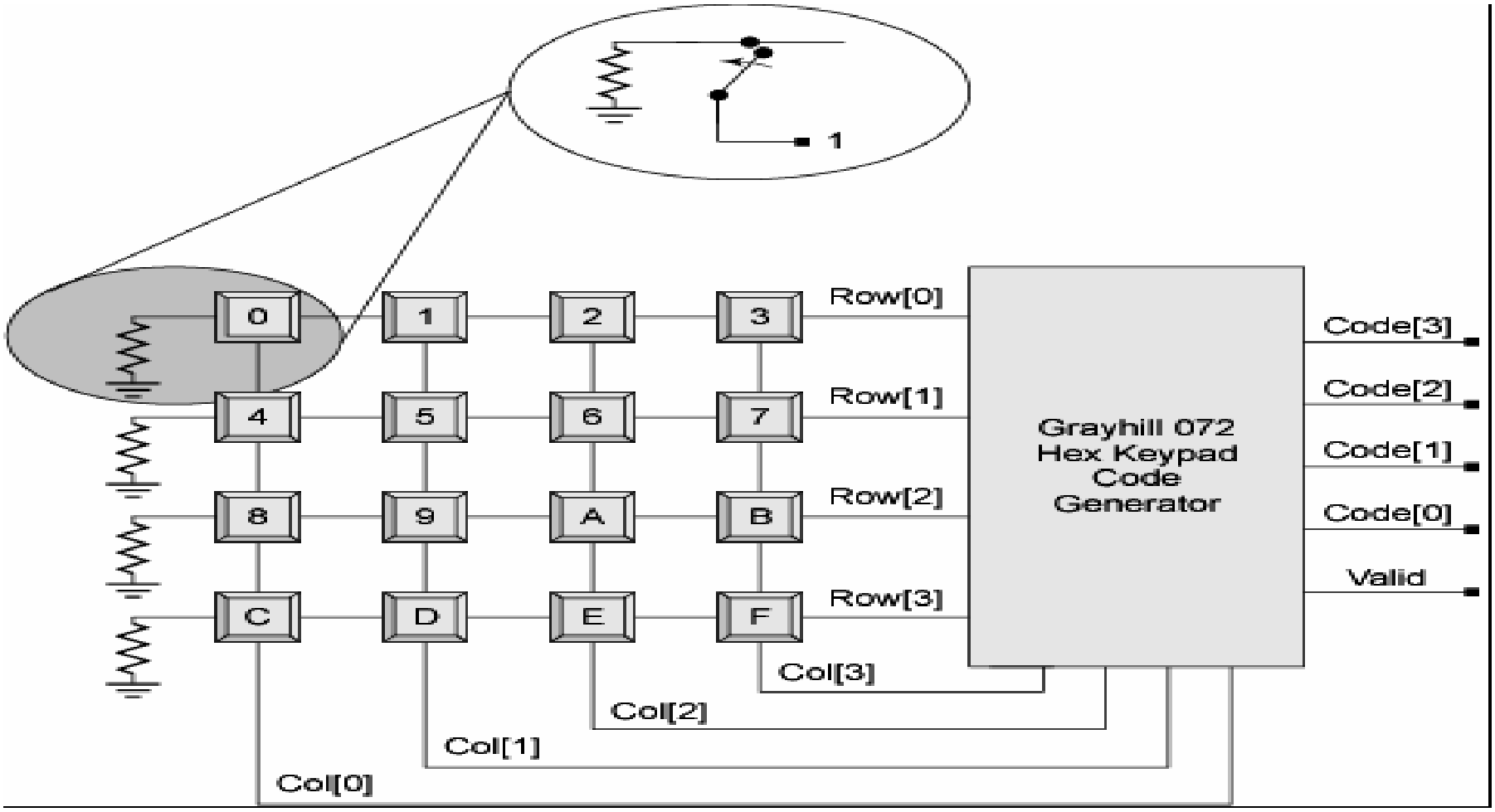
Clock domain interface



5.18 design Example: keypad Scanner and Encoder

- Keypad scanners are used to enter data manually in digital telephones, computer keyboards and other digital systems
- **A keypad scanner responds to a pressed key and forms a code that uniquely identifies the key that is pressed**
- Must take into account the asynchronous nature of the inputs and deal with switch debounce

Scanner/encoder for a hexadecimal keypad



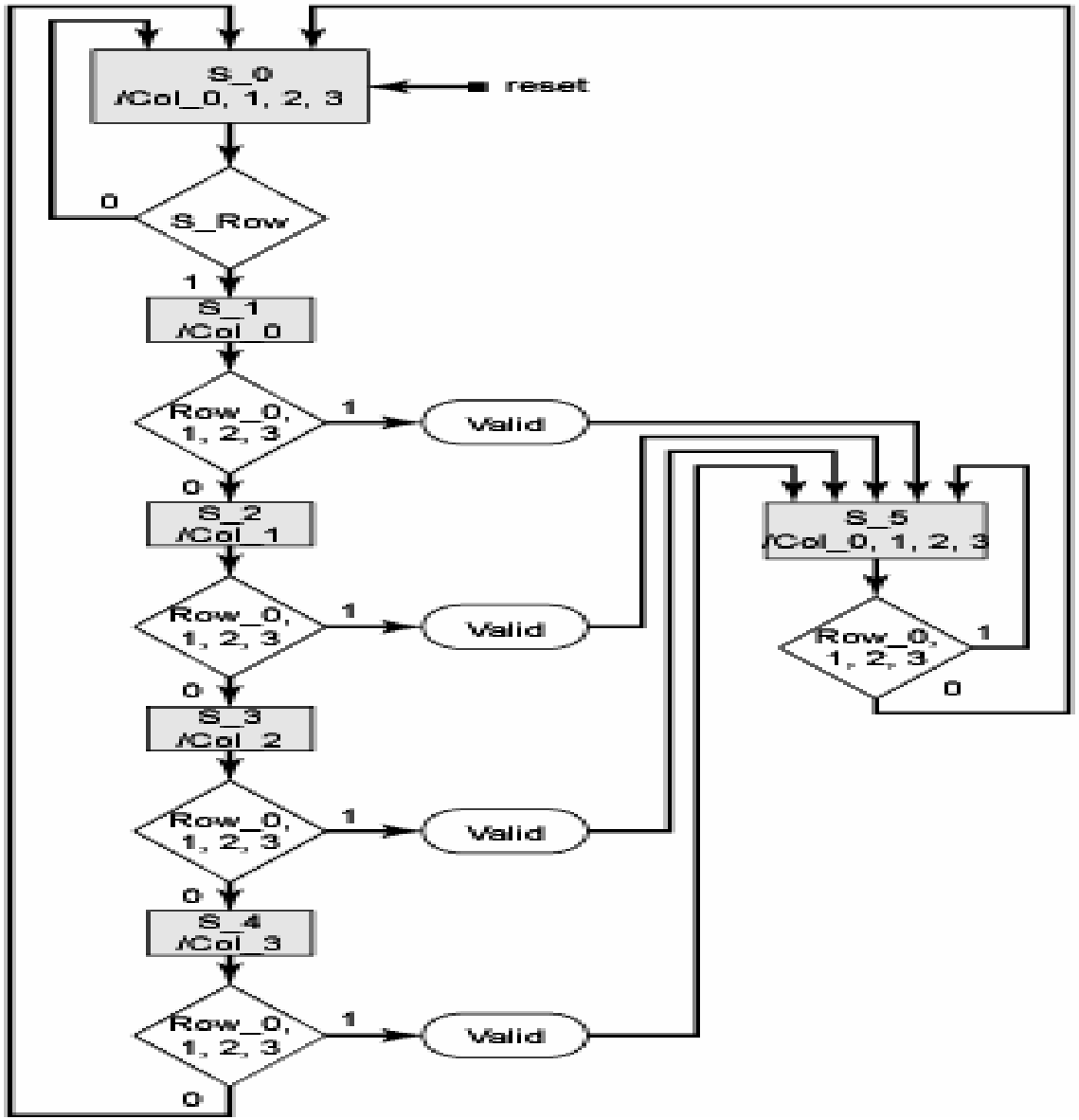
A keypad code generator

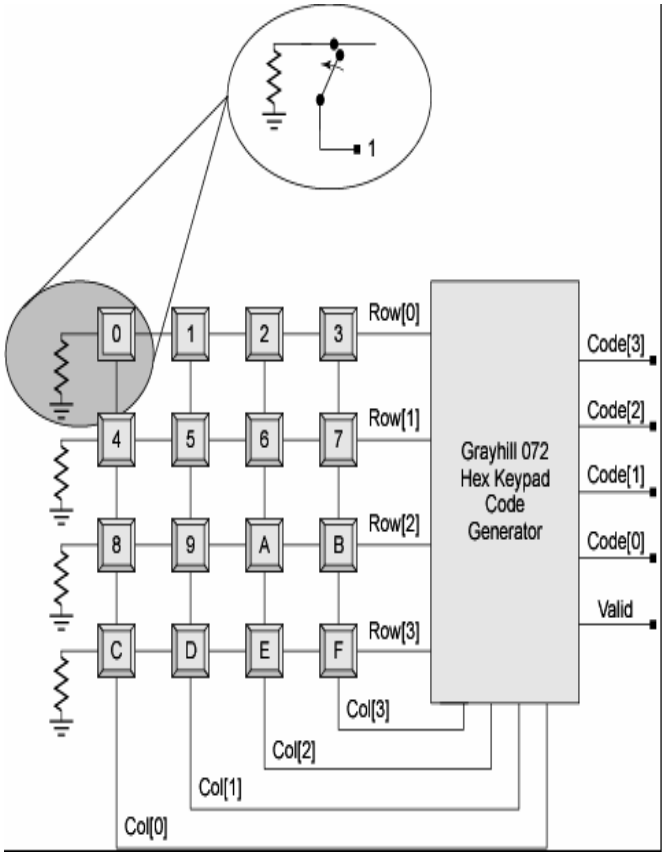
- A keypad code generator must implement a decoding scheme
- Detects whether a button is pressed
- Identifies the button that is pressed
- Generates an output consisting of the unique code of the button

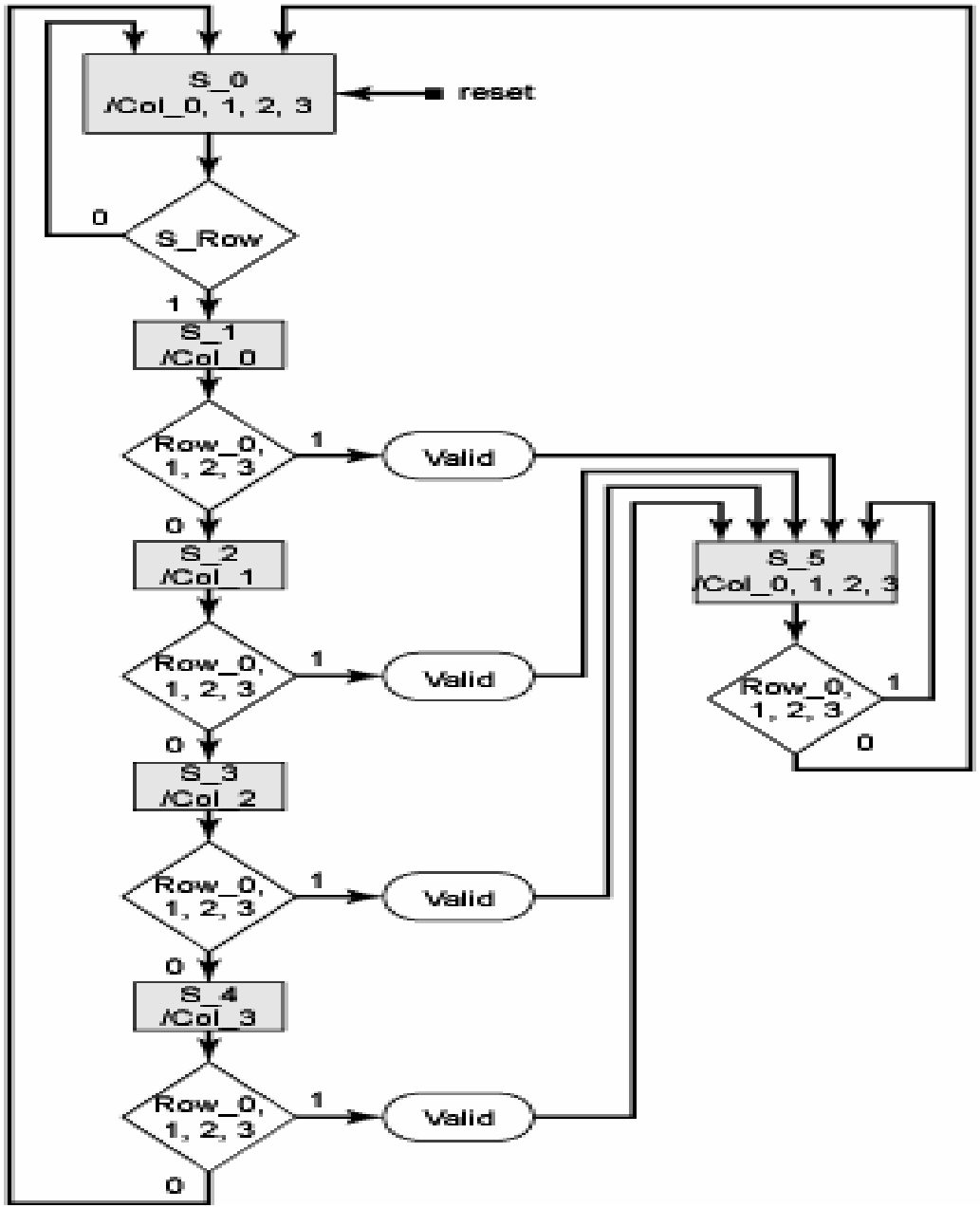
Keypad codes for a hexadecimal scanner

Key	Row[3:0]	Col[3:0]	Code
0	0001	0001	0000
1	0001	0010	0001
2	0001	0100	0010
3	0001	1000	0011
4	0010	0001	0100
5	0010	0010	0101
6	0010	0100	0110
7	0010	1000	0111
8	0100	0001	1000
9	0100	0010	1001
A	0100	0100	1010
B	0100	1000	1011
C	1000	0001	1100
D	1000	0010	1101
E	1000	0100	1110
F	1000	1000	1111

The behavior of the keypad scanner/ encoder



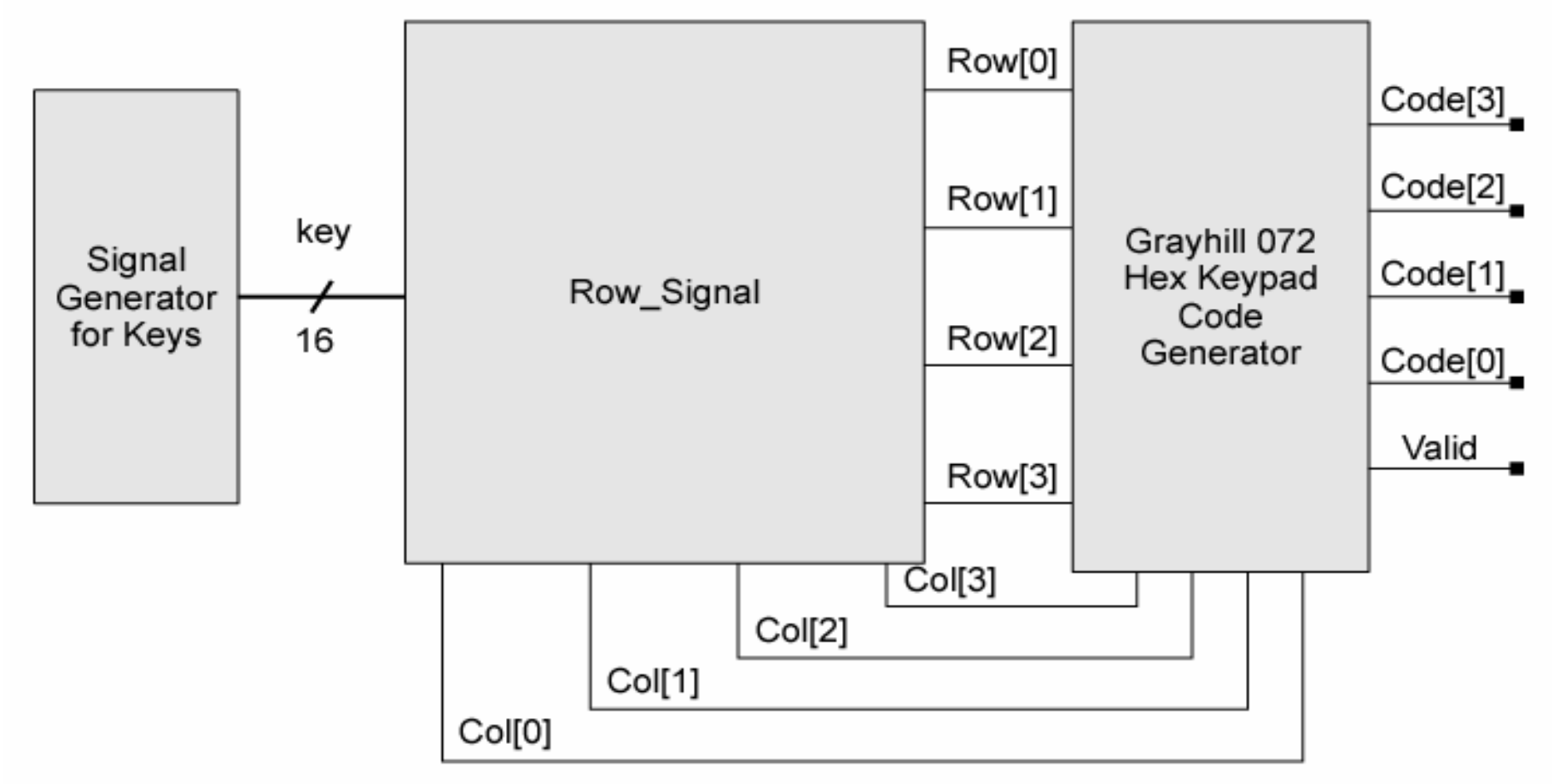

1
0
1
2
3
4
5
6
7
8
9
A
B
C
D
E
F
Row[0]
Row[1]
Row[2]
Row[3]
Col[3]
Col[2]
Col[1]
Col[0]
Grayhill 072
Hex Keypad
Code
Generator
Code[3]
Code[2]
Code[1]
Code[0]
Valid


S_0
/Col_0, 1, 2, 3
reset
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

The behavior of the keypad scanner/ encoder

- state S_0, with all column lines asserted, until one row line or more is asserted
- In S_1 only column 0 is asserted
- If a row is also asserted the output Valid is asserted for one clock and the machine moves to S_5, where it remains with all of the column lines asserted until the row is de-asserted
- Then the machine returns to S_0 for one cycle
- Note that all of the columns are asserted in S_5
- This eliminates the need to add two more states to the machine as it moves from S_2, S_3 and S_4 and awaits de-assertion of the row

Testbench organization for the Grayhill 072 hexadecimal keypad scanner/encoder



```
// Decode the asserted Row and Col
//   Grayhill 072 hex Keypad
//                Col[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          Col,Code;
    reg          [5:0] state,next_state;
```

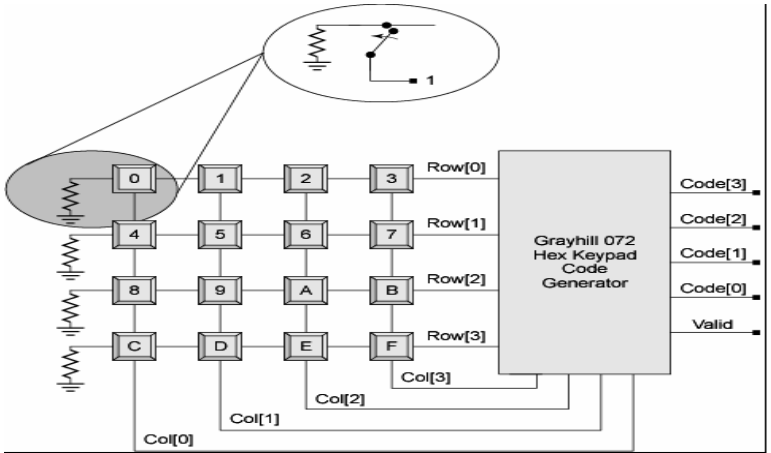
```
// One-hot
 parameter
   S_0=6’b000001,S_1=6’b000010,S_2=6’b000100;
 parameter
   S_3=6'b001000,S_4=6'b010000,S_5=6’b100000;
 assign
   Valid=((state==S_1)||(state==S_2)||(state==S_3)||
                (state==   S_4))&&Row;
// Does not matter if the row signal is not the
//              debounced 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 @ (negedge 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
 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          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[13] &&Col[3];
    Row[3]=Key[12]&&Col[0]||Key[13]&&Col[1]||Key[14]&&Col[2]||
              Key[15]&Col[3];
   end
endmodule
```



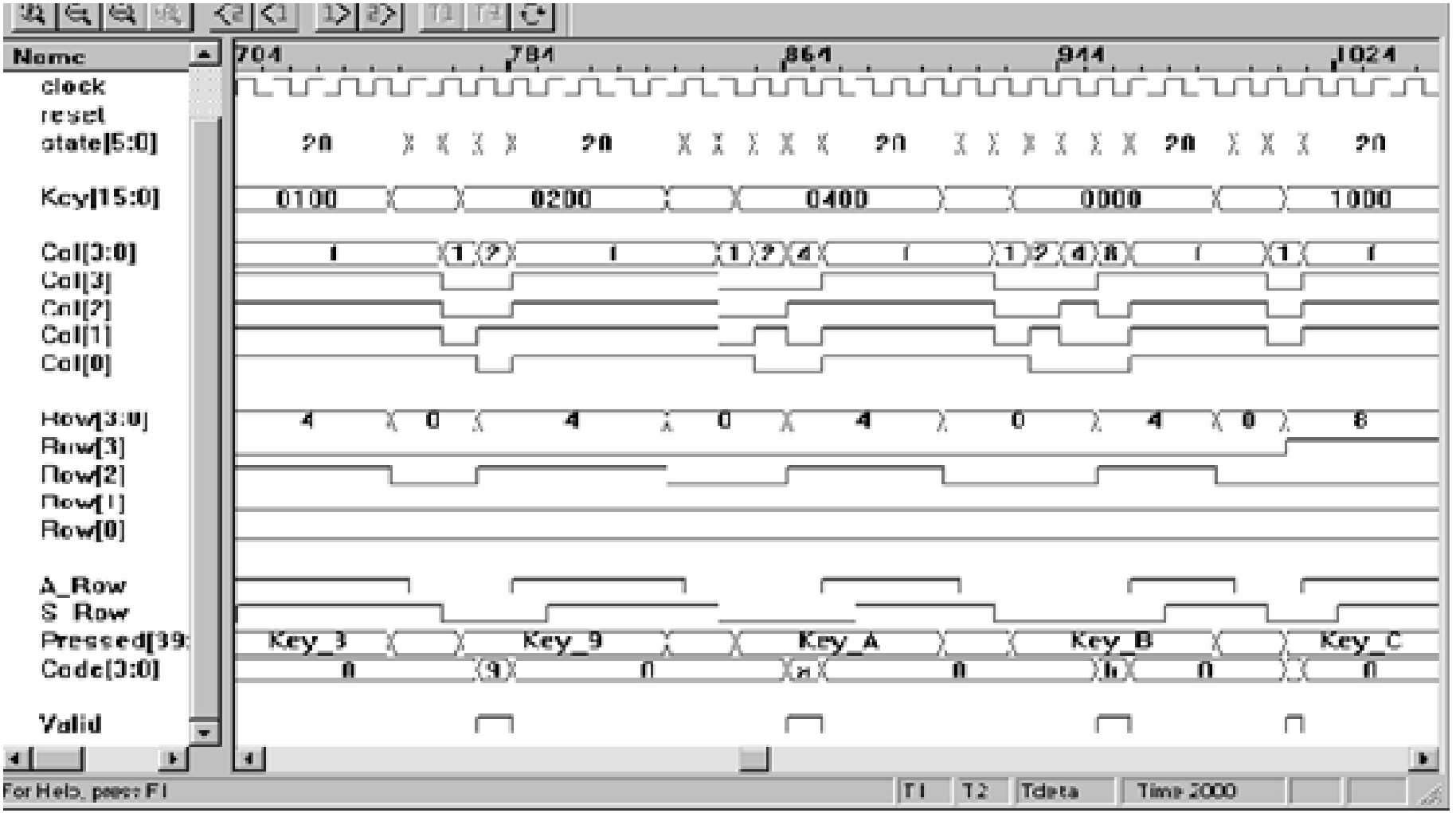
```
////////////////////////// Test Bench //////////////////////////
  module test_Hex_Keypad_Grayhill_072 ();
    wire [3:0]     Code;
    wire           Valid;
    wire [3:0]     Col;
    wire [3:0]     Row;
    reg            clock,reset;
    reg [15:0]     Key;
    integer        j,k;
    reg [39:0]     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
  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;
```

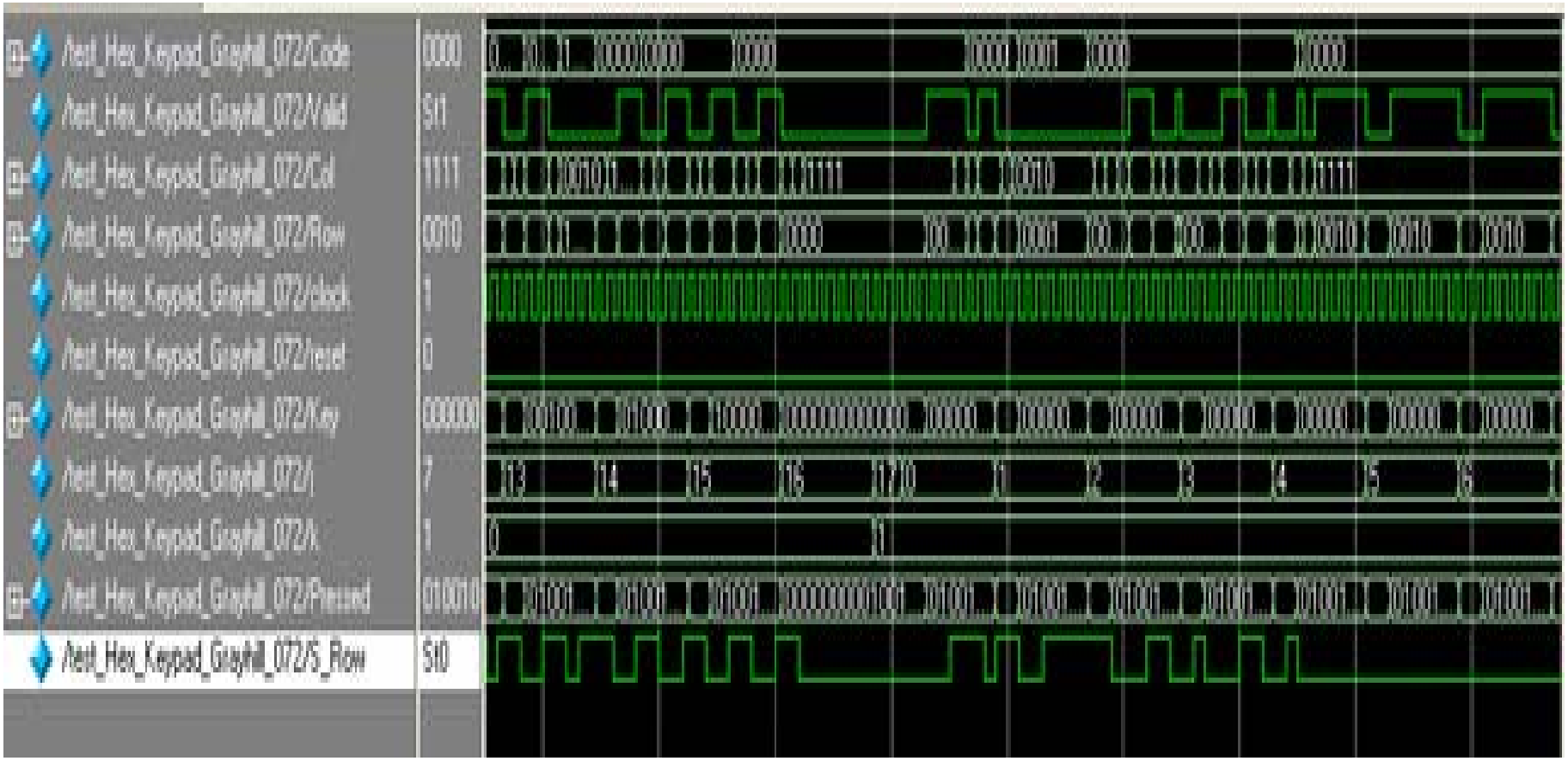
```
    16'h0040: Pressed=Key_6;
    16'h0080: Pressed=Key_7;
    16'h0100: Pressed=Key_8;
    16'h0200: Pressed=Key_9;
    16'h0400: Pressed=Key_A;
    16'h0800: Pressed=Key_B;
    16'h1000: Pressed=Key_C;
    16'h2000: Pressed=Key_D;
    16'h4000: Pressed=Key_E;
    16'h8000: Pressed=Key_F;
    default: Pressed=None;
  endcase
end

Hex_Keypad_Grayhill_072 M1
   (Code,Col,Valid,Row,S_Row,clock,reset);
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;#25 for (j=0;j<=16;j=j+1)
    begin
      #20 Key[j]=1; #60 Key=0;
    end
  end
end
endmodule
```

Simulation result

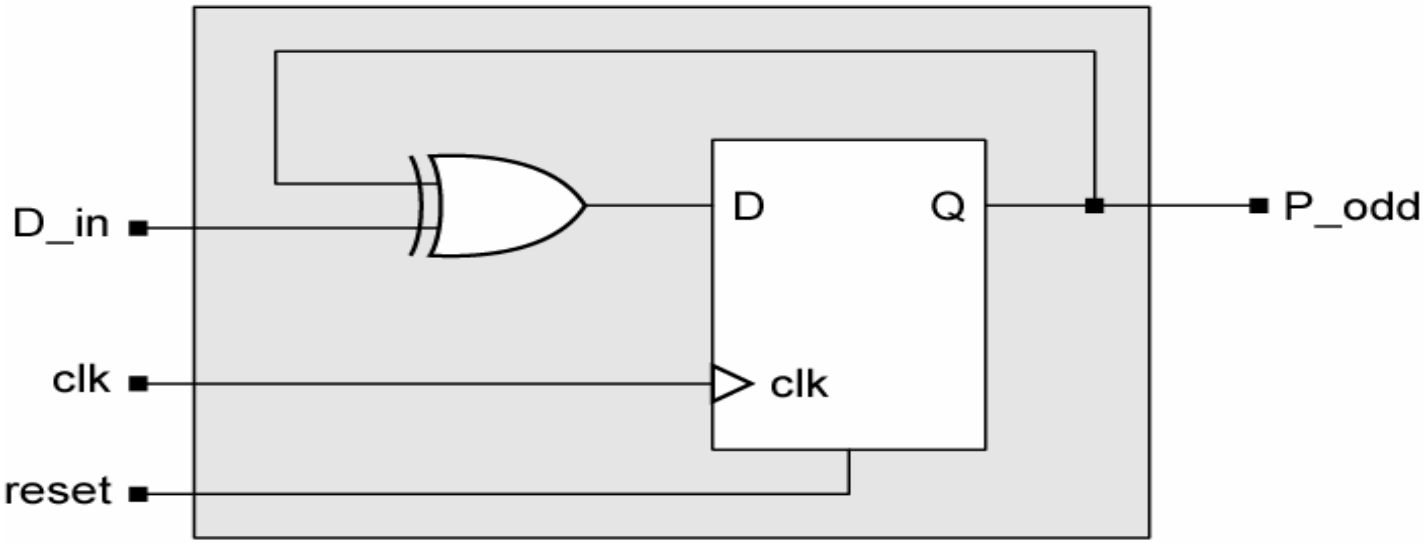


Simulation result



Problem 5-15

- Write a Verilog description of the circuit shown in Figure P5-15 and verify that the circuit- output, P_odd, is asserted if successive samples of D_in have an odd number of 1s.



Problem 5-16

Develop and verify a Verilog model of a 4-bit binary synchronous counter with the following specifications:

- negative edge-triggered synchronization
- synchronous load
- reset
- parallel load of data
- active-low enabled counting.

- **Problem 5-20**
- Develop and verify a Verilog model of a modulo-6 counter
- **Problem 5-23**
- Write a parameterized and portable Verilog model of an 8-bit ring counter whose movement is from its MSB to its LSB.