Chapter 6

Synthesis of Combinational and Sequential Logic

Synthesis tools' task

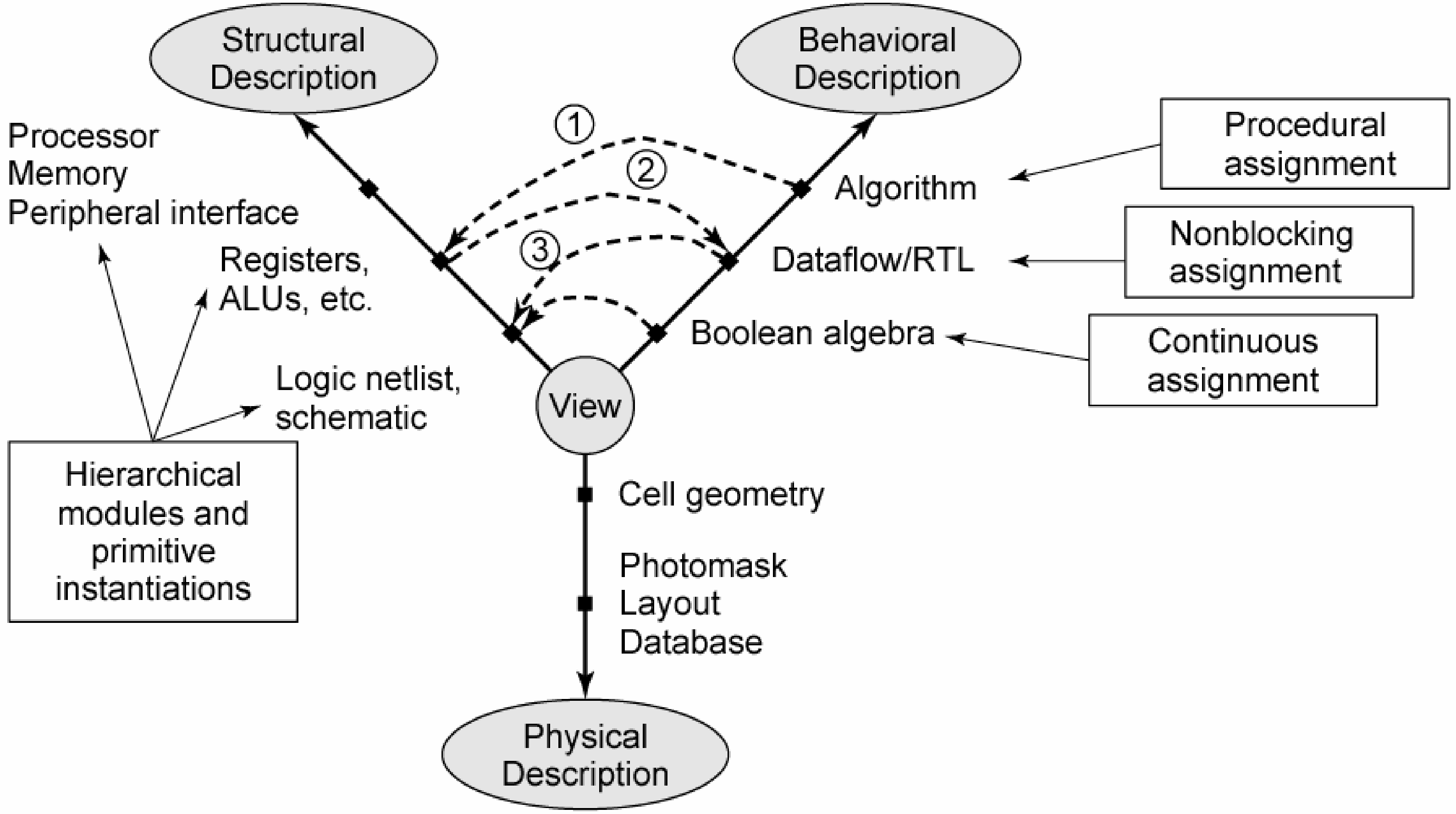
- Detect and eliminate redundant logic
- Detect combinational feedback loops
- Exploit don't-care conditions
- Detect unused states
- Detect and collapse equivalent states
- Make state assignments
- Synthesize optimal, multilevel realizations of logic subject to constraints on area and/or speed in a physical technology

6.1 Introduction to synthesis

- Circuit design **begins with specification** of the circuit's functionality and **ends with physical hardware**
- An HDL can facilitate the design process by providing different views of the circuit
- **Three common views:**
- behavioral, structural , physical

Y-chart

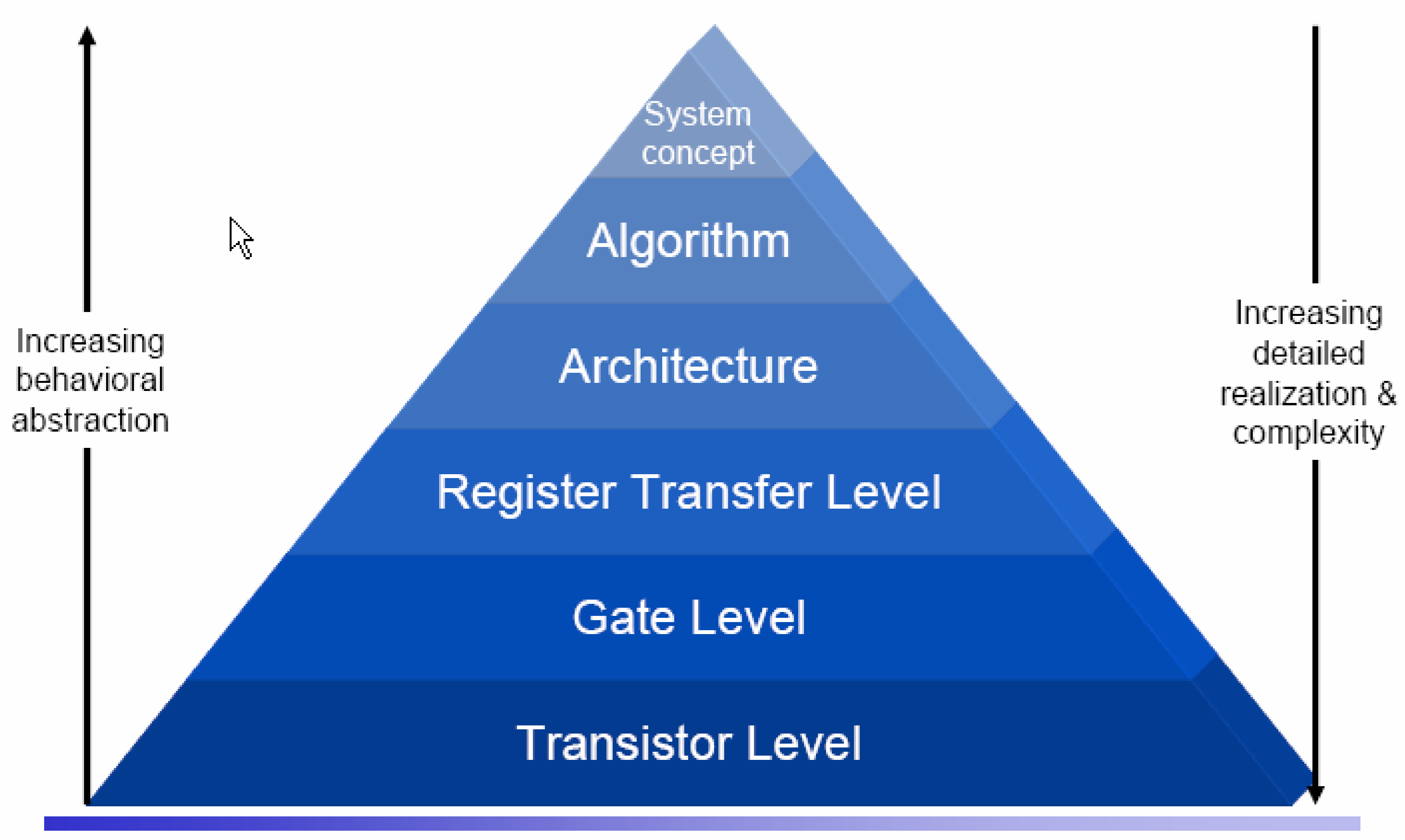
- representation of Verilog constructs supporting synthesis activity in three views of a circuit



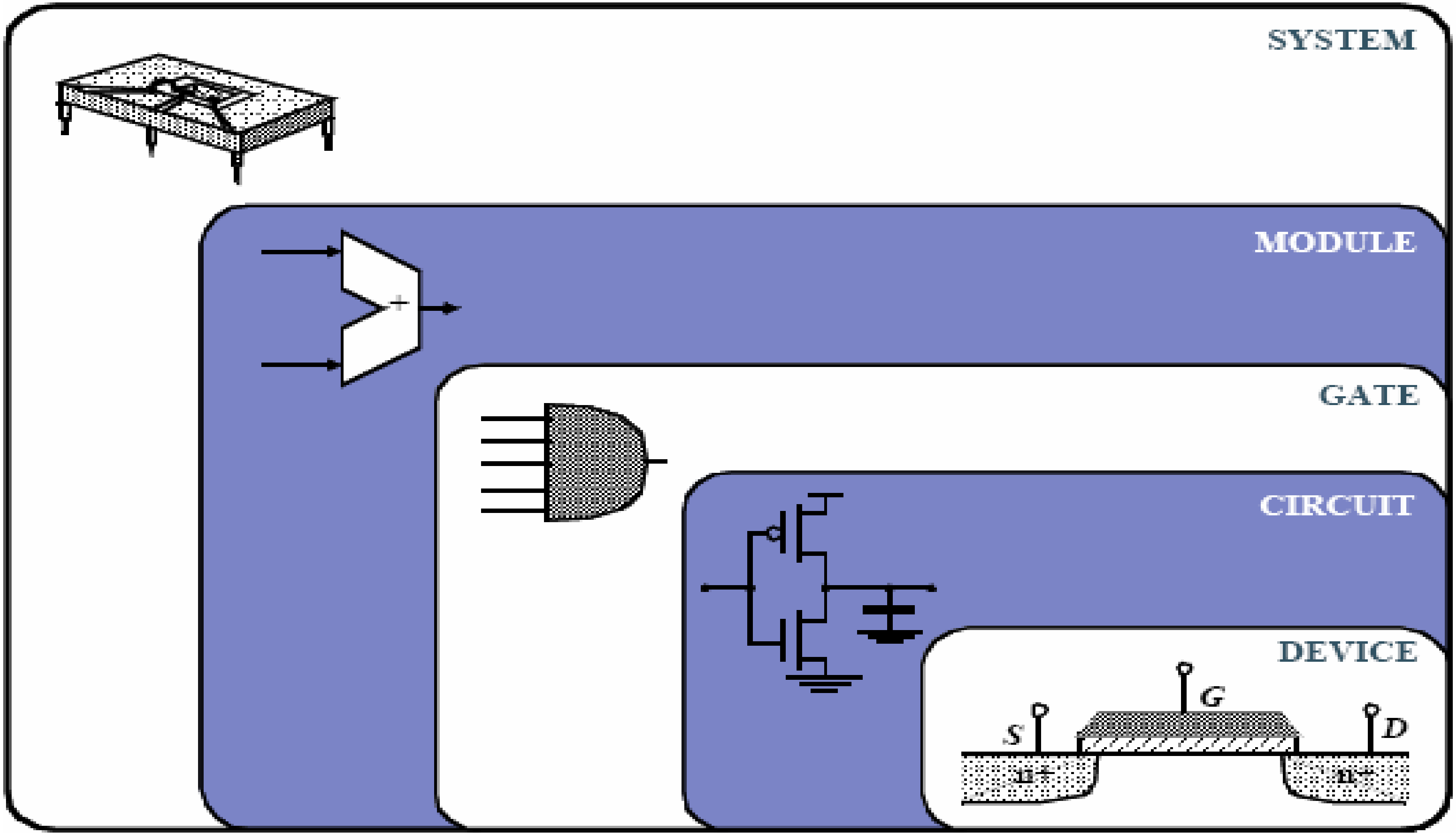
The chart shows a sequence of transformations

- (1) behavioral synthesis transforms an algorithm (behavioral view) to an architecture of registers
- (2) a Verilog model of this architecture is formed as a data flow/register transfer level (RTL) description
- (3) logic synthesis translates the data flow/RTL description into a Boolean representation and synthesizes it into a netlist

Design Abstraction Levels



Design Abstraction Levels



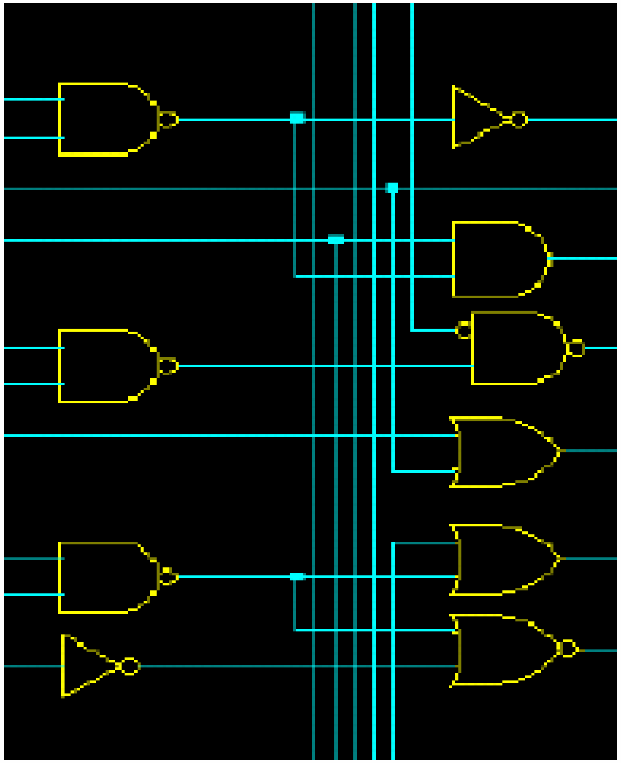
Architectural level

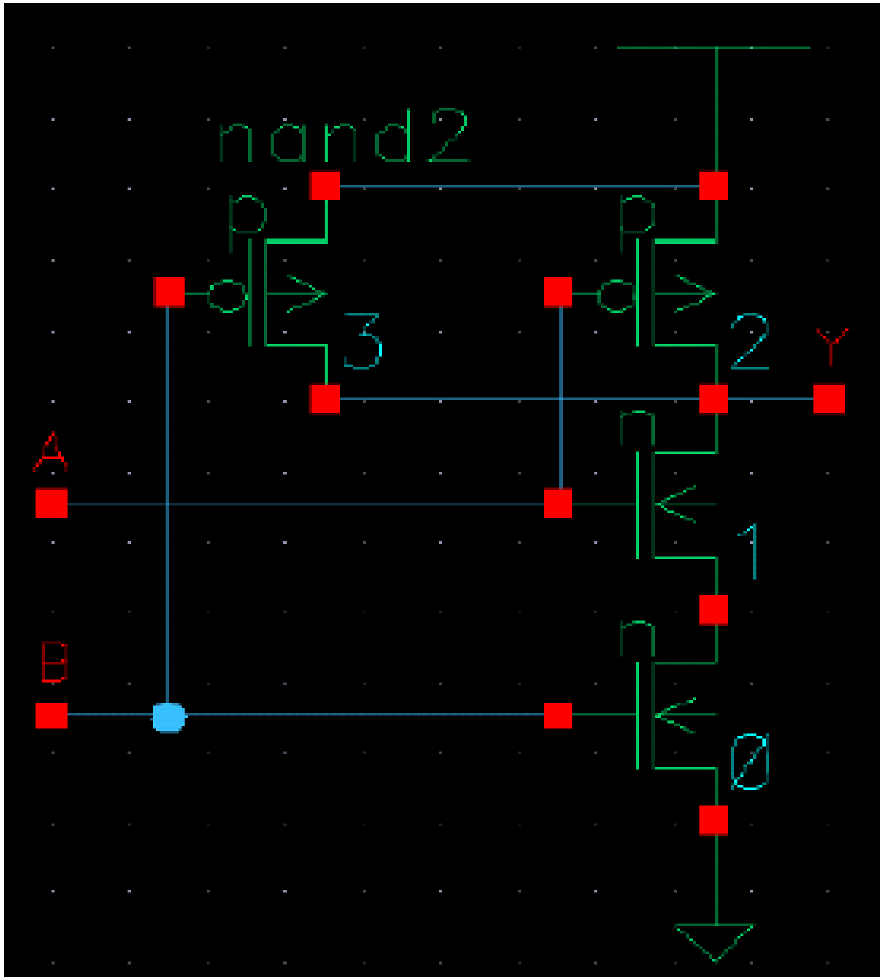
- The operations that must be executed by the circuit **to transform a sequence of inputs into a specified sequence of outputs**
- **Design challenge** is **to extract from the architectural description** ,a structure of computational resources that implements the functionality of the machine

Logical level

- Describes a set of variables and a set of Boolean functions
- The design task here is **to translate the Boolean descriptions into an optimized netlist** of combinational gates and storage registers that will implement the circuit at a satisfactory level of performance

Gate and Circuit Level Design

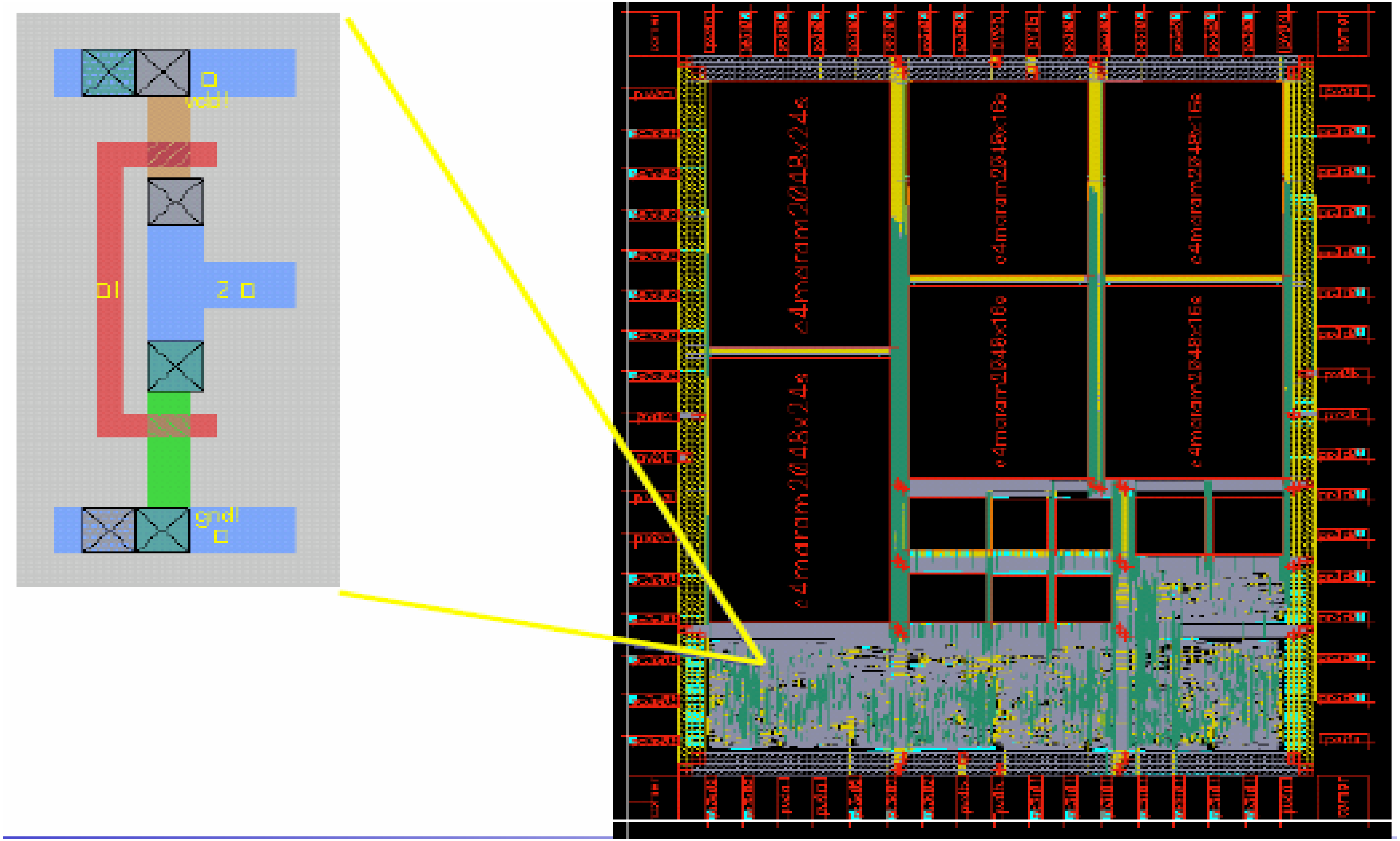




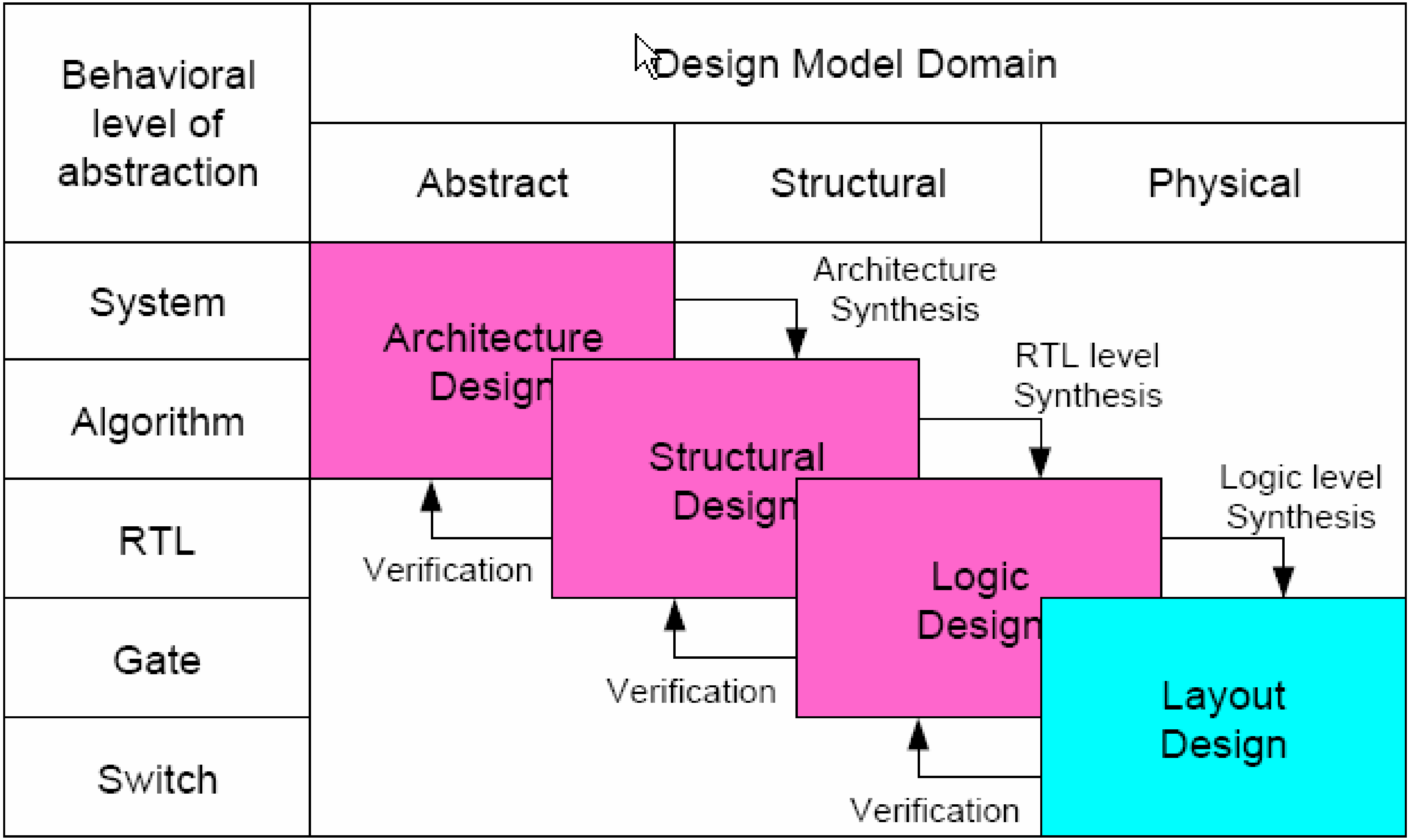
Geometrical model

- Describes the shapes that **define the doping regions of semiconductor materials** used to fabricate transistors
- HDLs do not treat geometric models

Physical Design



Verilog HDL in Design Domain



Behavioral level of abstraction

Design Model Domain

Abstract

Structural

Physical

System

Algorithm

RTL

Gate

Switch

Architecture Design

Architecture Synthesis

Structural Design

RTL level Synthesis

Logic Design

Logic level Synthesis

Layout Design

Verification

Verification

Verification

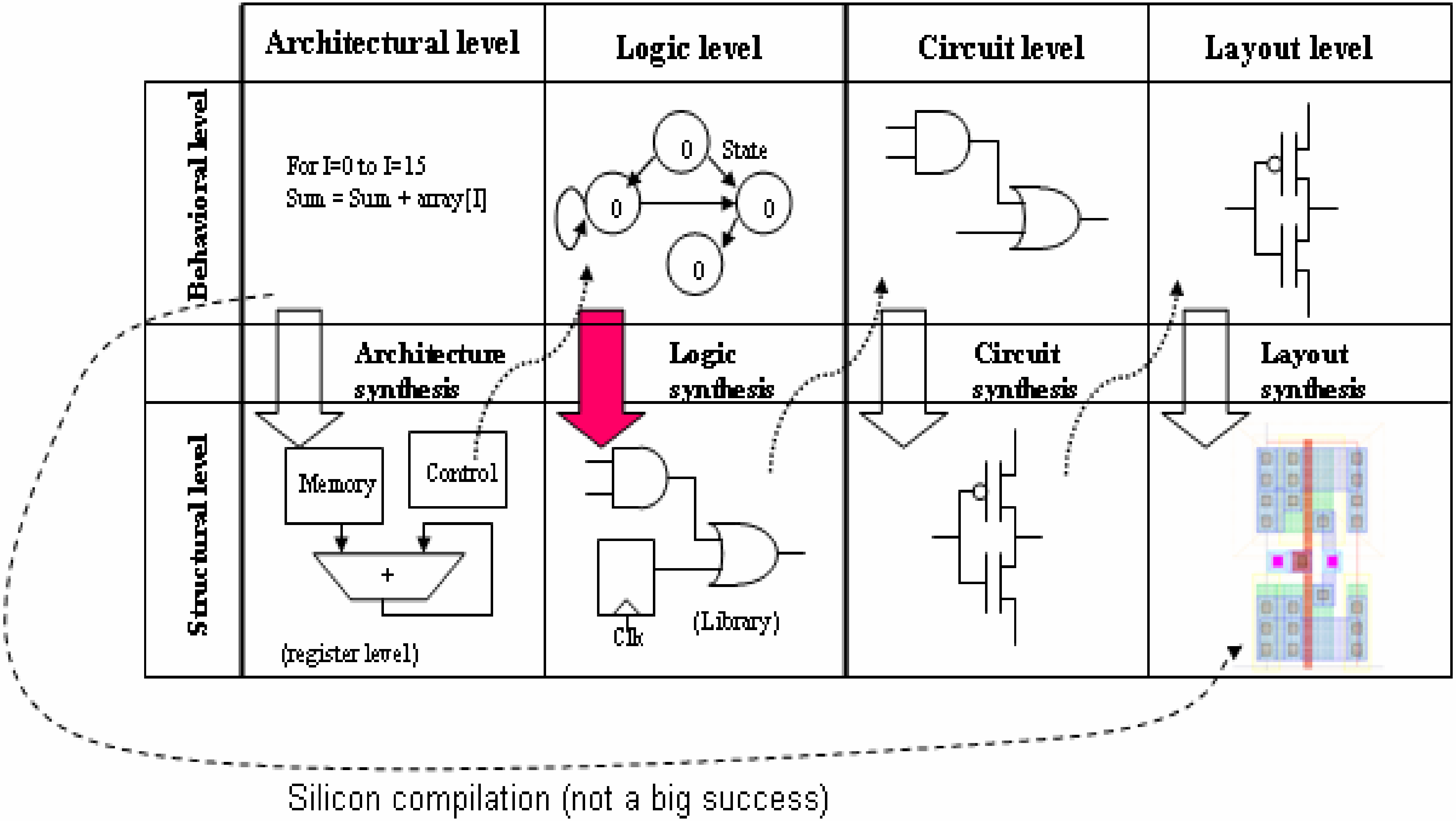
Three common views

- **A behavioral view** of an architectural model could be **an algorithm that specifies a sequence of data transformations**
- **A structural view** of the same model might consist of **a structure of datapath elements** (registers, memory, adders, and a controller) that implement the algorithm
- **Physical descriptions**, the **actual geometric patterns** of the physical devices that implement the circuit, will not be considered here

6.1.1 Logic synthesis

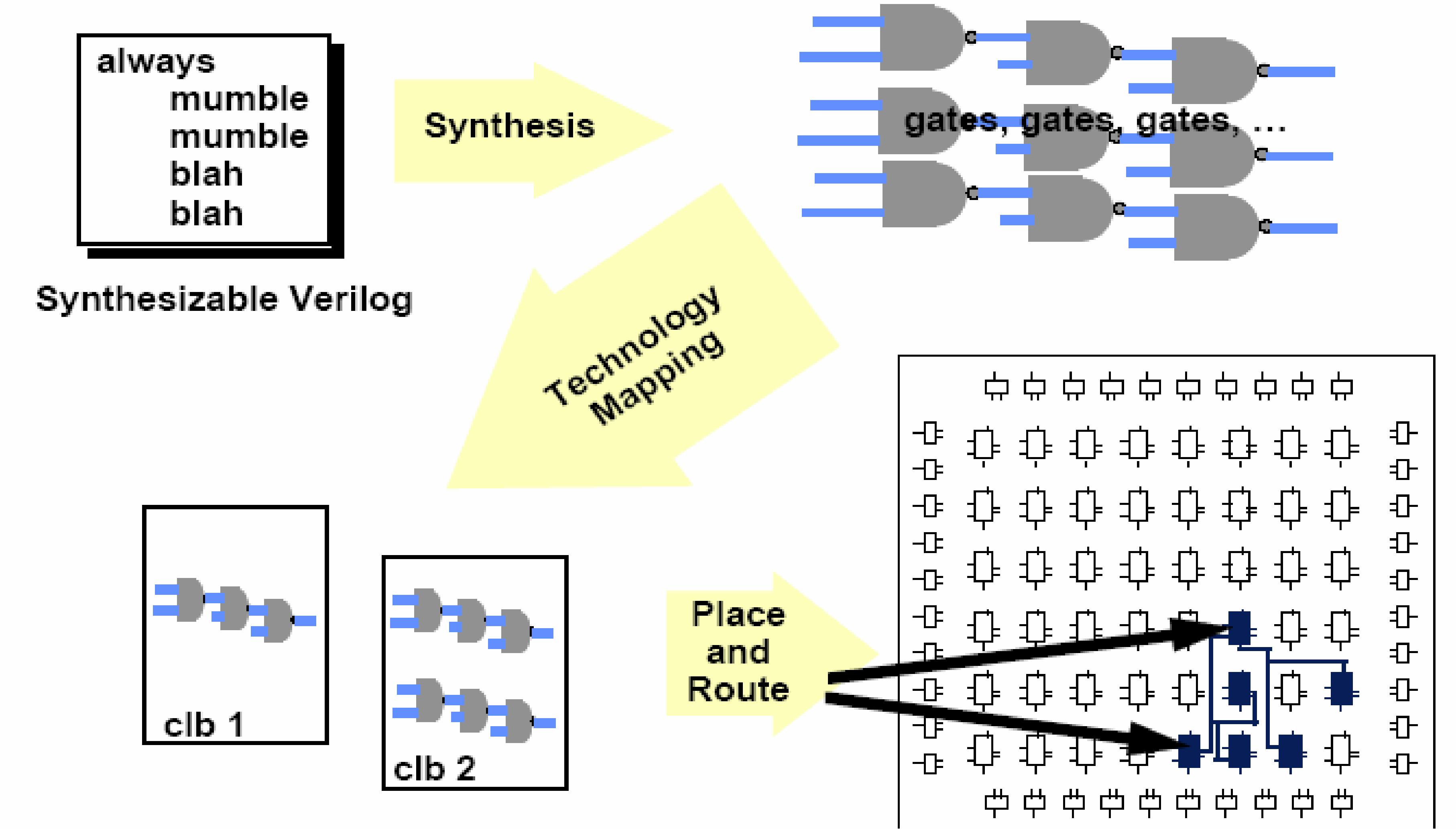
- Logic synthesis includes:
- Transforming a given netlist of primitives into an optimized netlist of Verilog primitives
- Mapping a generic optimal netlist into an equivalent circuit composed of physical resources in a target technology

Abstraction levels and synthesis



Synthesis and Technology Mapping

Simulation and Synthesis are components of a design methodology



```
always
        mumble
        mumble
        blah
        blah
```

Synthesizable Verilog

Synthesis

gates, gates, gates, …

Technology Mapping

clb 1

clb 2

Place and Route

Logic synthesis

- Three steps:
- **Synthesis =**

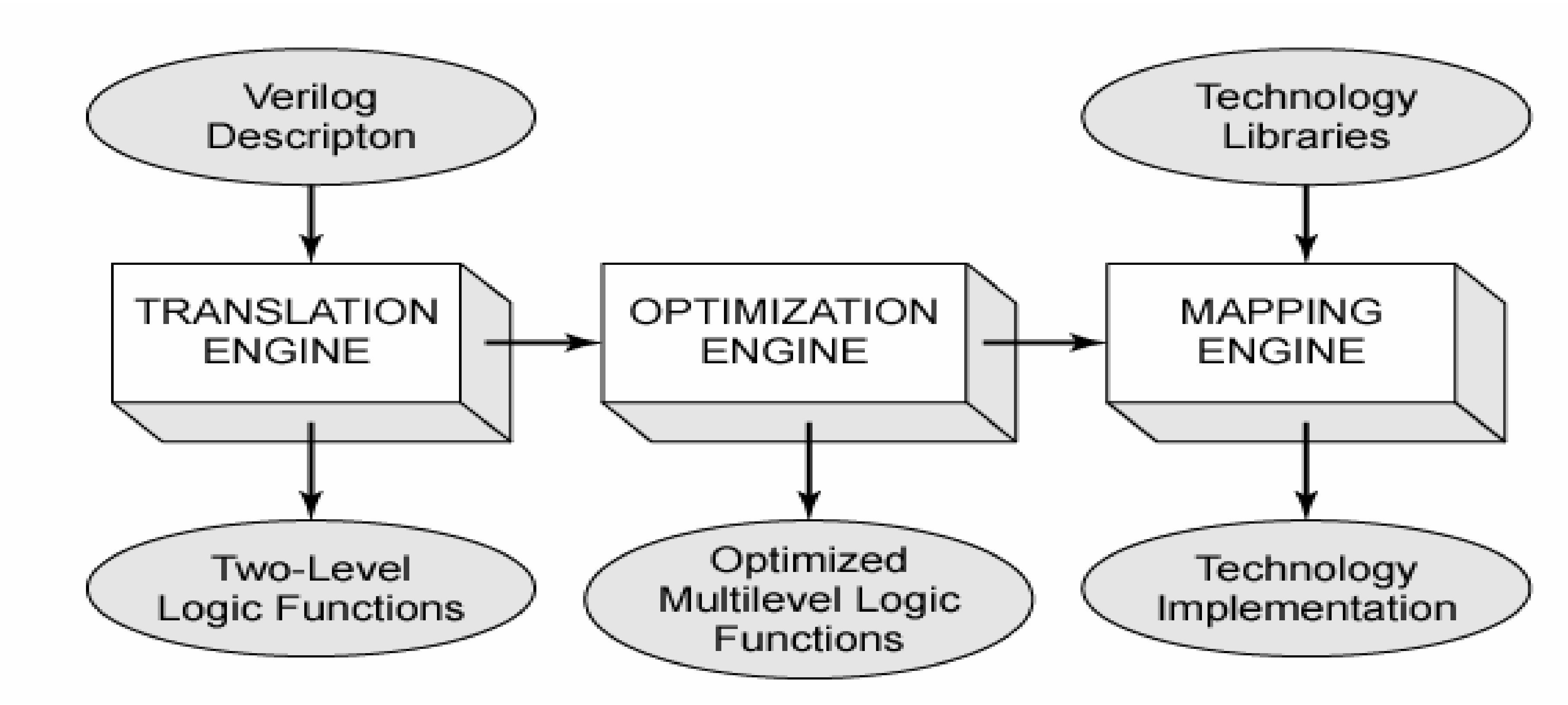
Translation

+ Logic optimization

+ mapping

The organization of a logic synthesis tool

The tool forms a hardware realization (technology implementation) from a Verilog RTL model or primitive netlist.



Synthesis

- **Synthesis creates a sequence of transformations**
- **From a higher level of abstraction to a lower one**
- There may be multiple realizations of a multi-input, multi-output combinational logic circuit
- But the transformations made in synthesis are guaranteed to maintain the input-output equivalence of the circuit and produce a testable circuit

Optimization

- The optimization process is based on an iterative search
- **Logic optimization is followed by performance optimization**, which seeks a circuit that has optimal performance in the physical technology
- By **removing redundant logic**, **sharing** internal logic, **and exploiting** input and output don’t-care conditions

Structure & Flattening optimization

① Decomposition
② Extraction
③ Factoring
④ Substitution
⑤ Elimination

■ The reason :

Types of component ,
Cost,
Delay ,
Fan in/out

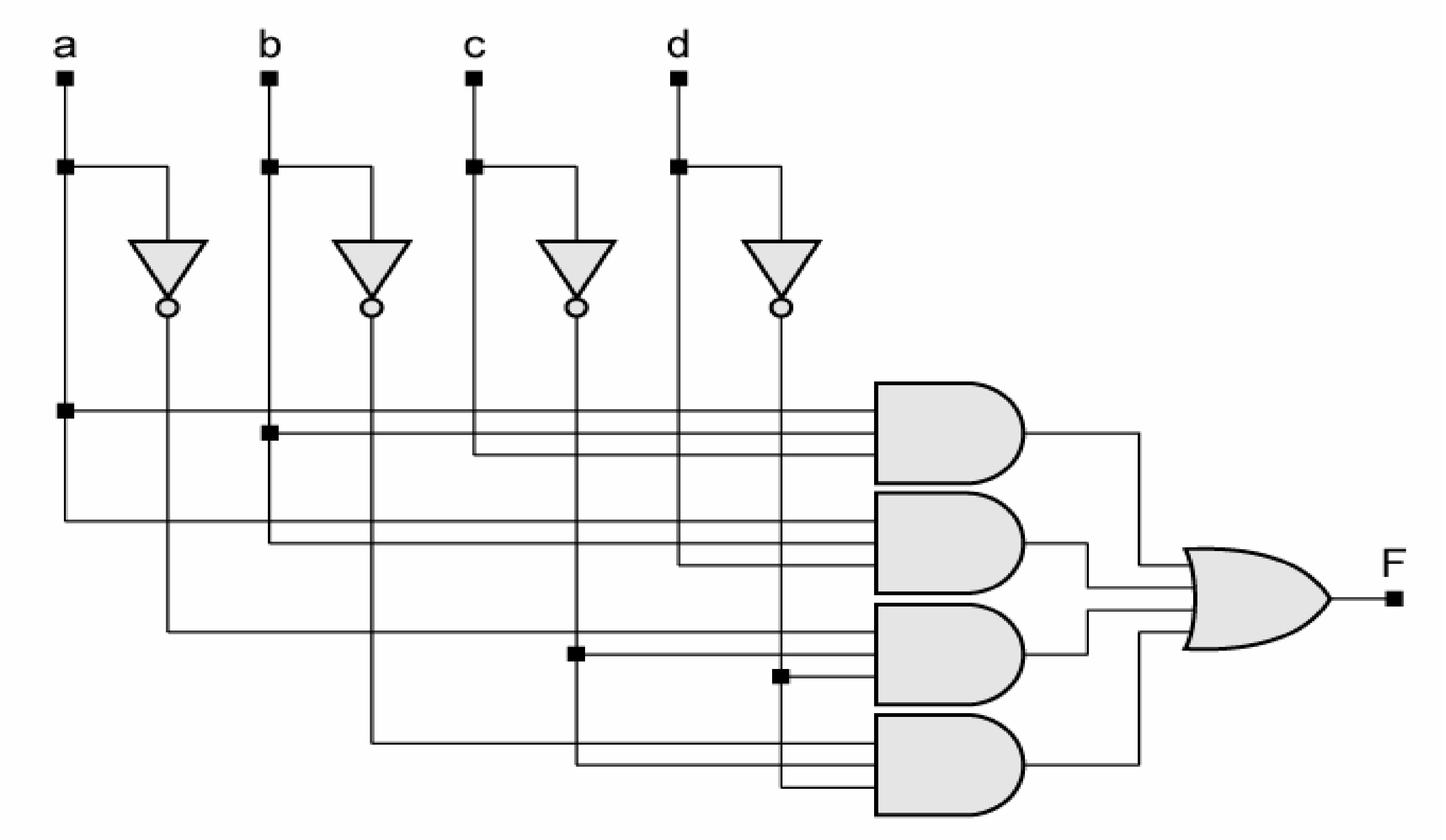
Decomposition

- The operation of decomposition transforms the circuit

  by expressing a single Boolean function in terms of new nodes.

Ex 6.1 decomposition

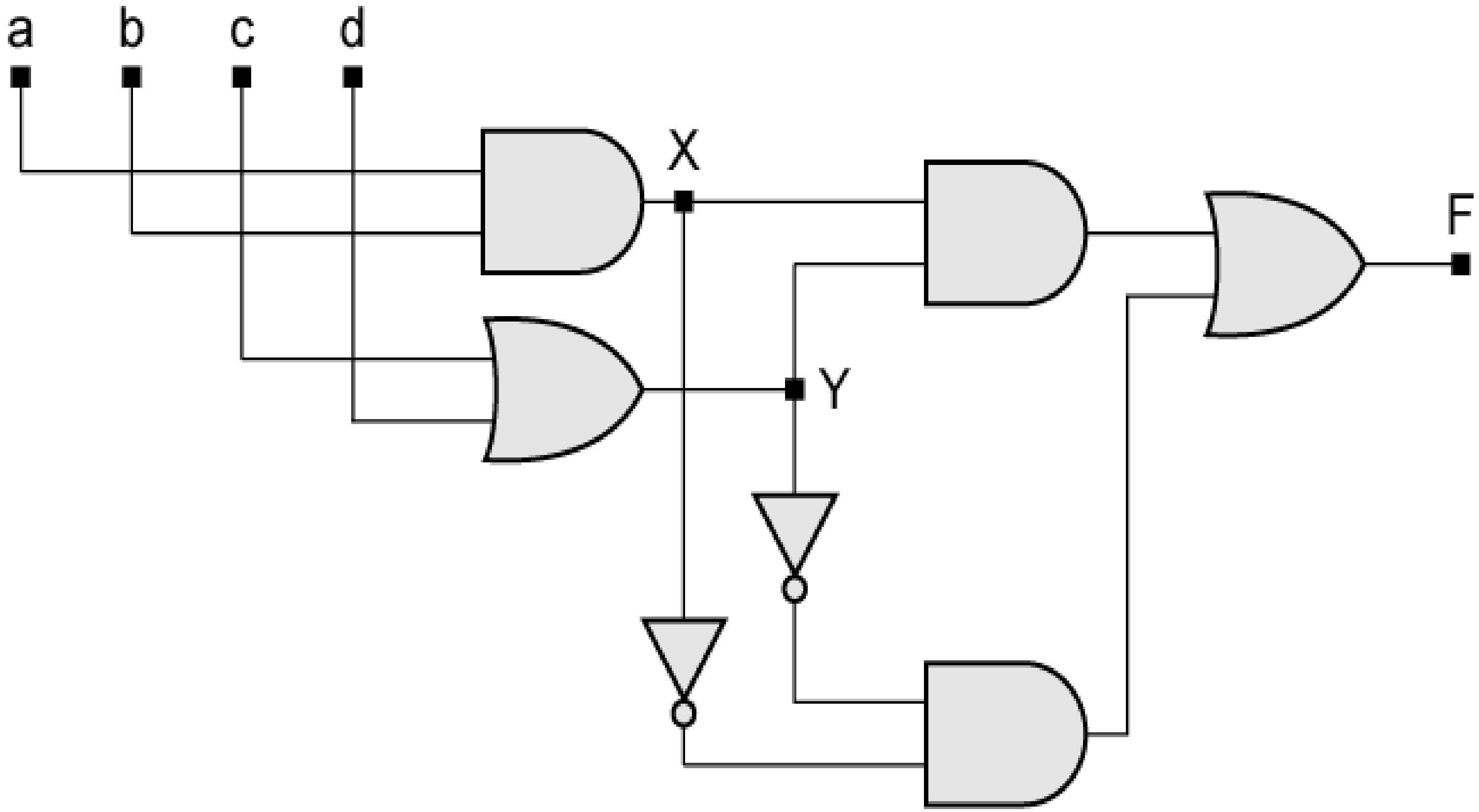
- F=abc+abd+a’b’c’+b’c’d’ (before decomposition)



F=XY+X'Y' (after decomposition)

X=ab ,Y=c+d,

Result: reduce Gates,wires , Chip area



Extraction

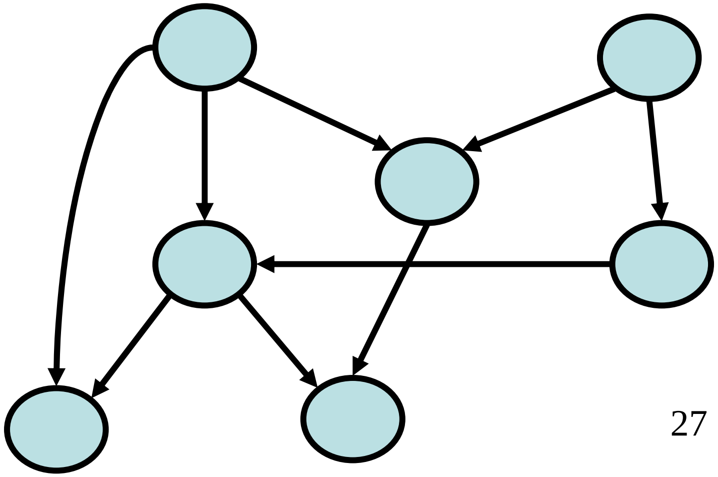
- The operation of extraction **expresses a set of functions in terms of intermediate nodes** by expressing each function in terms of its factors and then detecting which factors are shared among functions

Directed Acyclic Graph (DAG)

- **DAG – directed graph with no directed cycles**

Application

- Describe a process of a system
- *Design and Analysis of Algorithms*
- *Shortest Paths*



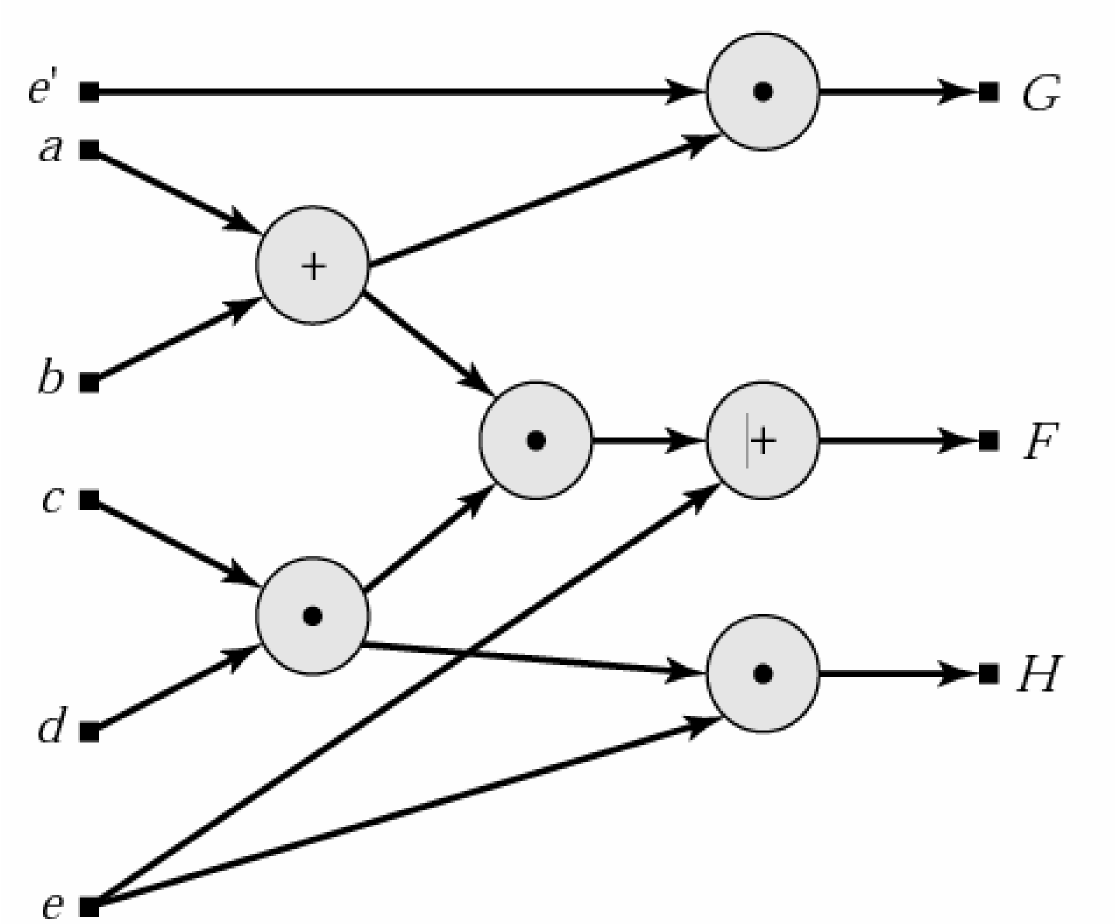
Ex 6.2 Extraction

- F=(a+b)cd+e

  G=(a+b)e’

  H=cde

(DAG of a set of functions before extraction)



By the new internal nodes X and Y to produce the new DAG

X=a+b

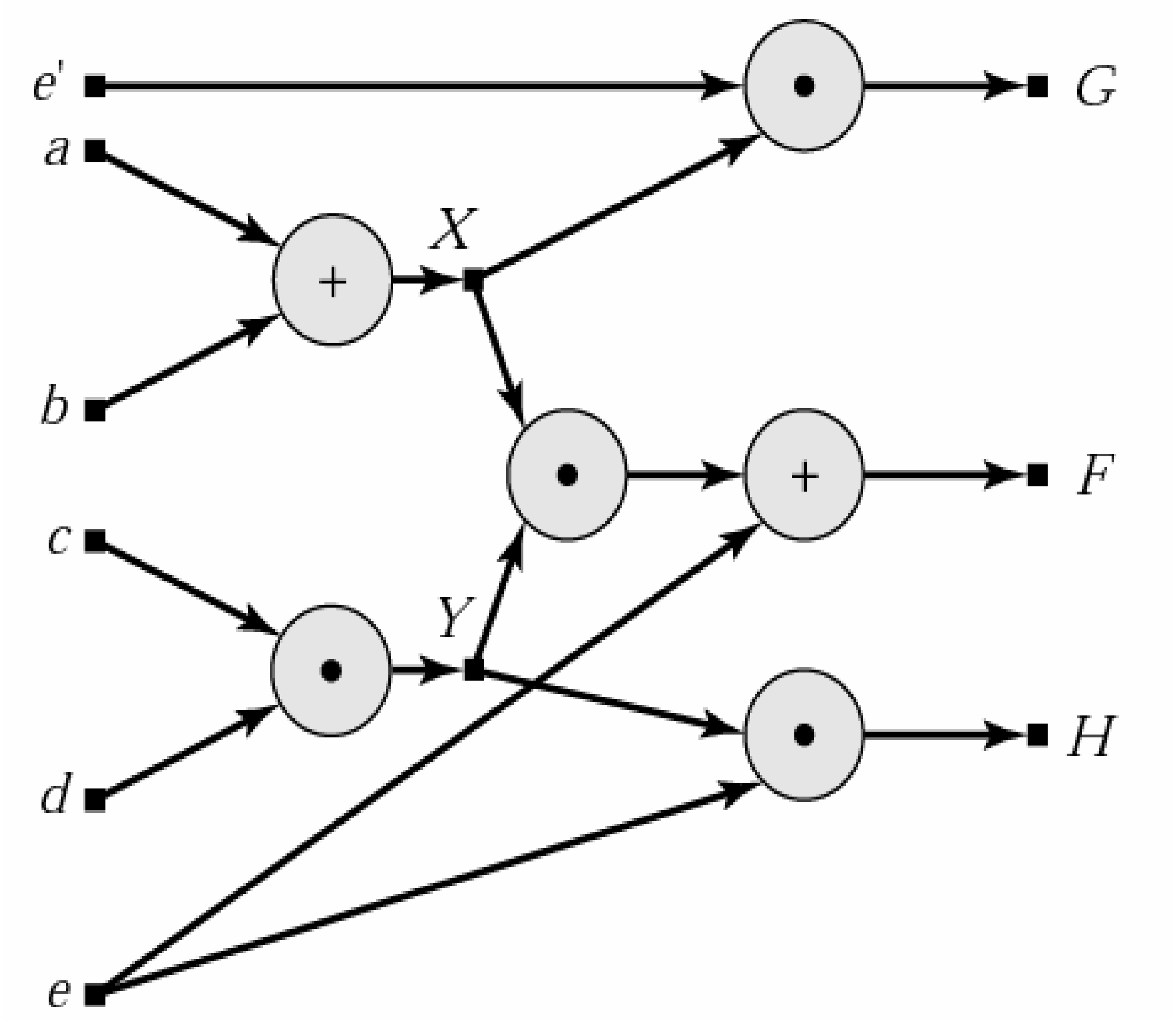
Y=cd

F=XY+e

G=Xe’

H=Ye

(DAG of a set of functions after extraction)



The optimization process

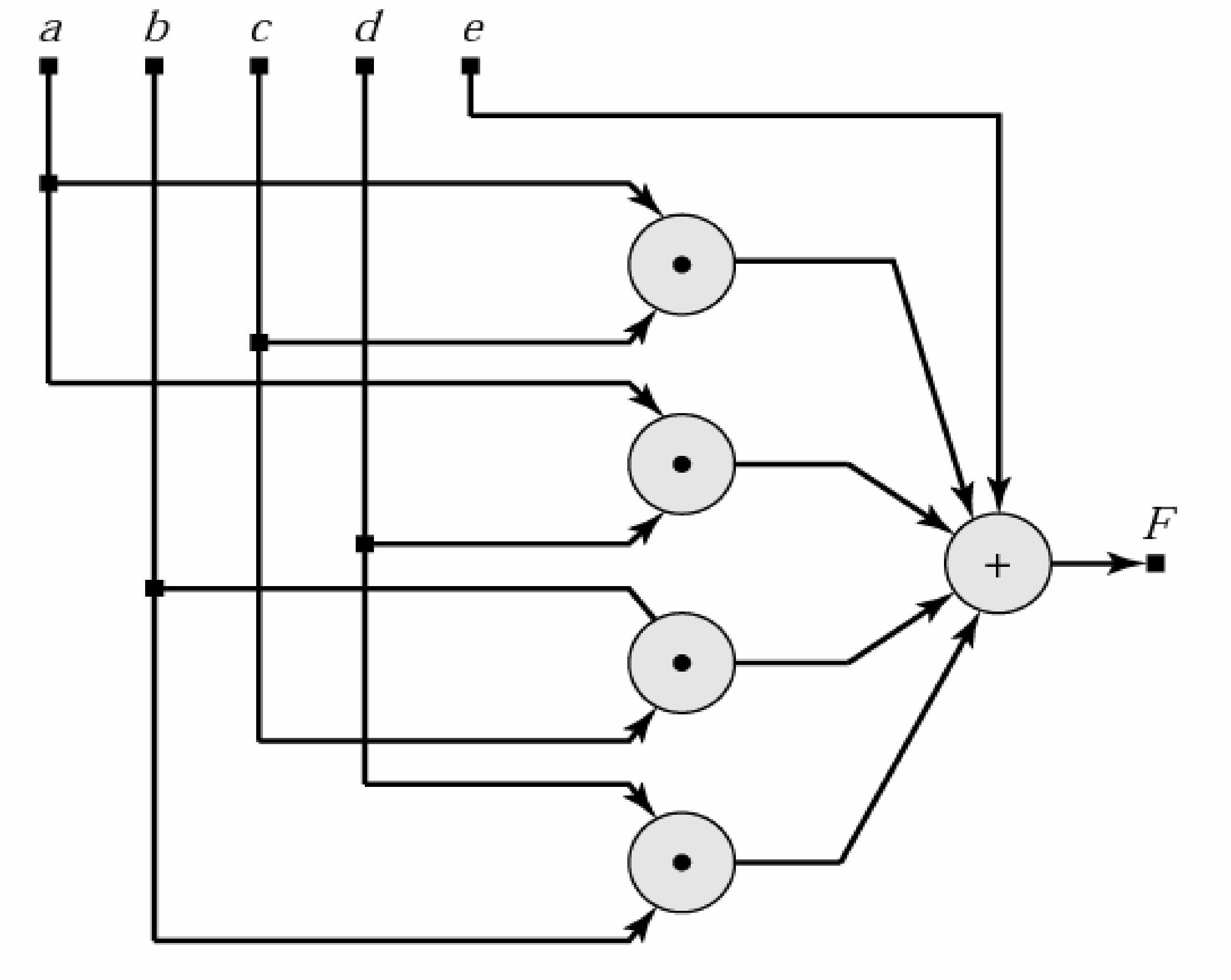
- The optimization process seeks a set of intermediate nodes
- to **eliminate replicated logic**
- **to optimize the circuit's delay and area**
- Because the intermediate nodes correspond to factors that are common to more than one Boolean function, and therefore can be shared

Factoring

- The task of finding the common factors among a set of functions is called factoring
- Factoring **produces a set of functions in a products of sum form**
- It creates a structural transformation of the circuit from a two-level realization to an equivalent multilevel realization that **uses less area, but is possibly slower**

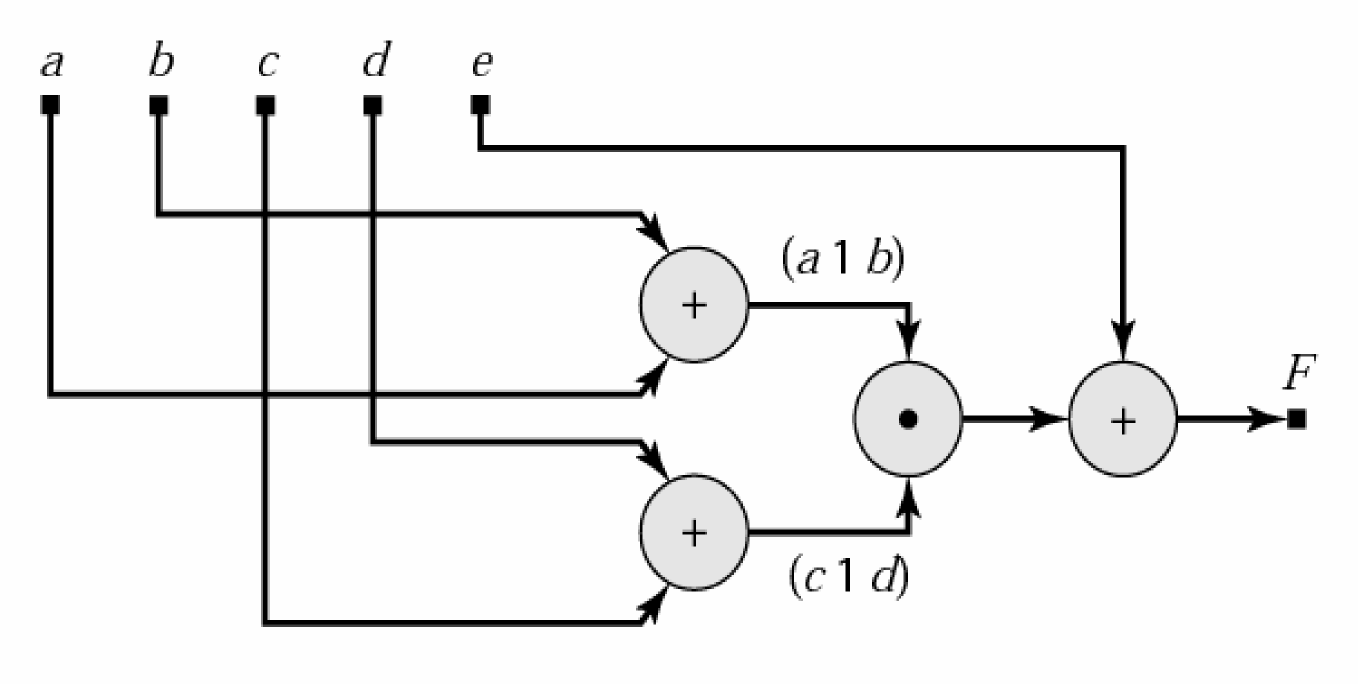
Ex 6.3 Factoring

- **F=ac+ad+bc+bd+e (before factoring)**



After Factoring

- **F=(a+b)(c+d)+e;**
- **Before:(w:14,G:5);Now:(w:9,G:4)**

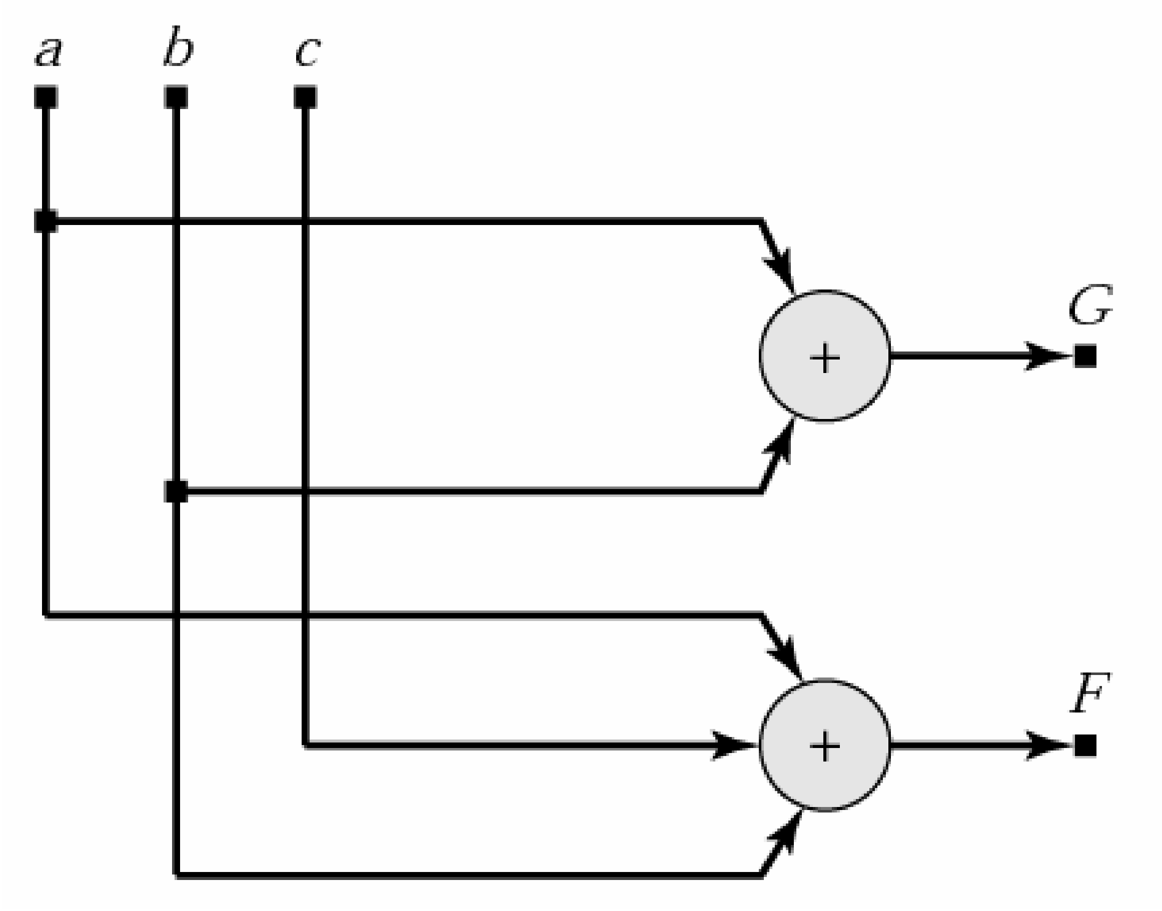


Factoring

- The substitution process expresses a Boolean function in terms of its inputs and another function
- Since both functions need to be implemented, this step **provides a potential reduction of replicated logic**

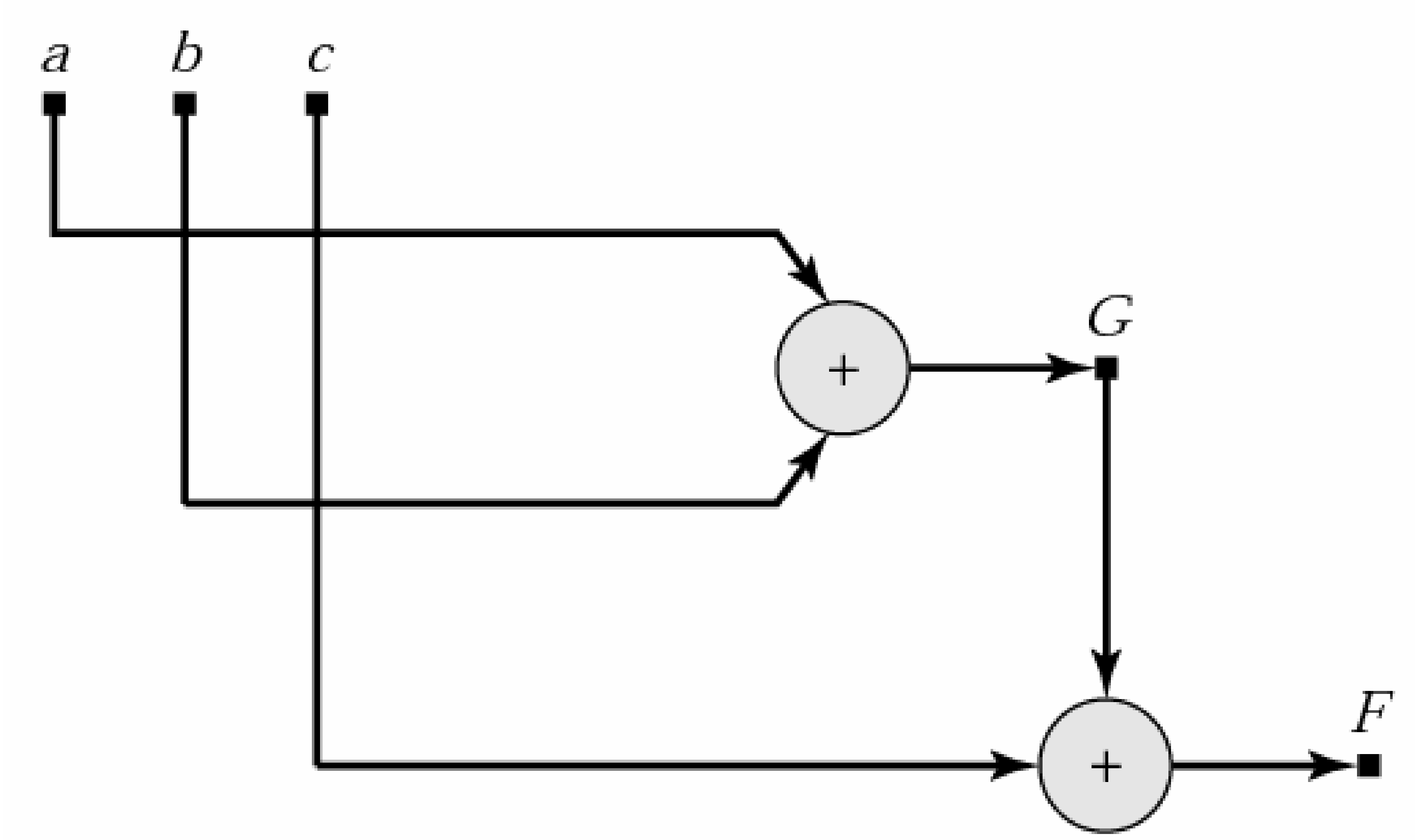
Ex 6.4 Substitution

- G=a+b
  F=a+b+c
  (before)



After substitution

- F=G+c



Elimination

- Elimination **removes a node in a function and reduces the structure of the circuit**
- This step is also referred to as **flattening the circuit**
- The transformation would ultimately **eliminate the area-efficient internal multilevel structure** and **create a faster two-level structure**

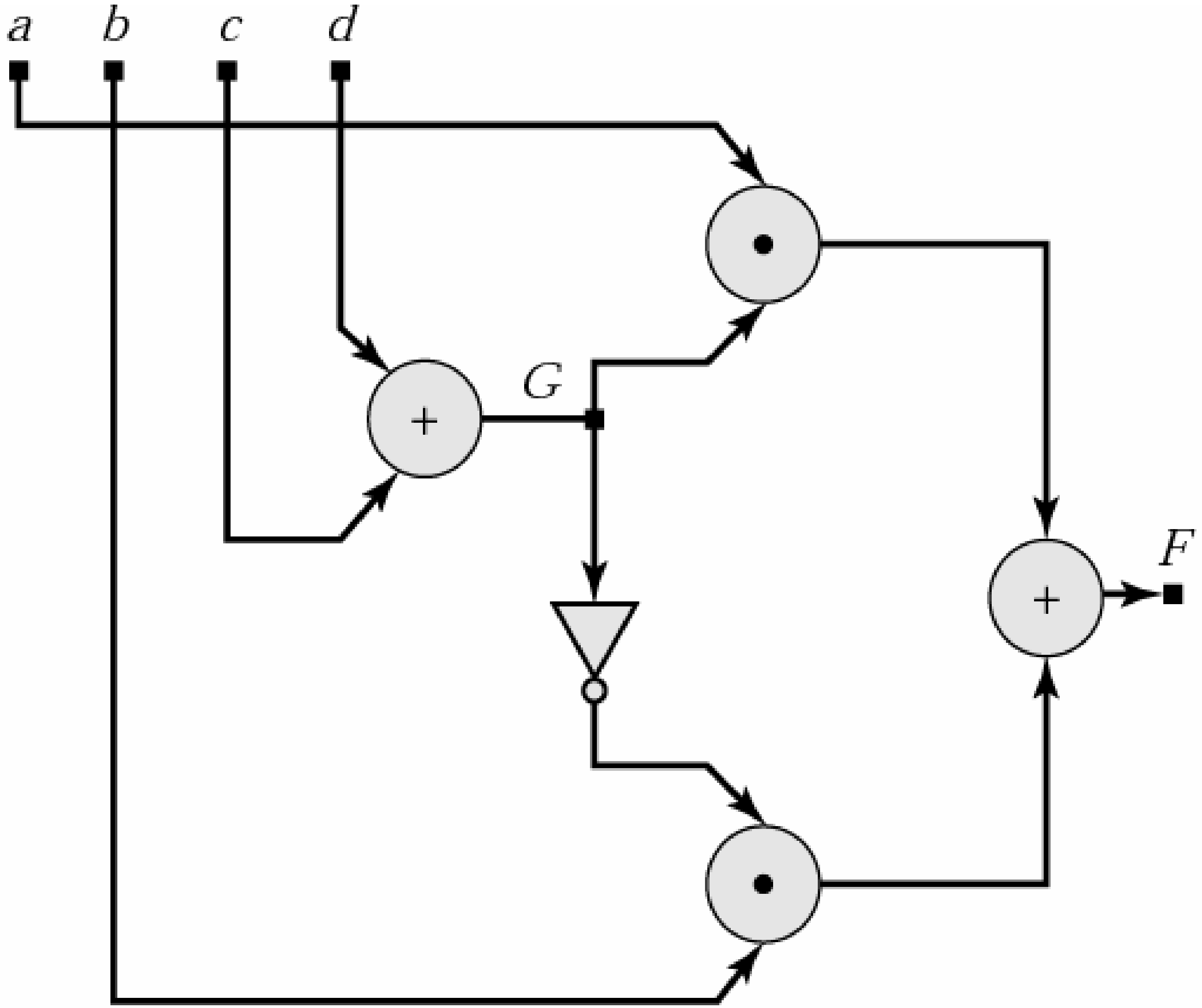
Ex. 6.5 Elimination

- F=Ga+G'b

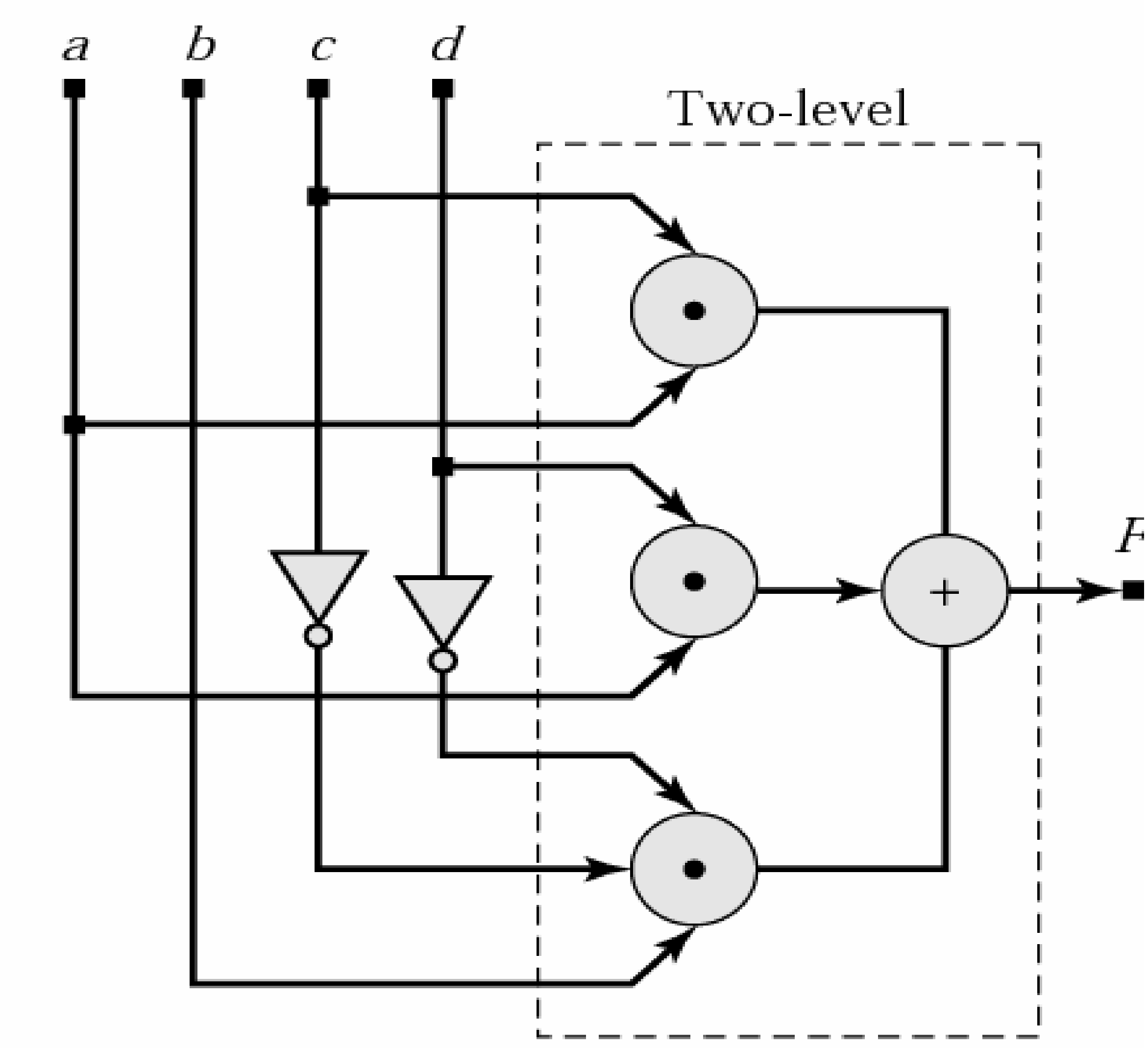
  G=c+d

  (before elimination)

Figure 6-11(a)



F=ac+ad+bc’d (after elimination)



Multilevel networks

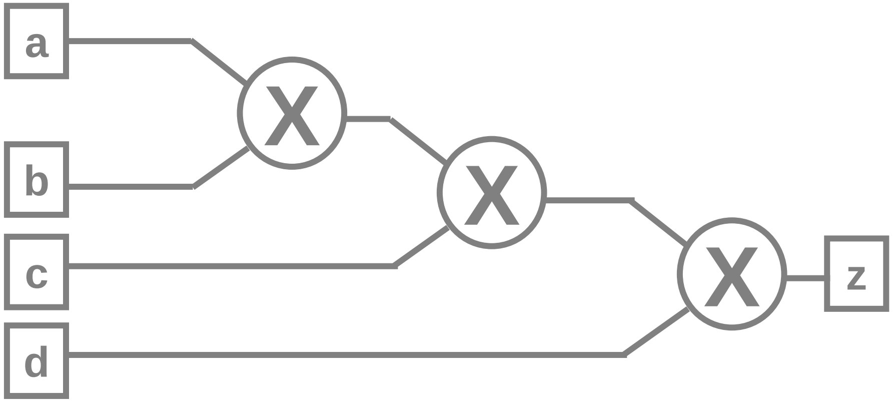
- Multilevel networks may be used because two-level networks might require **higher fan-in than is practical**
- **Multilevel circuits present an opportunity to share logic, but can be slower than a two-level counterpart**

Balancing Operators

- Use Parenthesis to Define Logic Groupings
  - Increases Performance & Utilization
  - Balances Delay from All Inputs to Output
  - Circuit Functionality Unchanged

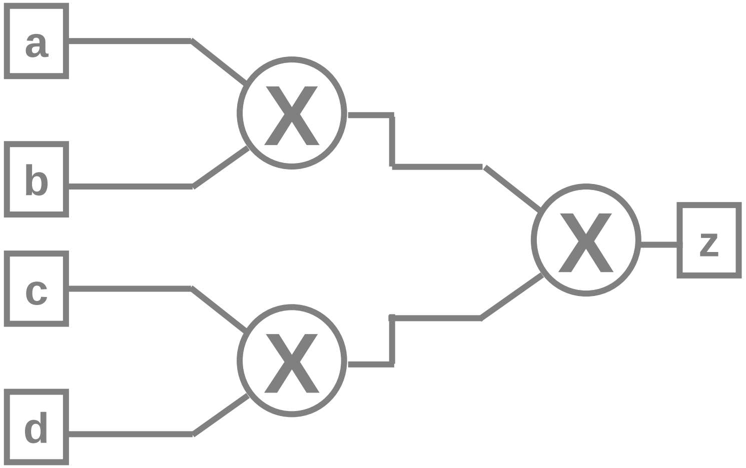
Unbalanced

```
out = a * b * c  * d;
```



Balanced

```
out = (a * b) * (c * d);
```

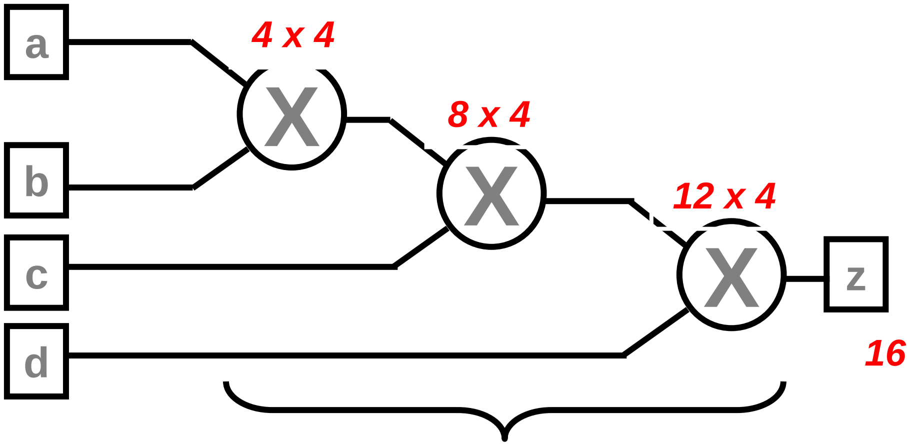


Balancing Operators: Example

- a, b, c, d: 4-bit vectors

Unbalanced

out = a * b * c * d

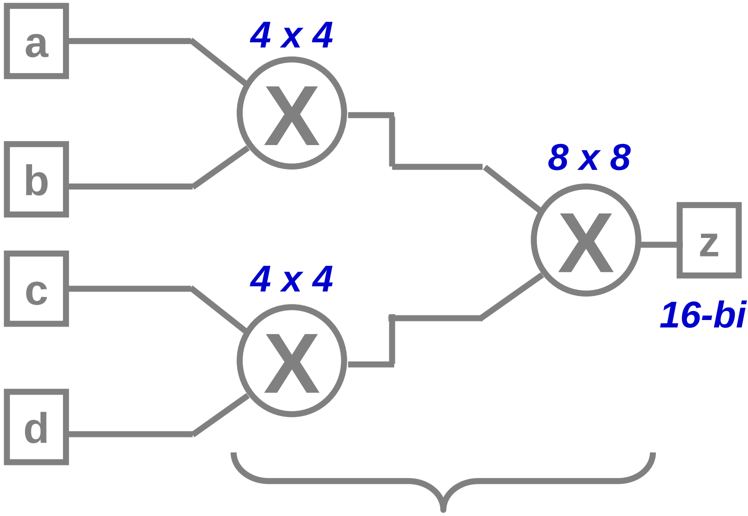


16-bit

Delay through 3
Stages of Multiply

Balanced

out = (a * b) * (c * d)



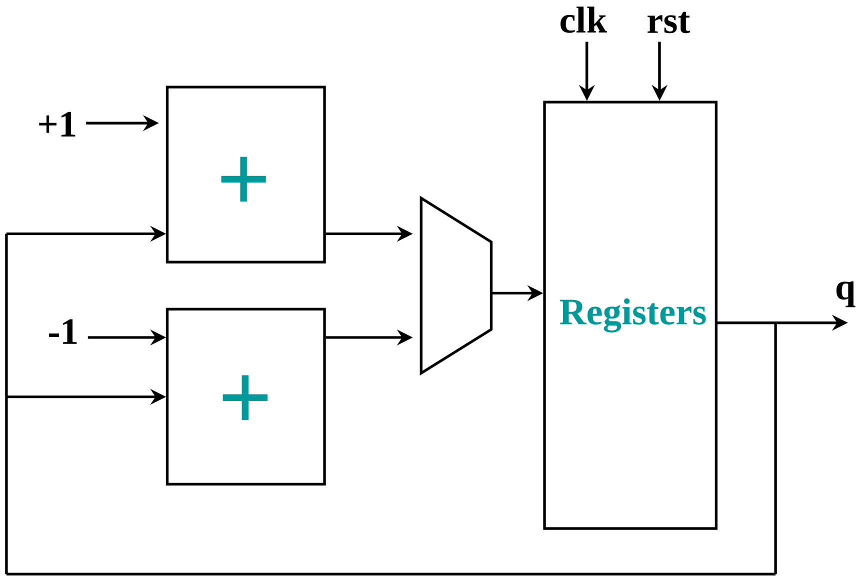
16-bit

Delay through 2
Stages of Multiply

Mutually Exclusive Operators

- *Up/Down Counter*
- *Quartus recognizes this code as a counter and will implement it efficiently, but not all tools will recognize it.*

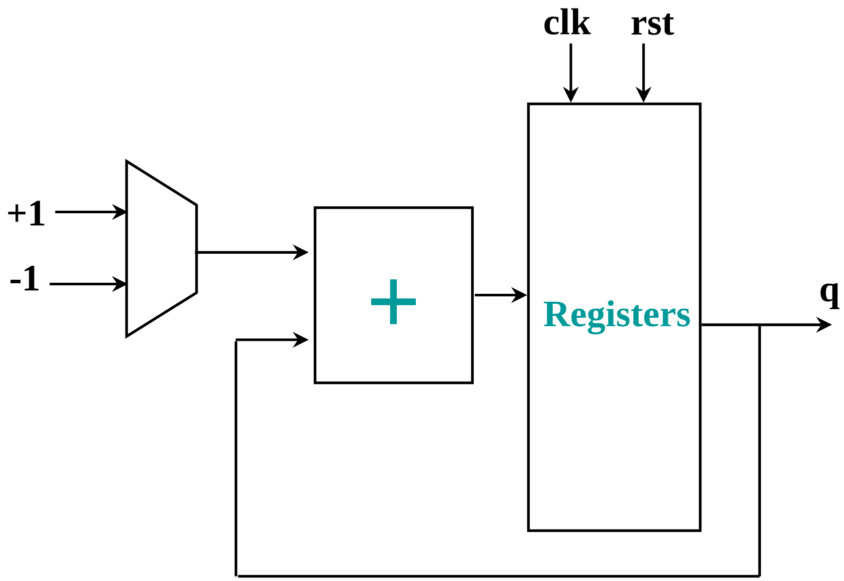
```
module test(rst, clk, updn, q);
input rst, clk, updn;
output [7:0] q;
reg [7:0] q;

always@(posedge clk or negedge rst)
begin
  if (!rst)
    q<=0;
  else if (updn)
         q<=q+1;
       else
         q<=q-1;
end
endmodule
```



Sharing Mutually Exclusive Operators

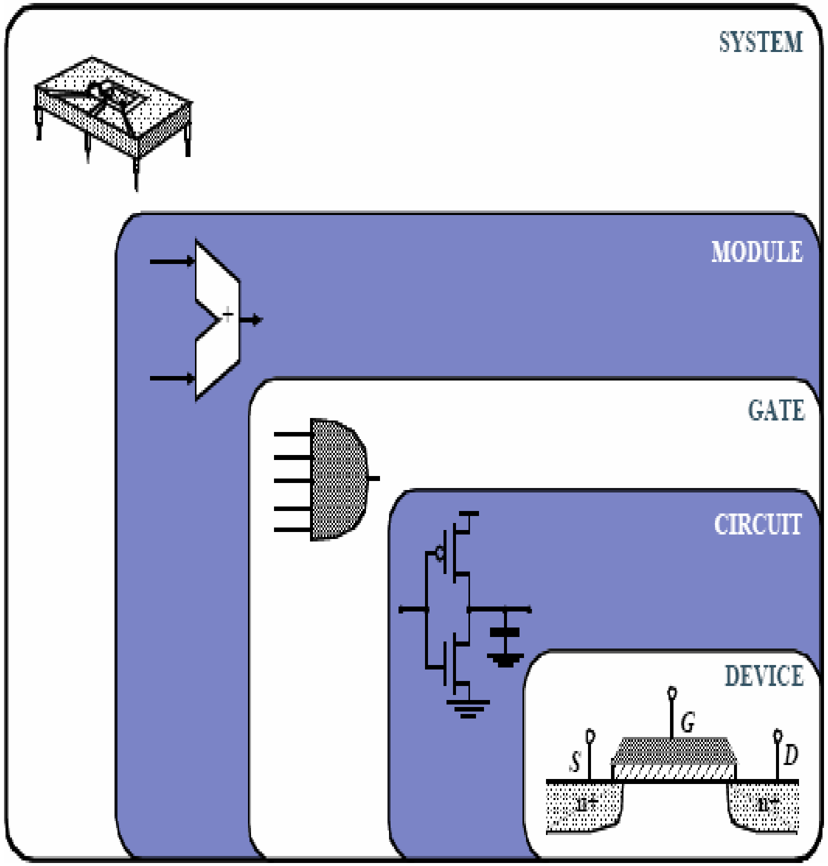
```
module test(rst, clk, updn, q);
input rst, clk, updn;
output [7:0] q;
reg [7:0] q;
integer dir;

always@(posedge clk or negedge
rst)begin
  if (!rst)
    q<=0;
  else if (updn)
         dir<=1;
       else
         dir<=-1;
  q<=q+dir;
end
endmodule
```

- *Up/Down Counter*
- *Only One Adder Required*



6.1.2 RTL synthesis

- RTL synthesis begins with an architecture and converts language-based RTL statements into a set of Boolean equations that can be optimized by a logic synthesis tool



RTL description

- RTL codes define:

① The register structure and register amount of the circuit

② The topology of the circuits

③ The logic function of the combination circuits between in/out to register or register to register

6.1.3 High-level synthesis

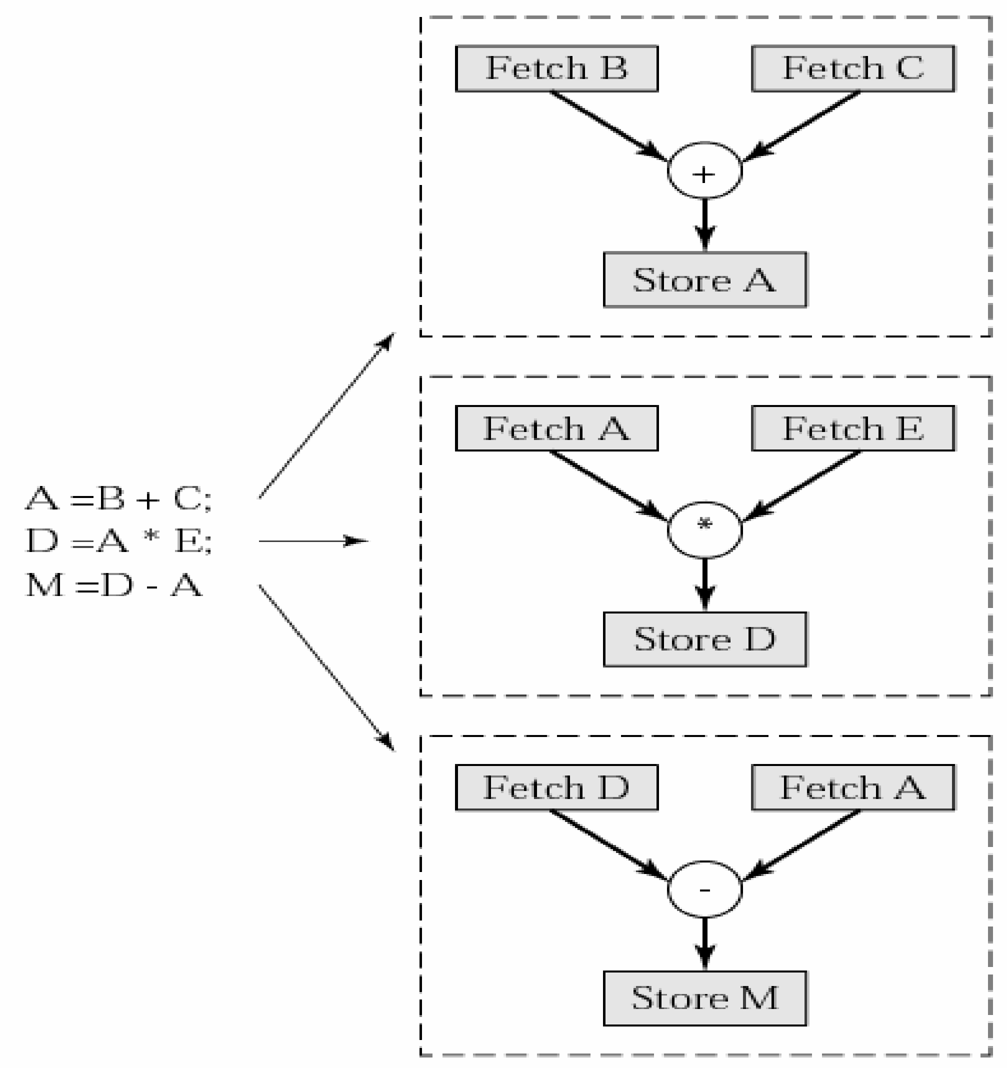
- Goal is to find an architecture whose resources **can be scheduled and allocated to implement an algorithm**
- The algorithm describes only the functionality, **not explicitly declare a structure of registers and datapaths**
- Many different architectures may implement the same functional specification

A behavioral synthesis

- **resource *allocation* and resource *scheduling***
- two main steps to create an architecture of datapath elements, control units, and memory:
- **The allocation step identifies the operators and infers the need for memory resources**
- The operations are assigned to specific clock cycles **to implement the ordered sequential activity flow of the algorithm**
- **Resource *scheduling*** determines the number of computational units

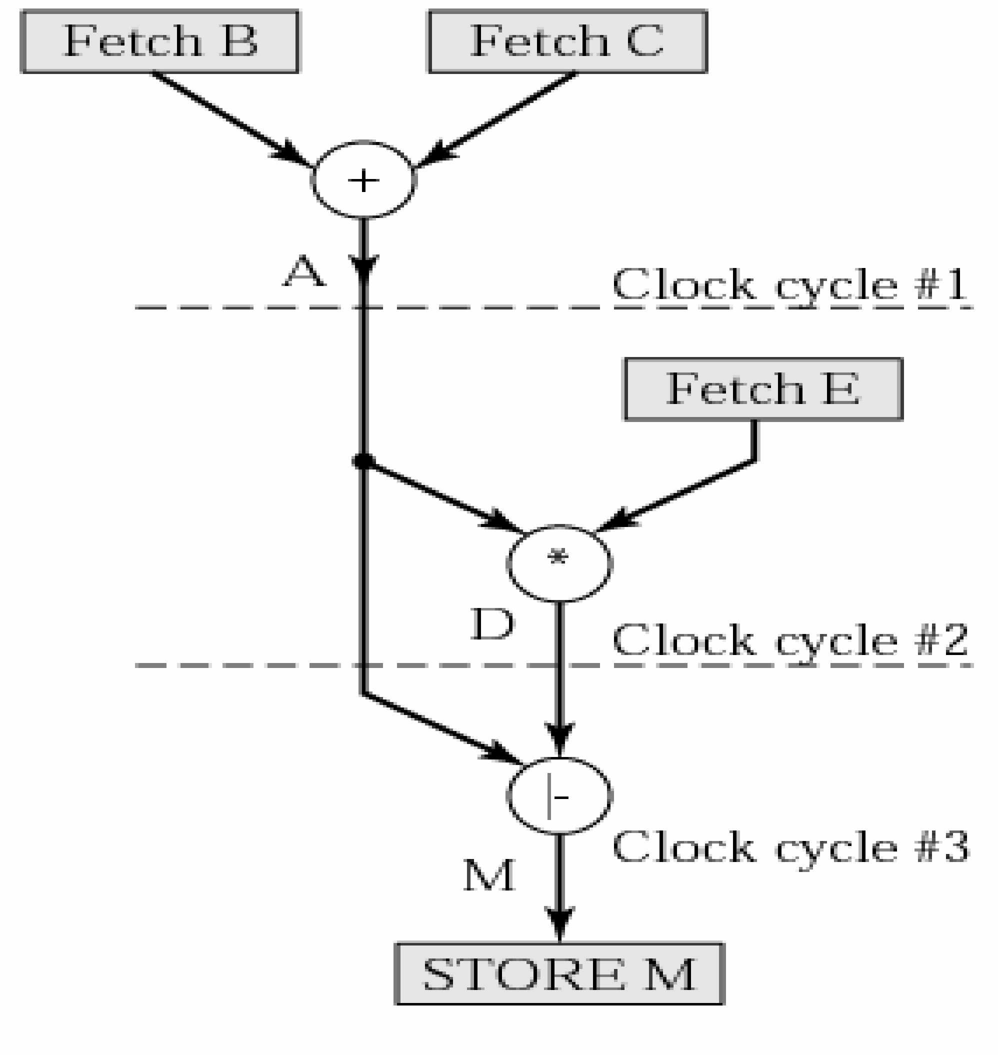
Representation of a behavioral model

parse trees



Representation of a behavioral model

Data Flow Grapl



6.2 Synthesis of combinational logic

- There are many ways to describe combinational logic, but some are not supported by synthesis tools

Synthesizable combinational logic can be described by:

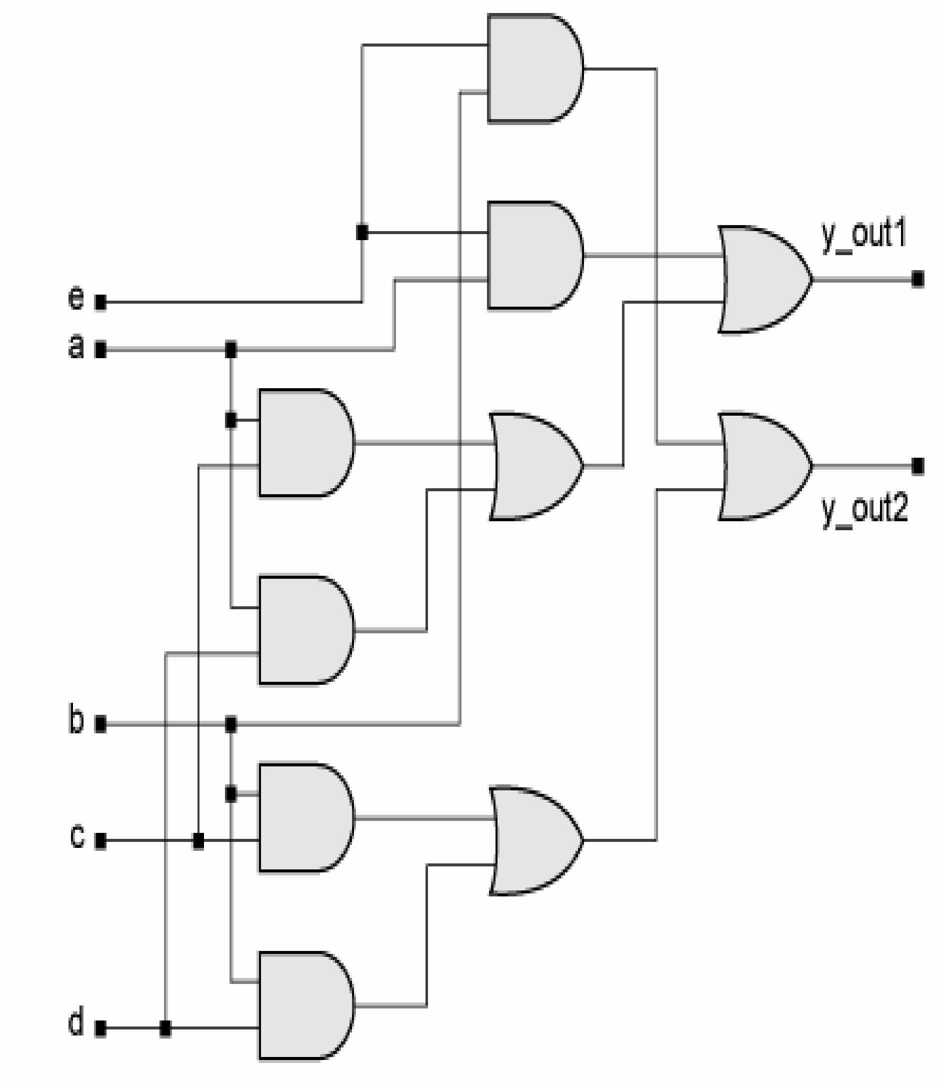
1. **a netlist of structural primitives**
2. **a set of continuous-assignment statement**
3. **a level-sensitive cyclic behavior**

1. Synthesizes a netlist

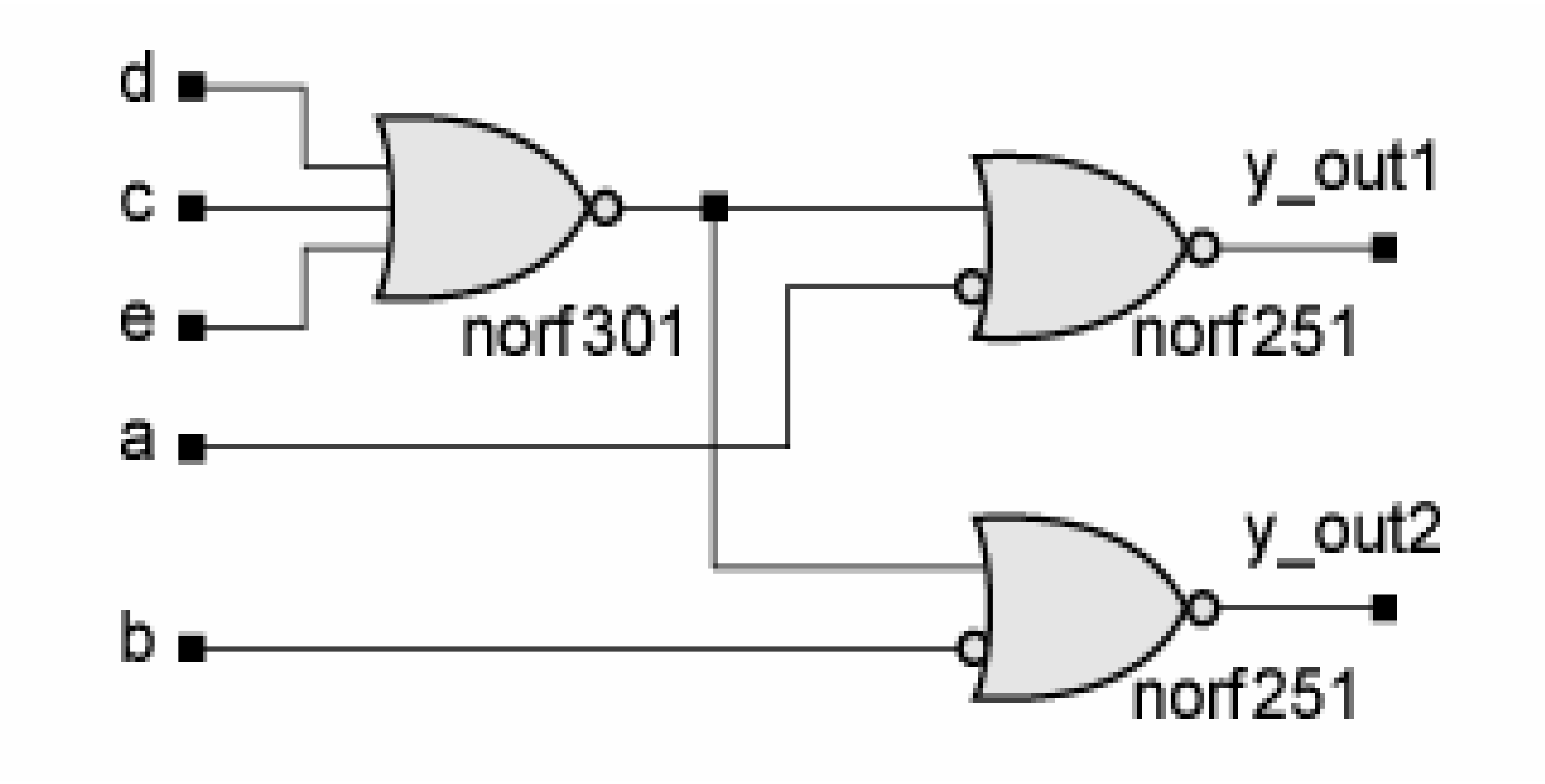
- A netlist of primitives should be synthesized to remove any redundant logic before mapping the design into a technology
- To ensure that the logic has been minimized correctly

Ex 6.6 Synthesizes a netlist : the pre-optimized circuit

```
module boole_opt(y_out1,
  y_out2, a, b, c, d, e);
  output      y_out1, y_out2;
  input       a, b, c, d, e;
  and         (y1,a,c);
  and         (y2,a,d);
  and         (y3,a,e);
  or          (y4,y1,y2);
  or          (y_out1,y3,y4);
  and         (y5,b,c);
  and         (y6,b,d);
  and         (y7,b,e);
  or          (y8,y5,y6);
  or          (y_out2,y7,y8);
endmodule
```



The synthesized circuit

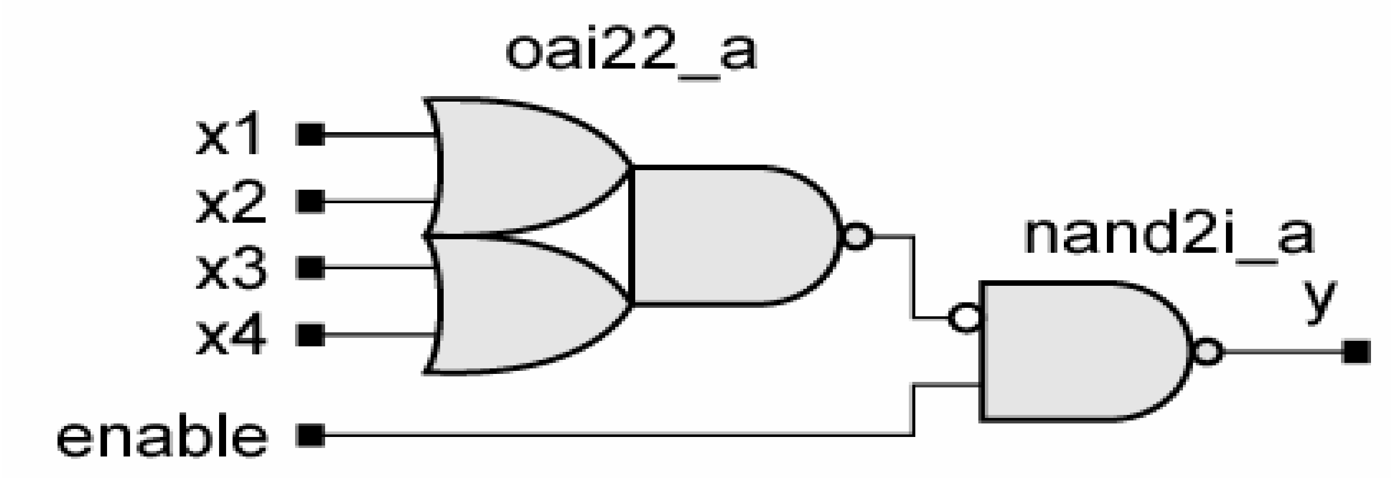


2. Continuous-assignment statements are synthesizable

- The expression in a continuous assignment statement will **be translated by the synthesis tool to an equivalent Boolean equation**
- The designer must verify that the continuous assignment correctly describes the logic, and that the synthesized multilevel logic circuit has the correct functionality

Example 6.7 The continuous assignment in or_nand

```
module or_nand (y, enable, x1,x2,x3,x4)
  output y;
  input enable, x1,x2,x3,x4;
  assign y=~(enable&(x1|x2)&(x3|x4));
endmodule
```



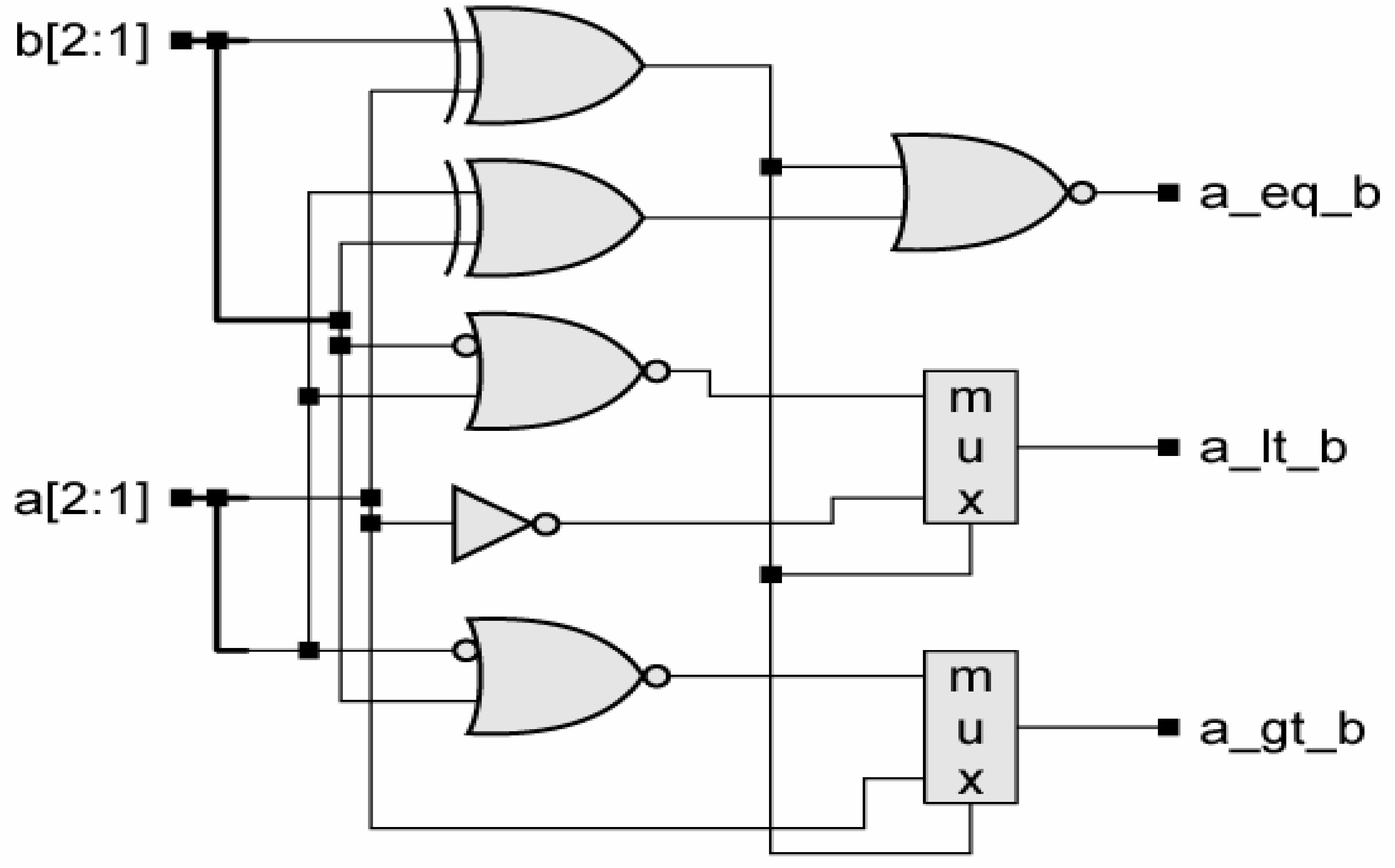
3. Level-sensitive cyclic behavior

- A level-sensitive cyclic behavior will synthesize to combinational logic **if it assigns a value to each output for every possible value of its inputs**
- The event control expression of the behavior must be sensitive to every input
- The every path of the activity flow must assign value to every output

Ex 6.8   2-bit comparator

```
module comparator (a_gt_b,a_lt_b,a_eq_b,a,b); // Alternative algorithm
    parameter   size=2;
    output      a_gt_b,a_lt_b,a_eq_b;
    input      [size:1]  a,b;
    reg         a_gt_b,a_lt_b,a_eq_b;
    integer     k;
    always @ (a or b)  begin: compare_loop
      for (k=size; k>0;k=k-1)
        begin
          if (a[k] != b[k]) begin
            a_gt_b=a[k]; a_lt_b=~a[k]; a_eq_b=0;
            disable compare_loop;
          end
        end
       a_gt_b=0; a_lt_b=0;a_eq_b=1;
   end
endmodule
```

Combinational logic synthesis of 2-bit comparator

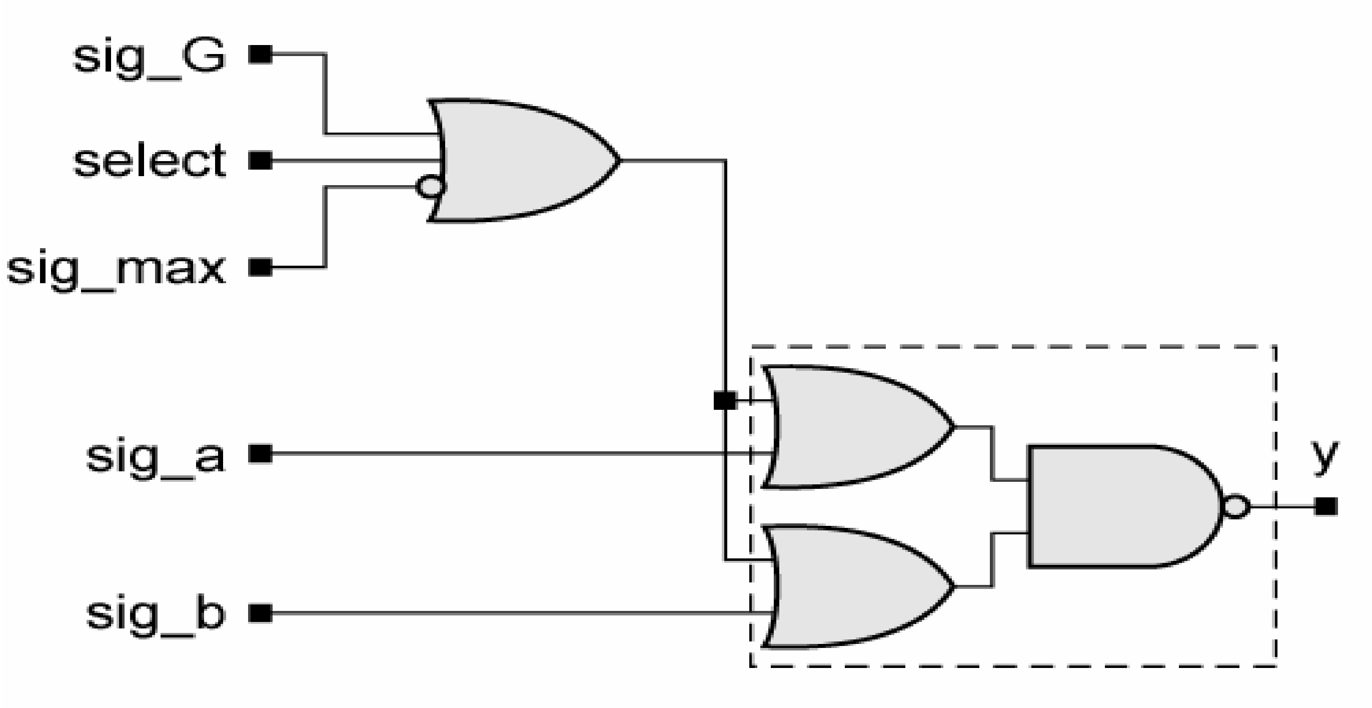


Timing

- Timing imposes a constraint on the speed of a circuit, but not on its functionality
- **The speed of ASIC is proportional to the physical geometry**
- **Faster parts require more area**
- Technology-dependent timing constructs are not to be included in the model

Example 6.9

```
module mux_logic (y,select,sig_G,sig_max,sig_a,sig_b);
  output y;
  input select,sig_G,sig_max,sig_a,sig_b;
  assign
  y=(select==1)||(sig_G==1)||(sig_max==0)?sig_a:sig_b;
endmodule
```



6.2.1 Synthesis of priority structures

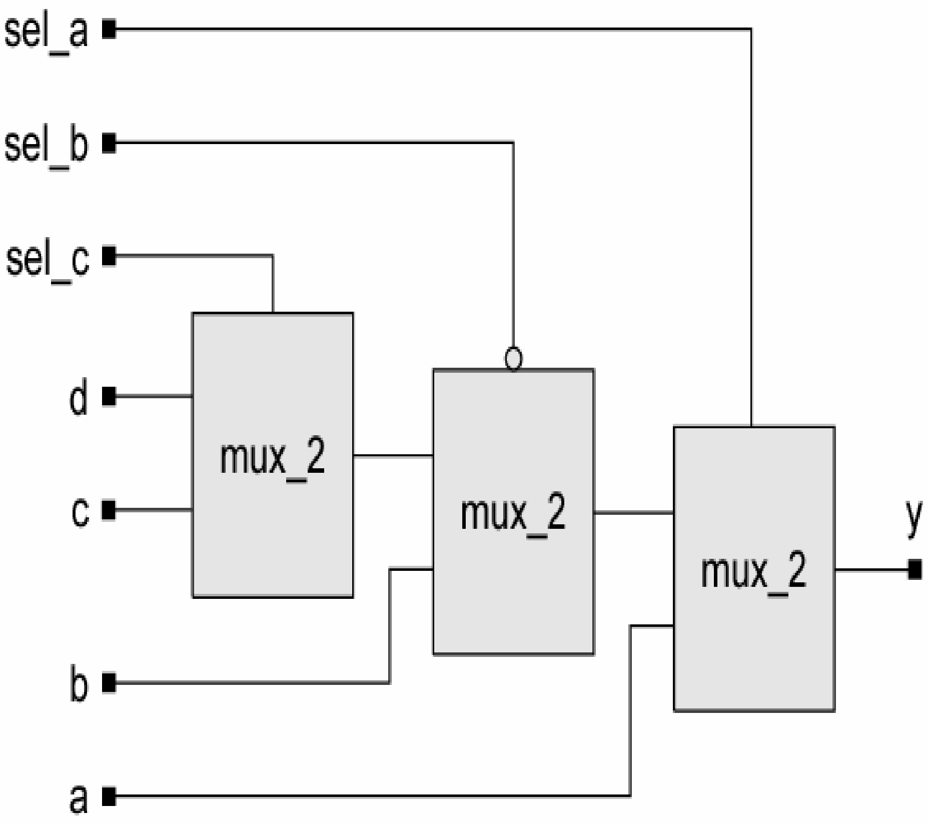
- A **case** statement **implicitly attaches higher priority** to the first item that it decodes than to the last one
- An **if** statement implies higher priority to the first branch than to the remaining branches.

Ex 6.10 Mux with priority

```
module
    mux_4pri(y,a,b,c,d,sel_a,s
    el_b,sel_c);
    output y;
    input
    a,b,c,d,sel_a,sel_b,sel_c;
    reg y;
    always @ (sel_a or sel_b
    or sel_c or a or b or c or d)
      begin
           if (sel_a==1) y=a; else
           if (sel_b==0) y=b; else
           if (sel_c==1) y=c; else
      end
endmodule
```

Ex 6.10 Mux with priority

```
module
    mux_4pri(y,a,b,c,d,sel_a,s
    el_b,sel_c);
    output y;
    input
    a,b,c,d,sel_a,sel_b,sel_c;
    reg y;
    always @ (sel_a or sel_b
    or sel_c or a or b or c or d)
      begin
          if (sel_a==1) y=a; else
          if (sel_b==0) y=b; else
          if (sel_c==1) y=c; else
      end
endmodule
```



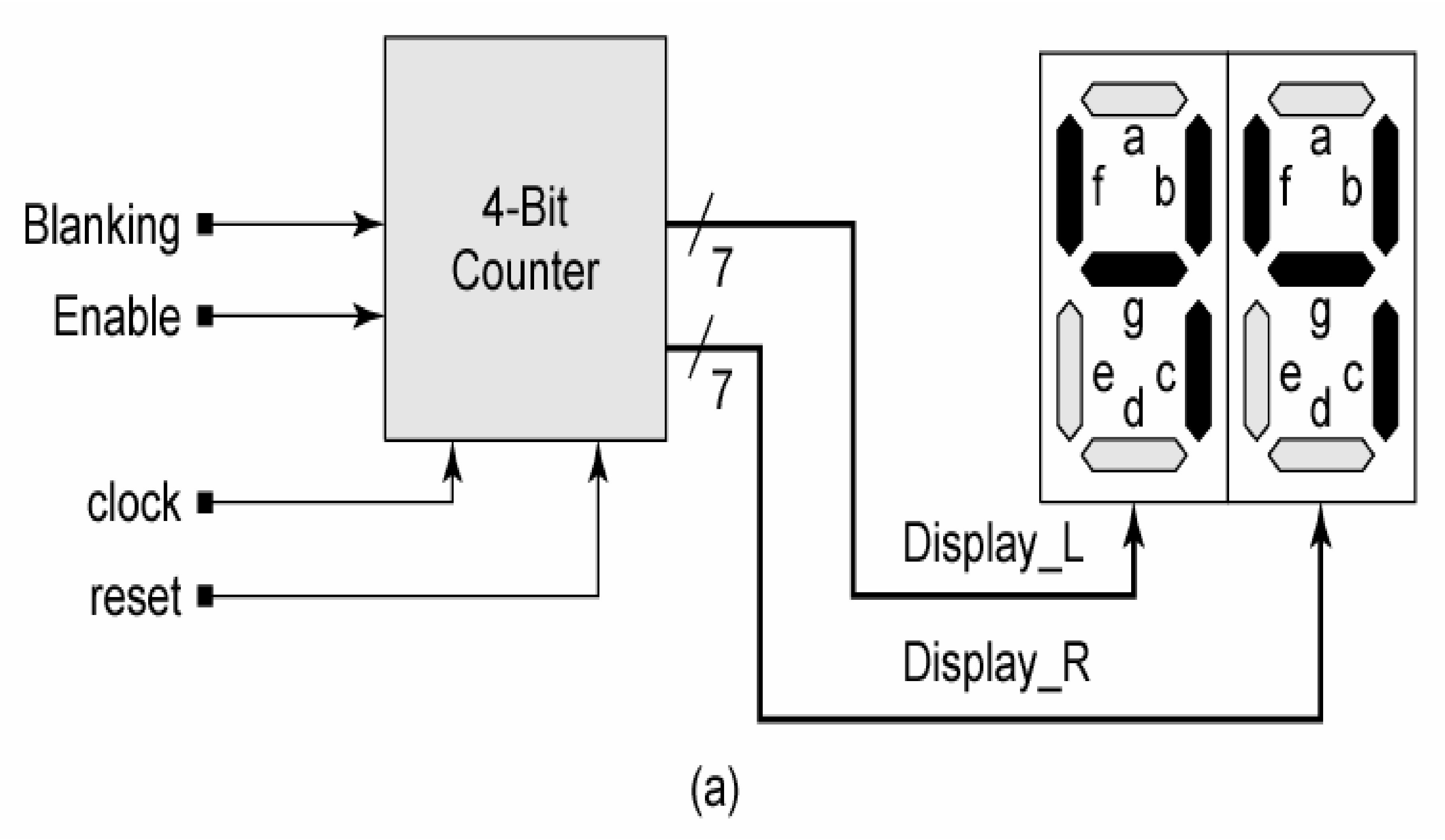
6.2.2 Exploiting logical don’t-care conditions

- When ***case***, ***if***, ***?... :*** are used, synthesized netlist should produce the same simulation results if ***default*** assignments are **purely 0 or 1**
- Simulation results may differ if the default or branch statement makes an explicit assignment of an **x** or a **z**
- **An assignment to x in a *case* or an *if* statement is treated as a don’t-care condition in synthesis**
- This may lead to a mismatch between the results of simulation and synthesis

Example 6.11

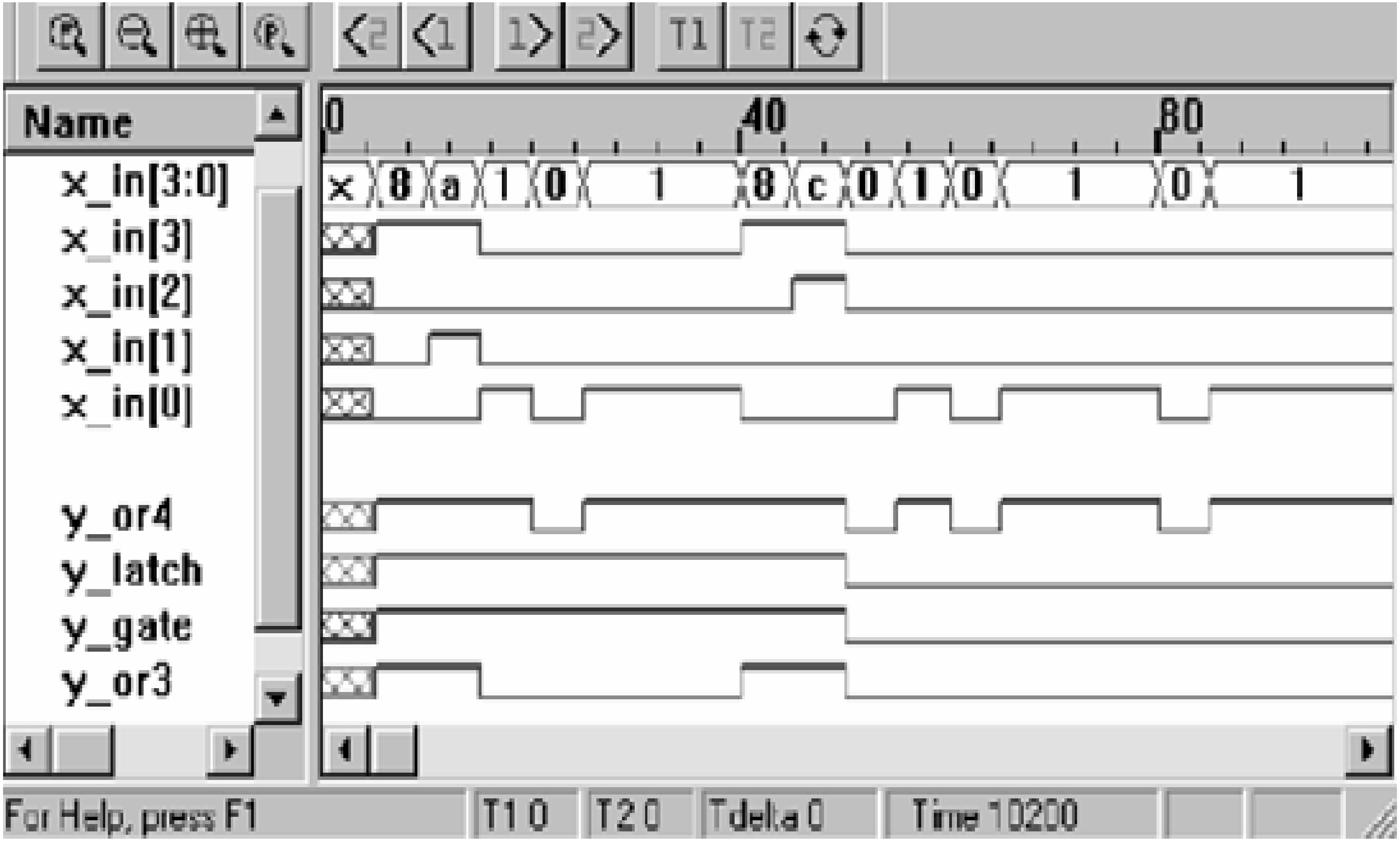
```
module Latched_Seven_Seg_Display
        (Display_L, Display_R, Blanking, Enable, clock, reset);
 output      [6:0]   Display_L, Display_R;
 input               Blanking, Enable, clock, reset;
 reg         [6:0]   Display_L, Display_R;
 reg         [3:0]   count; //   abc_defg
 parameter  BLANK = 7'b111_1111;
 parameter  ZERO  = 7'b000_0001; // h01
 parameter  ONE    = 7'b100_1111; // h4f
 parameter  TWO   = 7'b001_0010; // h12
 parameter  THREE = 7'b000_0110; // h06
 parameter  FOUR  = 7'b100_1100; // h4c
 parameter  FIVE   = 7'b010_0100; // h24
parameter     SIX   = 7’b010_0000;          // h20
parameter     SEVEN = 7’b000_1111;          // h0f
parameter     EIGHT = 7’b000_0000;          // h00
parameter     NINE  = 7’b000_0100;          // h04
always @ (posedge clock)
    if (reset) count<=0;
    else if (Enable) count<=count+1;
always @ (count or Blanking)
   if (Blanking) begin Display_L=BLANK; Display_R=BLANK; end
   else
    case (count)
         0 :     Display_L=ZERO; Display_R=ZERO; end
         2 :     Display_L=ZERO; Display_R=TWO; end
         4 :     Display_L=ZERO; Display_R=FOUR; end
         6 :     Display_L=ZERO; Display_R=SIX; end
         8 :     Display_L=ZERO; Display_R=EIGHT; end
        10 :     Display_L=ONE; Display_R=ZERO; end
        12 :     Display_L=ONE; Display_R=TWO; end
        14 :    Display_L=ONE; Display_R=FOUR; end
          //default : Display_L=BLANK; Display_R=BLANK; end
    endcase
endmodule
```

Seven-segment LED display



(a)

Simulation result



For Help, press F1

T1 0

T2 0

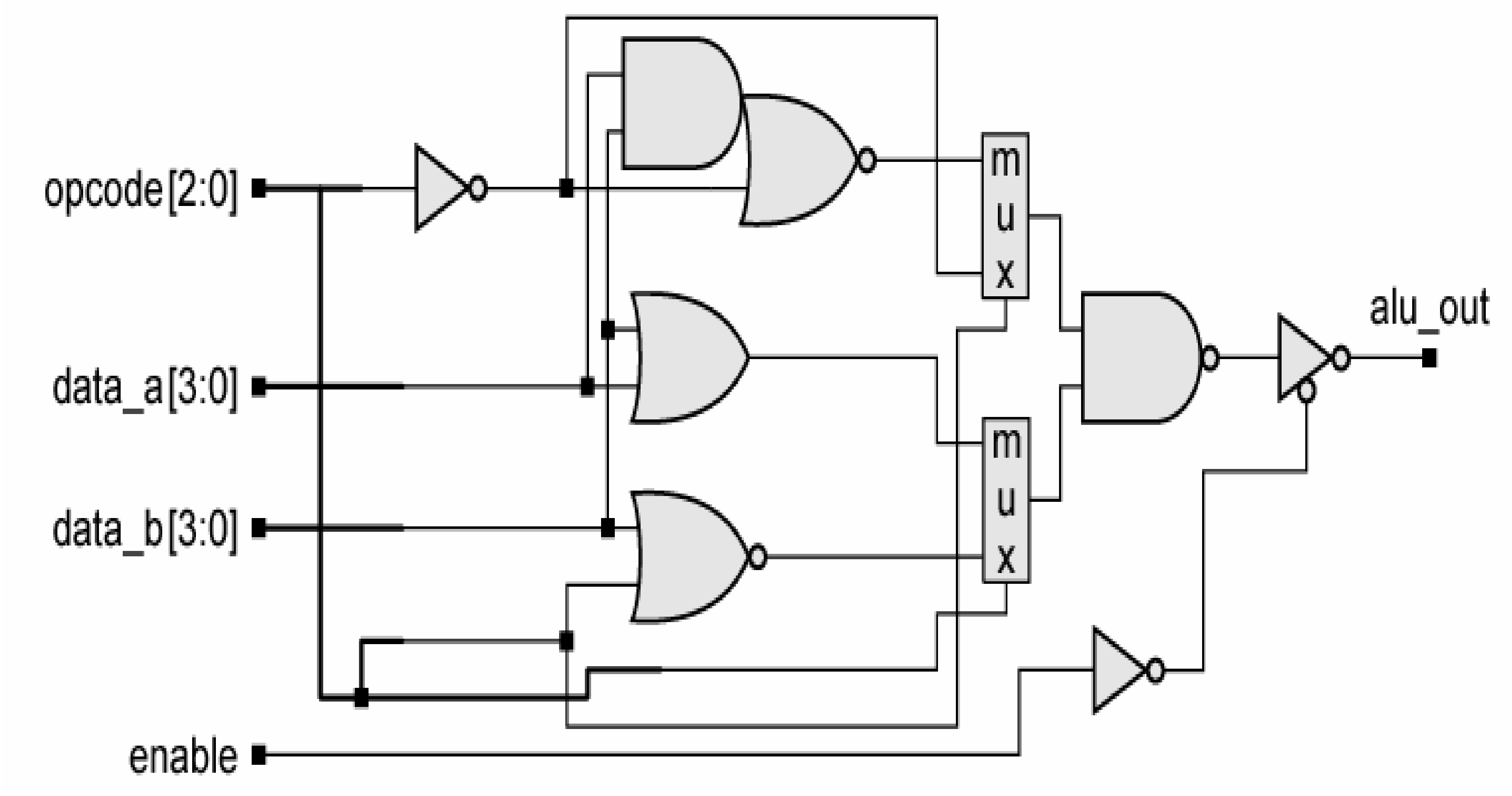
Tdelta 0

Time 10200

Example 6.12

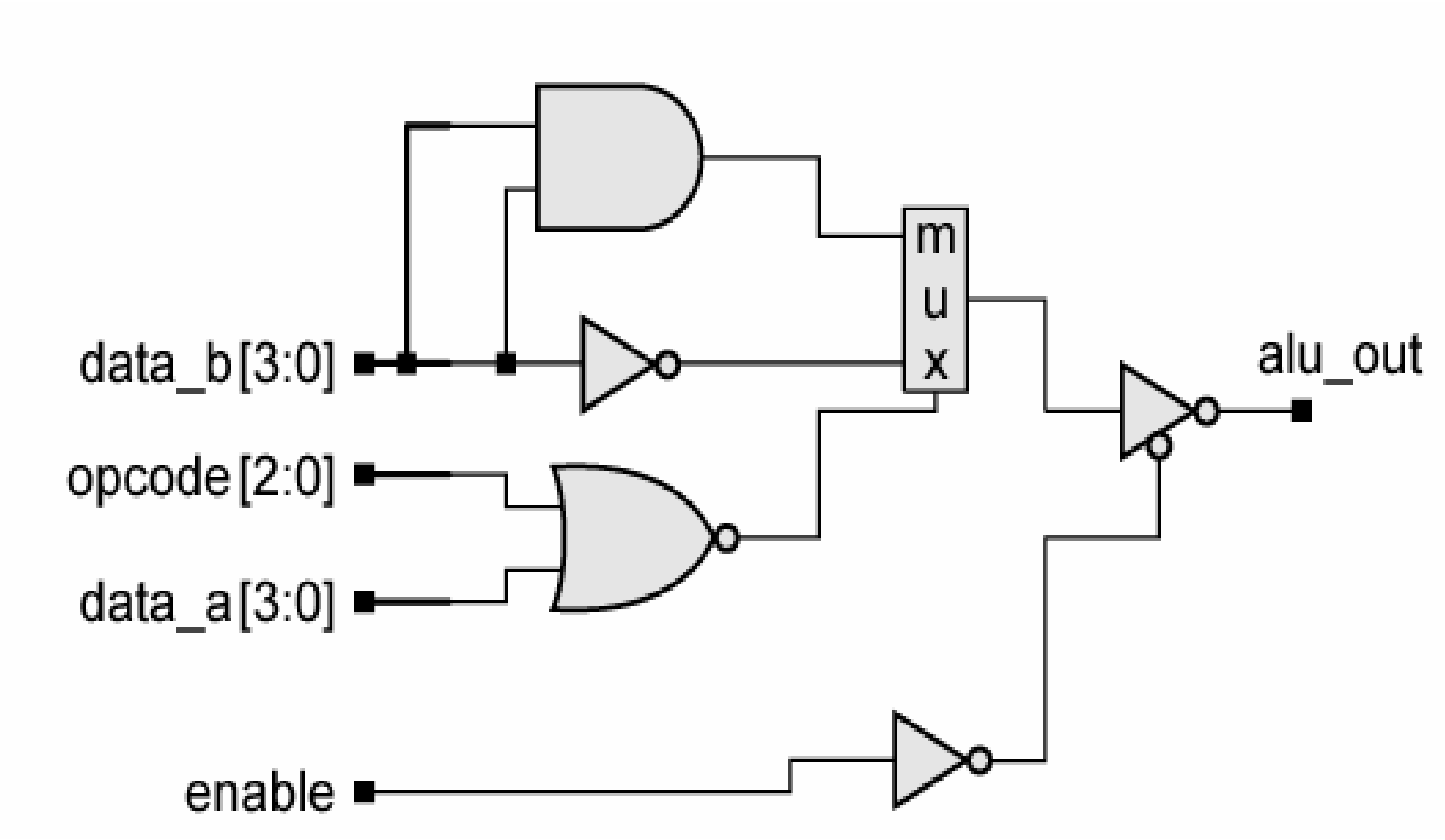
```
module alu_with_z1 (alu_out,data_a,data_b,enable,opcode);
   input [2:0] opcode;
   input [3:0] data_a,data_b;
   input        enable;
   output  alu_out;  //scalar for illustration
   reg  [3:0] alu_reg;
   assign alu_out=(enable==1) ? alu_reg : 4’z;
   always @ (opcode or data_a or data_b)
     case (opcode)
          3’b001: alu_reg=data_a | data_b;
          3’b010: alu_reg=data_a ^ data_b;
          3’b110: alu_reg=~data_b；
           default: alu_reg=4’b0;
     endcase
endmodule
```

Circuits synthesized from alu_with_z1



```
module alu_with_z2
    (alu_out,data_a,data_b,enable,opcode);
    input [2:0] opcode;
    input [3:0] data_a,data_b;
    input        enable;
    output  alu_out;  //scalar for illustration
    reg  [3:0] alu_reg;
    assign alu_out=(enable==1)?alu_reg:4’z;
    always @ (opcode or data_a or data_b)
      case (opcode)
           3’b001: alu_reg=data_a | data_b;
           3’b010: alu_reg=data_a ^ data_b;
           3’b110: alu_reg=~data_b；
            default: alu_reg=4’bx;
      endcase
endmodule
```

Circuits synthesized from alu_with_z2



6.2.3 ASIC cells and resource sharing

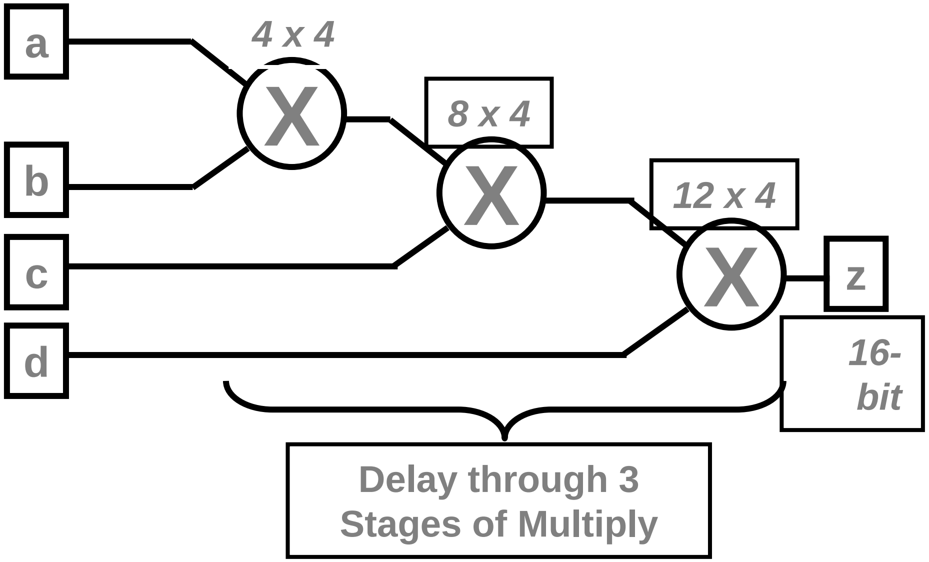
- An ASIC cell library contains cells that are more complex than combinational primitive gates
- The tool must share resources to minimize needless duplication of circuitry
- Use parentheses to control operator grouping for reducing the size of a circuit

Example: Balancing Operators

- a, b, c, d: 4-bit vectors

Unbalanced

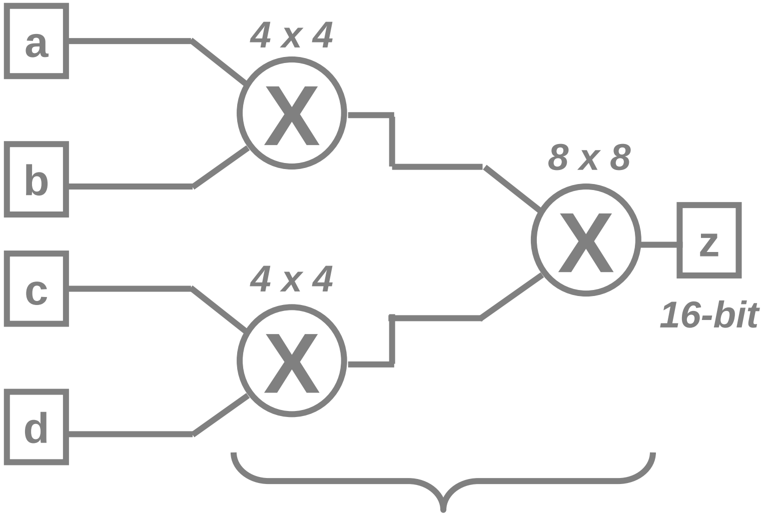
out = a * b * c * d



Delay through 3 Stages of Multiply

Balanced

out = (a * b) * (c * d)

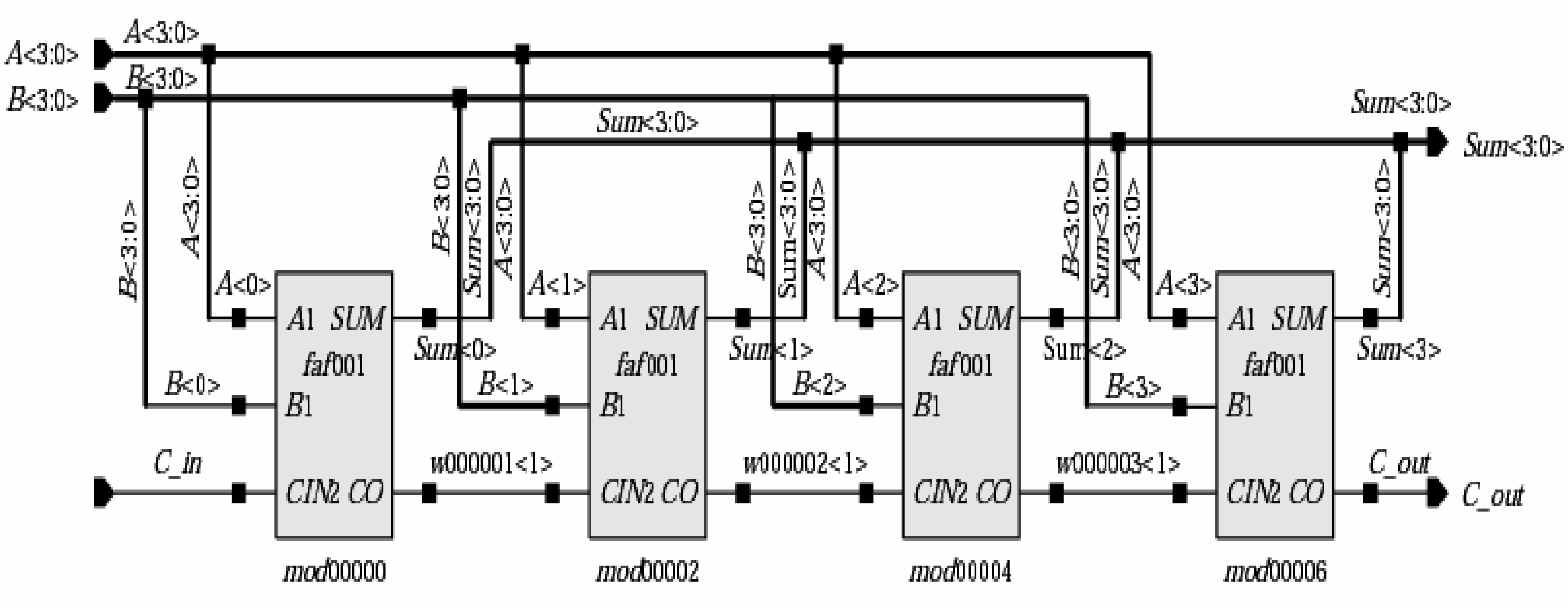


Delay through 2 Stages of Multiply

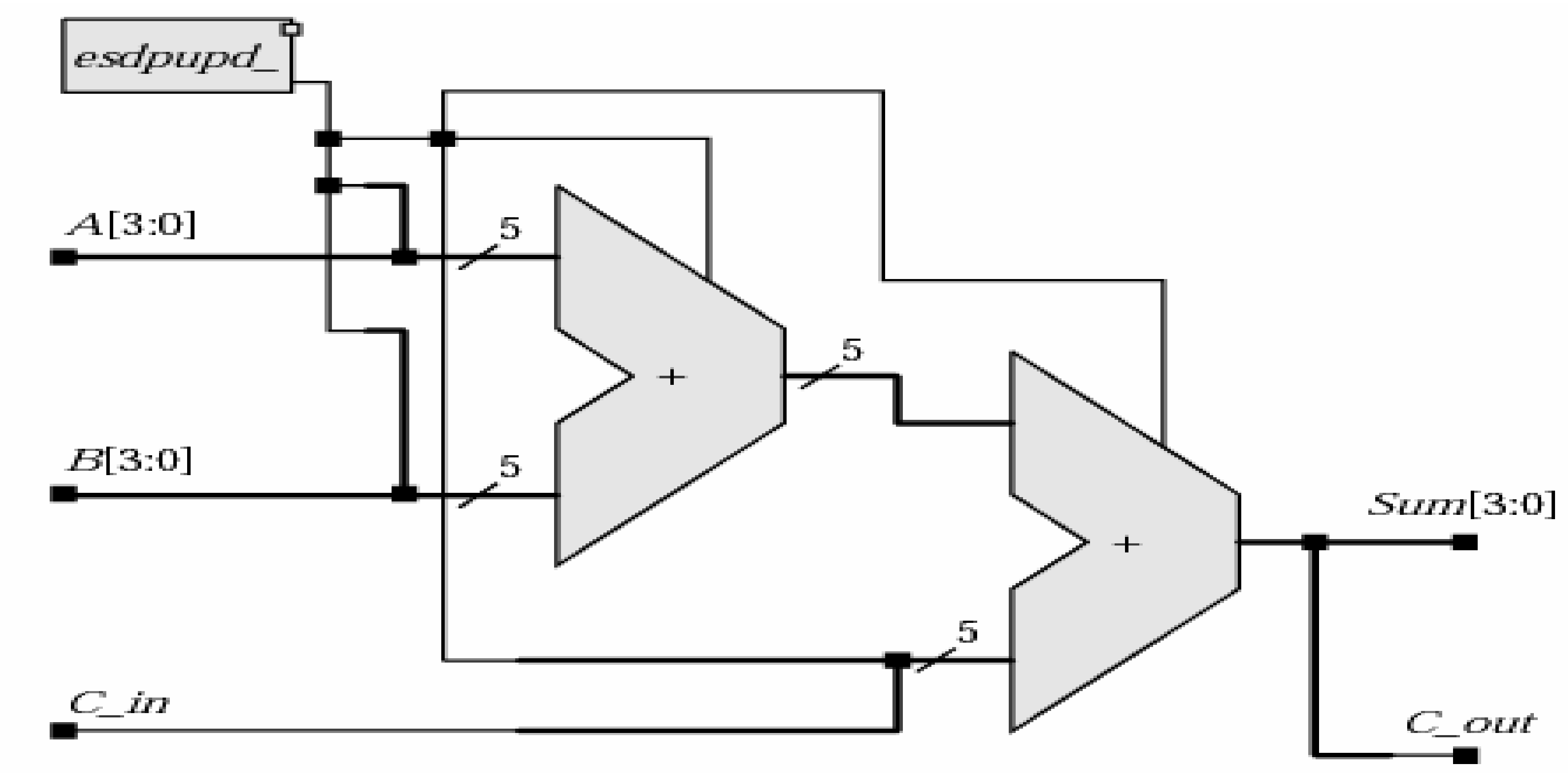
Example 6.13

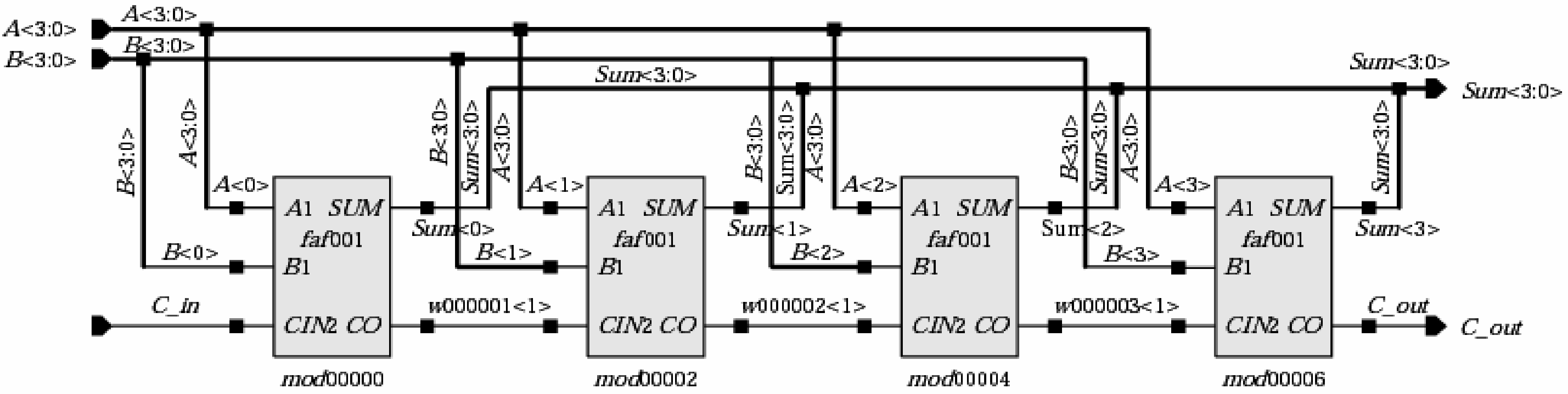
```
module badd_4 (Sum,C_out,A,B,C_in);
  output [3:0] Sum;
  output    C_out;
  input [3:0] A,B
  input     C_in;
  assign {C_out,Sum}=A+B+C_in;
endmodule
```

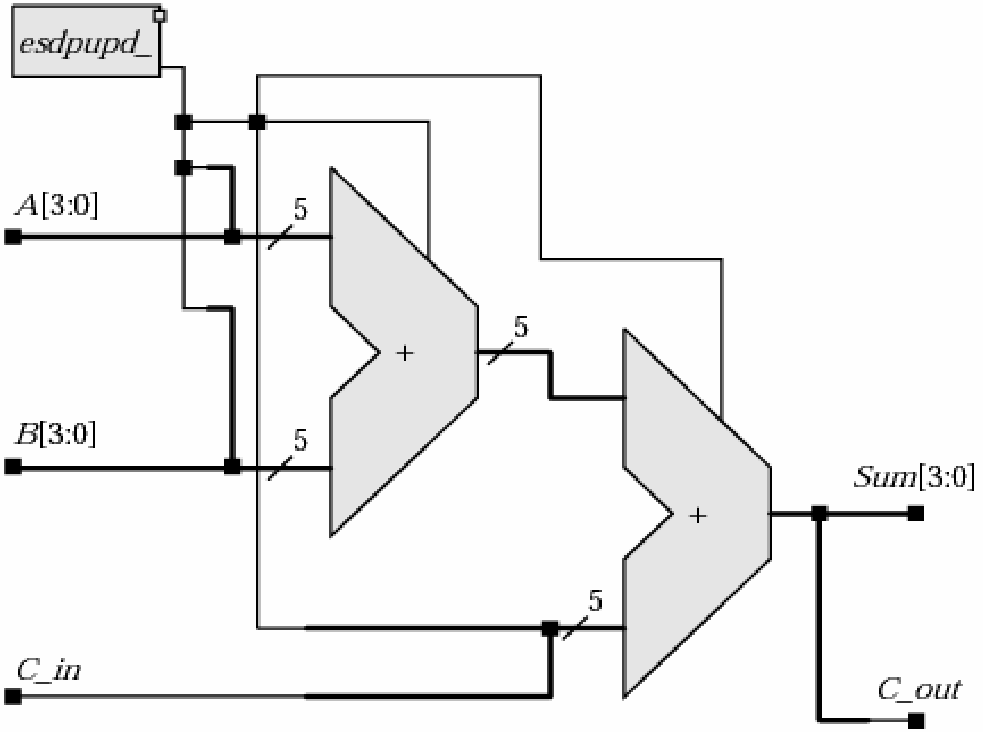
The synthesis result of the addition operator in a full-adder ASIC library cell



The synthesis result of the addition operator in 5-bit adder blocks




A<3:0>
B<3:0>
A<3:0>
B<3:0>
Sum<3:0>
Sum<3:0>
Sum<3:0>
B<3:0>
A<3:0>
A<0>
B<0>
C_in
A1 SUM
faf001
B1
CIN2 CO
mod00000
Sum<0>
B<3:0>
Sum<3:0>
A<3:0>
A<1>
B<1>
w000001<1>
mod00002
Sum<1>
B<3:0>
Sum<3:0>
A<3:0>
A<2>
B<2>
w000002<1>
mod00004
Sum<2>
B<3:0>
Sum<3:0>
A<3:0>
A<3>
B<3>
w000003<1>
mod00006
Sum<3:0>
Sum<3>
C_out
C_out


esdpupd_
A[3:0]
5
B[3:0]
5
+
5
+
5
Sum[3:0]
C_in
C_out

Share resources

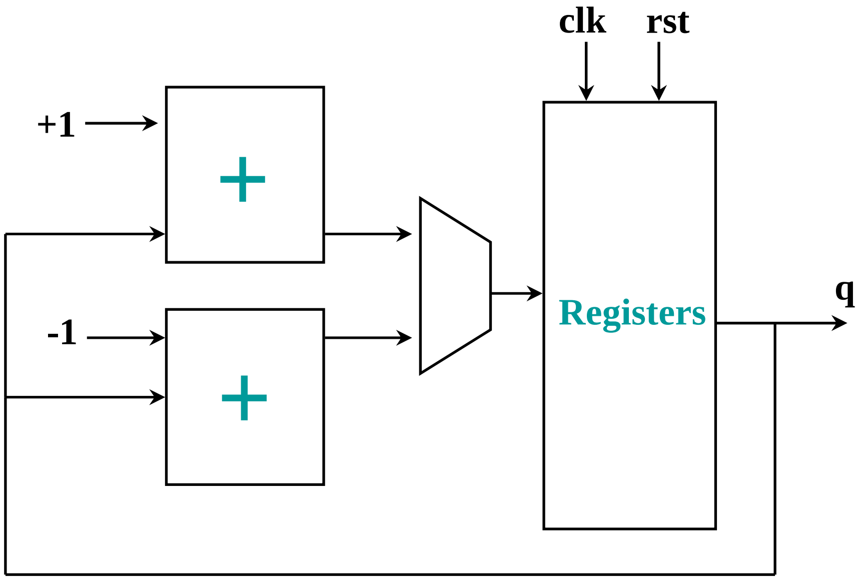
- For example:
- **assign y_out =**

  sel ? data_a + accum : data_a + data_b;
- The operators can be implemented by a shared adder whose input datapaths are multiplexed.
- **This feature is vendor-dependent**

Example: Share resources

- *Up/Down Counter*
- *Quartus recognizes this code as a counter and will implement it efficiently, but not all tools will recognize it.*

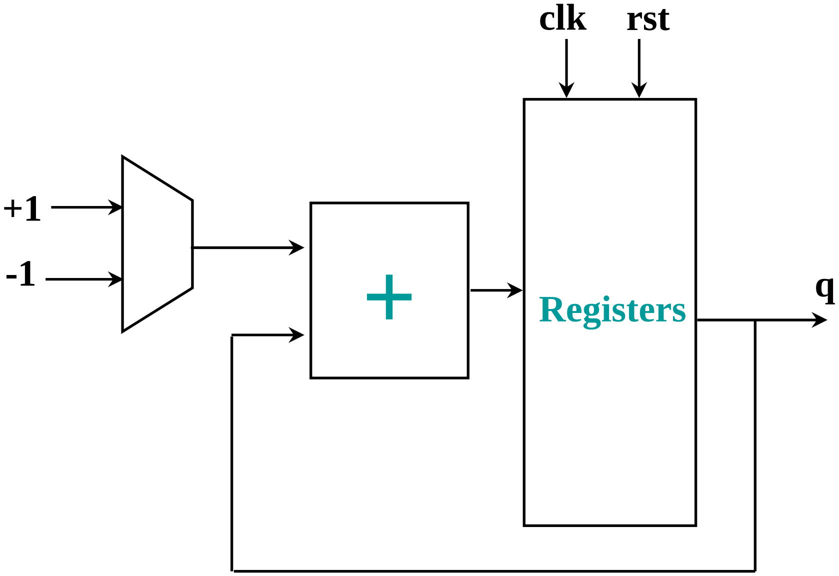
```
module test(rst, clk, updn, q);
input rst, clk, updn;
output [7:0] q;
reg [7:0] q;

always@(posedge clk or negedge rst)
begin
  if (!rst)
    q<=0;
  else if (updn)
        q<=q+1;
      else
        q<=q-1;
end
endmodule
```



Share resources

```
module test(rst, clk, updn, q);
input rst, clk, updn;
output [7:0] q;
reg [7:0] q;
integer dir;

always@(posedge clk or negedge
rst)begin
  if (!rst)
    q<=0;
  else if (updn)
         dir<=1;
       else
         dir<=-1;
  q<=q+dir;
end
endmodule
```

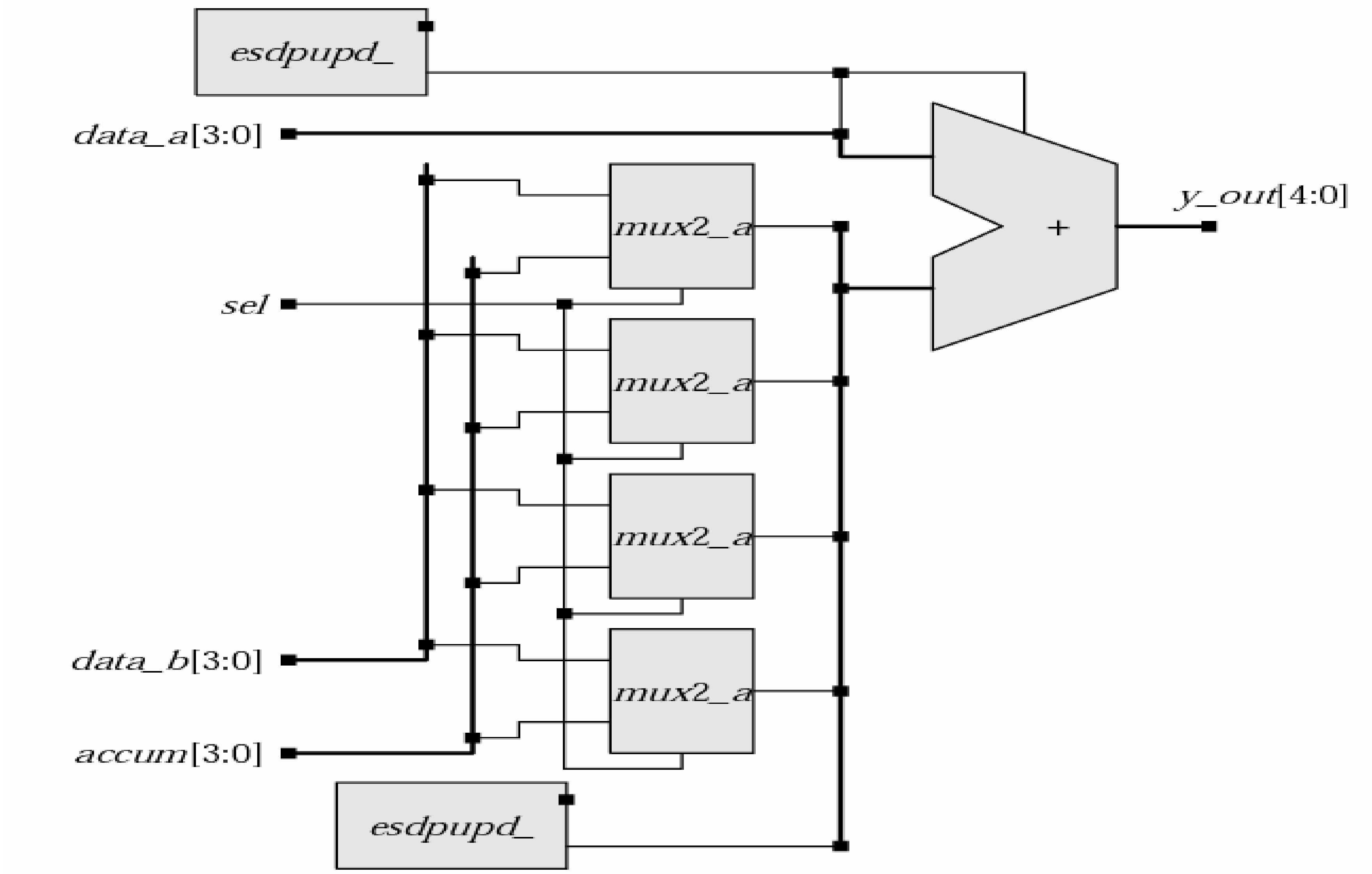
- *Up/Down Counter*
- *Only One Adder Required*



Ex 6.14 The use of parentheses in the description in res_share forces the synthesis tool to multiplex the datapaths

```
module res_share (y_out,sel,data_a,data_b,accum);
  output [4:0] y_out;
  input [3:0] data_a,data_b,accum;
  input   sel;
  assign y_out=data_a+(sel?accum:data_b);
endmodule
```

Implementation of a datapath with shared resources



6.3 Synthesis of sequential logic with latches

- Latches are synthesized in two ways: **intentionally and accidentally**
- A feedback-free netlist of combinational primitives, **a set of feedback-free continuous assignments** will synthesize into latch-free combinational logic
- **A continuous assignment using a conditional operator with feedback will synthesize into a latch**

Example 6.15

- **A cell of an SRAM memory** can be modeled by the following continuous assignment statement having feedback:

```
Assign
data_out=(CS_b==0)?(WE==0)?data_in:data_out:1’bz;
```

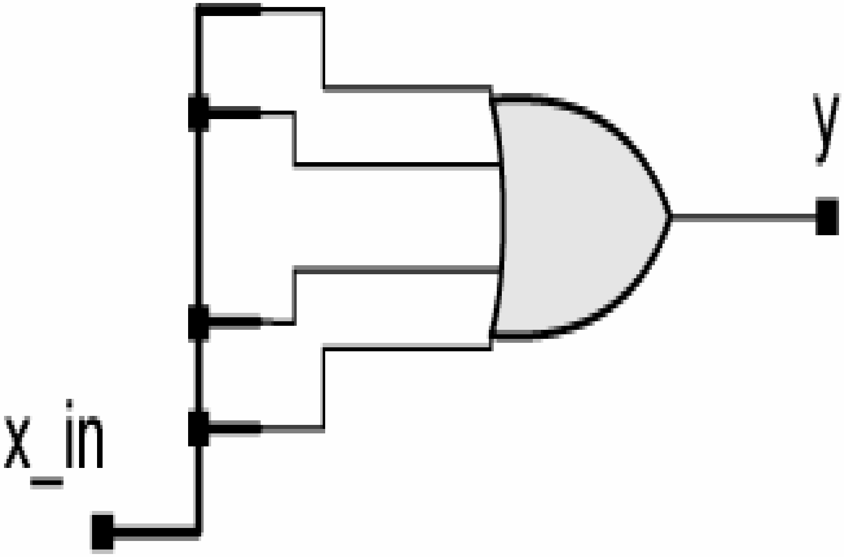
6.3.1 Accidental Synthesis of Latches

- **Example 6.16**

```
module or4_behav (y, x_in);
    parameter word_length=4;
    output y;
    input [word_length-1:0] x_in;
    reg y;
    integer k;
    always @ x_in
      begin: check_for_1
          y=0;
        for (k=0;k<=word_length-1;k=k+1)
            if (x_in[k]==1)
              begin y=1;
                disable check_for_1;
              end
        end
endmodule
```

Example 6.16

```
module or4_behav (y, x_in);
    parameter word_length=4;
    output y;
    input [word_length-1:0] x_in;
    reg y;
    integer k;
    always @ x_in
      begin: check_for_1
         y=0;
        for (k=0;k<=word_length-1;k=k+1)
          if (x_in[k]==1)
            begin y=1;
              disable check_for_1;
            end
        end
endmodule
```

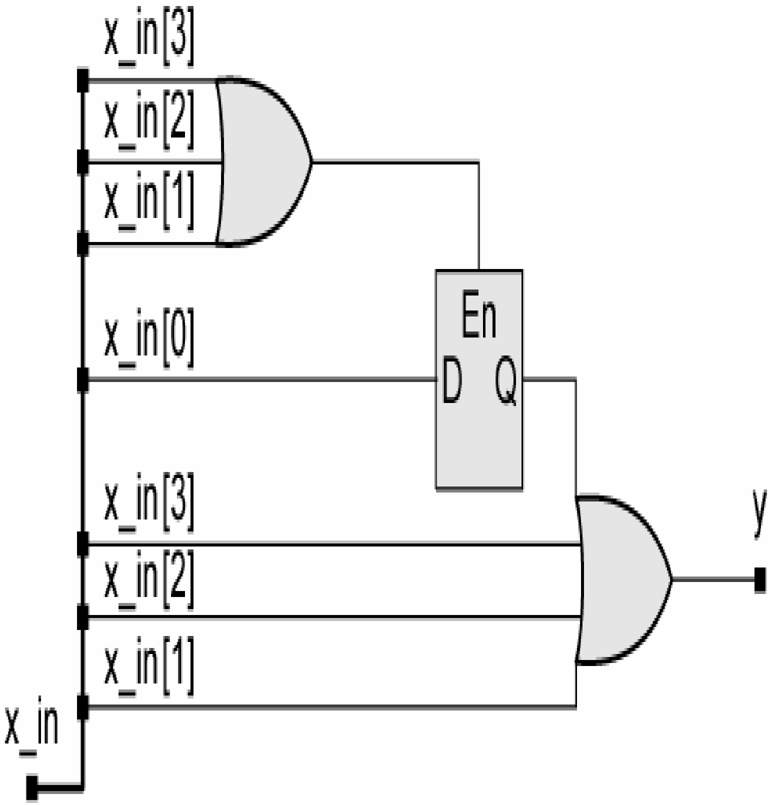


Example 6.16

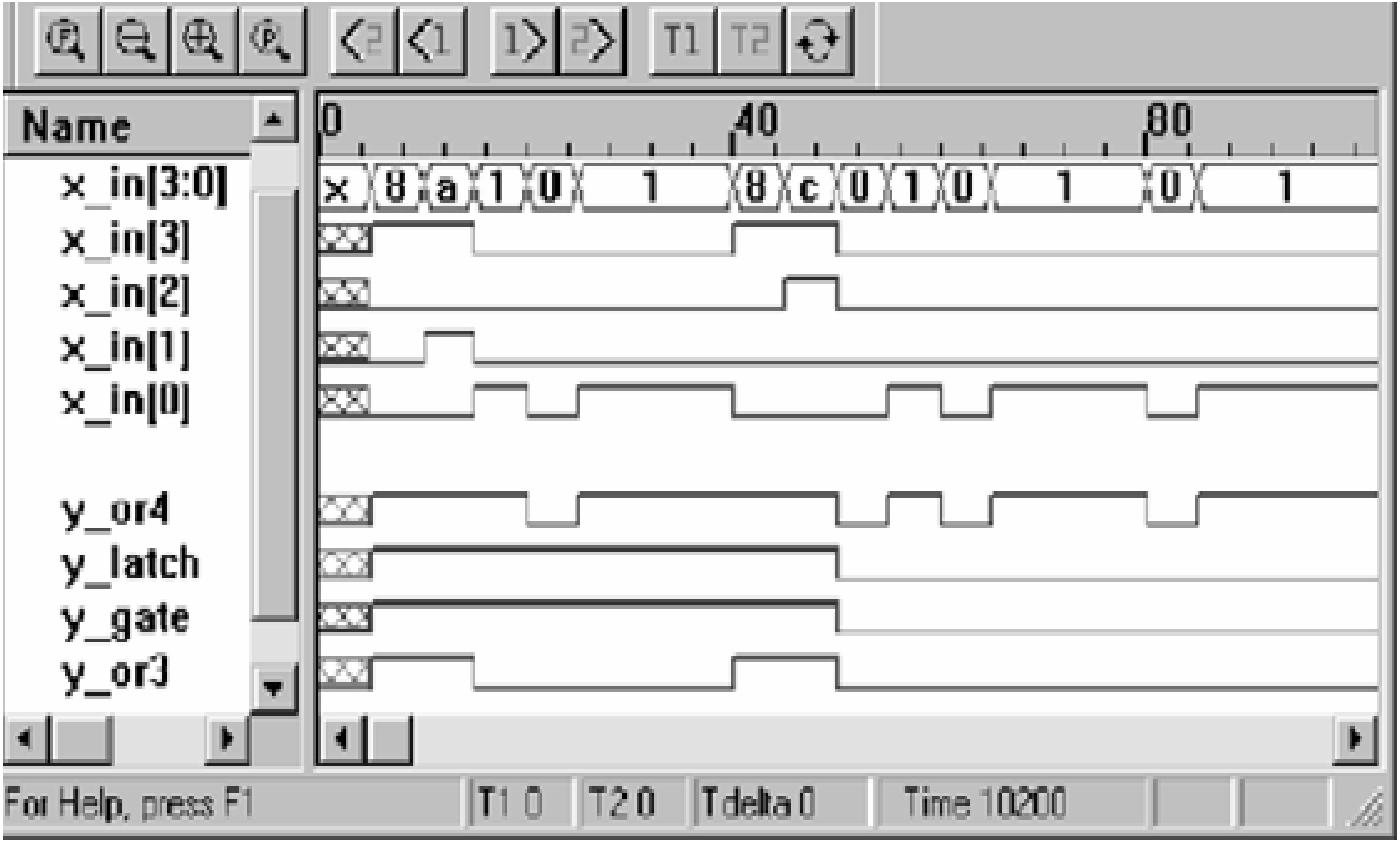
```
module or4_behav_latch (y, x_in);
   parameter word_length=4;
   output y;
   input [word_length-1:0] x_in;
   reg y;
   integer k;
   always @ (x_in[3:1])
     begin: check_for_1
       y=0;
       for (k=0;k<=word_length-1;k=k+1)
          if (x_in[k]==1)
            begin y=1;
            disable check_for_1;
            end
       end
endmodule
```

Example 6.16

```
module or4_behav_latch (y, x_in);
    parameter word_length=4;
    output y;
    input [word_length-1:0] x_in;
    reg y;
    integer k;
    always @ (x_in[3:1])
      begin: check_for_1
        y=0;
        for (k=0;k<=word_length-1;k=k+1)
            if (x_in[k]==1)
              begin y=1;
              disable check_for_1;
              end
       end
endmodule
```



Simulation result



To avoid latches

- All of the variables that are assigned value by the behavior must be assigned value under all events that affect the right-hand side expressions of the assignments that implement the logic
- **Failure to do so will produce a design with unwanted latches**

Remember

- If the output of **combinational logic** is not completely specified for all cases of the inputs, then a latch will be inferred, **a same signal will not appear on both sides of a level-sensitive cyclic behavior**

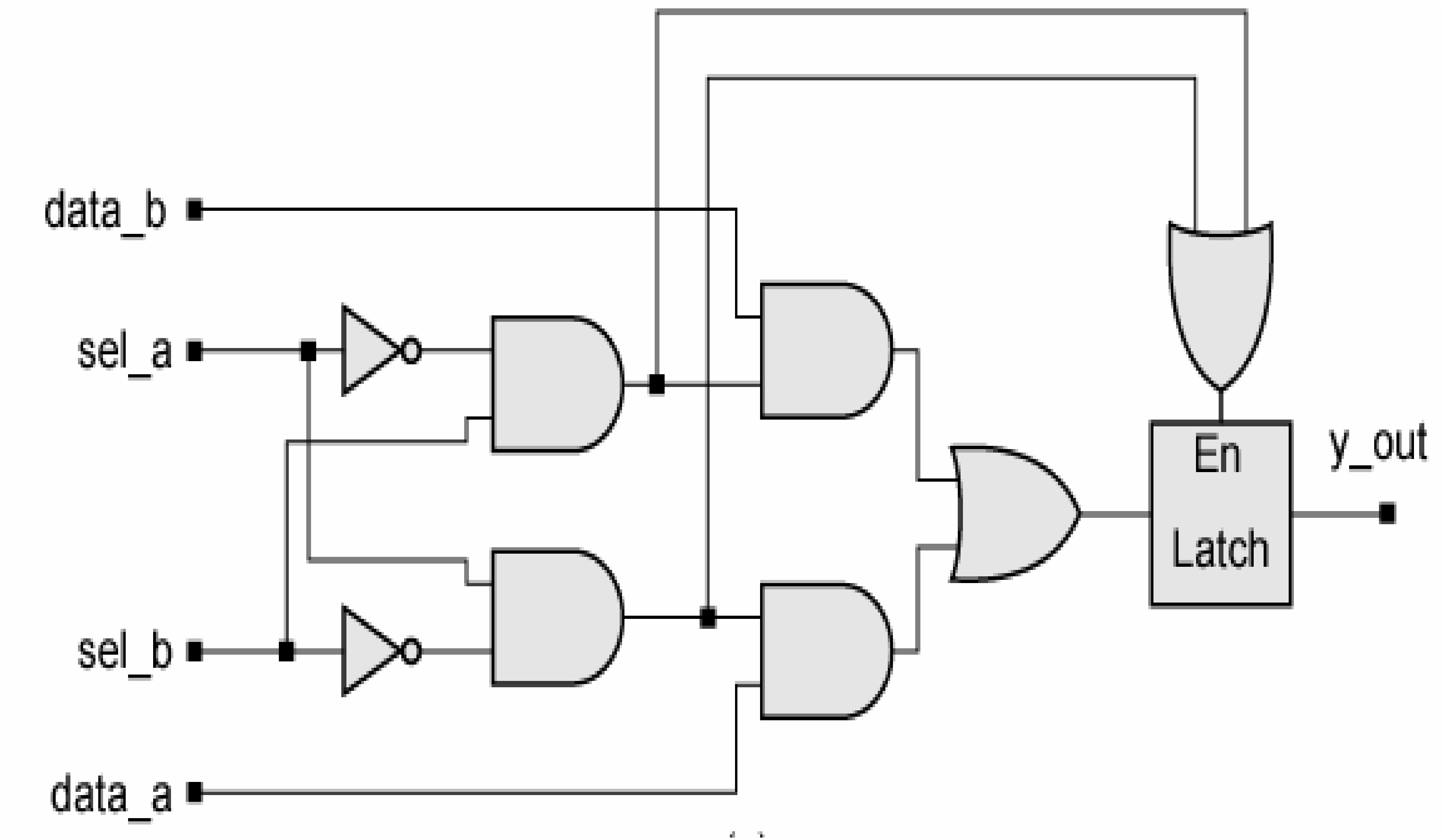
Synthesis tips

- A Verilog description of **combinational logic** must assign value to the outputs for all possible values of the inputs
- **When *case* or *if* statements are incompletely specified ,it may lead to synthesis of unwanted latches in the design**

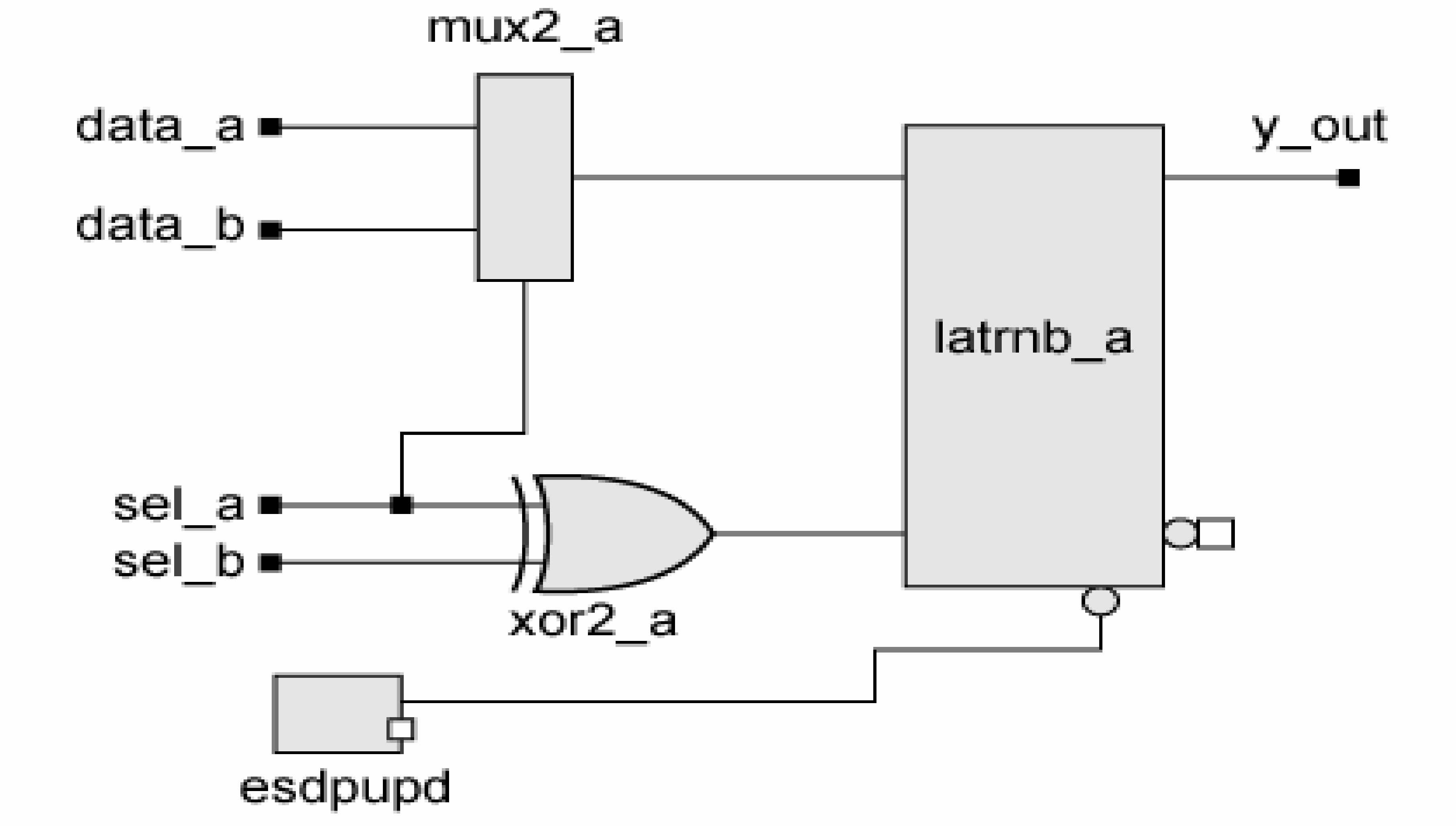
Example 6.17

```
module mux_latch
        (y_out,sel_a,sel_b,data_a,data_b);
  output y_out;
  input sel_a,sel_b,data_a,data_b;
  reg y_out;
  always @ (sel_a or sel_b or data_a or data_b)
    case ({sel_a,sel_b})
        2’b10: y_out=data_a;
        2’b01: y_out=data_b;
    endcase
endmodule
```

Synthesized circuit using generic parts



Synthesized circuit using parts from a cell library



6.3.2 Intentional synthesis of latches

- A synthesis tool infers the need for a latch when a register variable in a level-sensitive behavior is assigned value in some threads of activity, but not in others
- (e.g., an incomplete if statement in a behavior)

if & case statement

- **If a case statement has a default assignment with feedback ,** the synthesis tool will form a mux structure with feedback
- **If an if statement in a level-sensitive behavior assigns a variable to itself,** the result will be a mux structure with feedback
- **If the behavior is edge-sensitive,** incomplete case and conditional statements synthesize register variables to flip-flops
- **If the statements are completed with feedback**, the result is a register whose output is fed back through a mux at its datapath

Implemented of (? ... : ...)

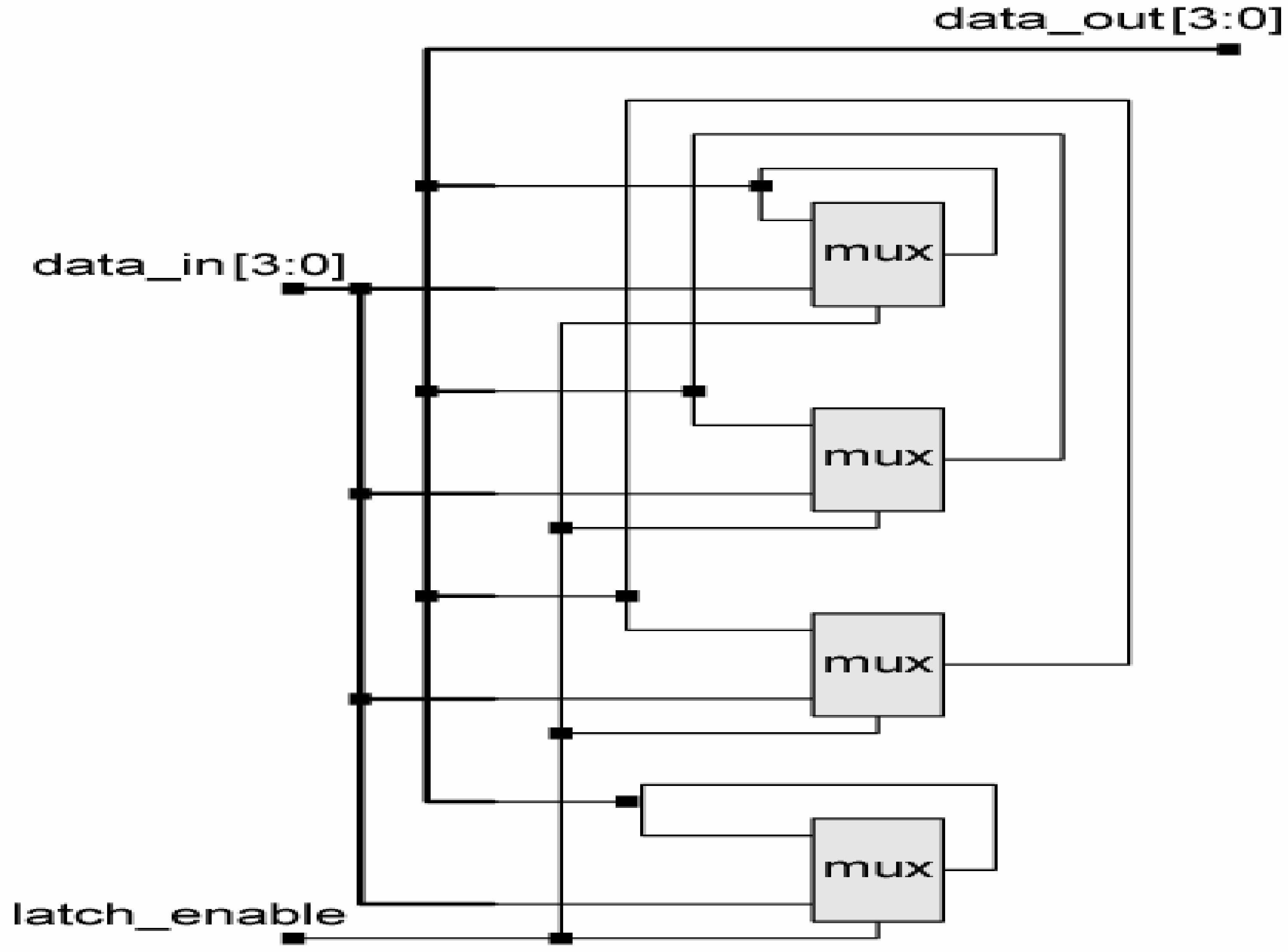
- when the conditional operator (**?** ... **:** ...) is implemented with feedback, but the actual implementation chosen by a synthesis tool depends on the context
- 1. If the conditional operator is used **in a continuous assignment**, the result will be a mux with feedback
- 2. If the conditional operator is used **in an edge-sensitive cyclic behavior**, the result will be a register with a gated data path in a feedback configuration with the output of the register
- 3. If it is used **in a level-sensitive cyclic behavior,** the results will be a hardware latch

Example 6.18

```
module latch_if (data_out,data_in,latch_enable);
    output [3:0] data_out;
    input [3:0] data_in;
    input  latch_enable;
    reg [3:0] data_out;
    always @ (latch_enable or data_in)
      if (latch_enable) data_out=data_in;
           else data_out=data_out;
endmodule
```

If an *if statement* in a level-sensitive behavior assigns a variable to itself, the result will be a mux structure with feedback

Synthesized circuit



Synthesis tips

- An ***if statement*** in a level-sensitive behavior will synthesize to a latch,

  if the statement assigns value to a register variable in some, but not all, branches

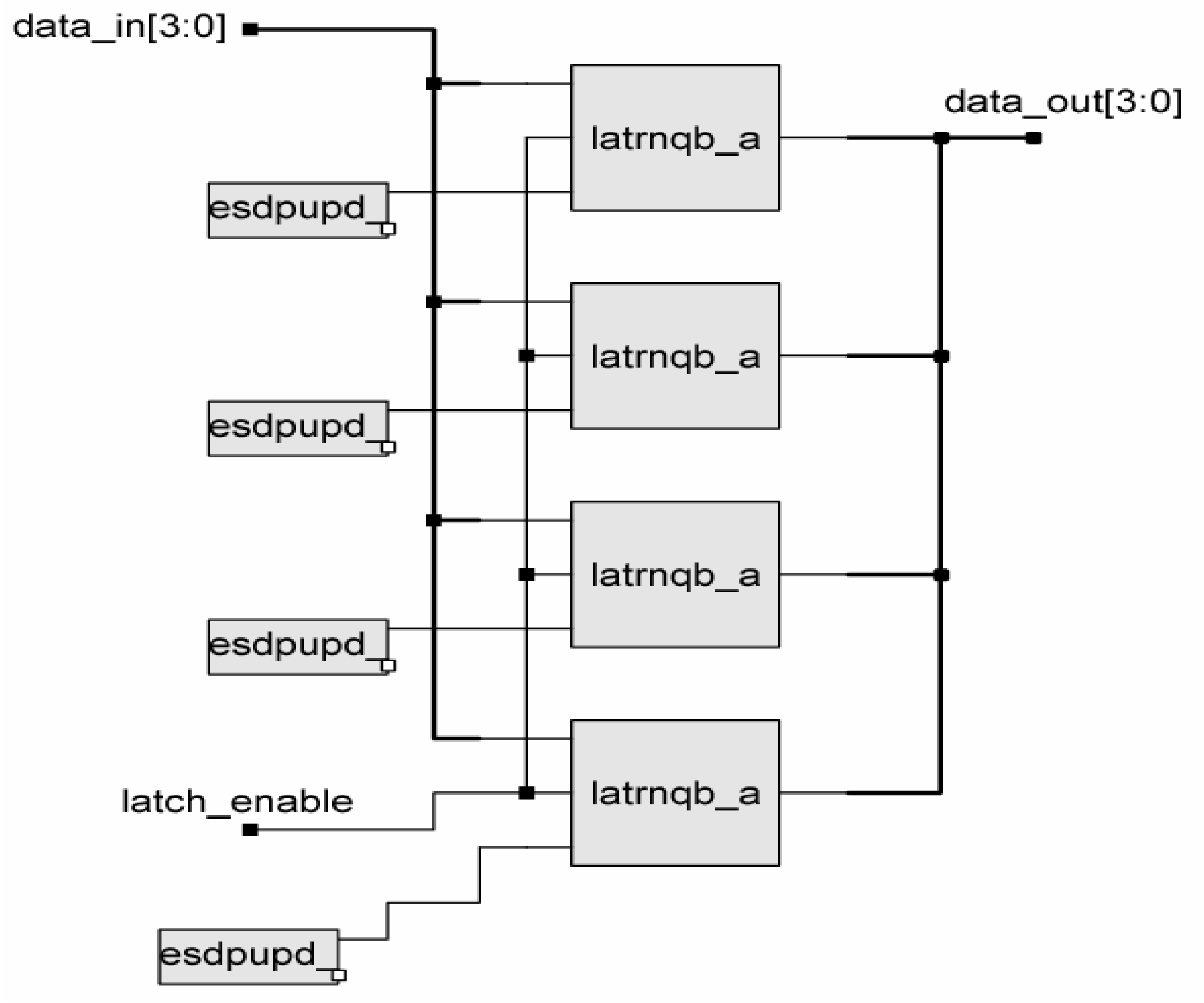
Example 6.19

```
module latch_if (data_out,data_in,latch_enable);
    output [3:0] data_out;
    input [3:0] data_in;
    input  latch_enable;
    reg [3:0] data_out;
    always @ (latch_enable or data_in)
      if (latch_enable) data_out=data_in;
endmodule
```

If *an if statement* is used in a level-sensitive cyclic behavior (if the statement assigns value to a register variable in some, but not all)

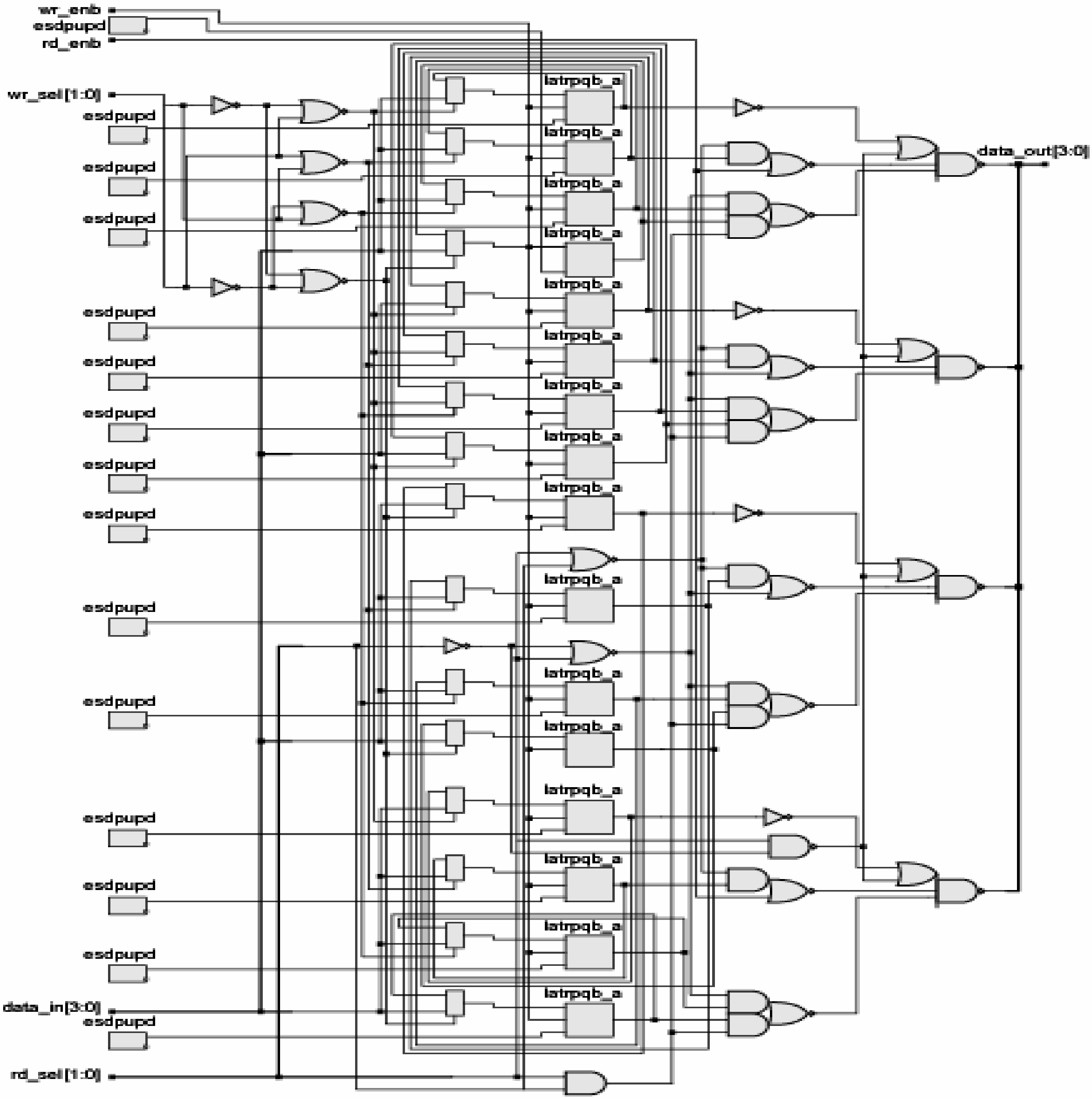
, the results will be a hardware latch

Synthesized circuit



Example 6.20
an array of four 4-bit words

```
module sn54170
    (data_out,data_in,wr_sel,rd_sel,wr_enb,rd_enb);
    output [3:0] data_out;
    input wr_enb,rd_enb;
    input [1:0] wr_sel,rd_sel;
    input [3:0] data_in;
    reg [3:0] latched_data;
    always @ (wr_enb or wr_sel or data_in)
      begin
         if (!wr_enb)  latched_data[wr_sel]=data_in;
      end
    assign data_out = (rd_enb) ? 4’b1111 : latched_data[rd_sel];
endmodule
```


wr_enb
esdpupd
rd_enb
wr_sel[1:0]
esdpupd
esdpupd
esdpupd
latrpqb_a
data_out[3:0]
data_in[3:0]
rd_sel[1:0]

6.4 Synthesis of three-state devices and bus interfaces

- Three-state device allow buses to be shared among multiple devices
- The preferred style for inferring a three-state bus driver **uses a continuous assignment statement**

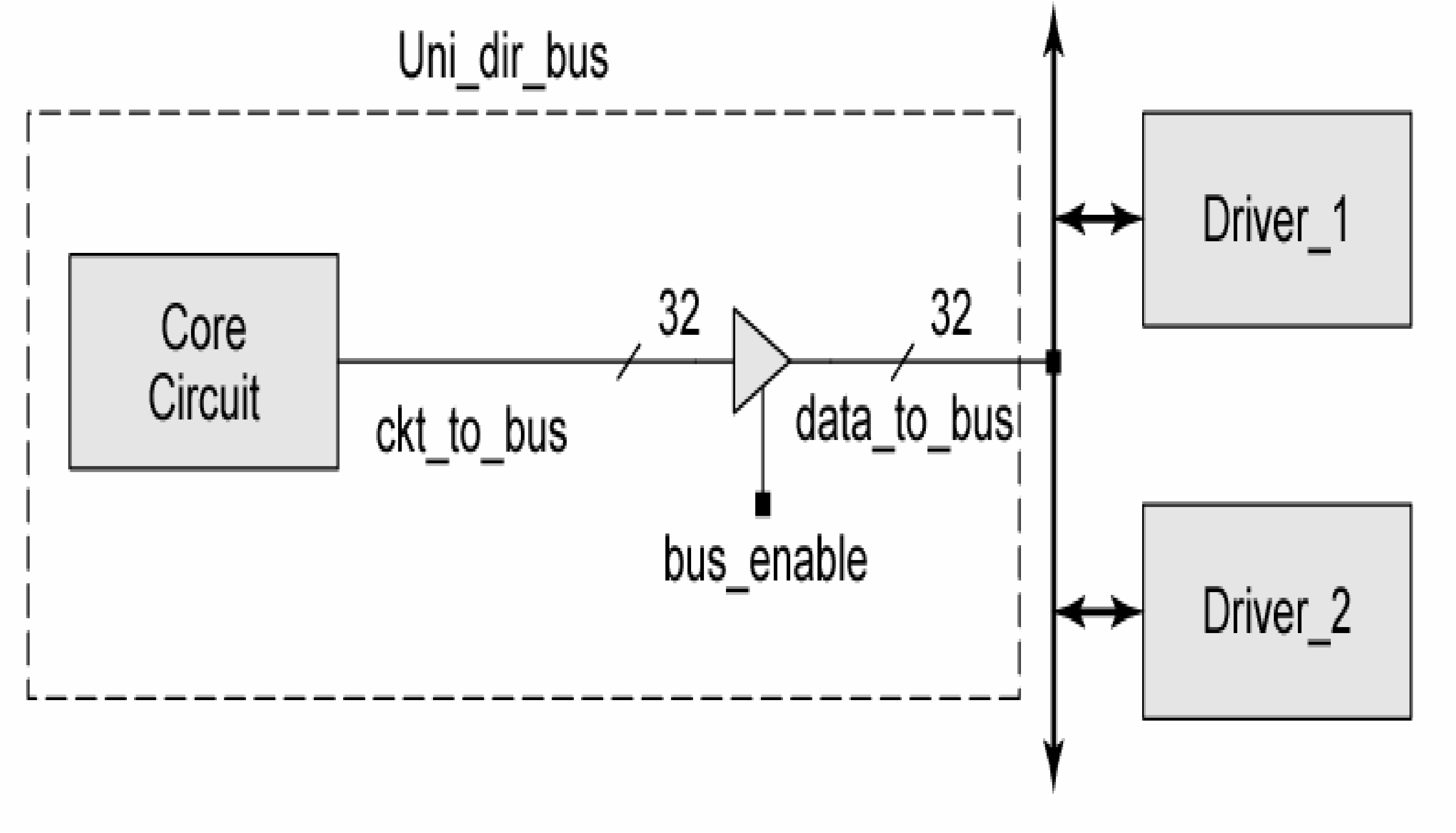
  that has one branch set to a three-state logic value (*z*)

Ex 6.21 a three-state unidirectional interface to a bus

```
module Uni_dir_bus (data_to_bus,bus_unable);
   input bus_enable;
   output [31:0] data_to_bus;
   reg [31:0] ckt_to_bus;
   assign
   data_to_bus=(bus_enabled) ?ckt_to_bus:32’bz;
  // Description of core circuit goes here to drive ckt_to_bus
endmodule
```

- a continuous assignment statement that has one branch set to a three-state logic value (***z***)

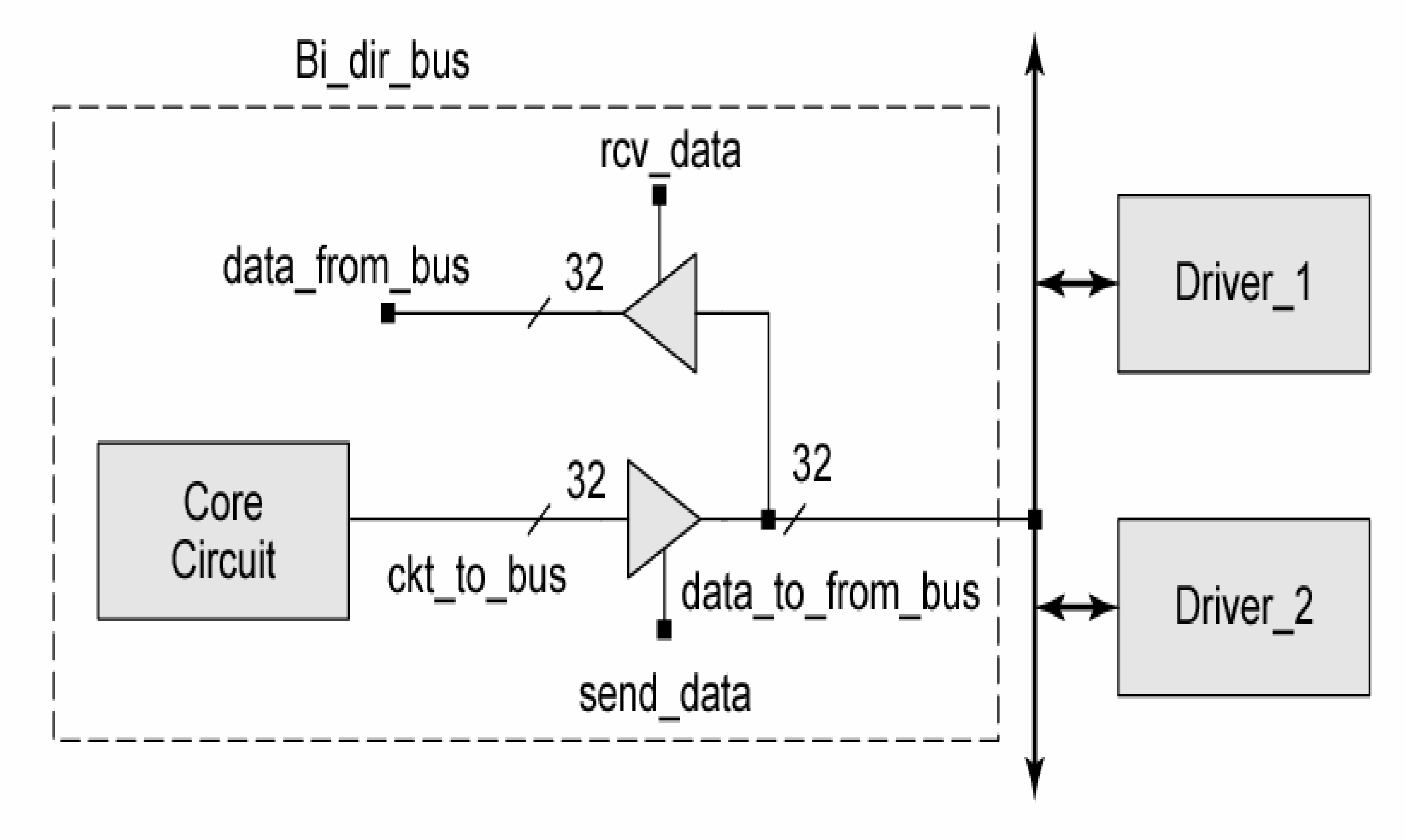
A three-state unidirectional interface to a bus



Ex 6.22 bidirectional interface to a bidirectional bus

```
module Bi_dir_bus (data_to_from_bus,send_data,rcv_data);
   inout [31:0] data_to_from_bus;
   input  send_data,rcv_data;
   wire [31:0] ckt_to_bus;
   wire [31:0] data_to_from_bus,data_from_bus;
   assign data_from_bus=(rcv_data)?data_to_from_bus:32’bz;
   assign data_to_from_bus=
             (send_data)?ckt_to_bus:data_to_from_bus;
// Behavior using data_from_bus and generating
// ckt_to_bus goes here
endmodule
```

bidirectional interface to a bidirectional bus



6.5 Synthesis of sequential logic with flip-flops

- Flip-flops are synthesized only from edge-sensitive cyclic behaviors
- Not every register variable that is assigned value in an edge-sensitive behavior synthesizes to a flip-flop

A register variable in an edge-sensitive behavior

- **Synthesized as a flip-flop:**

① If it is referenced outside the scope of the behavior

② If it is referenced within the behavior before it is assigned value

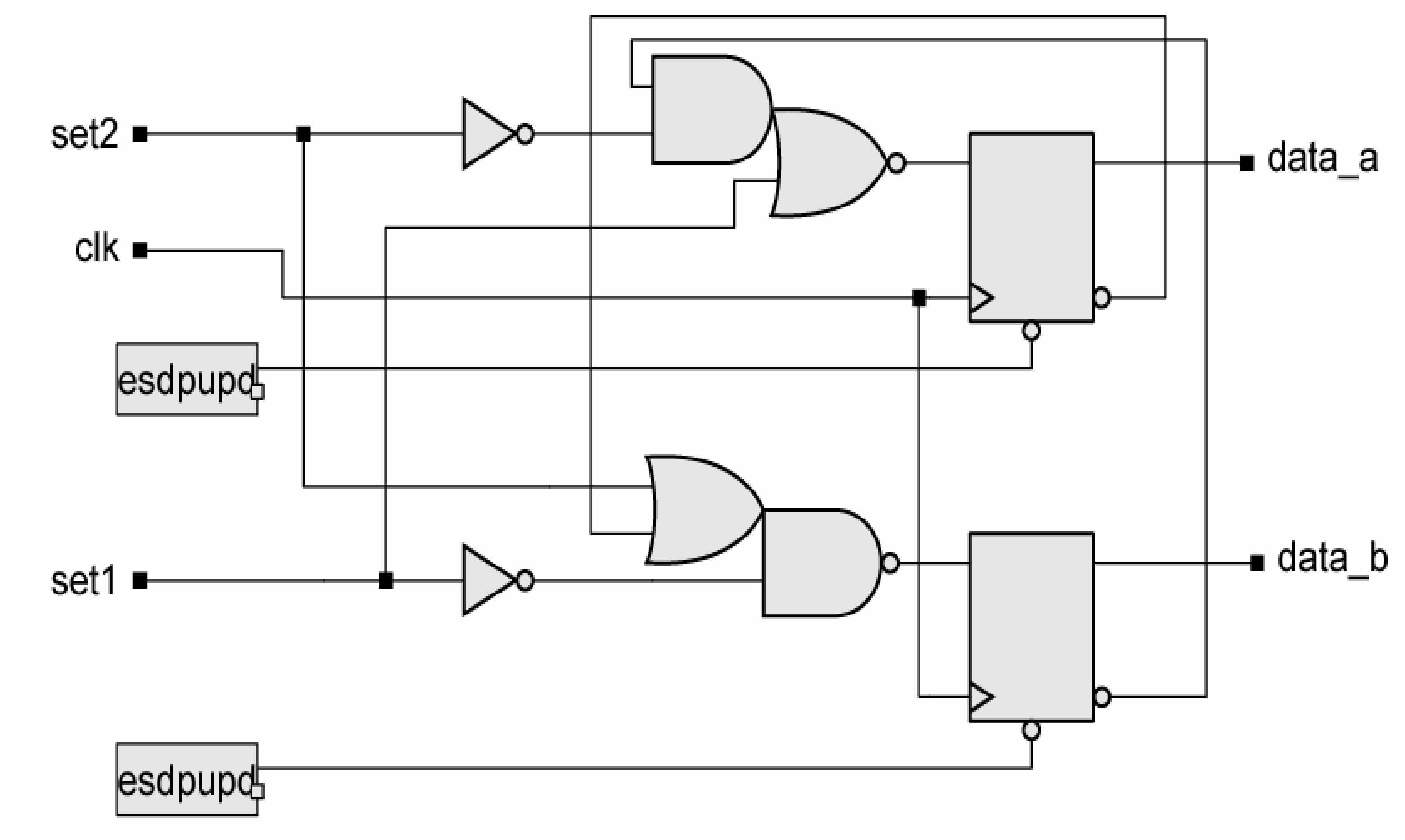
③ If it is assigned value in only some of the branches of the activity within the behavior

- **All of these situations imply the need for memory, or residual value**

Ex 6.23 A synchronous data swapping mechanism

```
module swap_synch (data_a,data_b,set1,set2,clk);
    output data_a,data_b;
    input clk,set1,set2,swap;
    reg data_a,data_b;
    always @ (posedge clk)
      begin
           if (set1) begin data_a<=1;data_b<=0; end
             else  if (set2) begin data_a<=0;data_b<=1; end
                     else begin
                            data_b<=data_a;  data_a<=data_b;
                             end
      end
endmodule
```

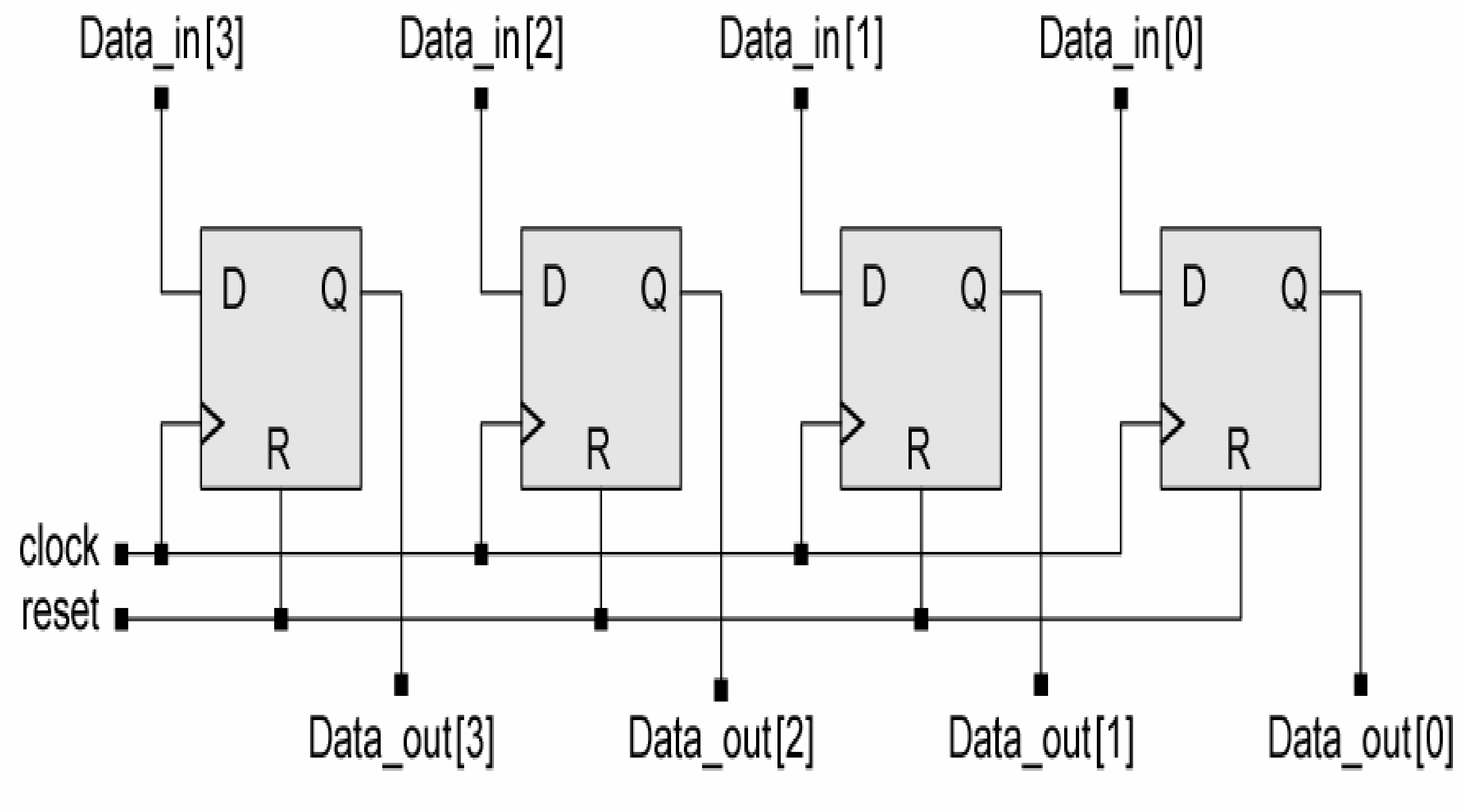
Fig 6-29 synthesis circuit



Ex 6.24 A 4-bit parallel-load data register

```
module D_reg4_a  (Data_out, clock, reset, Data_in);
   output            [3: 0] Data_out;
   input             [3: 0] Data_in;
   input                    clock, reset;
    reg              [3: 0] Data_out;
   always @  (posedge clock or posedge reset)
     begin
       if (reset == 1'b1) Data_out <= 4'b0;
         else  Data_out <= Data_in;
     end
endmodule
```

Synthesis circuit of a 4-bit parallel-load data register



Example 6.25

```
module empty_circuit (D_in,clk);
  input D_in;
  input clk;
  reg D_out;
  always @ (posedge clk) begin
    D_out<=D_in;
  end
endmodule
```

Example 6.25

```
module empty_circuit (D_in,clk);
  input D_in;
  input clk;
  reg D_out;
  always @ (posedge clk) begin
    D_out<=D_in;
  end
endmodule
```

The synthesis tool will eliminate D_out.

D_out is not referenced outside the scope of the behavior

Synthesis tips

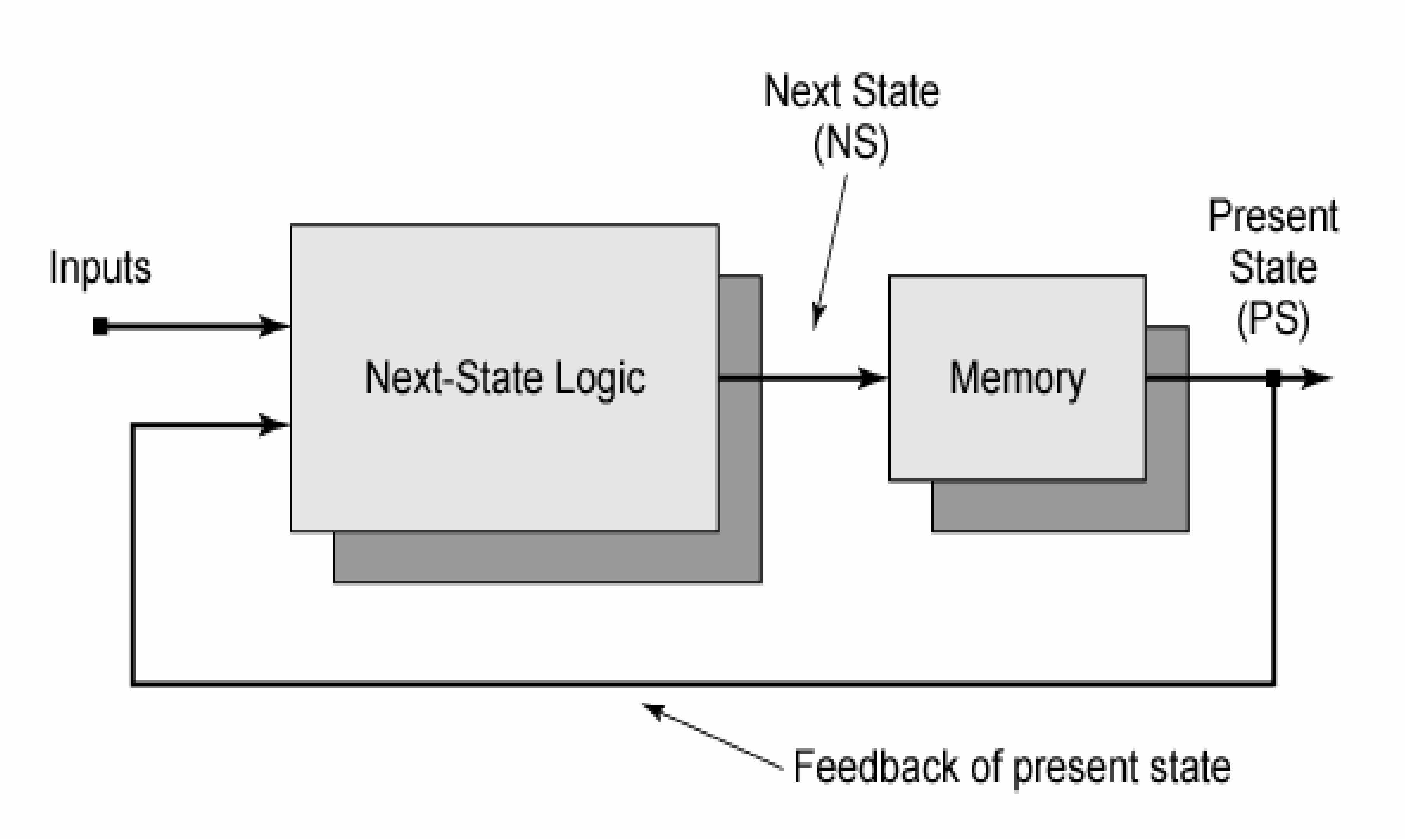
- A variable that is assigned value by a cyclic behavior before it is referenced within the behavior, **but is not referenced outside the behavior**, **will be eliminated by the synthesis process**
- **A variable that is assigned value by an edge-sensitive behavior and is referenced outside the behavior will be synthesized as the output of a flip-flop**

6.6 Synthesis of explicit state machines

- Explicit machine have **an explicitly declared state register and explicit logic** that governs the evolution of the state under the influence of the inputs
- Be described by two behaviors:

1. **an edge-sensitive behavior that synchronizes the evolution of the state**
2. **a level sensitive behavior that describe the next state and output logic**

State Machines



Finite State Machines

- **Moore State Machines:**

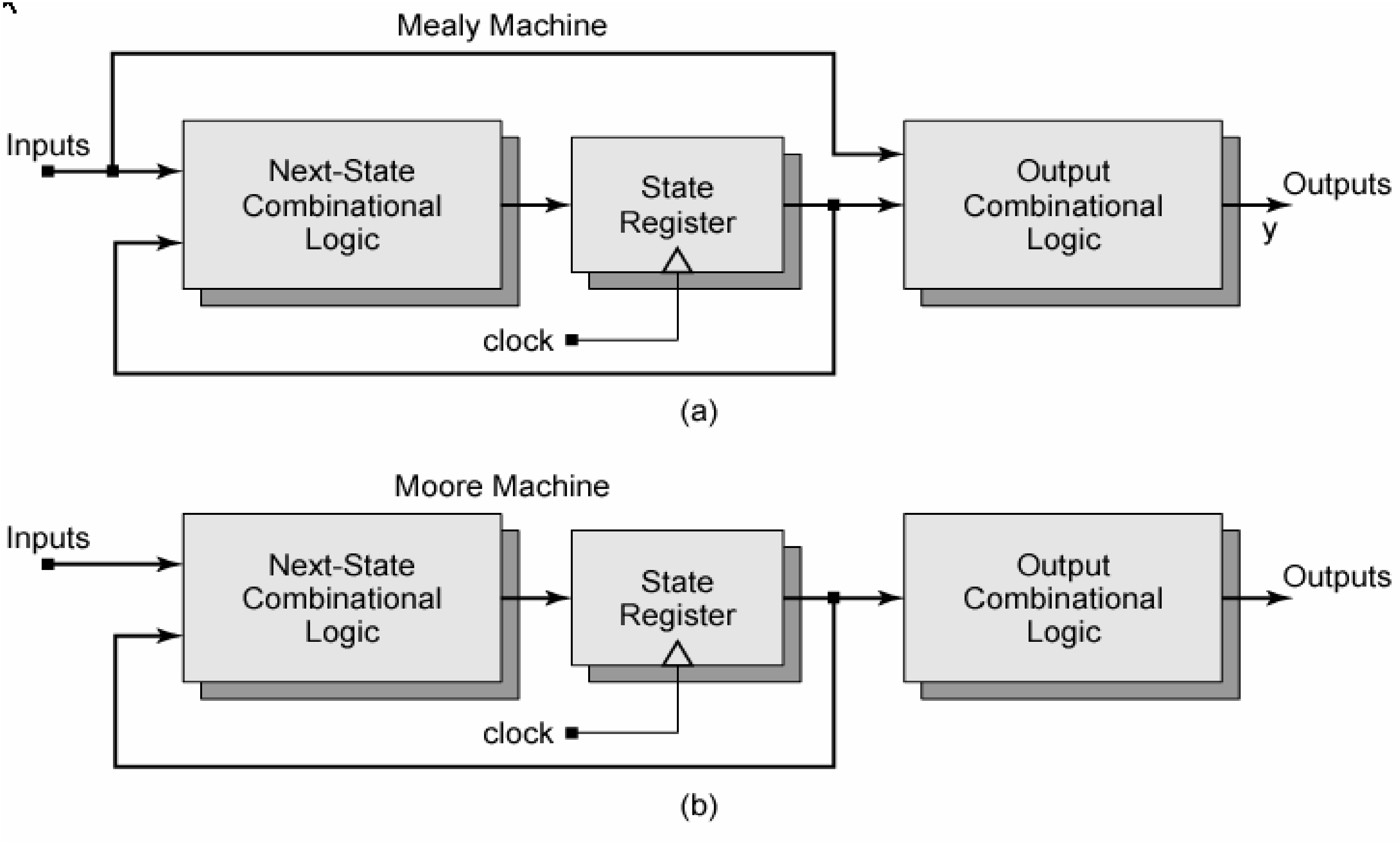
  Outputs Are a Function of the States Only

  Isolates the Outputs from the Inputs

- **Mealy State Machines:**

  Outputs are a Function of the Inputs as well as the States.

Mealy & Moore State Machine


Mealy Machine
Inputs
Next-State Combinational Logic
State Register
clock
Output Combinational Logic
Outputs
y
(a)
Moore Machine
Inputs
Next-State Combinational Logic
State Register
clock
Output Combinational Logic
Outputs
(b)

Synthesis Tip

- **Use two cyclic behaviors to describe an explicit state machine:**
- **a level-sensitive behavior to describe the combinational logic for the next state and outputs:**

  Use the procedural assignment operator(=)
- **an edge-sensitive behavior to synchronize the state transitions:**

  Use the nonblocking assignment operator (<=)

6.6.1 Synthesis of a BCD-to-excess-3 code converter

✓ An Excess-3 code word is obtained by adding 3 to the decimal value of the BCD word and taking the binary equivalent of the result

Input BCD A B C D	Output Excess-3 WXYZ
0 0 0 0	0 0 1 1
0 0 0 1	0 1 0 0
0 0 1 0	0 1 0 1
0 0 1 1	0 1 1 0
0 1 0 0	0 1 1 1
0 1 0 1	1 0 0 0
0 1 1 0	1 0 0 1
0 1 1 1	1 0 1 0
1 0 0 0	1 0 1 1
1 0 0 1	1 0 1 1

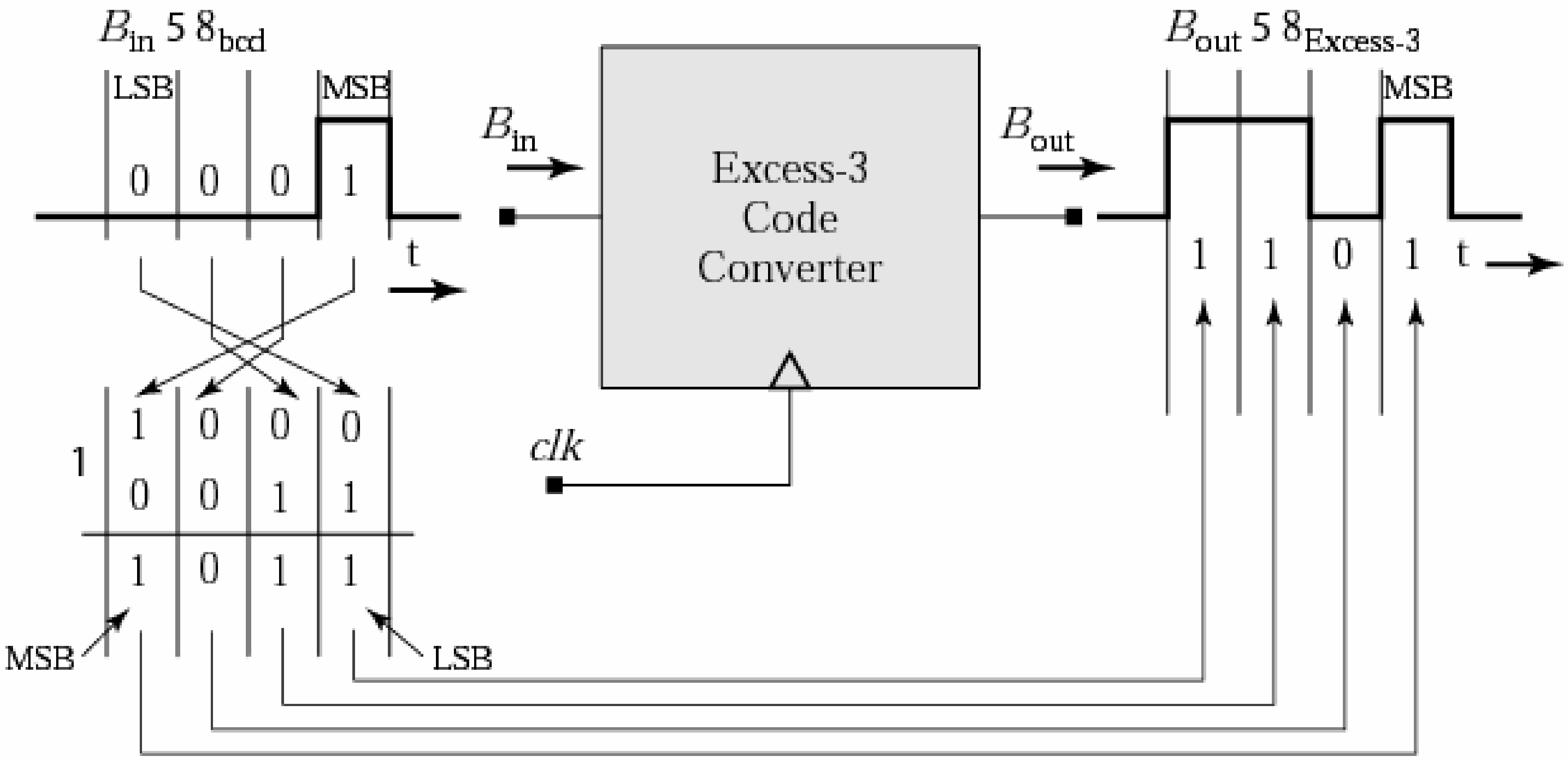
Formulation: BCD to Excess 3

Conversion of 4-bit codes can be most easily formulated by a truth table

- Variables
  - BCD:
    A,B,C,D
- Variables
  - Excess-3
    W,X,Y,Z
- Don’t Cares
  - BCD
- 1010 to 1111

Input BCD	Output Excess-3
A B C D	WXYZ
0 0 0 0	0 0 1 1
0 0 0 1	0 1 0 0
0 0 1 0	0 1 0 1
0 0 1 1	0 1 1 0
0 1 0 0	0 1 1 1
0 1 0 1	1 0 0 0
0 1 1 0	1 0 0 1
0 1 1 1	1 0 1 0
1 0 0 0	1 0 1 1
1 0 0 1	1 0 1 1

A BCD-to-Excess-3 Code Converter



Basic steps

(1) forming a state-transition graph

(2) defining a state table

(3) choosing a state assignment

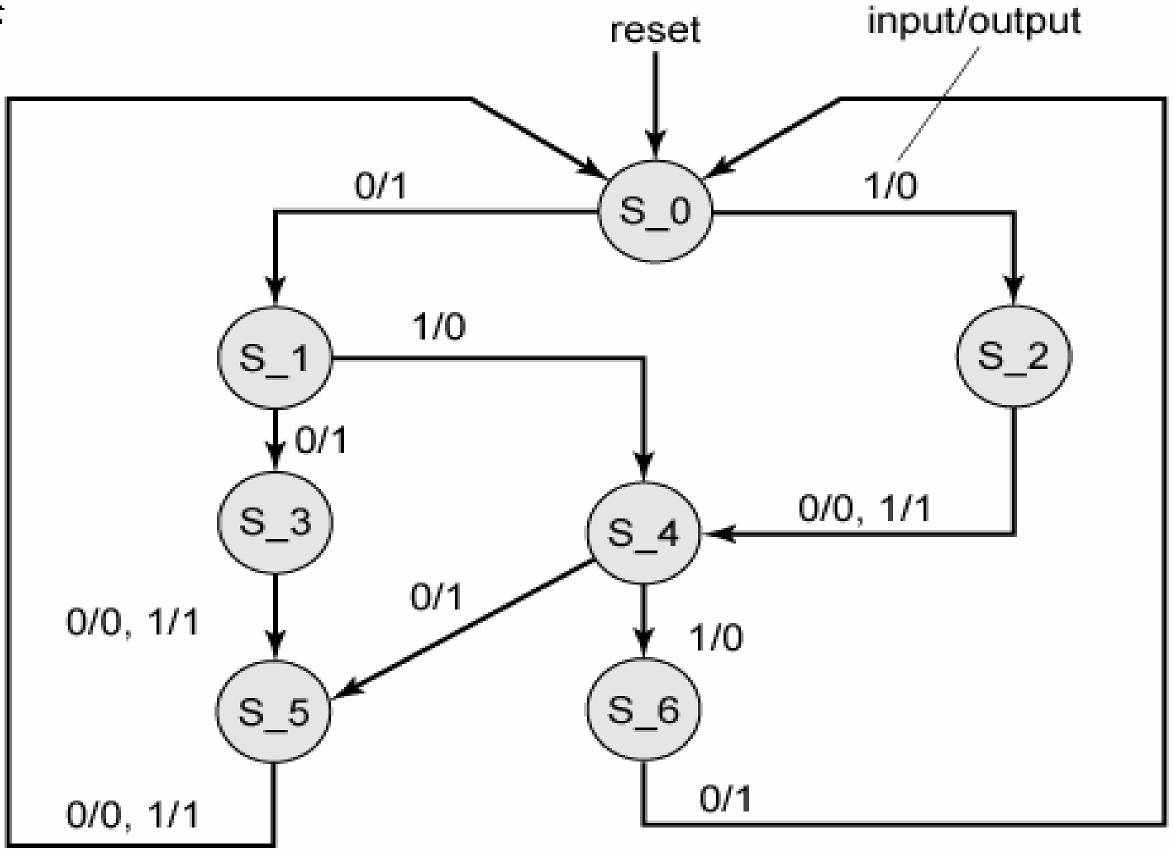
(4) encoding the next state and output table

(5) developing and minimizing the Karnaugh maps for the encoded state bits and output

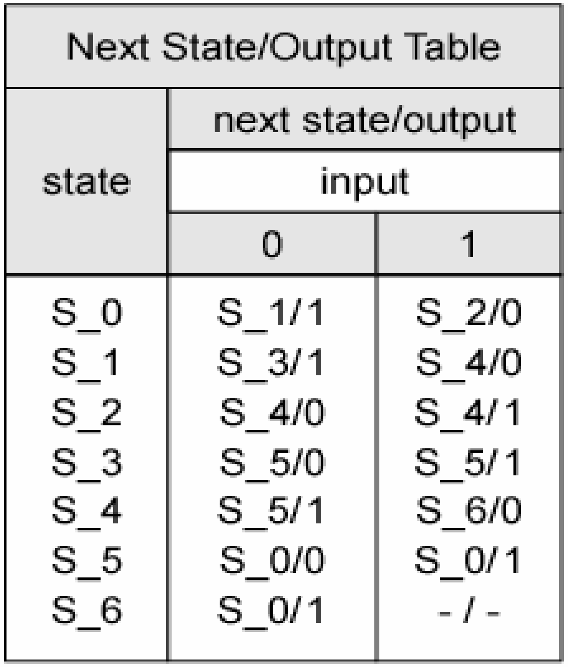
(6) creating the circuit's implementation (schematic).

Mealy type FSM:
(a) state transition
(b) the machine's state table

Input BCD A B C D	Output Excess-3 WXYZ
0 0 0 0	0 0 1 1
0 0 0 1	0 1 0 0
0 0 1 0	0 1 0 1
0 0 1 1	0 1 1 0
0 1 0 0	0 1 1 1
0 1 0 1	1 0 0 0
0 1 1 0	1 0 0 1
0 1 1 1	1 0 1 0
1 0 0 0	1 0 1 1
1 0 0 1	1 0 1 1



(a)

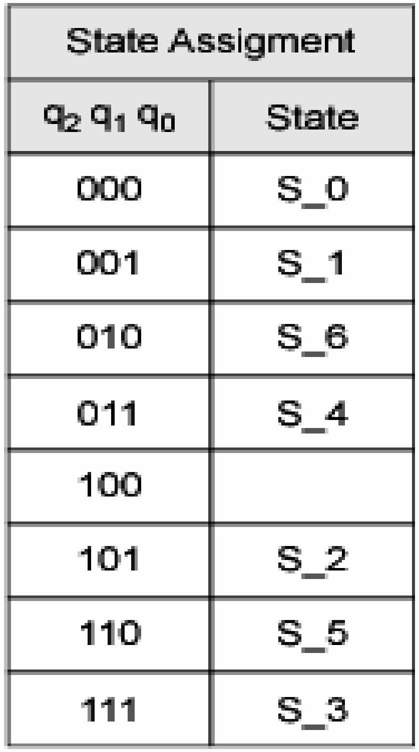


Next State/Output Table

state	next state/output	
	input	
	0	1
S_0	S_1/1	S_2/0
S_1	S_3/1	S_4/0
S_2	S_4/0	S_4/1
S_3	S_5/0	S_5/1
S_4	S_5/1	S_6/0
S_5	S_0/0	S_0/1
S_6	S_0/1	- / -

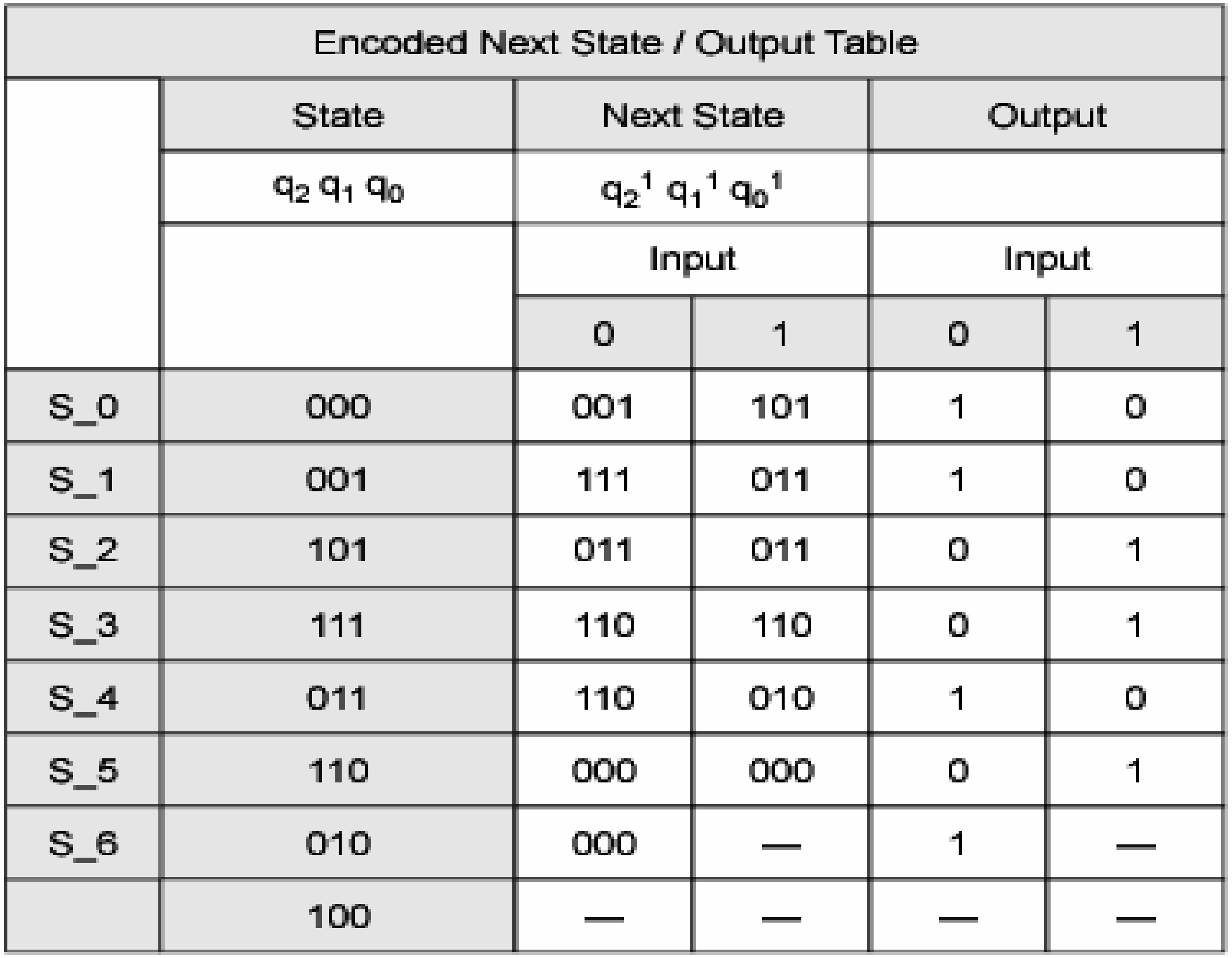
(b)

BCD-to-Excess-3 Code Converter
(a) state assignment
(b) endcoded next state & out table



State Assigment	
$q_2\, q_1\, q_0$	State
000	S_0
001	S_1
010	S_6
011	S_4
100	
101	S_2
110	S_5
111	S_3

(a)



Encoded Next State / Output Table					
	State	Next State		Output	
	$q_2\, q_1\, q_0$	$q_2^1\, q_1^1\, q_0^1$			
		Input		Input	
		0	1	0	1
S_0	000	001	101	1	0
S_1	001	111	011	1	0
S_2	101	011	011	0	1
S_3	111	110	110	0	1
S_4	011	110	010	1	0
S_5	110	000	000	0	1
S_6	010	000	—	1	—
	100	—	—	—	—

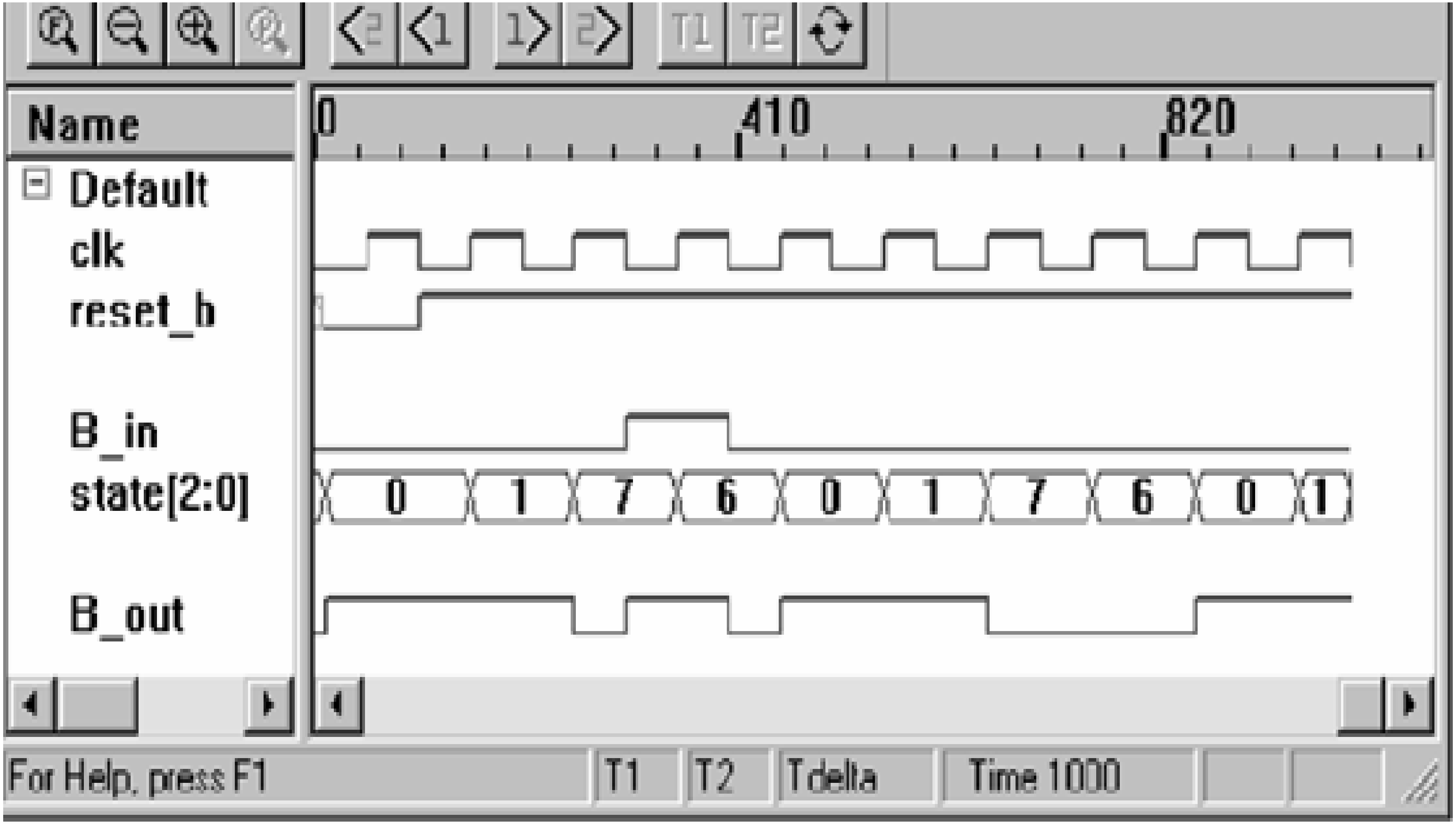
(b)

A BCD-to-Excess-3 Code Converter

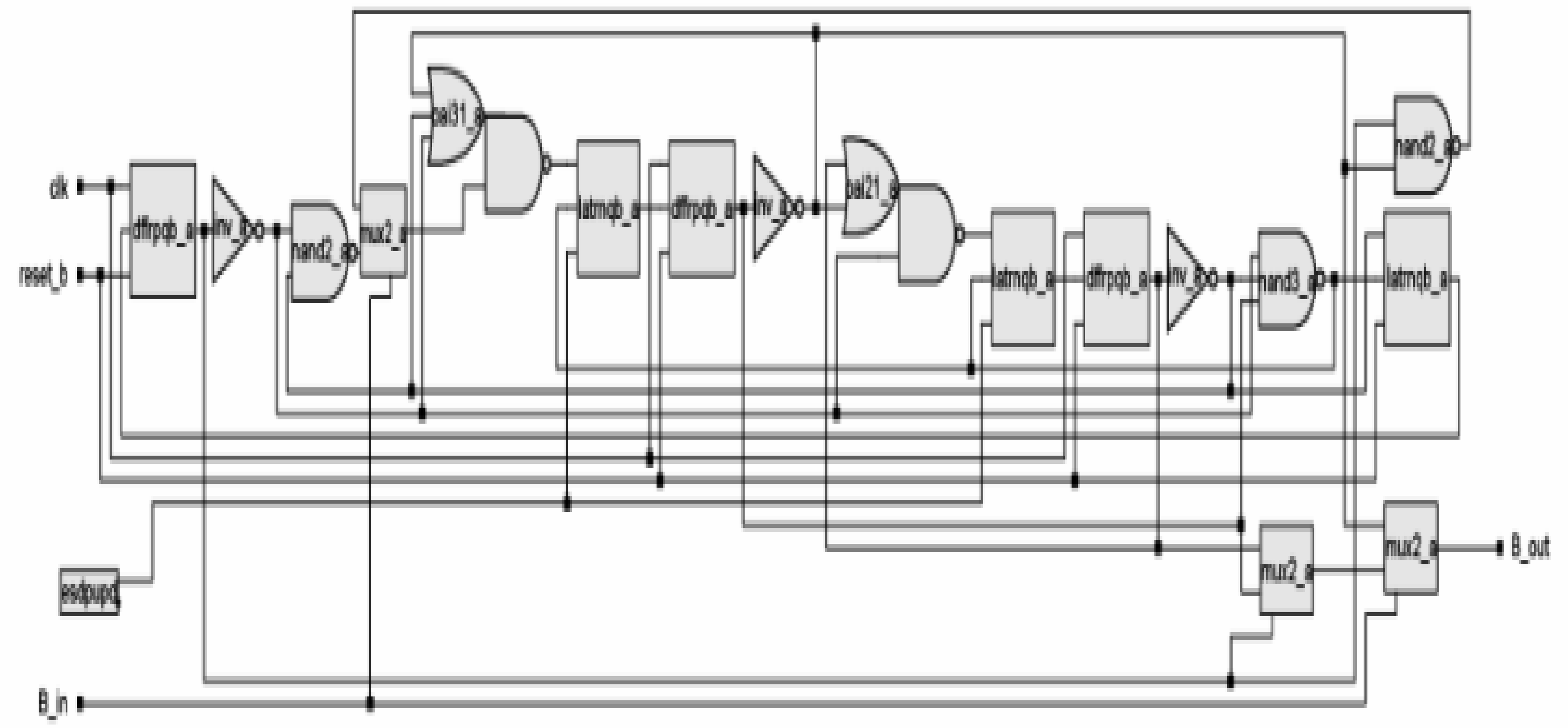
```
module BCD_to_Excess_3b (B_out, B_in,clk,reset_b);
    output B_out;
    input B_in,clk,reset_b;
    parameter   S_0=3’b000,      // state assignment
                S_1=3’b001,
                S_2=3’b101,
                S_3=3’b111,
                S_4=3’b011,
                S_5=3’b110,
                S_6=3’b010,
                dont_care_state=3’bx,
                dont_care_out=1’bx;
    reg [2:0]   state,next_state;
    reg         B_out;
    always @ (posedge clk or negedge reset_b)
      if (reset_b==0) stae<=S_0; else state<=next_state;
```

```
always @ (state or B_in) begin  // determine next state
    B_out=0;
   case (state)
     S_0: if (B_in==0) begin next_state=S_1;B_out=1; end
          else if (B_in==1) begin next_state=S_2; end
     S_1: if (B_in==0) begin next_state=S_3;B_out=1; end
          else if (B_in==1) begin next_state=S_4; end
     S_2: begin next_state=S_4; B_out=B_in; end
     S_3: begin next_state=S_5; B_out=B_in; end
     S_4: if (B_in==0) begin next_state=S_5;B_out=1; end
          else if (B_in==1) begin next_state=S_6; end
     S_5: begin next_state=S_0; B_out=B_in; end
     S_6: begin next_state=S_0; B_out=1; end
    endcase
  end
endmodule
```

Simulation of BCD_to_Excess_3b



Circuit synthesized from BCD_to_Excess_3b

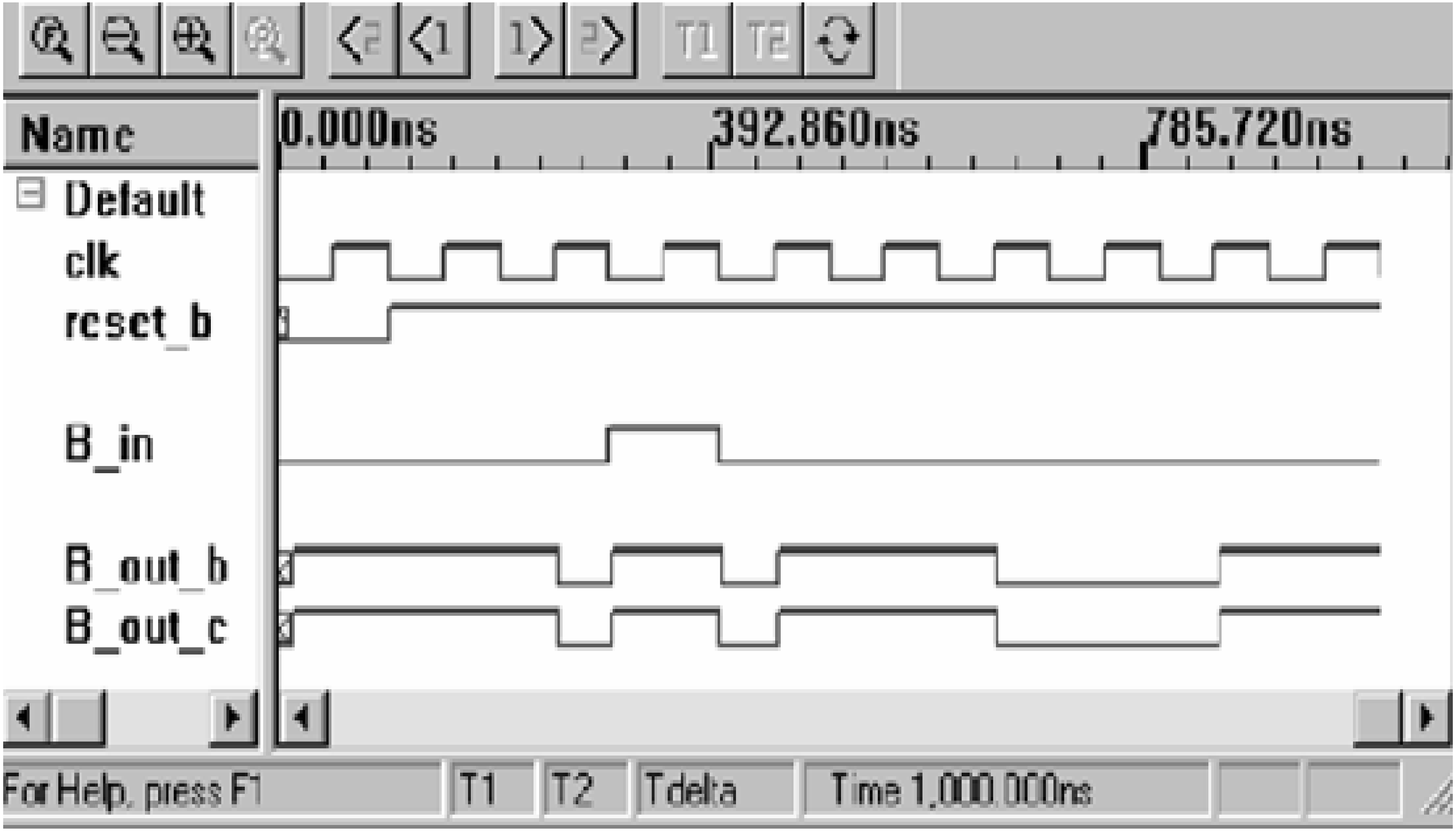


(a)

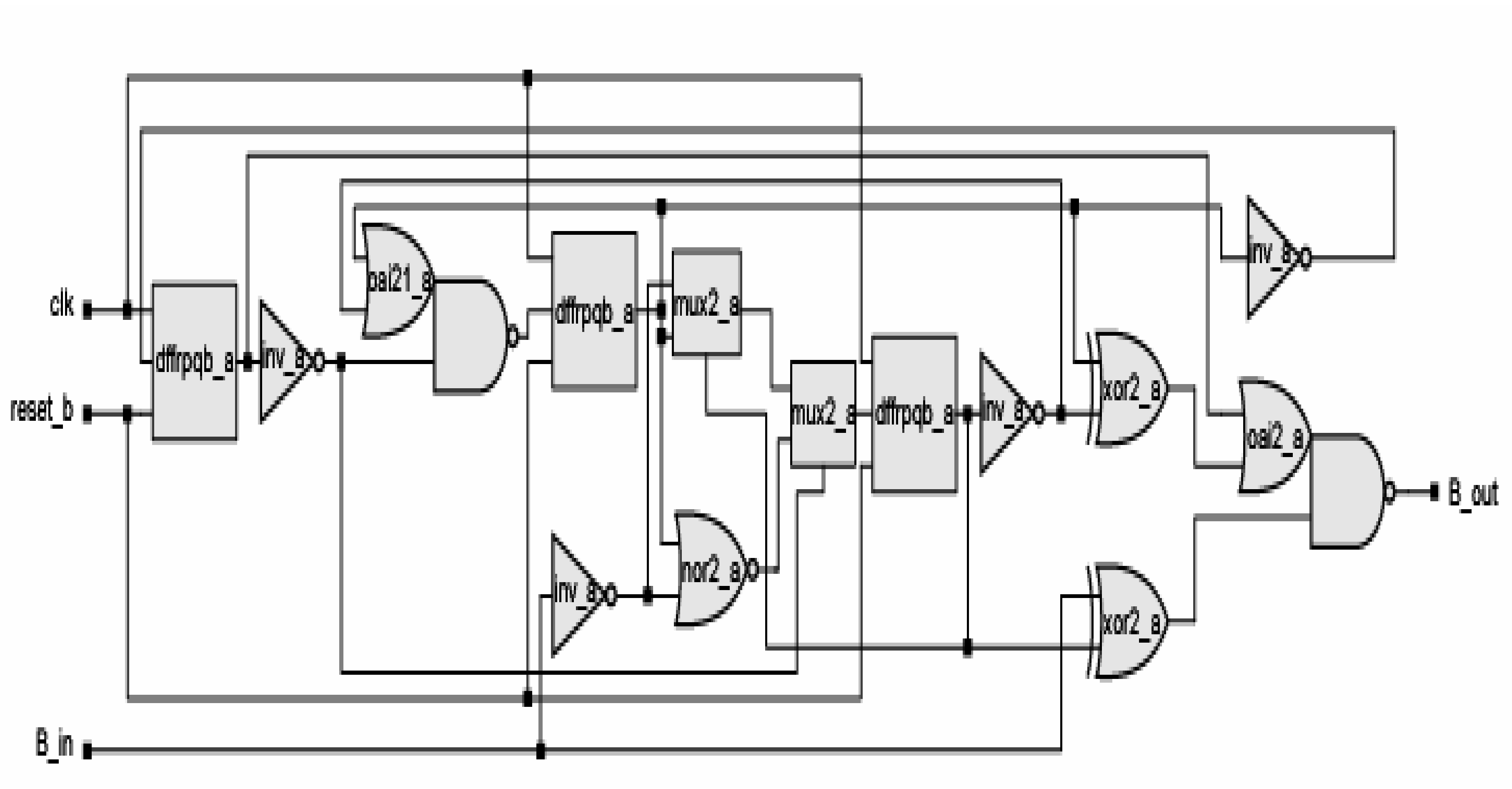
```
module BCD_to_Excess_3c (B_out, B_in,clk,reset_b);
   output B_out;
   input B_in,clk,reset_b;
   parameter   S_0=3'b000,      // state assignment
               S_1=3'b001,
               S_2=3'b101,
               S_3=3'b111,
               S_4=3'b011,
               S_5=3'b110,
               S_6=3'b010,
               dont_care_state=3'bx,
               dont_care_out=1'bx;
   reg [2:0]   state,next_state;
   reg         B_out;
   always @ (posedge clk or negedge reset_b)
     if (reset_b==0) stae<=S_0; else state<=next_state;
```

```
always @ (state or B_in) begin
     B_out=0;
     case (state)
        S_0: if (B_in==0) begin next_state=S_1;B_out=1; end
                else if (B_in==1) begin next_state=S_2; end
        S_1: if (B_in==0) begin next_state=S_3;B_out=1; end
                else if (B_in==1) begin next_state=S_4; end
        S_2: begin next_state=S_4; B_out=B_in; end
        S_3: begin next_state=S_5; B_out=B_in; end
        S_4: if (B_in==0) begin next_state=S_5;B_out=1; end
                else if (B_in==1) begin next_state=S_6; end
        S_5: begin next_state=S_0; B_out=B_in; end
        S_6: begin next_state=S_0; B_out=1; end
           default: beign next_state=dont_care_state;
          B_out=dont_care_out;
        end
      endcase
    end
endmodule
```

Simulation of BCD_to_Excess_3c



Circuit synthesized from BCD_to_Excess_3c



Synthesis Tip

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

6.6.2 Synthesis of a mealy-type NRZ-to-Manchester line code converter

- **A serial line code converter** converts a non-return-to-zero (NRZ) bit stream into a Manchester encoded bit stream
- Line codes are used in data transmission or storage systems **to reduce the effects of noise** in serial communications channels and/or **to reduce the width of channel datapaths**

Manchester encoding

(Self Synchronizing code)

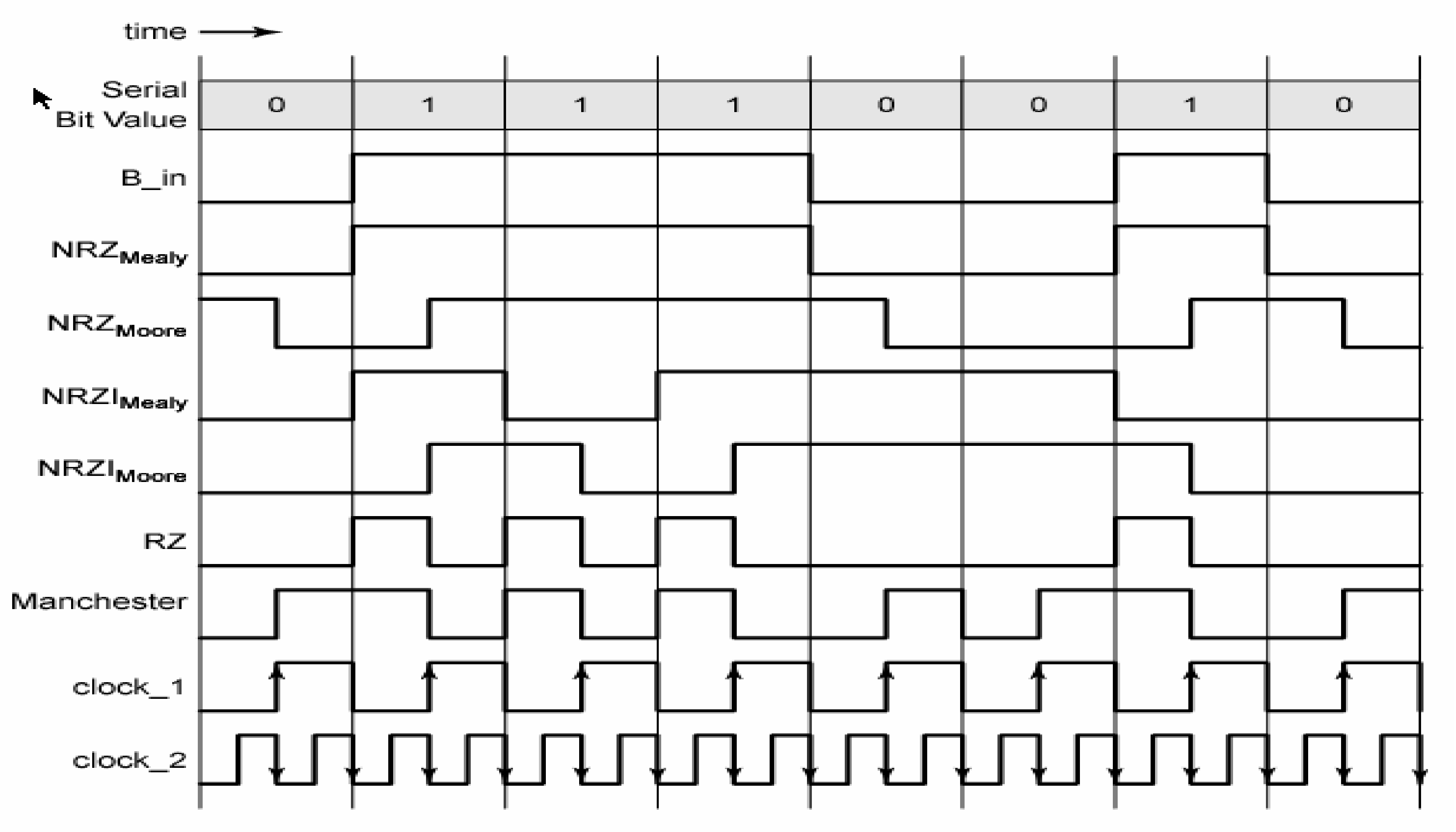
- There is **always** a mid-bit transition {which is used as a clocking mechanism}.
- **The direction of the mid-bit transition represents the digital data**.

0 ⬄ low-to-high transition

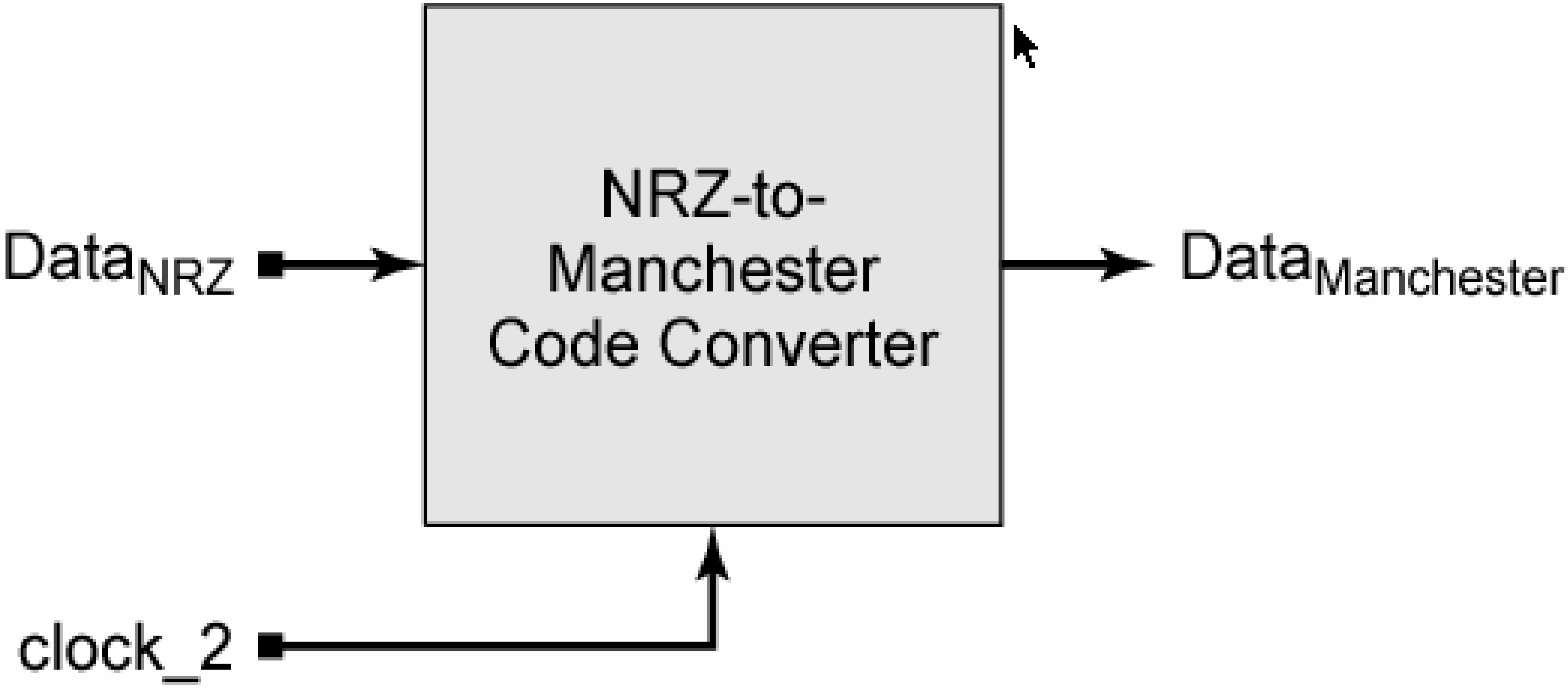
1 ⬄ high-to-low transition

There may be a second transition at the beginning of the bit interval.

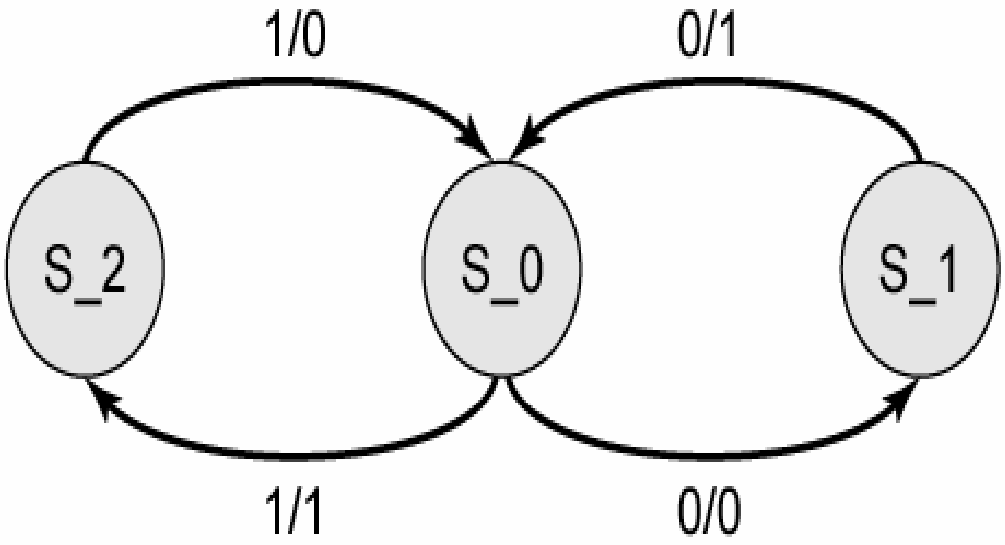
The signal waveform of the NRZ-to-Manchester line code



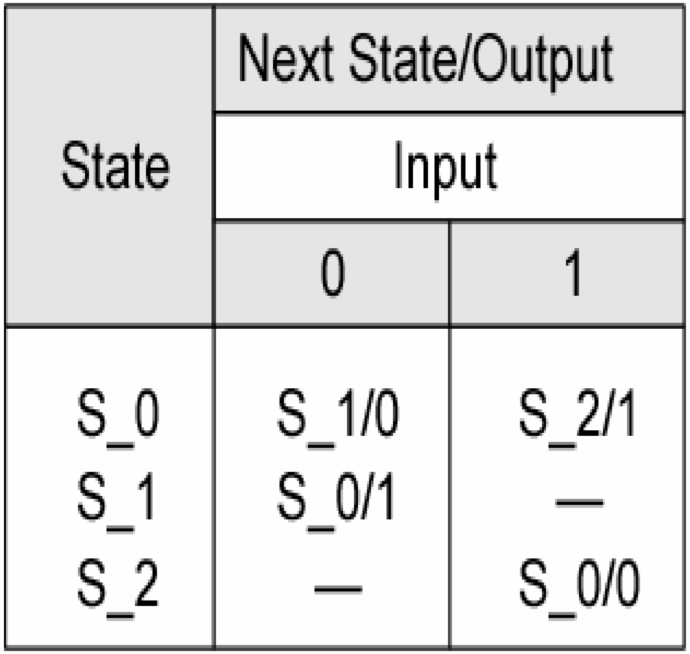
A Mealy-type NRZ-to-Manchester line code converter



A Mealy-type NRZ-to-Manchester line code converter



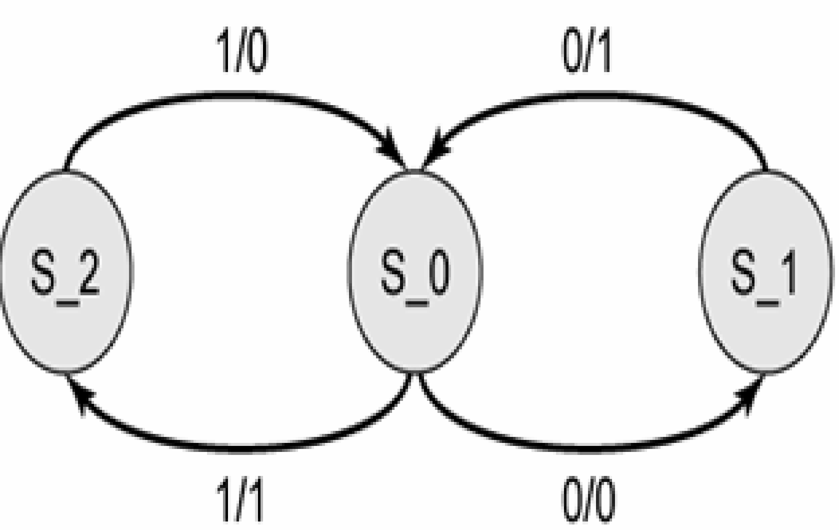
(a)

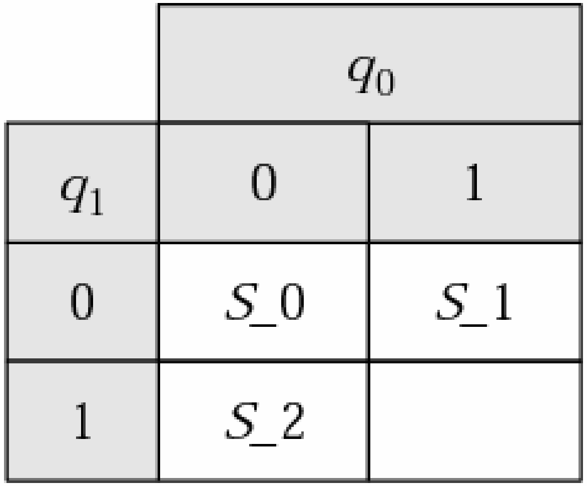


State	Next State/Output	
	Input	
	0	1
S_0	S_1/0	S_2/1
S_1	S_0/1	—
S_2	—	S_0/0

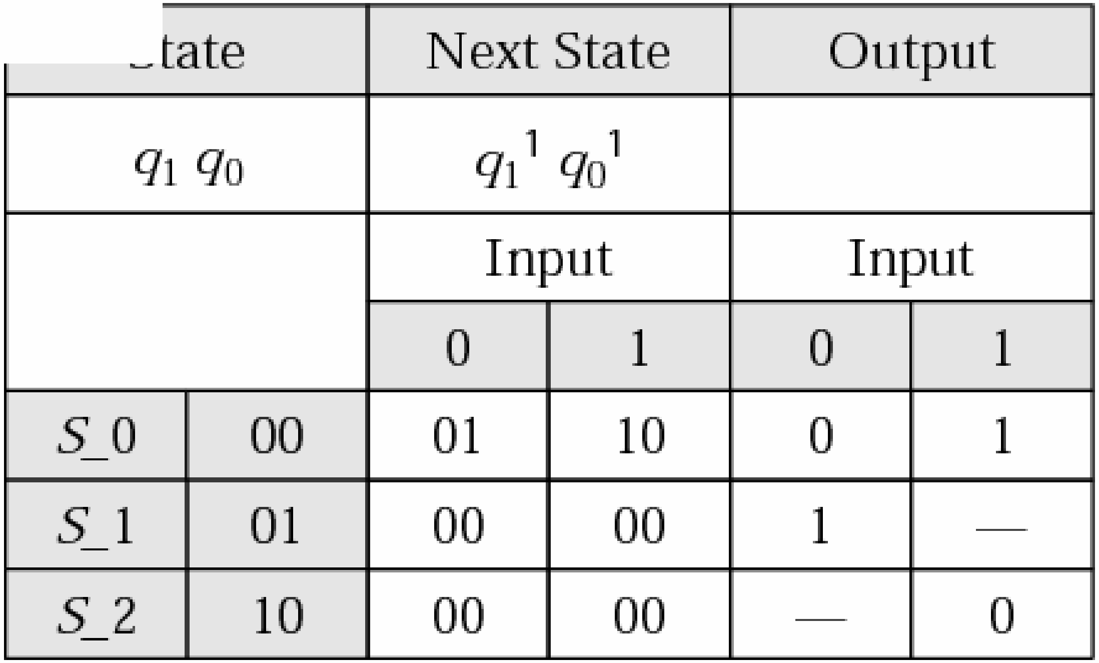
(b)

(a) state assignment
(b) endcoded next state & out table



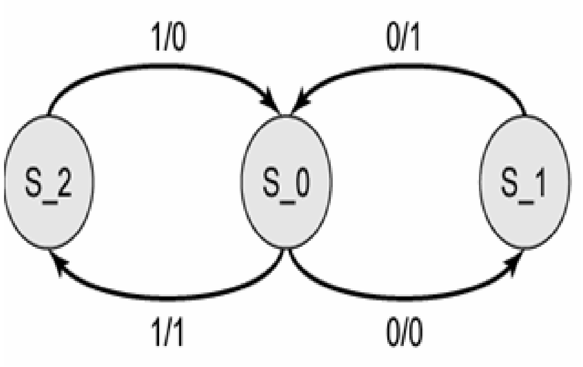


	q_0	
q_1	0	1
0	S_0	S_1
1	S_2	

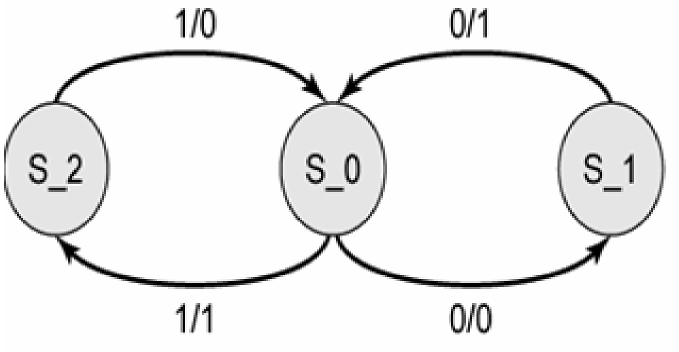


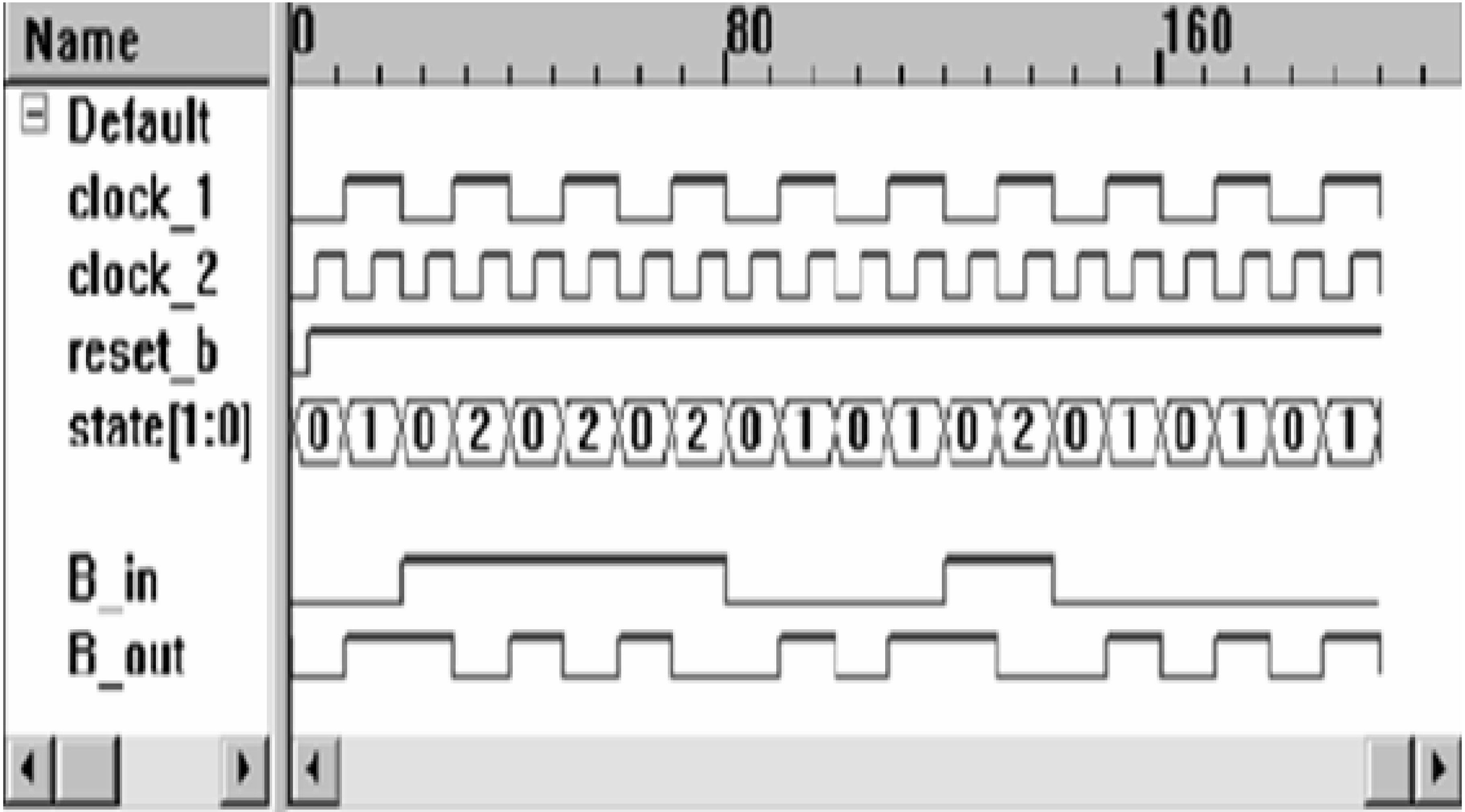
State		Next State		Output	
$q_1\ q_0$		$q_1^1\ q_0^1$			
		Input		Input	
		0	1	0	1
S_0	00	01	10	0	1
S_1	01	00	00	1	—
S_2	10	00	00	—	0

```
module NRZ_2_Manchester_Mealy
         (B_out,B_in,clock,reset_b);
  output B_out;
  input B_in;
  input clock,reset_b;
  reg [1:0] state,next_state;
  reg B_out;
parameter S_0=0, S_1=1, S_2=2,   dont_care_state=2'bx,
    dont_care_out=1'bx;
 always @ (negedge clock or negedge reset_b)
    if (reset_b==0) state<=S_0; else state<=next_state;
 always @ (state or B_in)  begin
    B_out=0;
   case (state)
     S_0: if (B_in==0) next_state=S_1;
           else if (B_in==1) begin next_state=S_2; B_out=1; end
     S_1: begin next_state=S_0; B_out=1; end
     S_2: begin next_state=S_0; end
     default: begin next_state=dont_care_state; B_out=dont_care_out; end
    endcase
  end
 endmodule
```

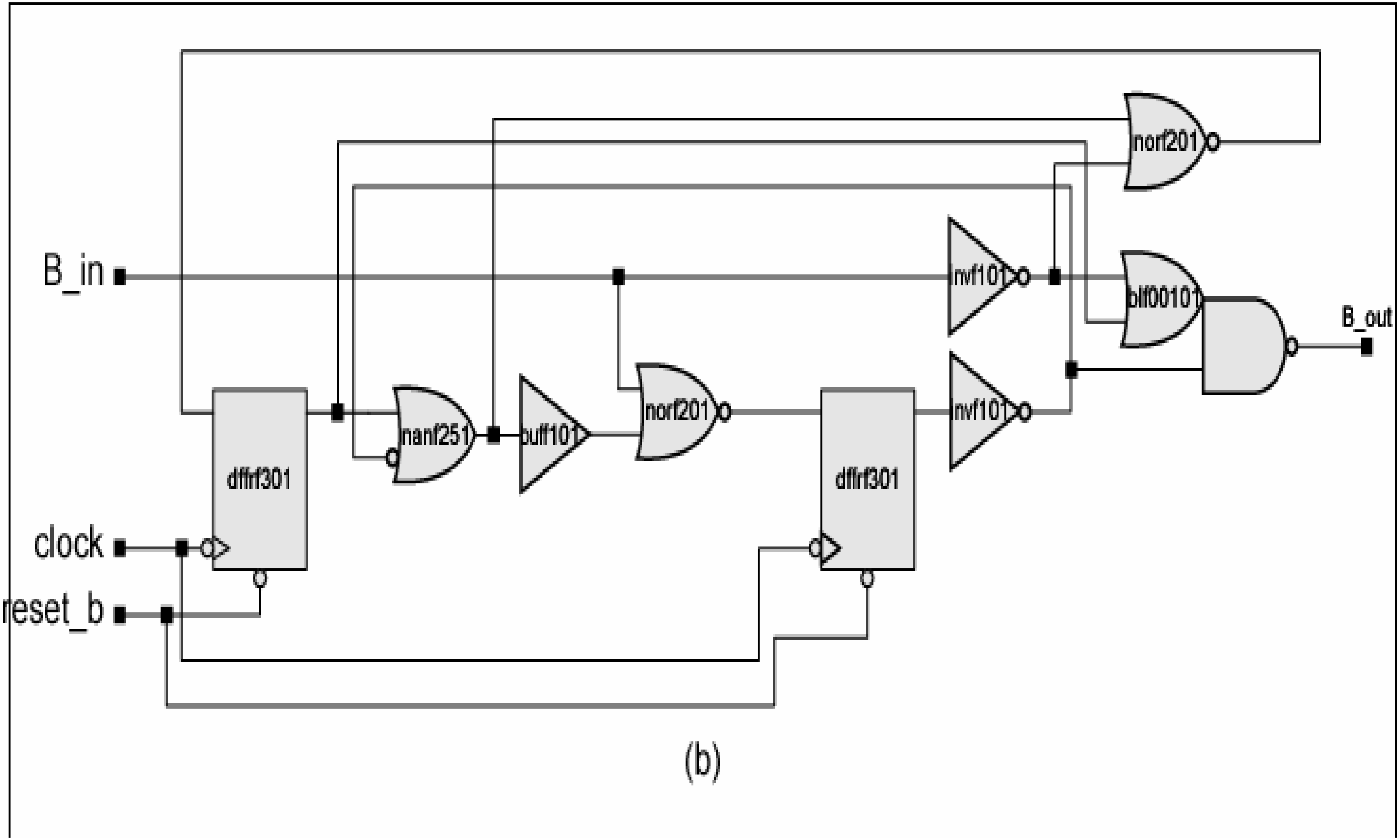


Simulation


1/0
0/1
S_2
S_0
S_1
1/1
0/0


Name
0
80
160
Default
clock_1
clock_2
reset_b
state[1:0]
0 1 0 2 0 2 0 2 0 1 0 1 0 2 0 1 0 1 0 1
B_in
B_out

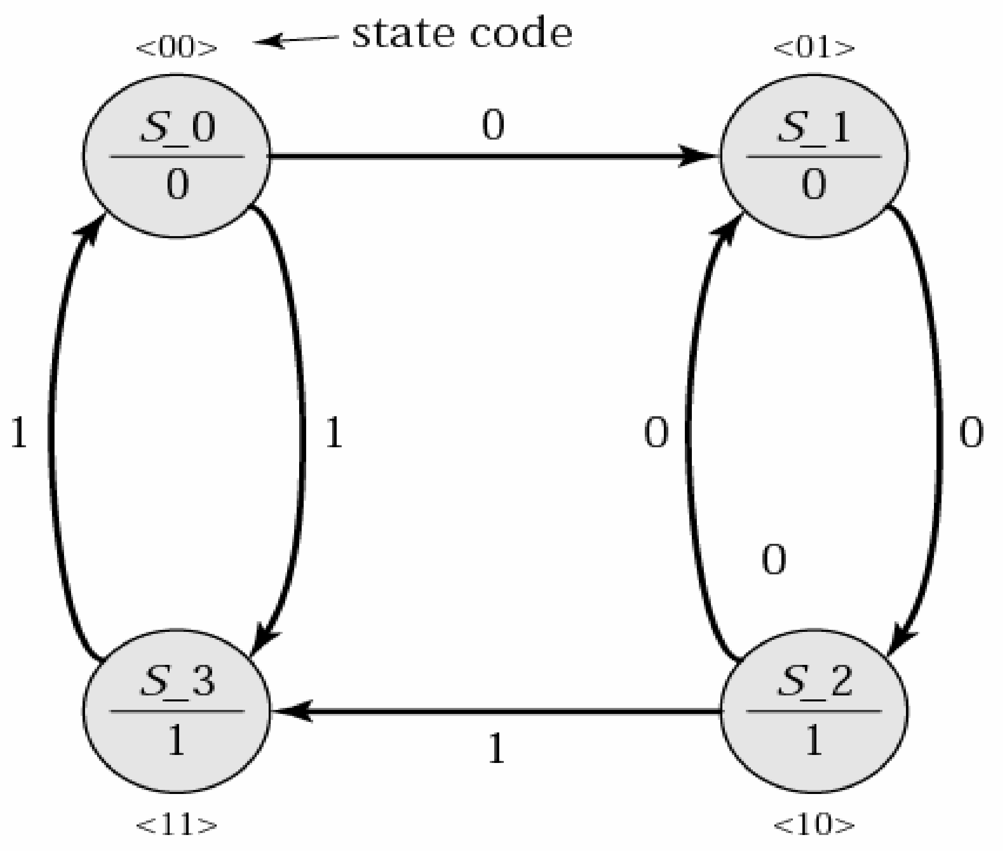
Schematic



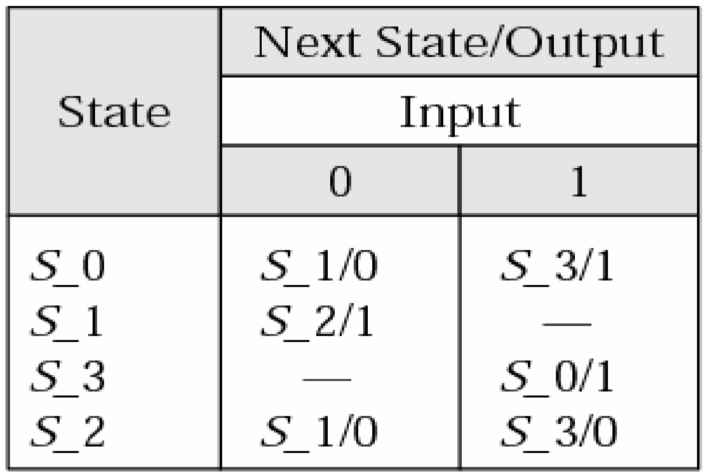
B_in
clock
reset_b
dffrf301
nanf251
buff101
norf201
dffrf301
invf101
invf101
norf201
blf00101
B_out

(b)

6.6.3 Synthesis of a Moore-type NRZ-to-Manchester line code converter



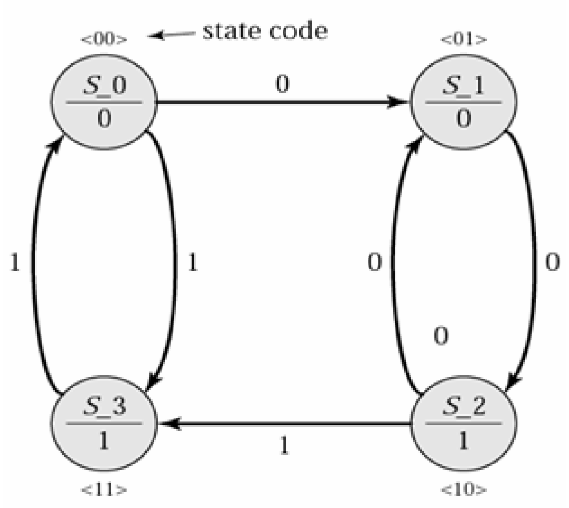
(a)



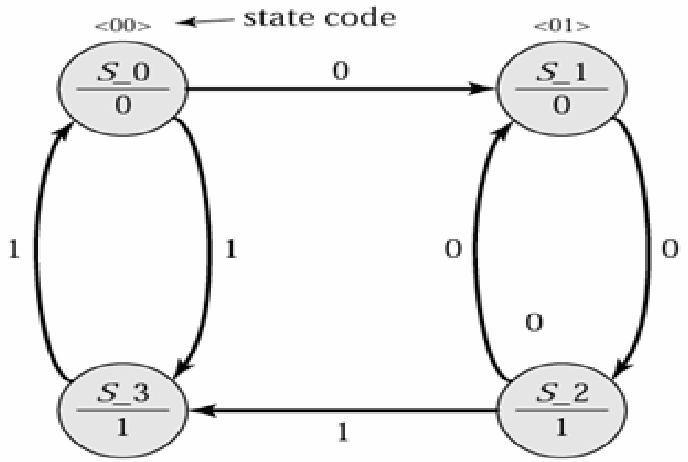
State	Next State/Output	
	Input	
	0	1
*S*_0	*S*_1/0	*S*_3/1
*S*_1	*S*_2/1	—
*S*_3	—	*S*_0/1
*S*_2	*S*_1/0	*S*_3/0

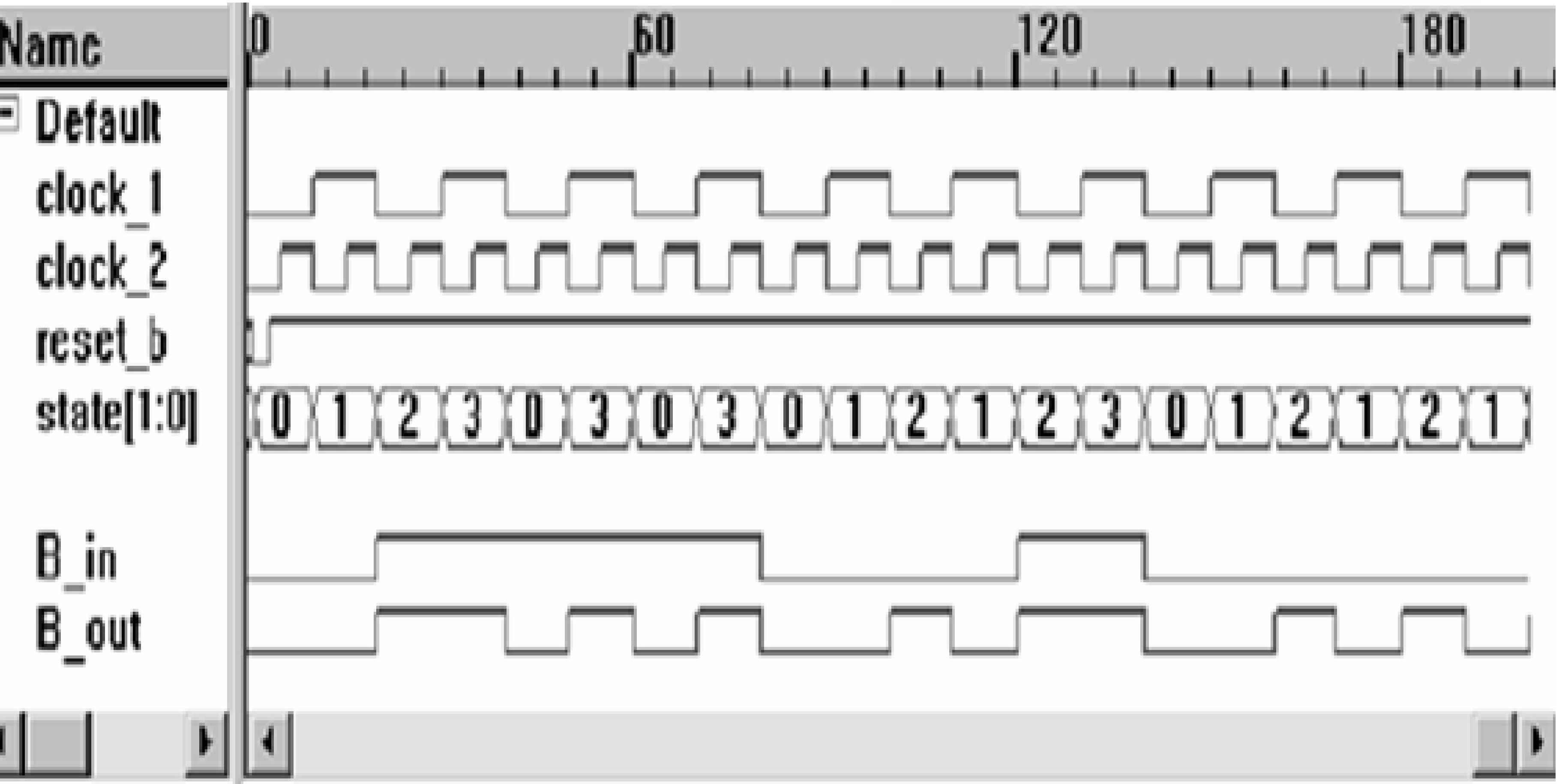
(b)

```
module NRZ_2_Manchester_Moore (B_out,B_in,clock,reset_b);
 output B_out;
 input B_in,  input clock,reset_b;
 reg [1:0] state,next_state;
 reg B_out;
 parameterS_0=0, S_1=1, S_2=2, S_3=3;
always @ (negedge clock or negedge reset_b)
   if (reset_b==0) state<=S_0; else state<=next_state;
always @ (state or B_in)  begin
  case (state)
   S_0: begin if (B_in==0) next_state=S_1;
        else next_state=S_3;
        end
   S_1: begin next_state=S_2; end
   S_2: begin B_out=1;
      if (B_in==0) next_state=S_1;
        else next_state=S_3;
       end
   S_3: begin B_out=1; next_state=S_0; end
  endcase
 end
endmodule
```

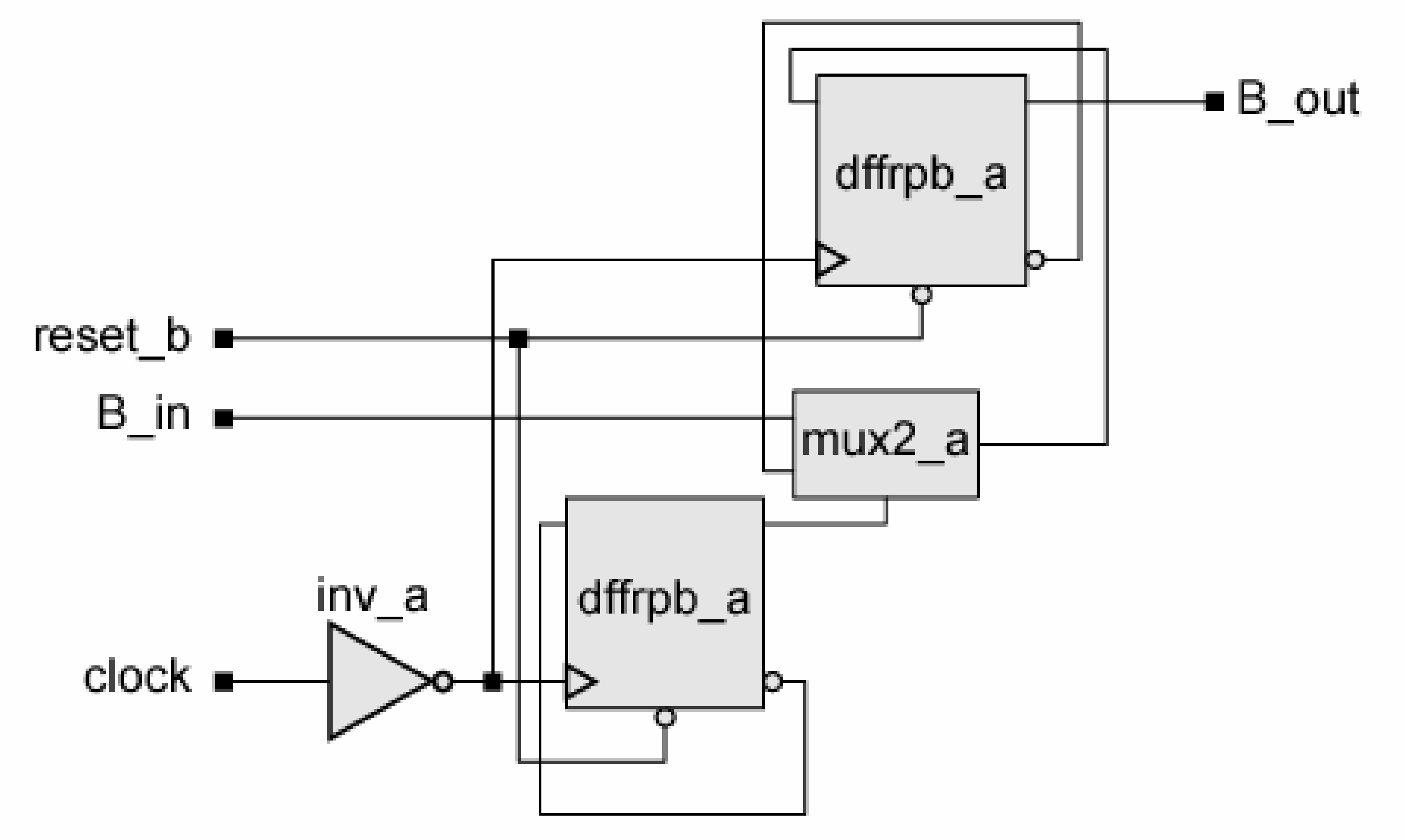


Simulation


state code
<00>
<01>
S_0
0
S_1
0
0
1
1
0
0
0
S_3
1
S_2
1
1


Name
0
60
120
180
Default
clock_1
clock_2
reset_b
state[1:0]
0 1 2 3 0 3 0 3 0 1 2 1 2 3 0 1 2 1 2 1
B_in
B_out

ASIC synthesis circuit



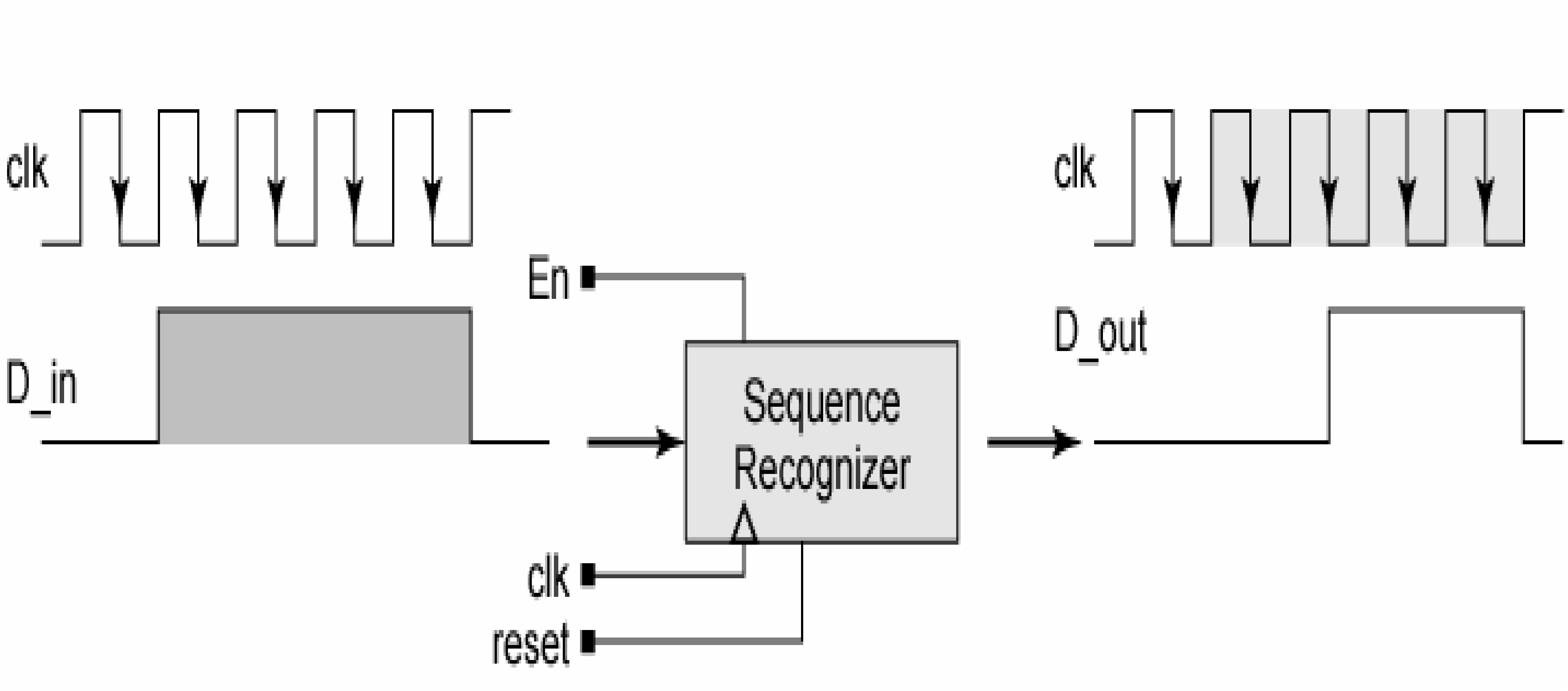
6.6.4 Synthesis of a sequence recognizer

- **To describe sequence recognizers:**
- **1.To clarify the semantics of how the machine receives input bits**
- **2. To distinguish between resetting and non-resetting machines**

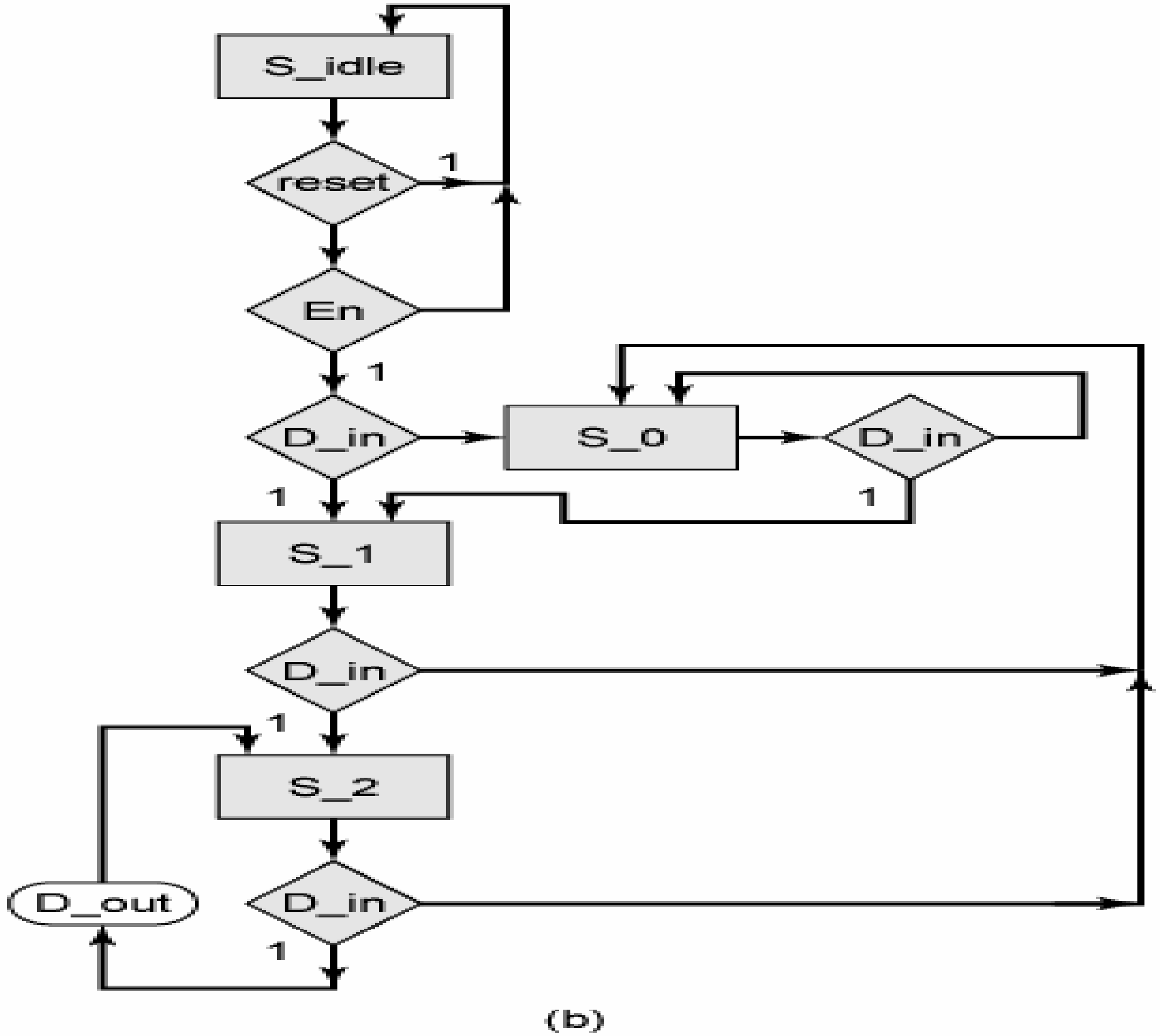
A non-resetting machine continues to assert its output if the input bit pattern is overlapping

Example 6.26

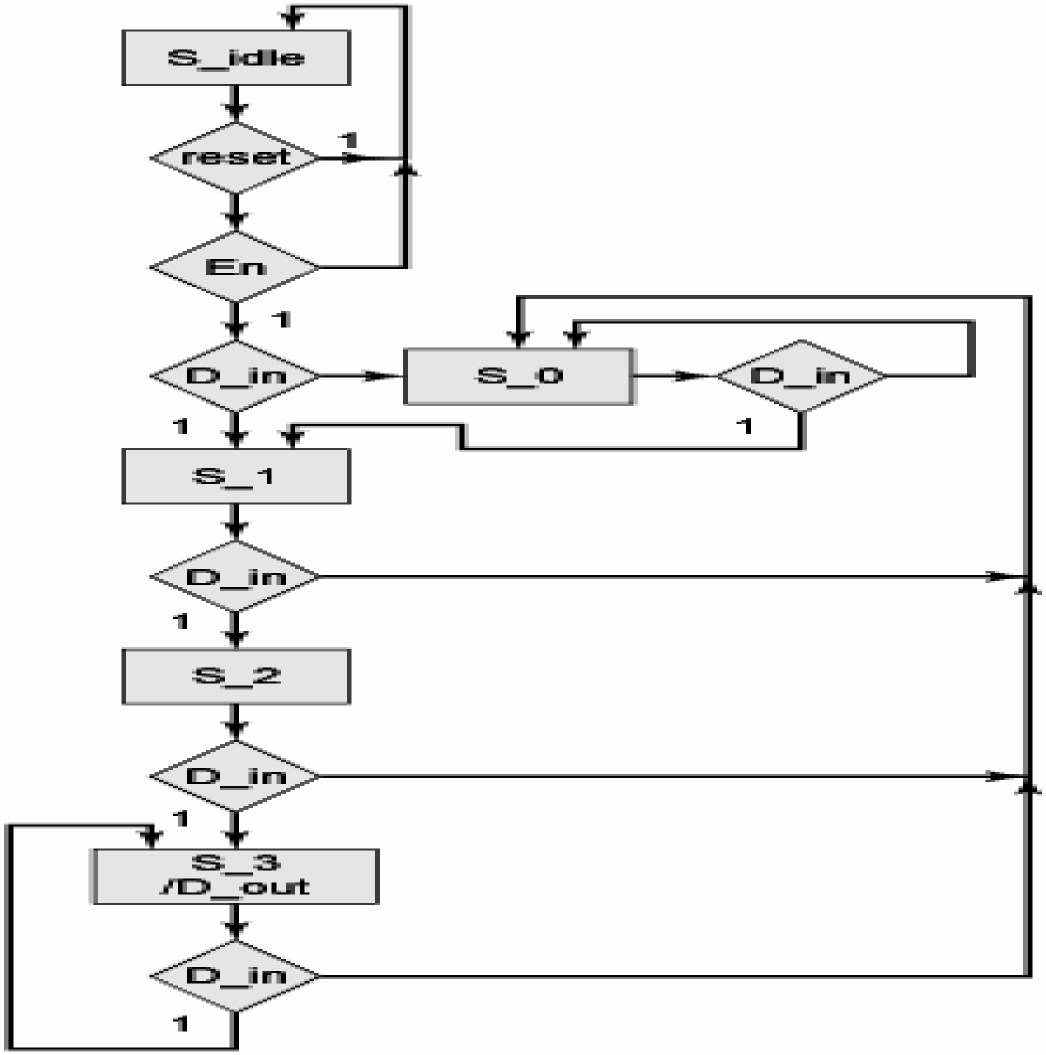
- The sequence recognizer is to sample the serial input, D_in, **on the falling edge of the clock and assert D_out if three successive samples are 1**



ASM chart for a Mealy-type explicit FSM implementation



ASM chart for a Moore-type explicit FSM implementation



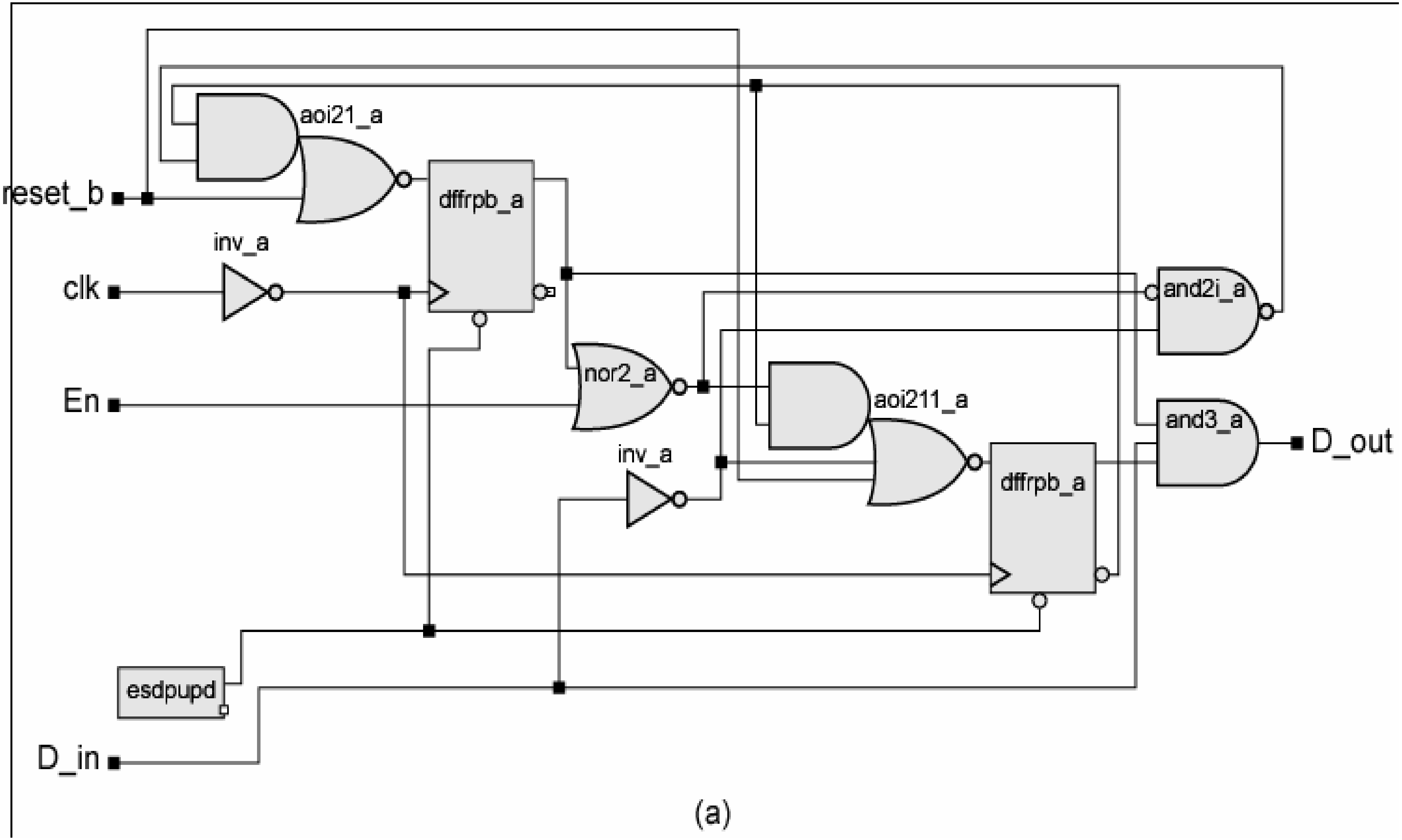
(c)

Mealy-type sequence recognizer

```
module Seq_Rec_3_1s_Mealy (D_out, D_in, En, clk, reset);
    output D_out;
    input D_in, En;
    input clk, reset;
    parameter S_idle=0;   // Binary code
    parameter S_0=1;
    parameter S_1=2;
    parameter S_2=3;
    reg [1:0] state, next_state;
    always @ (negedge clk)
    if (reset==1) state<=S_idle; else state<=next_state;
```

```
always @ (state or D_in) begin
    case (state)
        S_idle:   if ((En==1)&&(D_in==1) next_state=S_1; else
                  if ((En==1)&&(D_in==0) next_state=S_0;
                    else         next_state=S_idle;
         S_0:     if (D_in==0) next_state=S_0;
                    else   if (D_in==1) next_state=S_1;
                    else   next_state=S_idle;
         S_1:     if (D_in==0) next_state=S_0;
                    else if (D_in==1) next_state=S_2;
                     else         next_state=S_idle;
         S_2:     if (D_in==0) next_state=S_0;
                      else  if (D_in==1) next_state=S_2;
                      else        next_state=S_idle;
         default:                 next_state=S_idle;
      endcase
    end
assign D_out=((state=S_2) && (D_in==1));     // Mealy output
endmodule
```

ASIC synthesis circuit



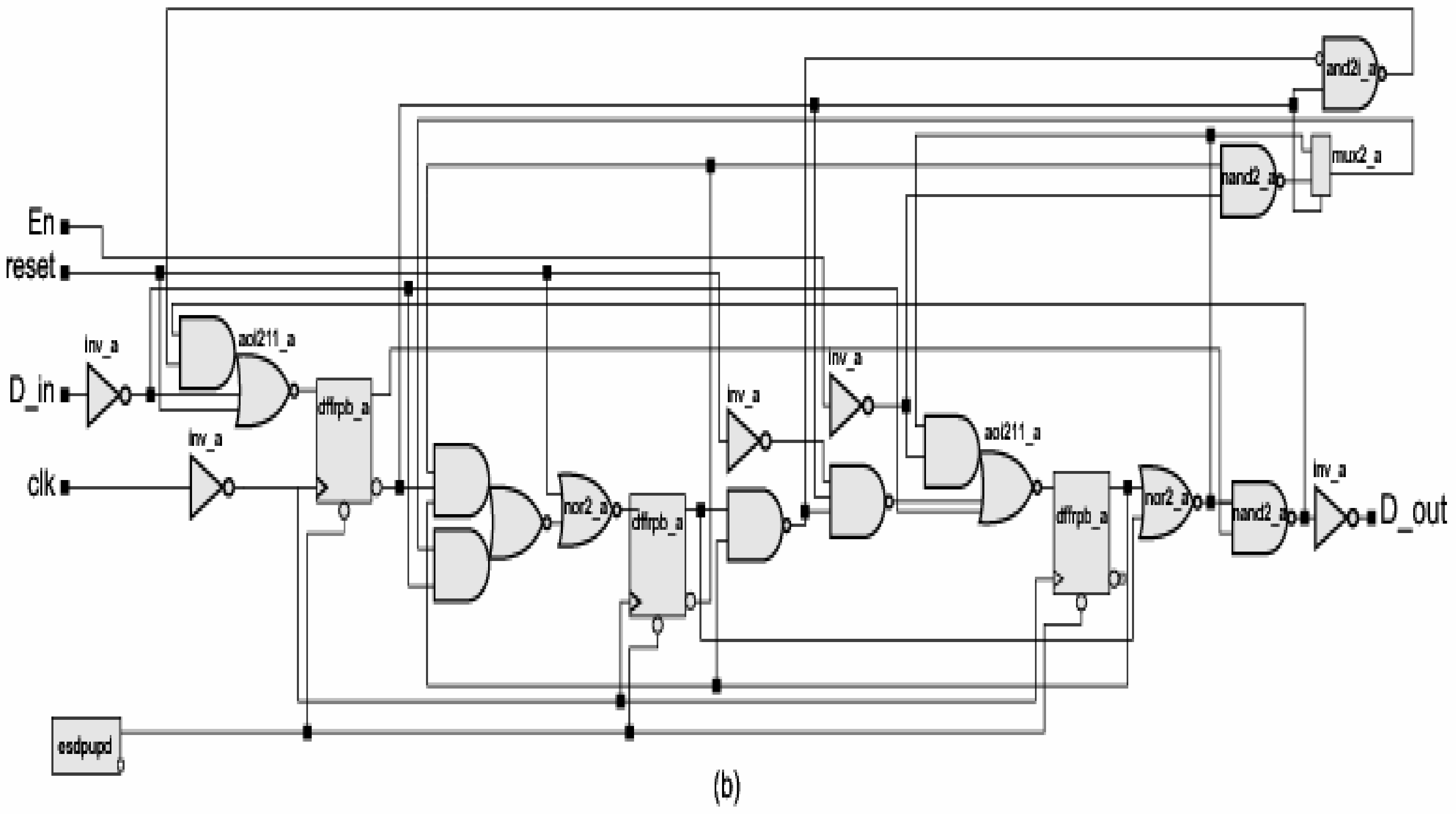
(a)

Moore-type sequence recognizer

```
module Seq_Rec_3_1s_Moore (D_out, D_in, En, clk, reset);
    output D_out;
    input D_in, En;
    input clk, reset;
    parameter S_idle=0;   // One-hot
    parameter S_0=1;
    parameter S_1=2;
    parameter S_2=3;
    parameter S_3=4;
    reg [2:0] state, next_state;
    always @ (negedge clk)
    if (reset==1) state<=S_idle; else state<=next_state;
```

```
always @ (state or D_in) begin
 case (state)
           S_idle:   if ((En==1)&&(D_in==1) next_state=S_1; else
                     if ((En==1)&&(D_in==0) next_state=S_0;
                     else                                   next_state=S_idle;
            S_0:     if (D_in==0) next_state=S_0; else
                     if (D_in==1) next_state=S_1;
                     else                next_state=S_idle;
            S_1:     if (D_in==0) next_state=S_0; else
                     if (D_in==1) next_state=S_2;
                     else                next_state=S_idle;
            S_2,S_3: if (D_in==0) next_state=S_0; else
                       if (D_in==1) next_state=S_3;
                       else                next_state=S_idle;
            default:                  next_state=S_idle;
         endcase
      end
assign D_out=(state=S_3);      // Moore output
endmodule
```

ASIC synthesis circuit



(b)

Testbench

```
module t_Seq_Rec_3_1s ();
     reg D_in_NRZ,D_in_RZ,En,clk,reset;
     wire Mealy_NRZ;
     wire Mealy_RZ;
     wire Moore_NRZ;
     wire Moore_RZ;
     Seq_Rec_3_1s_Mealy M0 (Mealy_NRZ,D_in_NRZ,En,clk,reset);
     Seq_Rec_3_1s_Mealy M1 (Mealy_RZ,D_in_RZ,En,clk,reset);
     Seq_Rec_3_1s_Moore M2 (Moore_NRZ,D_in_NRZ,En,clk,reset);
     Seq_Rec_3_1s_Moore M3 (Moore_RZ,D_in_RZ,En,clk,reset);
   initial #275 $finish;
   initial begin #5 reset=1; #1 reset=0; end
   initial begin clk=0; forever #10 clk=~clk; end
   initial begin
      #5 En=1;
      #50 En=0; end
```

```
 initial
  fork
    begin #10 D_in_NRZ=0; #25 D_in_NRZ=1; #80 D_in_NRZ=0;   end
    begin #135 D_in_NRZ=1; #40 D_in_NRZ=0; end
    begin #195 D_in_NRZ=1'bx; #60 D_in_NRZ=0; end
   join
 initial
    fork
     #10 D_in_RZ=0;
     #35 D_in_RZ=1;  #45 D_in_RZ=0;
     #55 D_in_RZ=1;  #65 D_in_RZ=0;
     #75 D_in_RZ=1;  #85 D_in_RZ=0;
     #95 D_in_RZ=1;  #105 D_in_RZ=0;
     #35 D_in_RZ=1;  #145 D_in_RZ=0;
     #135 D_in_RZ=1;  #45 D_in_RZ=0;
     #155 D_in_RZ=1;  #165 D_in_RZ=0;
     #195 D_in_RZ=1'bx;  #250 D_in_RZ=0;
   join
endmodule
```

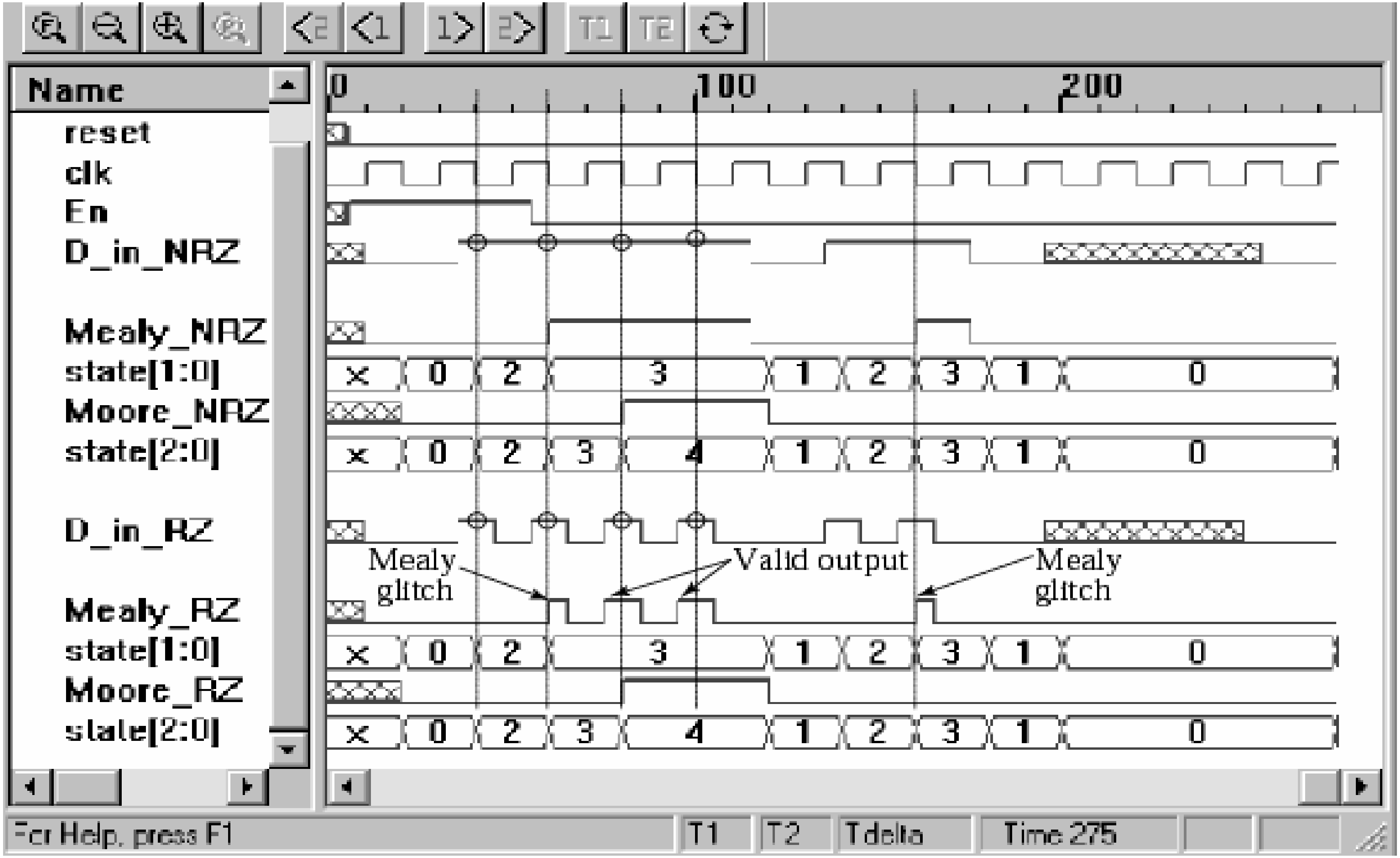
fork

- Groups statements into **a parallel block**, so that they are **executed concurrently**.
- **fork [: Label [Declarations...]]**

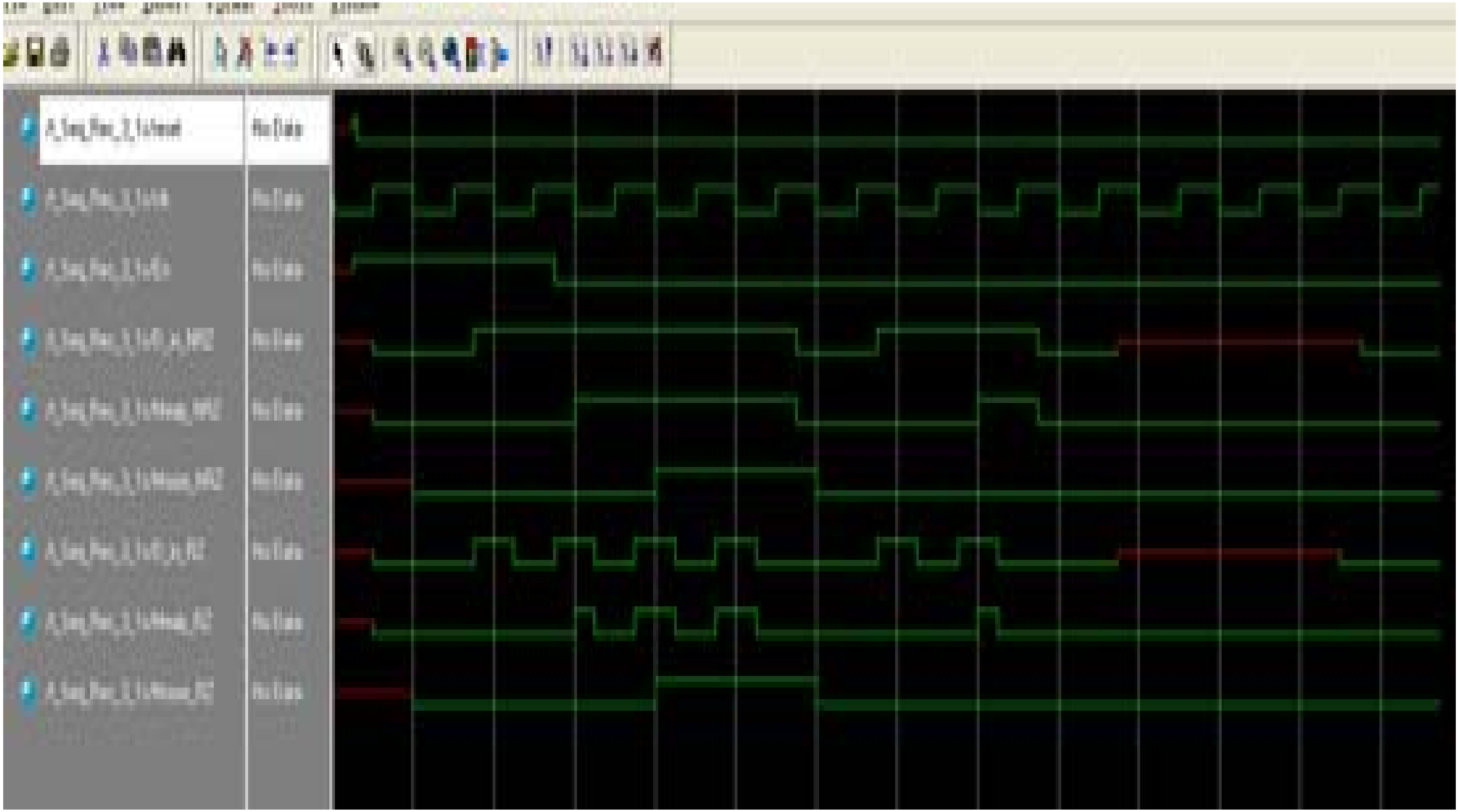
  Statements...

  Join
- Declaration = {either} Register Parameter Event
- **Not synthesizable.**

Simulation


Name
reset
clk
En
D_in_NRZ
Mealy_NRZ
state[1:0]
Moore_NRZ
state[2:0]
D_in_RZ
Mealy_RZ
state[1:0]
Moore_RZ
state[2:0]
0
100
200
x 0 2 3 1 2 3 1 0
x 0 2 3 4 1 2 3 1 0
x 0 2 3 1 2 3 1 0
x 0 2 3 4 1 2 3 1 0
Mealy glitch
Valid output
Mealy glitch
For Help, press F1
T1
T2
Tdelta
Time 275

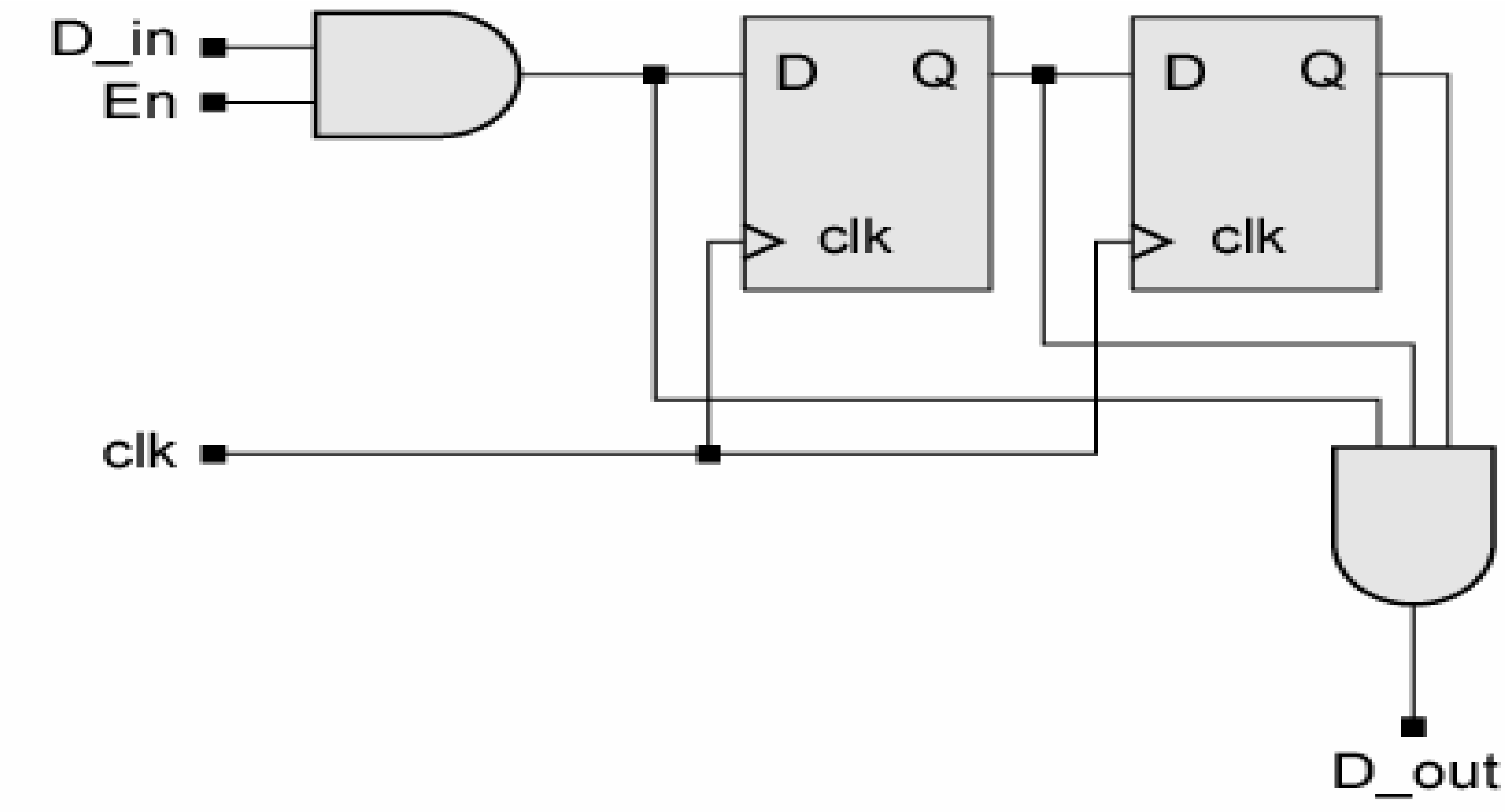
Simulation result



Alternative approach for Mealy-type

```
module Seq_Rec_3_1s_Mealy_Shft_Reg (D_out, D_in, En,
   clk, reset);
 output D_out;
 input D_in,En;
 input clk,reset;
 parameter Empty=2’b00;
 reg [1:0] Data;
 always @ (negedge clk)
 if (reset==1) Data<=Empty;
     else if (En==1) Data<={D_in,Data[1]};
 assign D_out=((Data==2’b11)&&(D_in==1)); // mealy_output
endmodule
```

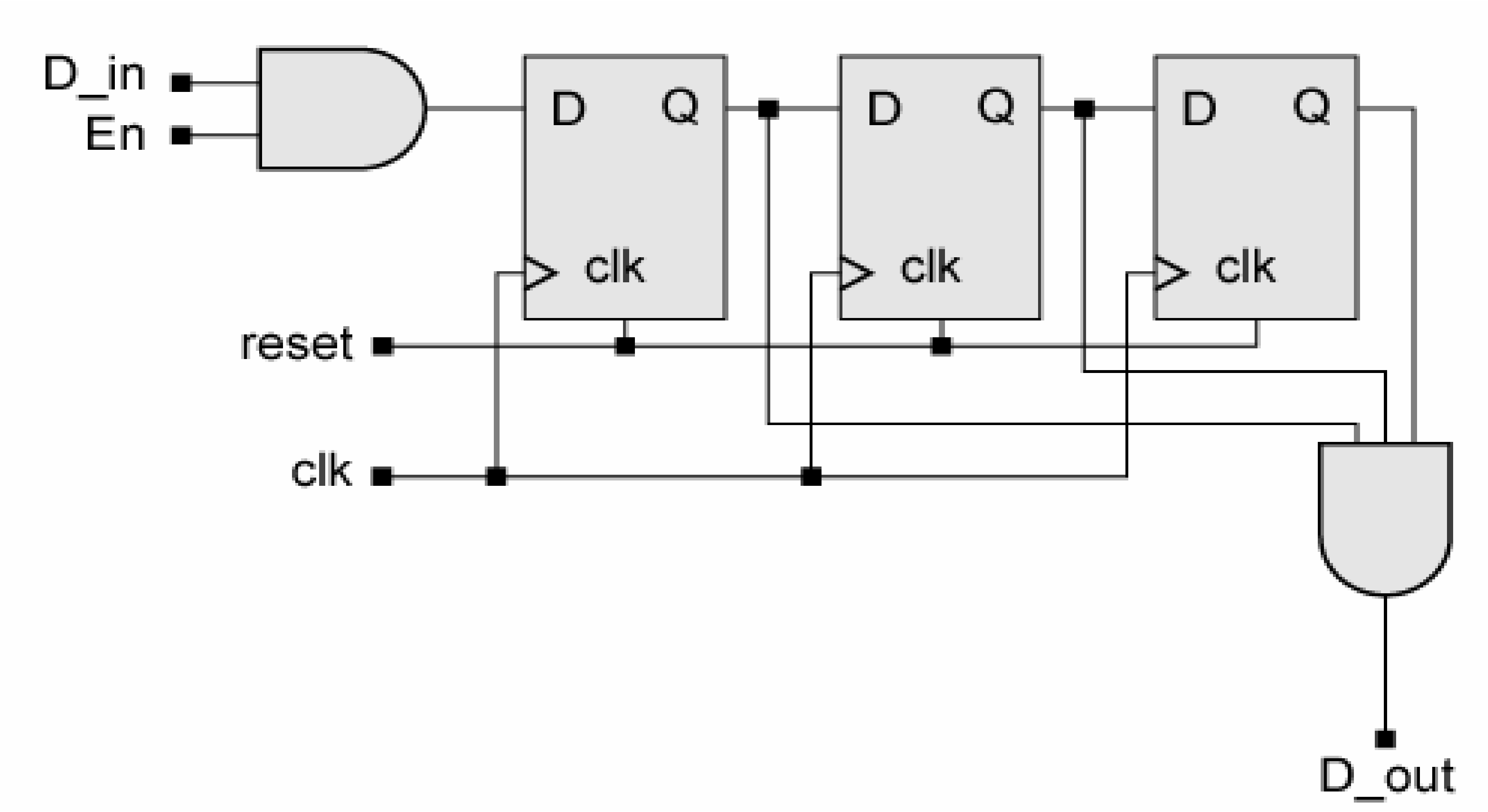
Mealy sequence recognizers for detecting three successive 1s in a serial bit stream



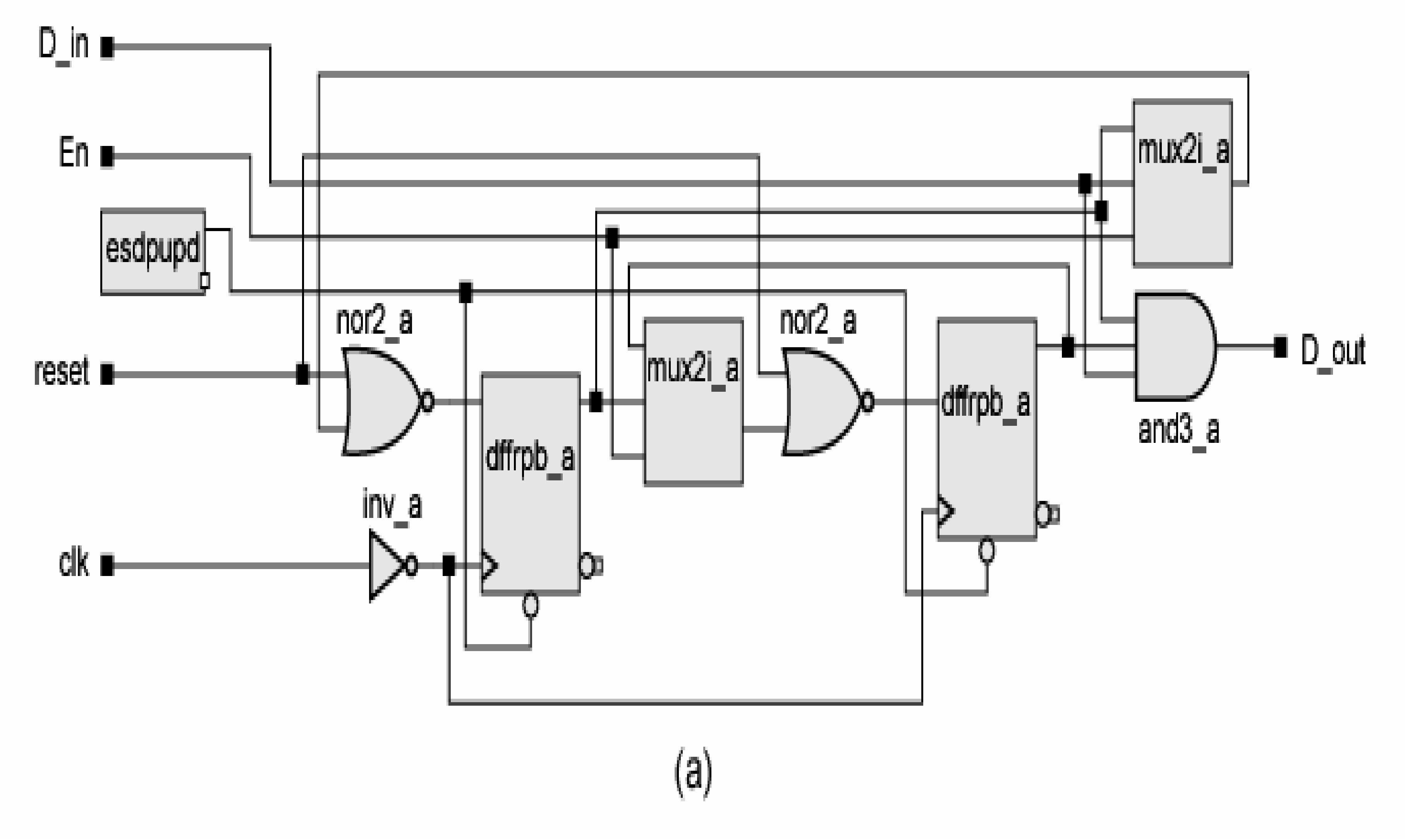
Alternative approach for Moore-type

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

 always @ (negedge clk)
 if (reset==1) Data<=Empty; else if (En==1)
    Data<={D_in,Data[2:1]};
 assign D_out=((Data==3'b111);      // Moore output
endmodule
```

Moore sequence recognizers for detecting three successive 1s in a serial bit stream

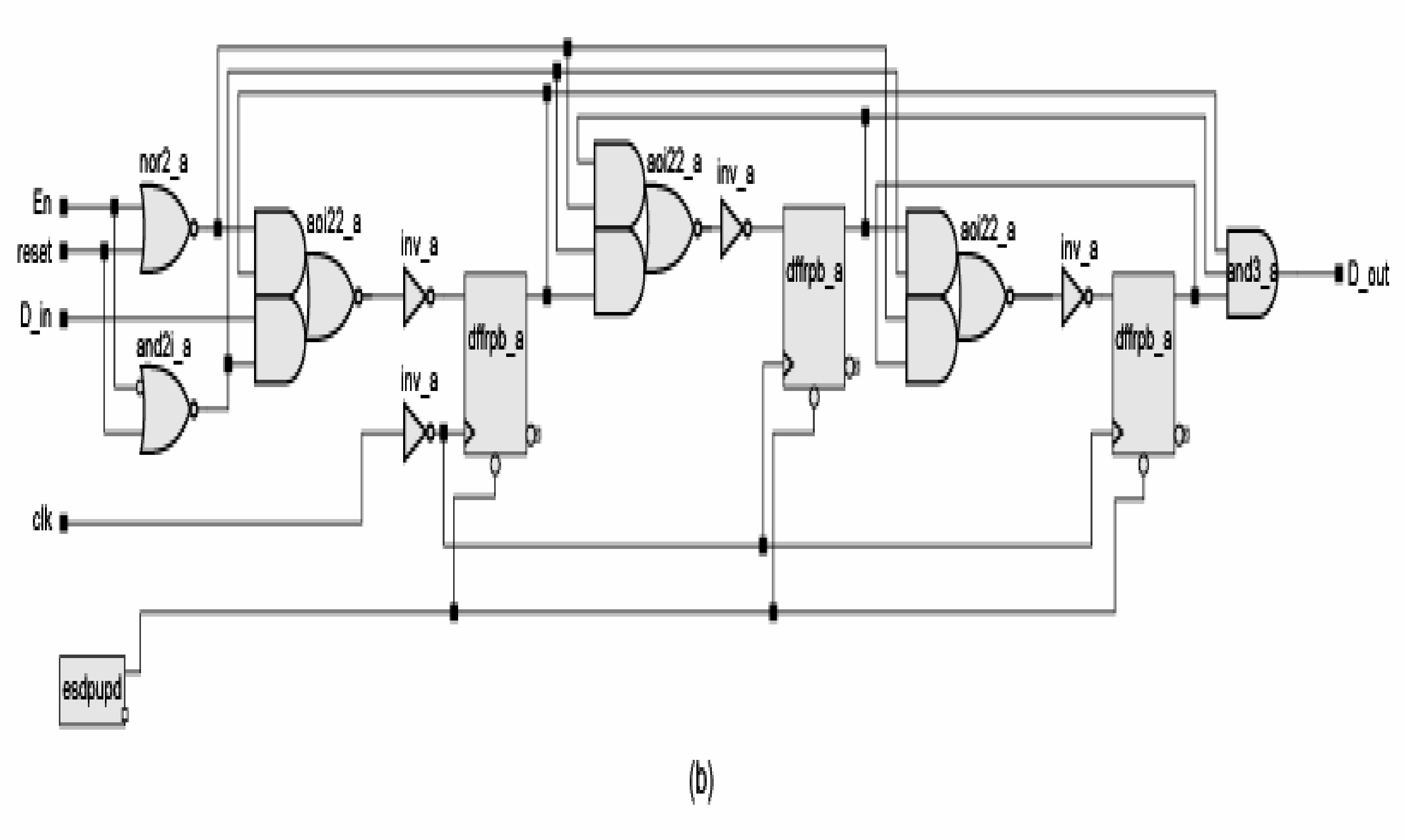


Synthesis circuit

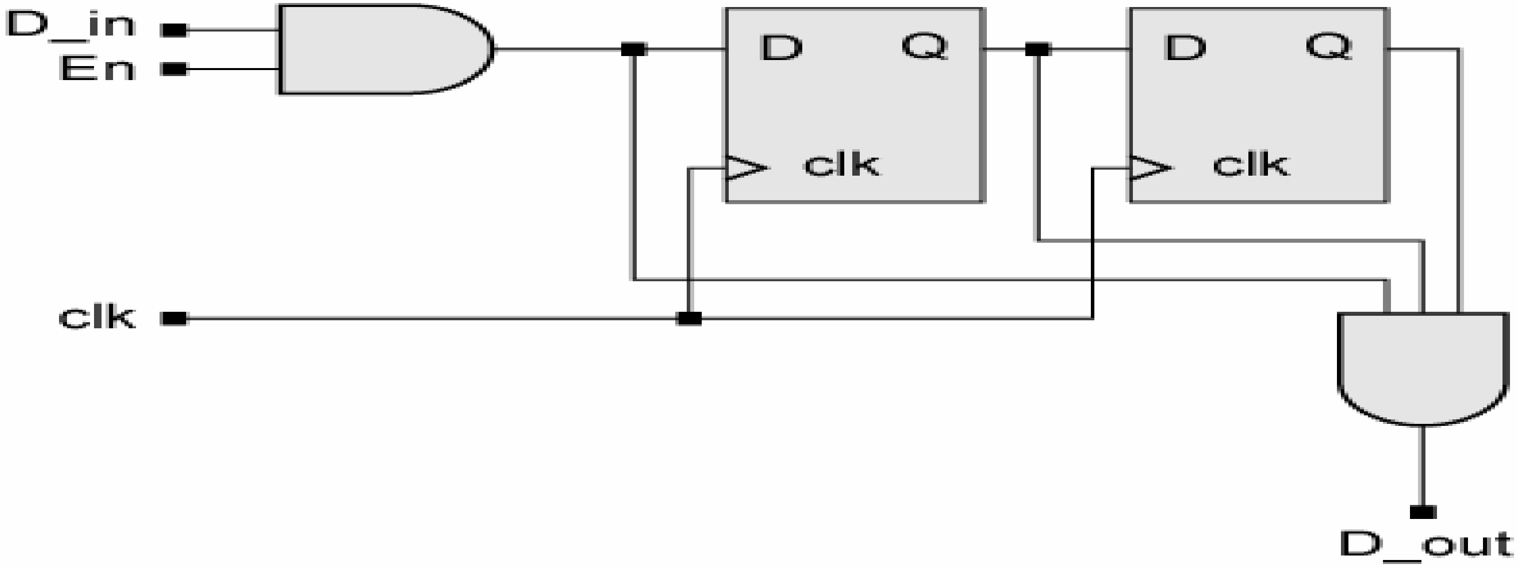


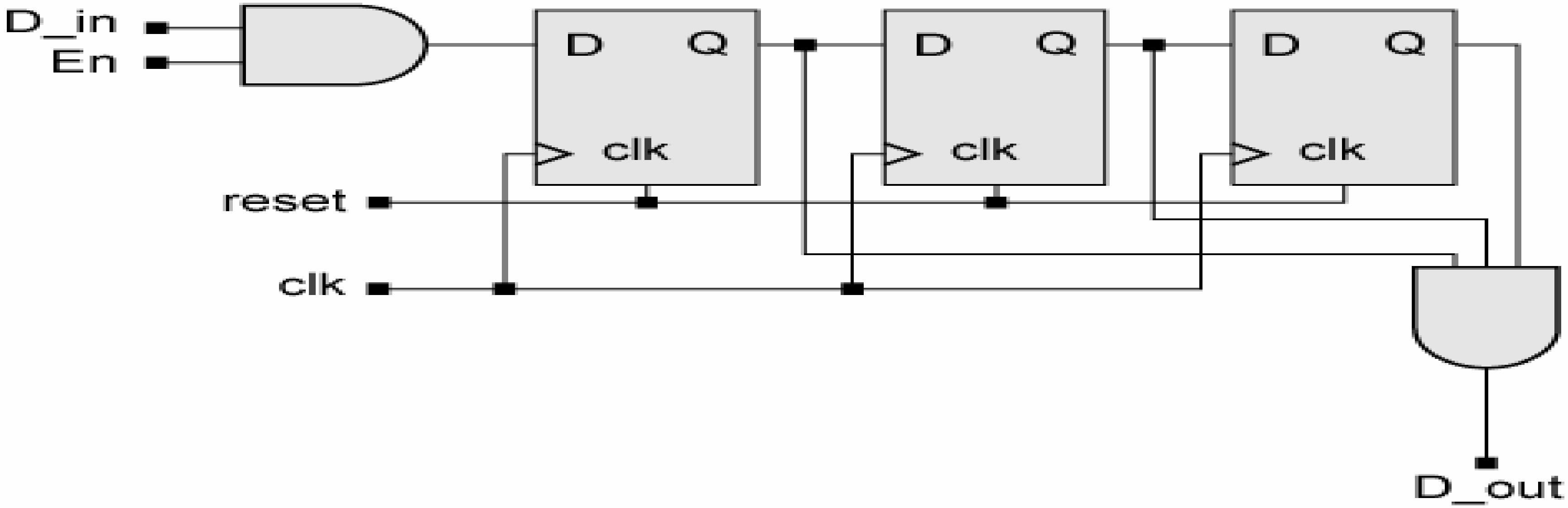
(a)

Synthesis circuit

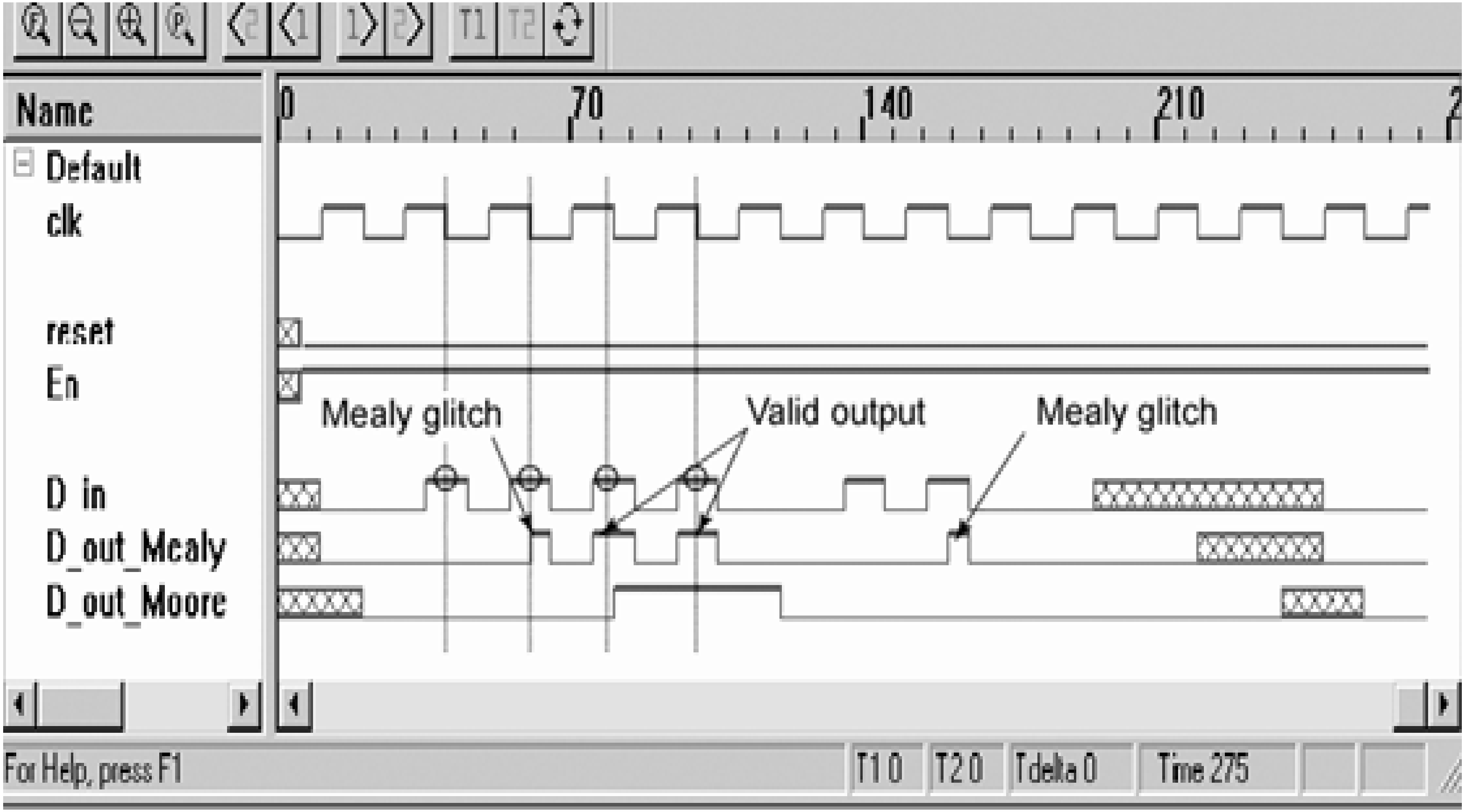


The difference of Mealy,Moore sequence recognizers





Simulation result

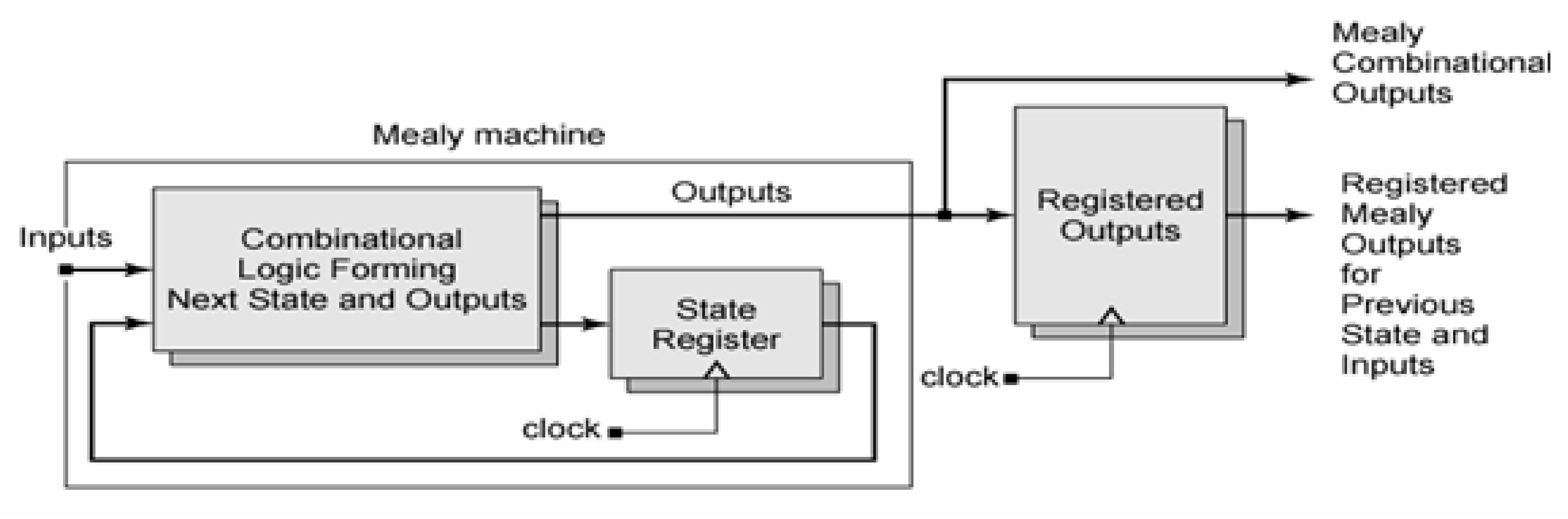


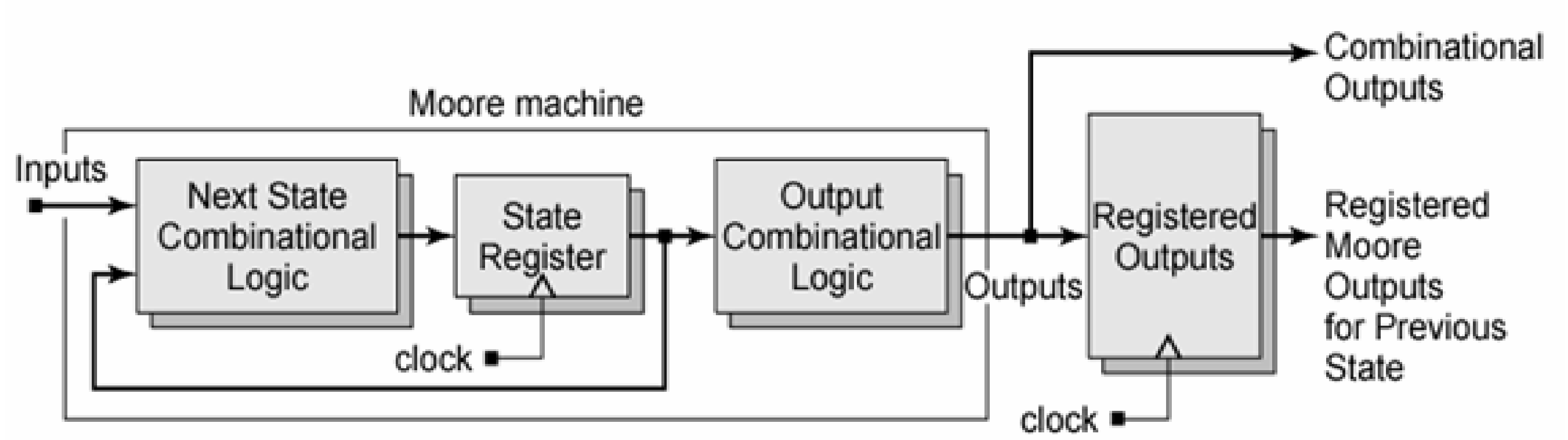
(c)

6.7 Registered logic

- Variables whose values are assigned synchronously with a clock signal are said to be registered
- **Registered signals are updated at the active edges of the clock** and are stable (i.e., they cannot glitch)
- The outputs of a Moore-type state machine are not registered, but they **cannot glitch** with changes at the machine’s input

Registered outputs in a Mealy,moore machine


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

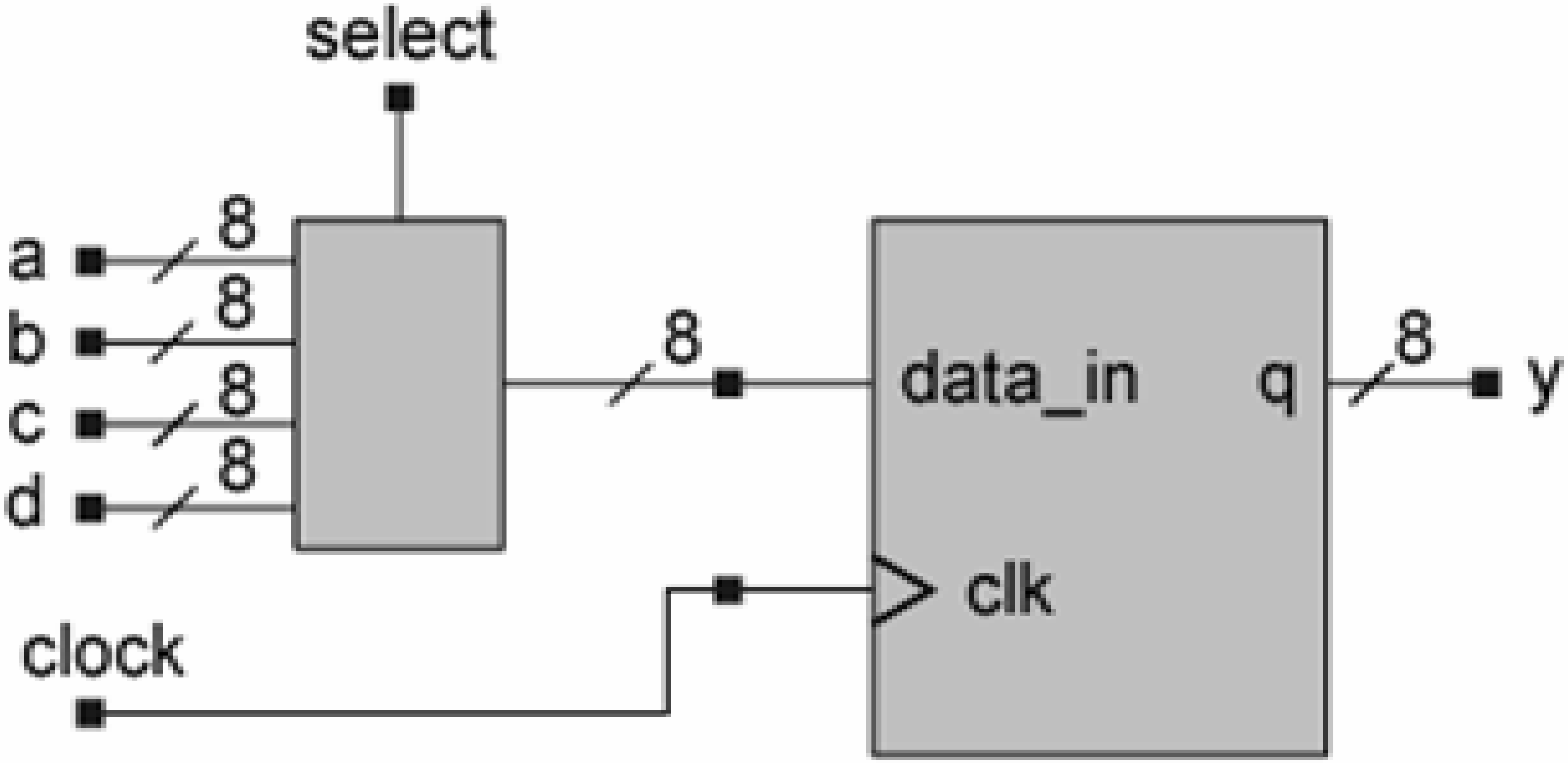

Moore machine
Inputs
Next State Combinational Logic
State Register
clock
Output Combinational Logic
Outputs
Registered Outputs
clock
Combinational Outputs
Registered Moore Outputs for Previous State

(b)

Example 6.27

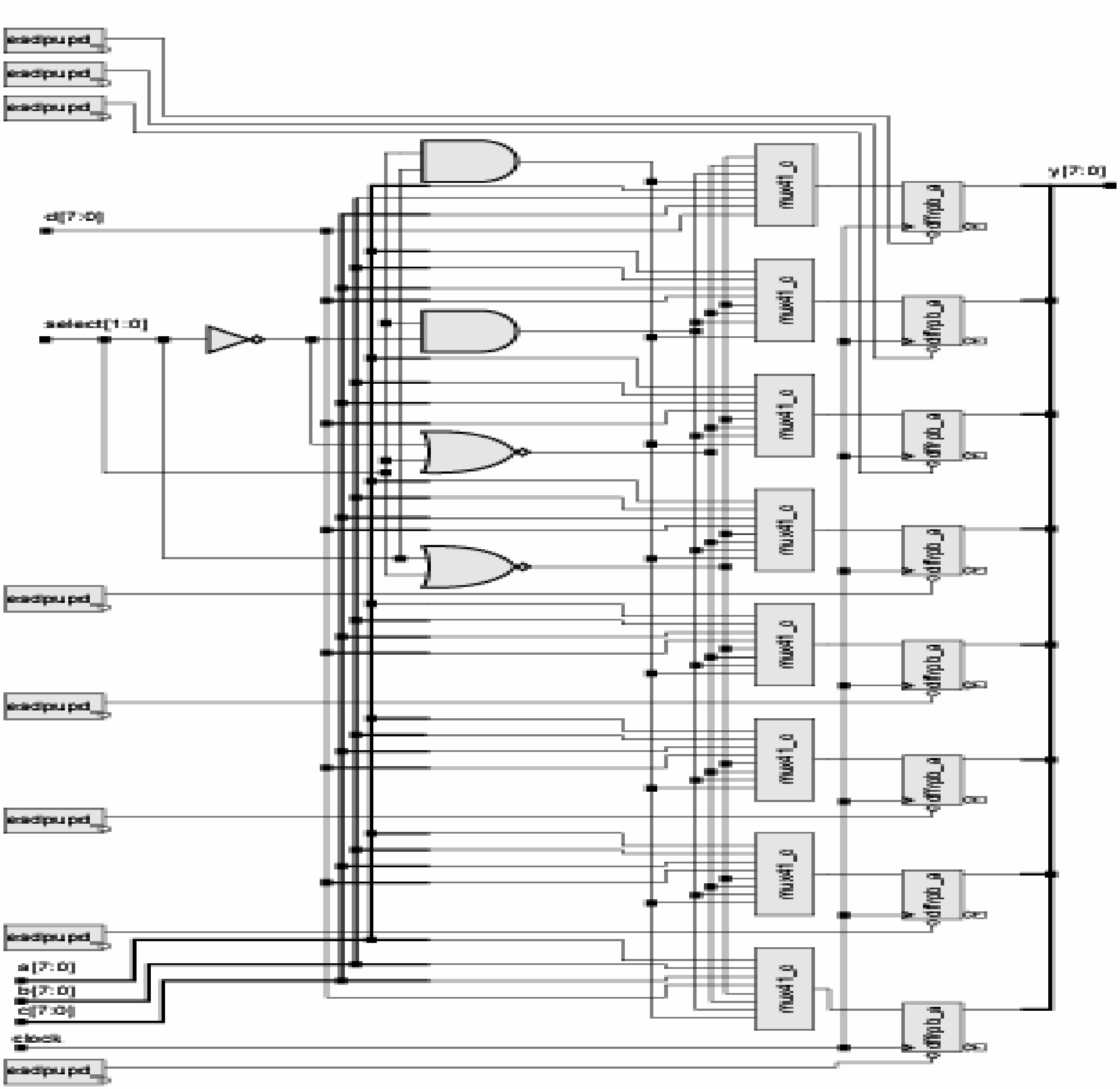
```
module mux_reg (y,a,b,c,d,select,clock);
  output [7:0] y;
  input [7:0] a,b,c,d;
  input [1:0] select;
  input clock;
  reg y;
  always @ (posedge clock)
    case (select)
         0: y<=a;      // non-blocking
         1: y<=b;
         2: y<=c;
         3: y<=d;
         default: y<=8'bx;
     endcase
endmodule
```

Example 6.27



(a)

Synthesized circuit



(b)

Ex 6.28 Registered output of sequence recognizers

```
module Seq_Rec_3_1s_Mealy (D_out, D_in, En, clk, reset);
    output D_out;
    input D_in, En;
    input clk, reset;
    parameter S_idle=0;   // Binary code
    parameter S_0=1;
    parameter S_1=2;
    parameter S_2=3;
    reg [1:0] state, next_state;
    reg D_out_reg;
    always @ (negedge clk)
      if (reset==1) state<=S_idle,D_out_reg<=0;
       else
       begin state<=next_state;
          D_out_reg<= ((state==S_2)&&(next_state==S_2)&& (D_in==1));
       end
```

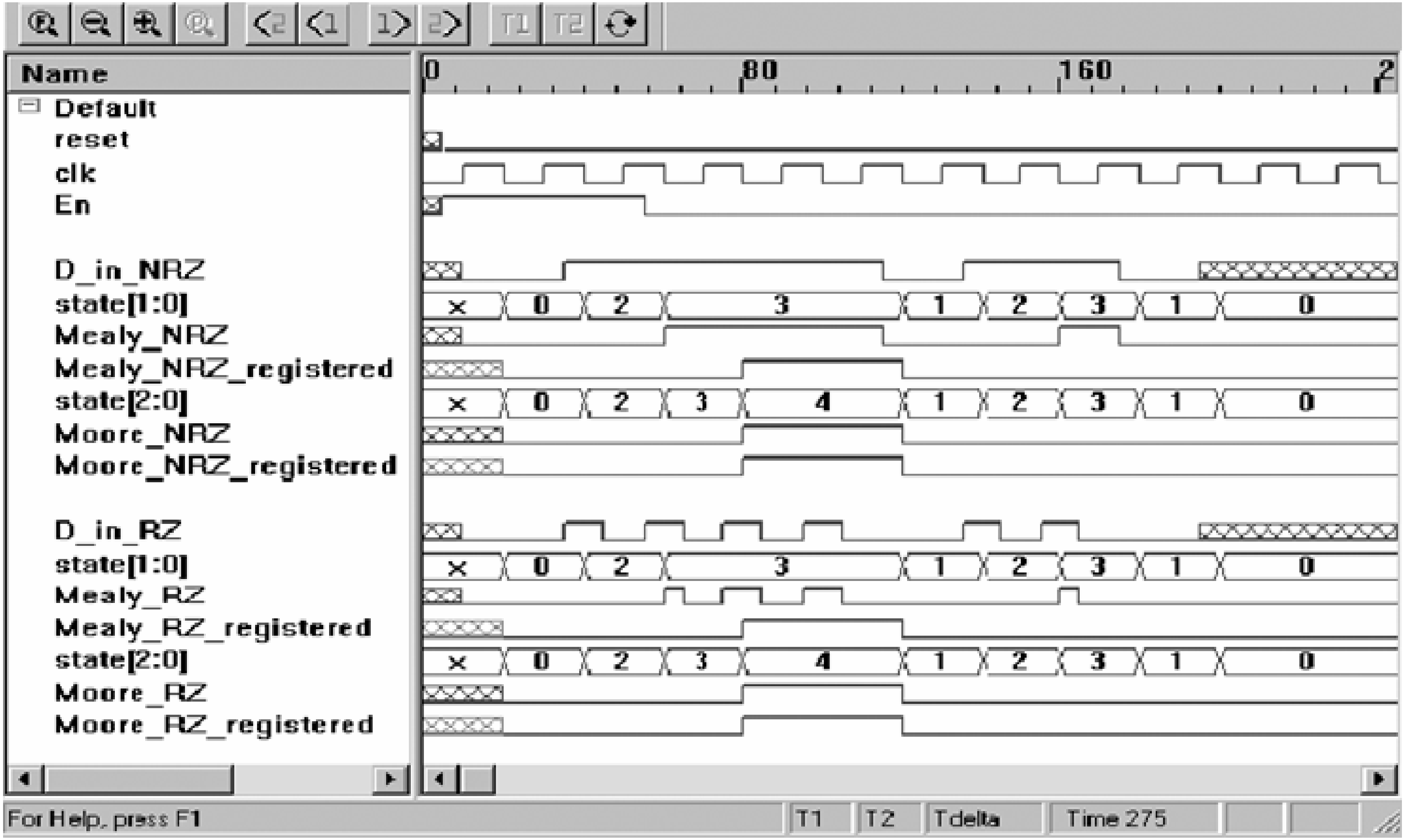
```
always @ (state or D_in) begin
       case (state)
            S_idle:   if ((En==1)&&(D_in==1)) next_state=S_1;
                         else
                      if ((En==1)&&(D_in==0)) next_state=S_0;
                         else                              next_state=S_idle;
             S_0:     if (D_in==0) next_state=S_0; else
                      if (D_in==1) next_state=S_1;
                          else          next_state=S_idle;
             S_1:     if (D_in==0) next_state=S_0; else
                      if (D_in==1) next_state=S_2;
                          else          next_state=S_idle;
             S_2,S_3: if (D_in==0) next_state=S_0; else
                        if (D_in==1) next_state=S_3;
                           else          next_state=S_idle;
             default:                    next_state=S_idle;
          endcase
       end
assign D_out=D_out_reg;      // Mealy output
endmodule
```

Moore-type sequence recognizer

```
module Seq_Rec_3_1s_Moore (D_out, D_in, En, clk, reset);
    output D_out;
    input D_in, En;
    input clk, reset;
    parameter S_idle=0;   // Binary code
    parameter S_0=1;
    parameter S_1=2;
    parameter S_2=3;
    reg [1:0] state, next_state;
    reg D_out_reg;
    always @ (negedge clk)
      if (reset==1) begin state<=S_idle; D_out_reg<=0; end
      else
       begin
            state<=next_state; D_out_reg<=(next_state==S_3);
       end
```

```
always @ (state or D_in) begin
      case (state)
           S_idle:   if ((En==1)&&(D_in==1)) next_state=S_1; else
                     if ((En==1)&&(D_in==0)) next_state=S_0;
                         else                              next_state=S_idle;
            S_0:     if (D_in==0) next_state=S_0; else
                     if (D_in==1) next_state=S_1;
                        else         next_state=S_idle;
            S_1:     if (D_in==0) next_state=S_0; else
                     if (D_in==1) next_state=S_2;
                         else         next_state=S_idle;
            S_2,S_3: if (D_in==0) next_state=S_0; else
                       if (D_in==1) next_state=S_3;
                          else         next_state=S_idle;
            default:                  next_state=S_idle;
         endcase
      end
assign D_out=D_out_reg;      // Moore output
endmodule
```

Simulation result



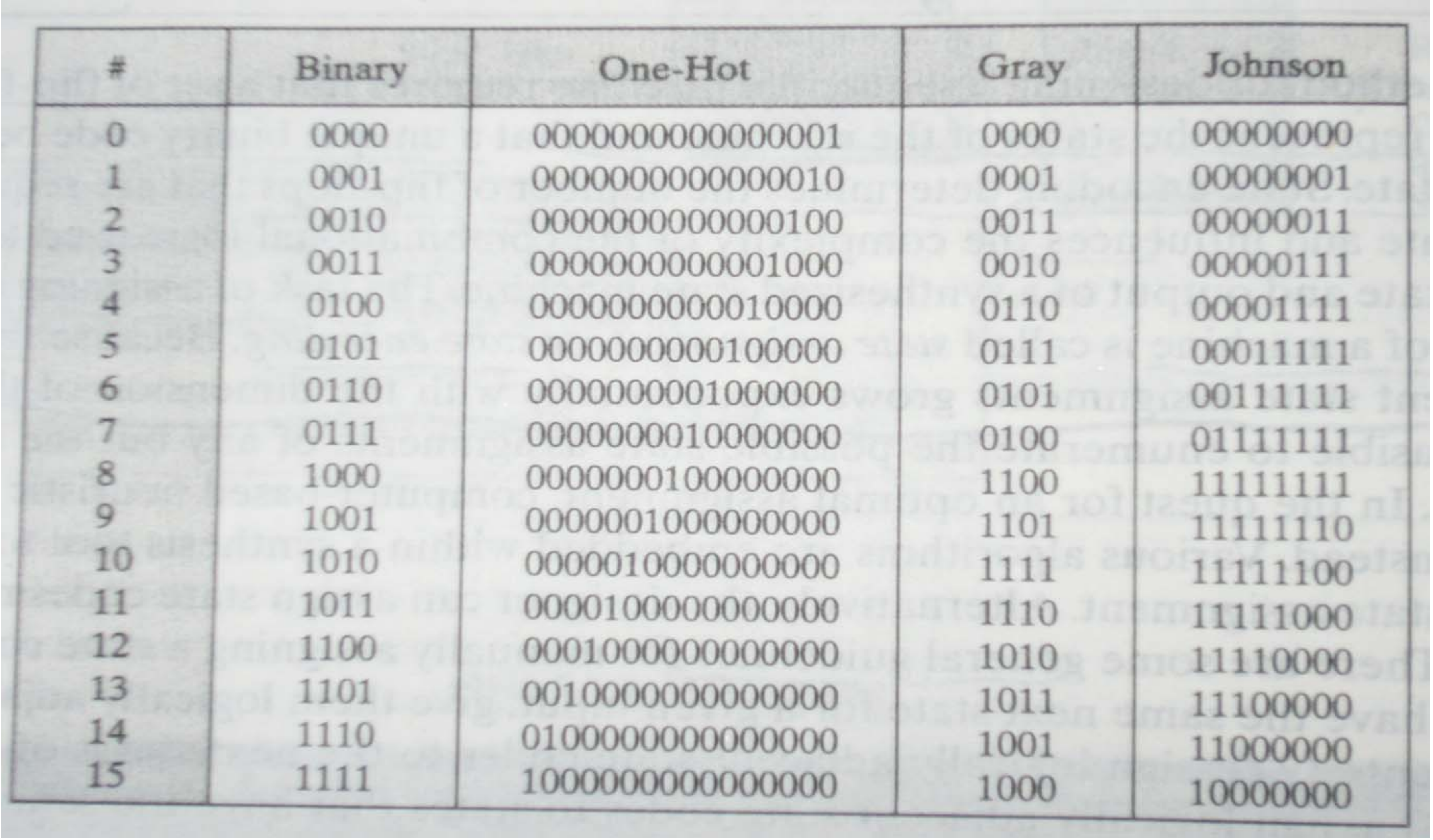
6.8 State encoding

- A set of flip-flops represent the states of the machine and a unique binary code be assigned to each state
- **Assigning a code to the states of a machine is called state assignment (state encoding)**
- **State encoding determines the number of flip-flops**

General guideline of state assignment

- If two states have the same next state for a given input, give them logically adjacent state assignments,
- Assign logically adjacent state codes to states that have the same output for a given input.

Commonly used state-assignment codes



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

Number of state assignment

- The number of flip-flops must be sufficient to represent the number of states as a binary number
- A state machine with N states will require at least $\log_2 N$ flip-flops

6.9 Synthesis of implicit state machines, registers, and counters

- **An implicit state machine has one or more clock-synchronized event-control expressions in a behavior**
- Multiple event-control expressions of an implicit FSM will **be synchronized to the same edge of the same clock**

6.9.1 Implicit state machines

- **The state is defined implicitly by the evolution of activity within a cyclic behavior**
- May contain multiple clock-synchronized event-control expressions within the same behavior
- Being a more general style of design than explicit FSMs
- **limitation:** each state may be entered from only one other state

The multiple event-control expressions separate the activity of the behavior into distinct clock cycles of the machine

```
always @ (posedge clk)     // Synchronized event before first
                           // assignment
  begin
   reg_a<=reg_b;    // Executes in first clock cycle
   reg_c<=reg_d;    // Executes in first clock cycle
   @ (posedge clk)          // Begins second clock cycle
   begin
     reg_g<=reg_f; // Executes in second clock cycle
     reg_m<=reg_r; // Executes in second clock cycle
   end
  end
```

6.9.2 Synthesis of counters

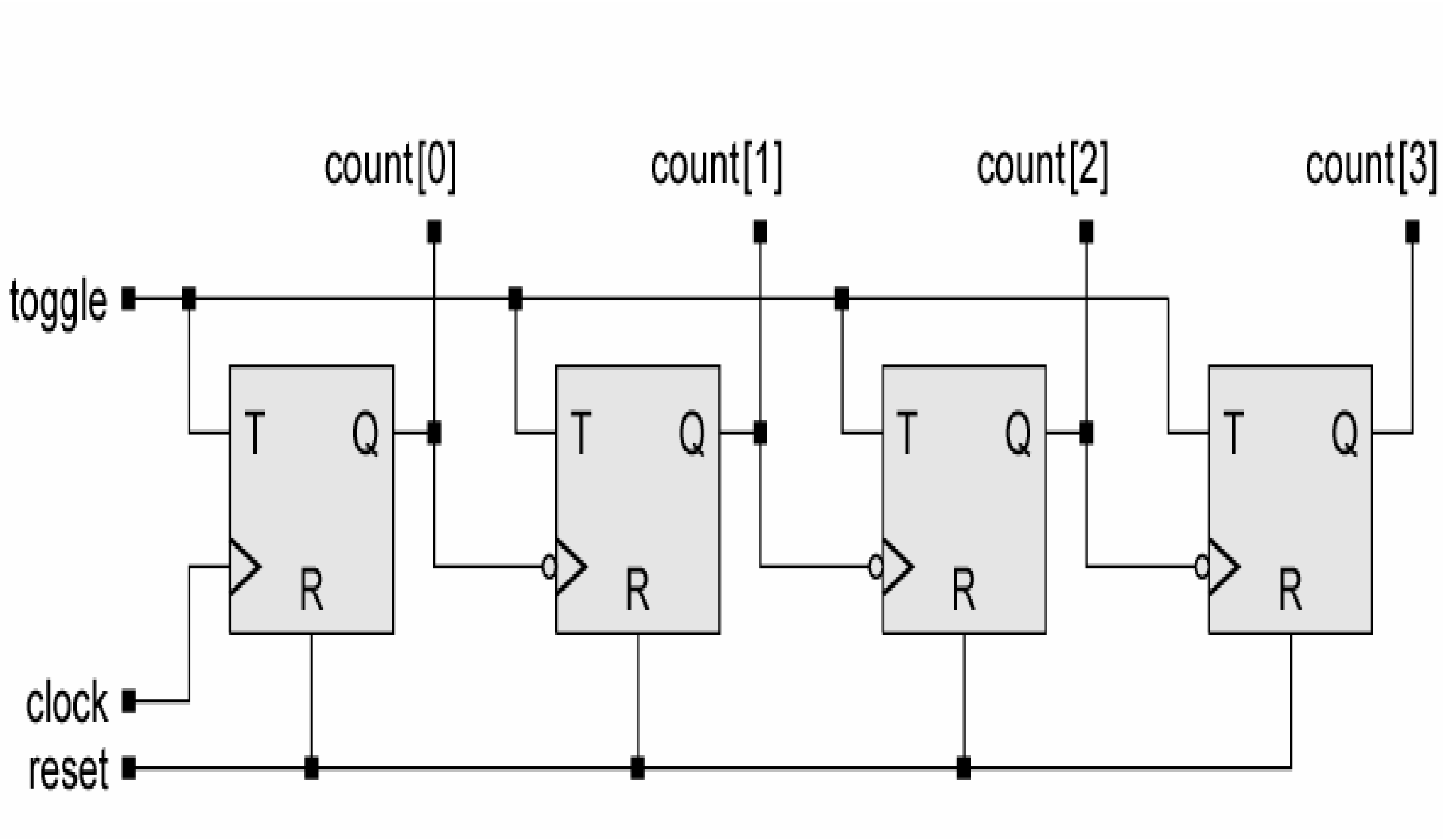
- Synthesize a variety of counters and shift registers as single-cycle implicit state machines

Ex 6.29 a 4-bit ripple counter

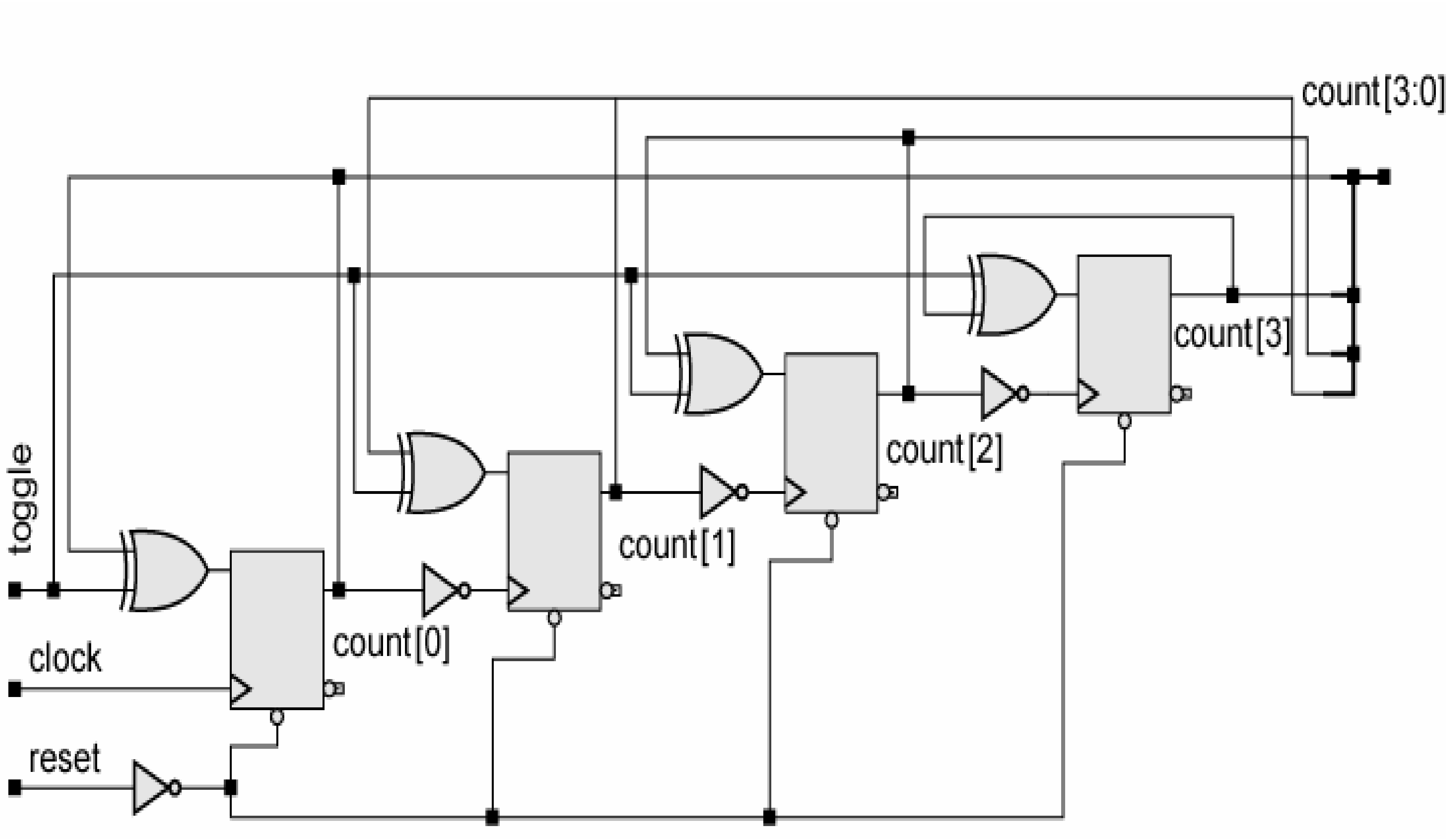
1. can be implemented with **T-type flip-flops**
2. **has limited practical application**
3. the output count is also **subject to glitches during the transitions**

```
module ripple_counter (count, toggle, clock, reset,);
 output  [3: 0]     count;
 input    toggle, clock, reset;
 reg       [3: 0]     count;
 wire      c0, c1, c2;
       assign c0 = count[0];
       assign c1 = count[1];
       assign c2 = count[2];
       always @  (posedge reset or posedge clock)
         if (reset == 1'b1) count[0] <= 1'b0; else
           if (toggle == 1'b1) count[0] <= ~count[0];
       always @  (posedge reset or negedge c0)
         if (reset == 1'b1) count[1] <= 1'b0; else
           if (toggle == 1'b1) count[1] <= ~count[1];
       always @  (posedge reset or negedge c1)
         if (reset == 1'b1) count[2] <= 1'b0; else
           if (toggle == 1'b1) count[2] <= ~count[2];
       always @  (posedge reset or negedge c2)
         if (reset == 1'b1) count[3] <= 1'b0; else
           if (toggle == 1'b1) count[3] <= ~count[3];
 endmodule
```

Structure of a 4-bit ripple counter



Synthesized circuit for 4-bit ripple counter



6.9.3 Synthesis of registers

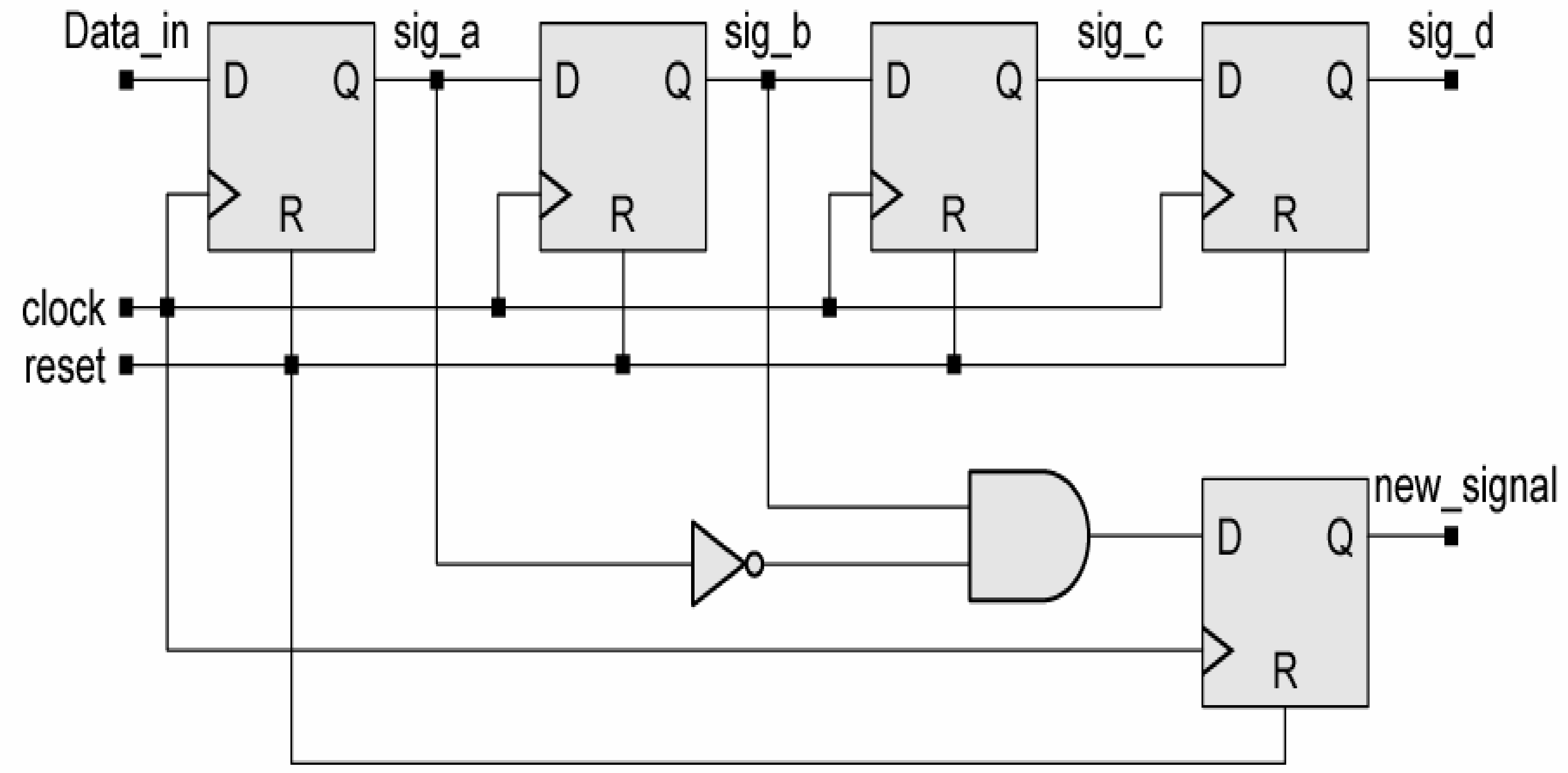
- Storage elements in a sequential machine can be implemented with flip-flops or with latches, **depending on the clocking scheme used by the machine**

Ex 6.31: Shift register with combinational logic forming the register variable

```
module shifter_1 (sig_d, new_signal, Data_in, clock, reset);
    output sig_d, new_signal;
    input Data_in, clock, reset;
    reg sig_a, sig_b, sig_c, sig_d, new_signal;
    always @ (posedge reset or posedge clock)
      begin if (reset==1'b1)
         begin
              sig_a<=0;
              sig_b<=0;
              sig_c<=0;
              sig_d<=0;
     end else
    begin
      sig_a<=Data_in;
      sig_b<=sig_a;
      sig_c<=sig_b;
      sig_d<=sig_c;
      new_signal<=(~sig_a) & sig_b;
    end
  end
endmodule
```

How many flip-flops?

Generic structure

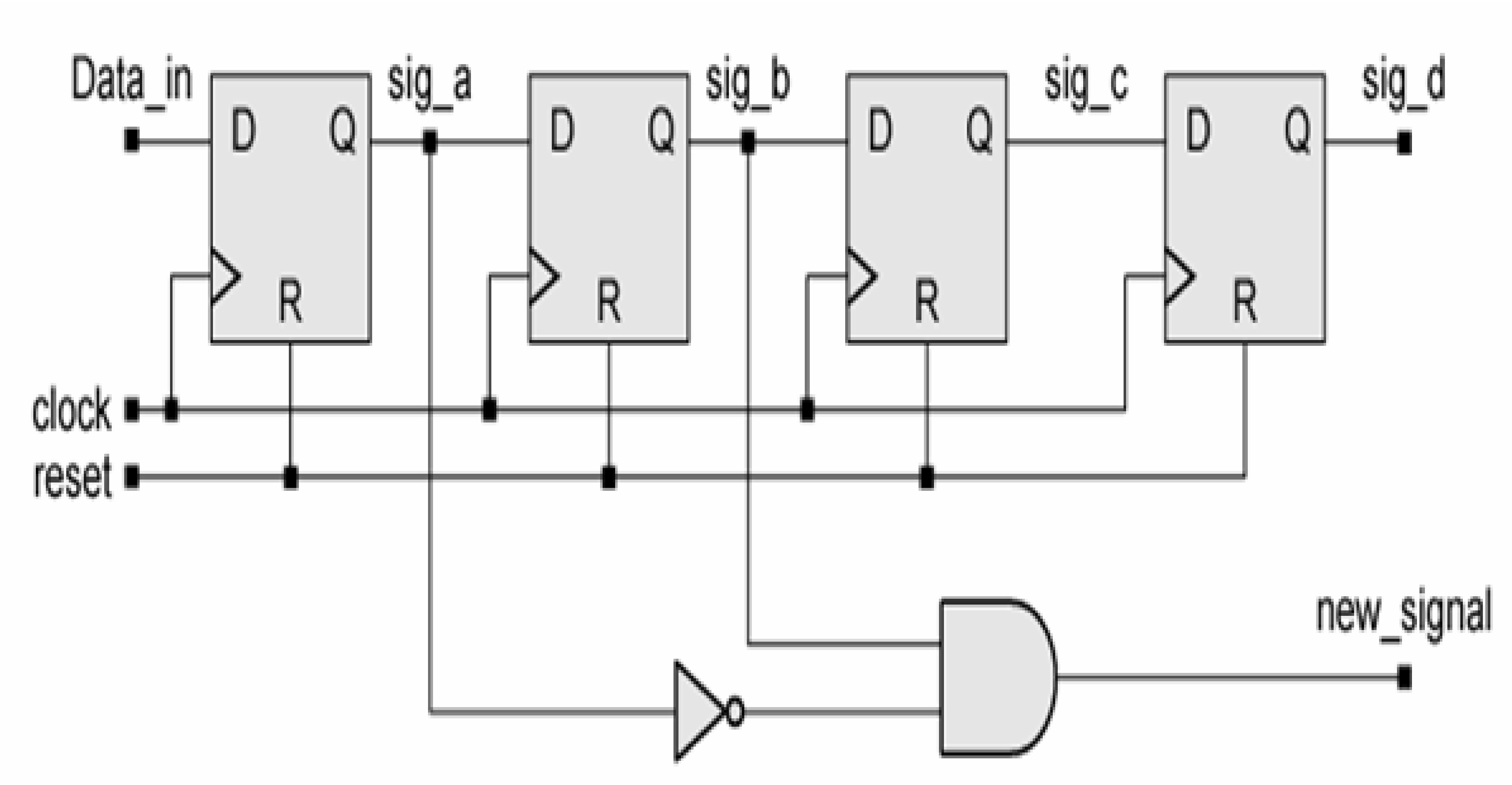


Example 6.32

```
   module shifter_2 (sig_d, new_signal, Data_in, clock, reset);
   output sig_d, new_signal;
   input Data_in, clock, reset;
   reg sig_a, sig_b, sig_c, sig_d, new_signal;
   always @ (posedge reset or posedge clock)
    begin if (reset==1'b1)
      begin
          sig_a<=0;
          sig_b<=0;
          sig_c<=0;
          sig_d<=0;
      end
     else  begin
               sig_a<=shift_input;
               sig_b<=sig_a;
               sig_c<=sig_b;
               sig_d<=sig_c;
           end
    end
  assign new_signal<=(~sig_a) & sig_b;
endmodule
```

How many flip-flops?

In shifter_2, new_signal is formed outside of the behavior in a continuous assignment, and is synthesized as the combinational logic



Ex 6.33 Accumulator-1

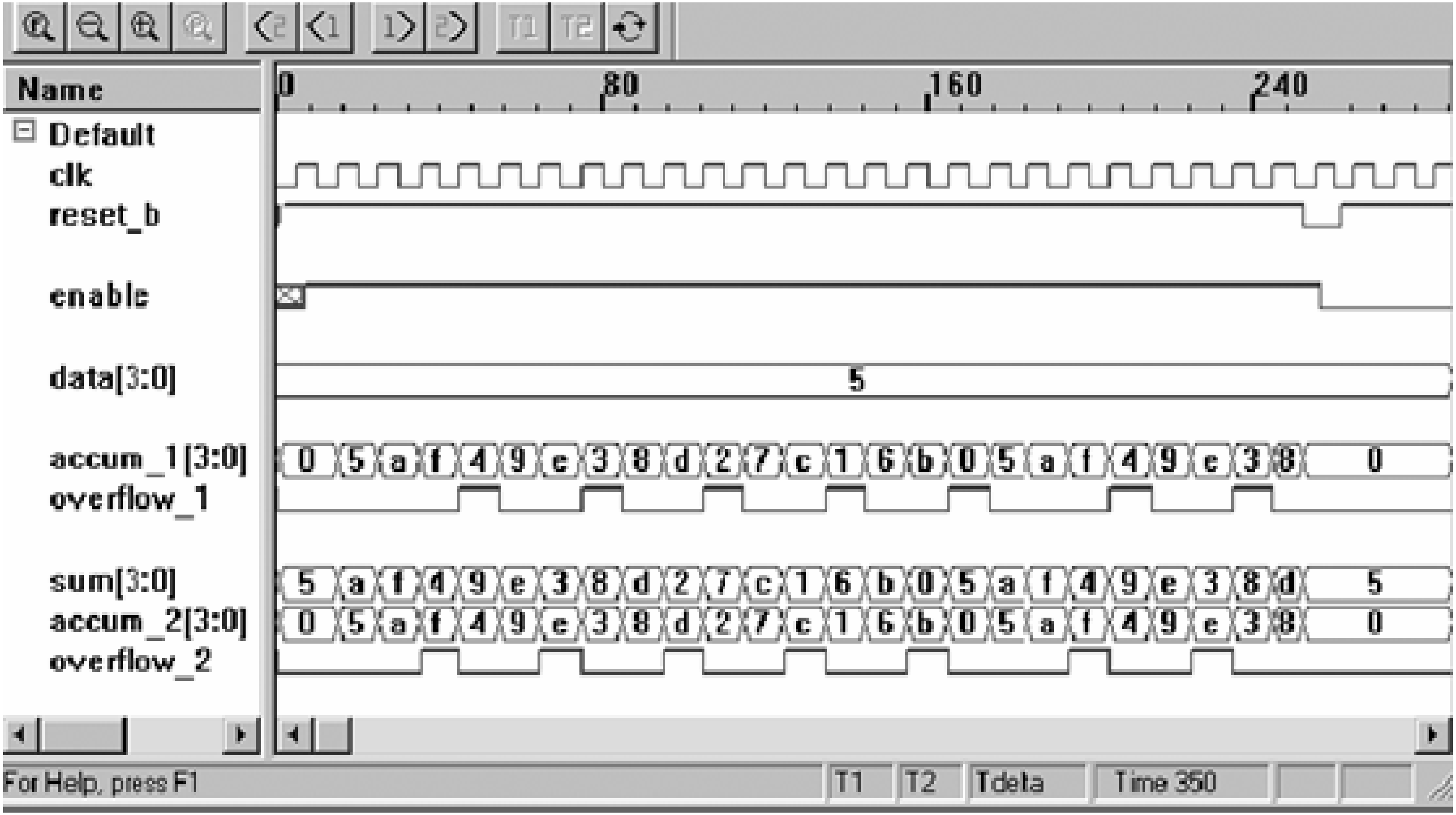
```
module Add_Accum_1 (accum, overflow, data,
   enable, clk, reset_b);
  output [3:0] accum;
  output overflow;
  input [3:0] data;
  input enable, clk, reset_b;
  reg accum, overflow;
  always @ (posedge clk or negedge reset_b)
   if (reset_b==0)
     begin accum<=0; overflow<=0; end
   else if (enable)  {overflow, accum} <=accum+data;
              //concatenate the carry and sum into one
endmodule
```

Accumulator-2

```
module Add_Accum_2
                (accum, overflow, data, enable, clk, reset_b);
  output [3:0] accum;
  output overflow;
  input [3:0] data;
  input enable, clk, reset_b;
  reg overflow;
  wire [3:0] sum;
  assign {overflow, sum}=accum+data;
      //concatenate the carry and sum into one
  always @ (posedge clk or negedge reset_b)
    if (reset_b==0) begin accum<=0;
    else if (enable) accum<=sum;
endmodule
```

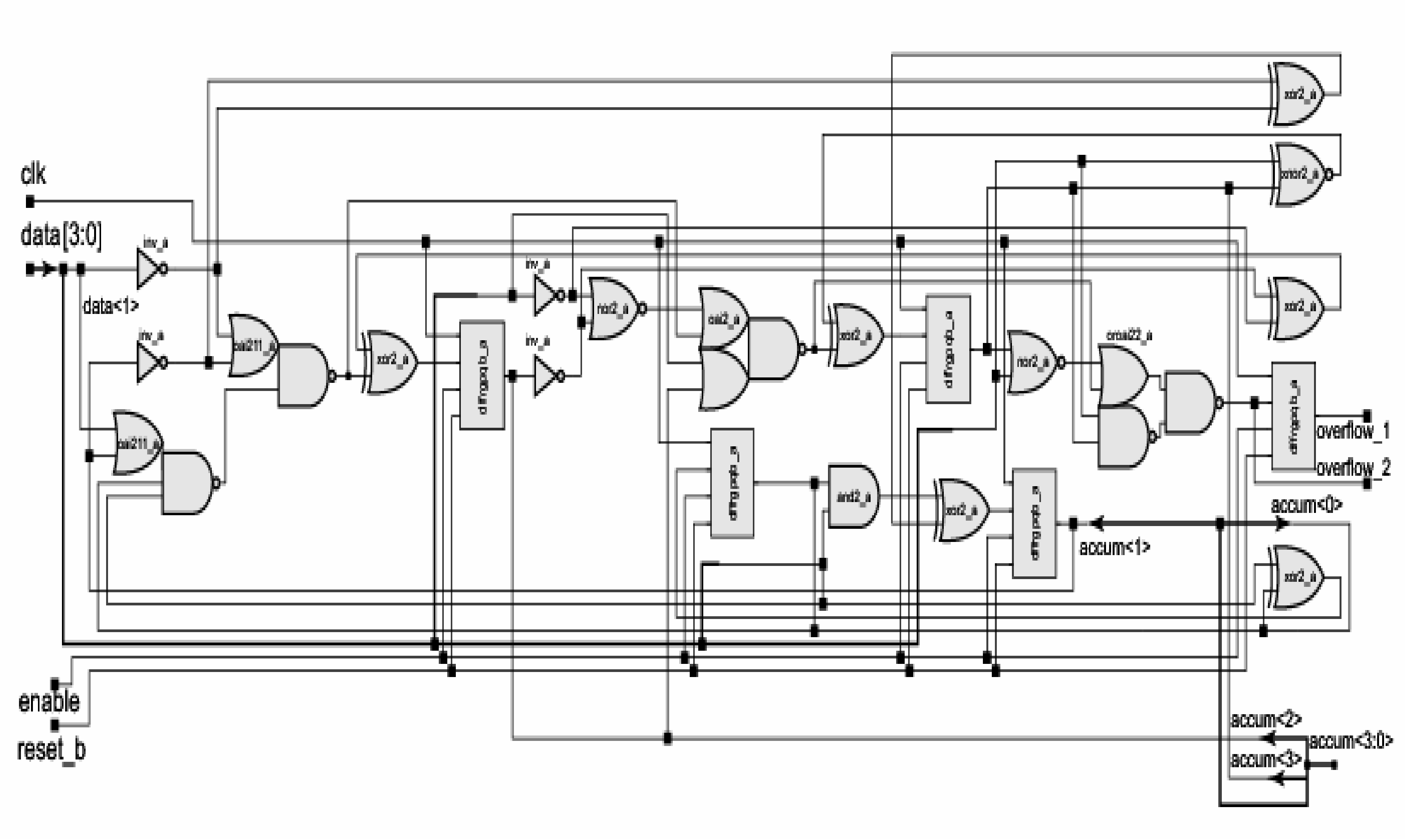
ACC 1-2's difference ?

Simulation result


Name
0
80
160
240
Default
clk
reset_b
enable
data[3:0]
5
accum_1[3:0]
0 5 a f 4 9 e 3 8 d 2 7 c 1 6 b 0 5 a f 4 9 e 3 8 0
overflow_1
sum[3:0]
5 a f 4 9 e 3 8 d 2 7 c 1 6 b 0 5 a f 4 9 e 3 8 d 5
accum_2[3:0]
0 5 a f 4 9 e 3 8 d 2 7 c 1 6 b 0 5 a f 4 9 e 3 8 0
overflow_2
For Help, press F1
T1
T2
Tdelta
Time 350

(a)

Synthesis circuit



6.10 Resets

- Every sequential module in a design should have a reset signal.
- Otherwise, the initial state of the machine and its operation cannot be controlled
- **If an initial state cannot be assigned, the machine cannot be tested for manufacturing defects**

6.11 Synthesis of gated clocks and clock enables

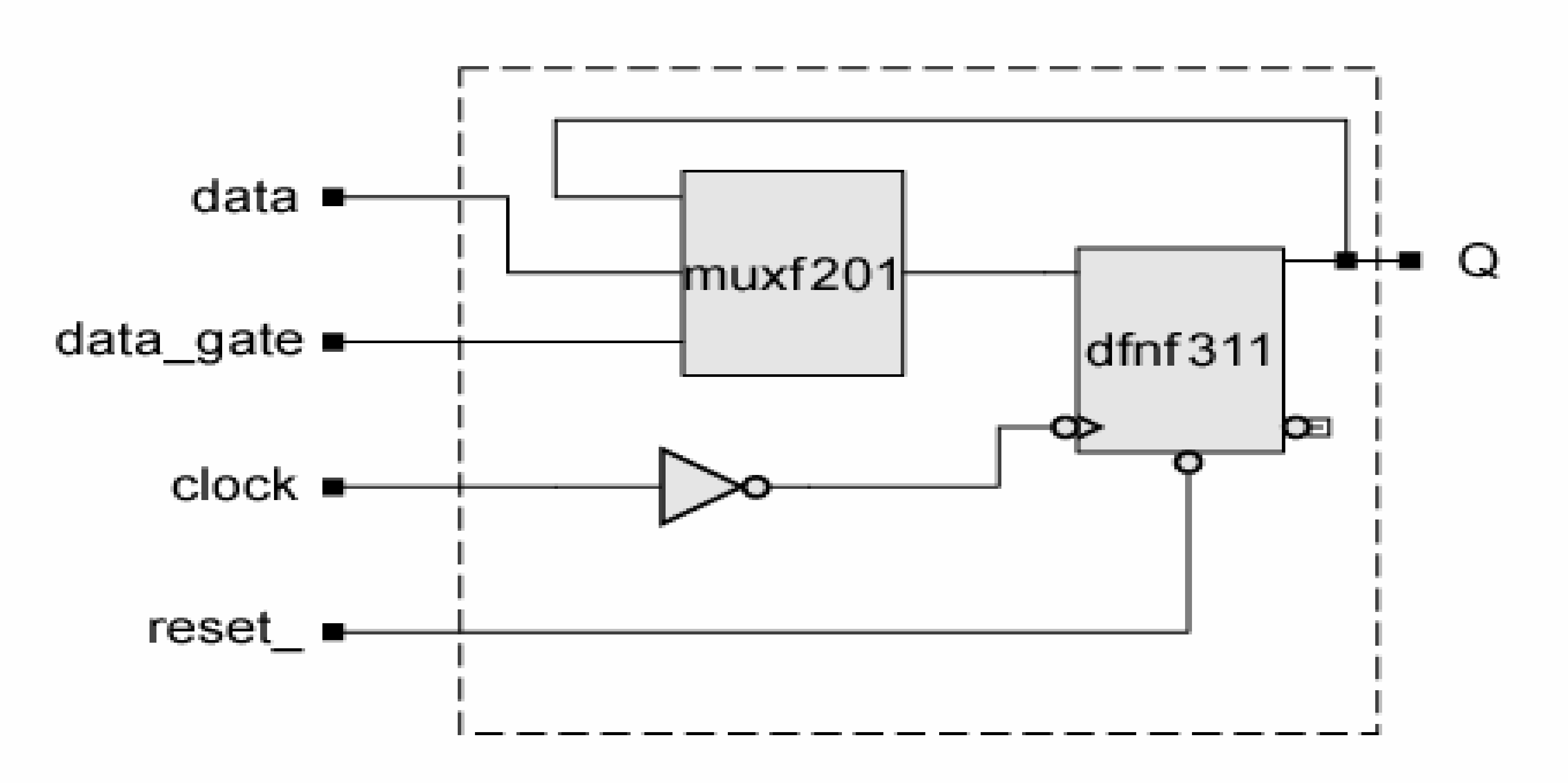
- Designers **avoid using gated clocks**
- Low-power designs deliberately disable clocks **to reduce or eliminate power wasted** by useless switching of transistors
- **Improperly gated clocks can cause the clock signal to violate a flip-flop’s constraint on the minimum width of the clock pulse**

The recommended way to write a Verilog description that synthesizes a gated clock

```
module best_gated_clock
            (clock, reset_, data_gate, data, Q);
   input clock, reset_, data, data_gate;
   output Q;
   reg Q;
   always @ (posedge clock or negedge reset_)
     if (reset_==0) Q<=0;
      else if (data_gate) Q<=data;
                                    // infers storage
endmodule
```

Synthesis of Gated Clocks

```
always @ (posedge clk)
  if (enable == 1) q_out <= Data_in;
```



6.12 Anticipating the results of synthesis

- The results of synthesis should not be taken at face value.
- It is advisable to anticipate what the synthesis process will produce and then examine the results against those expectations
- The section will cover some of **the basic rules that will help the designer anticipate the results of synthesis and write Verilog descriptions** that infer the desired results

6.12.1 Synthesis of data types

- Nets that are primary inputs or outputs will be retained in the design
- **Internal nets may be eliminated** by the action of the synthesis tool
- Integers are stored as 32-bit data objects
- Use sized numbers (e.g., 8’b0110_1110) to reduce the size of the register required to hold a parameter
- ***x*** **or** ***z*** **have no hardware counterpart**

6.12.2 Operator grouping

- All Verilog operators may be used in expressions forming a binary or Boolean value
- Some operators may be treated in a special way by the technology mapper
- **+,-,<,>, = may be mapped directly to a library element**
- Otherwise, the synthesis tool will convert the operator into an equivalent set of Boolean equations
- **Some operators must be restricted for successful synthesis**

Restricted operators

- **Shift operators (<<,>>) are synthesizable, when the shift is by a constant number of bits**
- The reduction, bitwise, and logical connective operators are each equivalent to operations performed by a logic gate

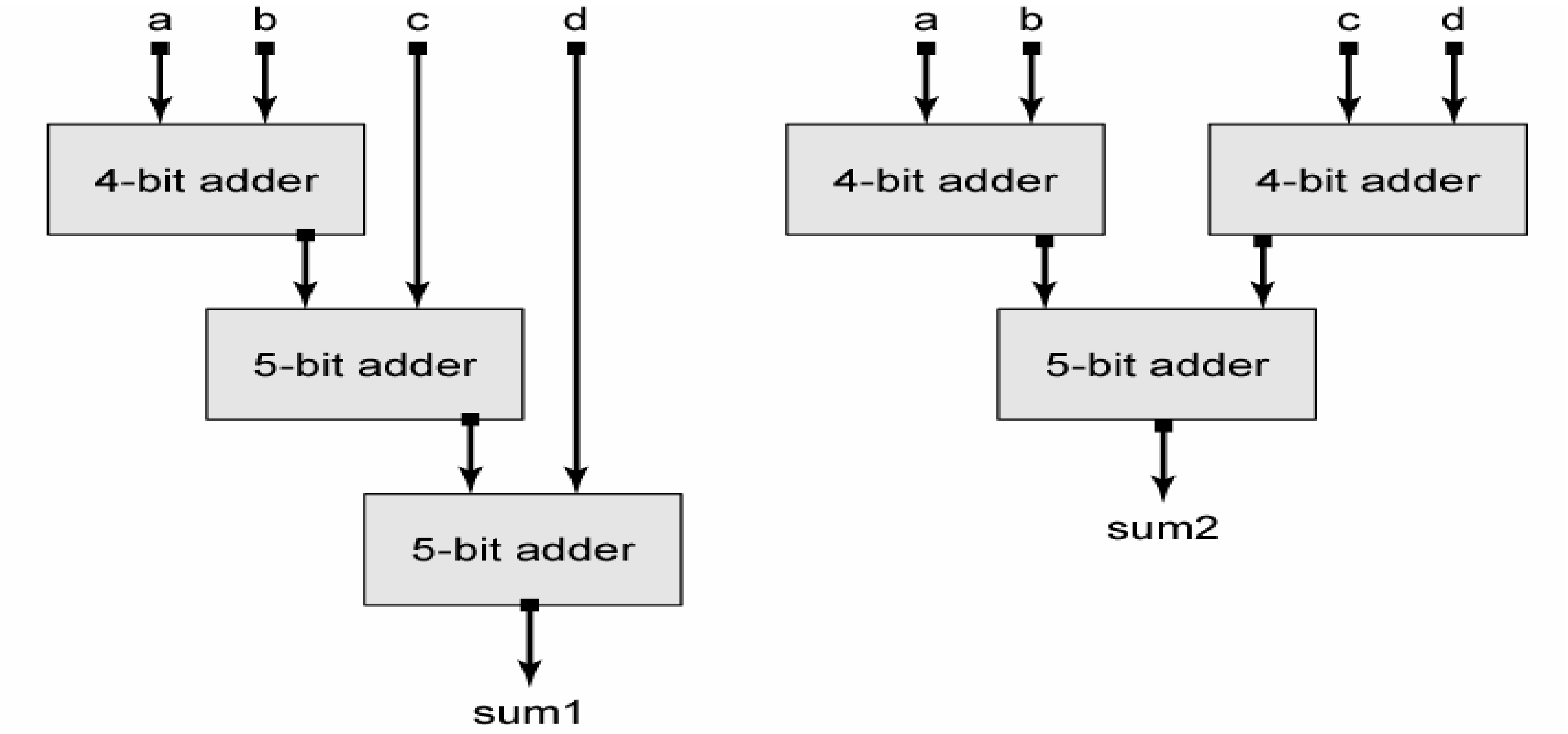
  (& ~& | ~| ^ ~^ ^~ reduction (~^ and ^~ are equivalent))
- **The conditional operator (?...:) synthesizes into library muxes or into gates that implement the functionality of a mux**
- When an expression has multiple operators, the architecture of the synthesized result will reflect the parsing of the compiler and the precedence of the operators

Ex 6.35 Operator group

```
module operator_group (sum1, sum2, a, b, c, d);
  output [4:0] sum1, sum2;
  input [3:0] a, b, c, d;

  assign sum1=a+b+c+d;
  assign sum2=(a+b)+(c+d);
endmodule
```

For speed,sum2 will be approximately 30% faster.

For low power, input d forming sum1 could be used for the signal that changes more frequently.



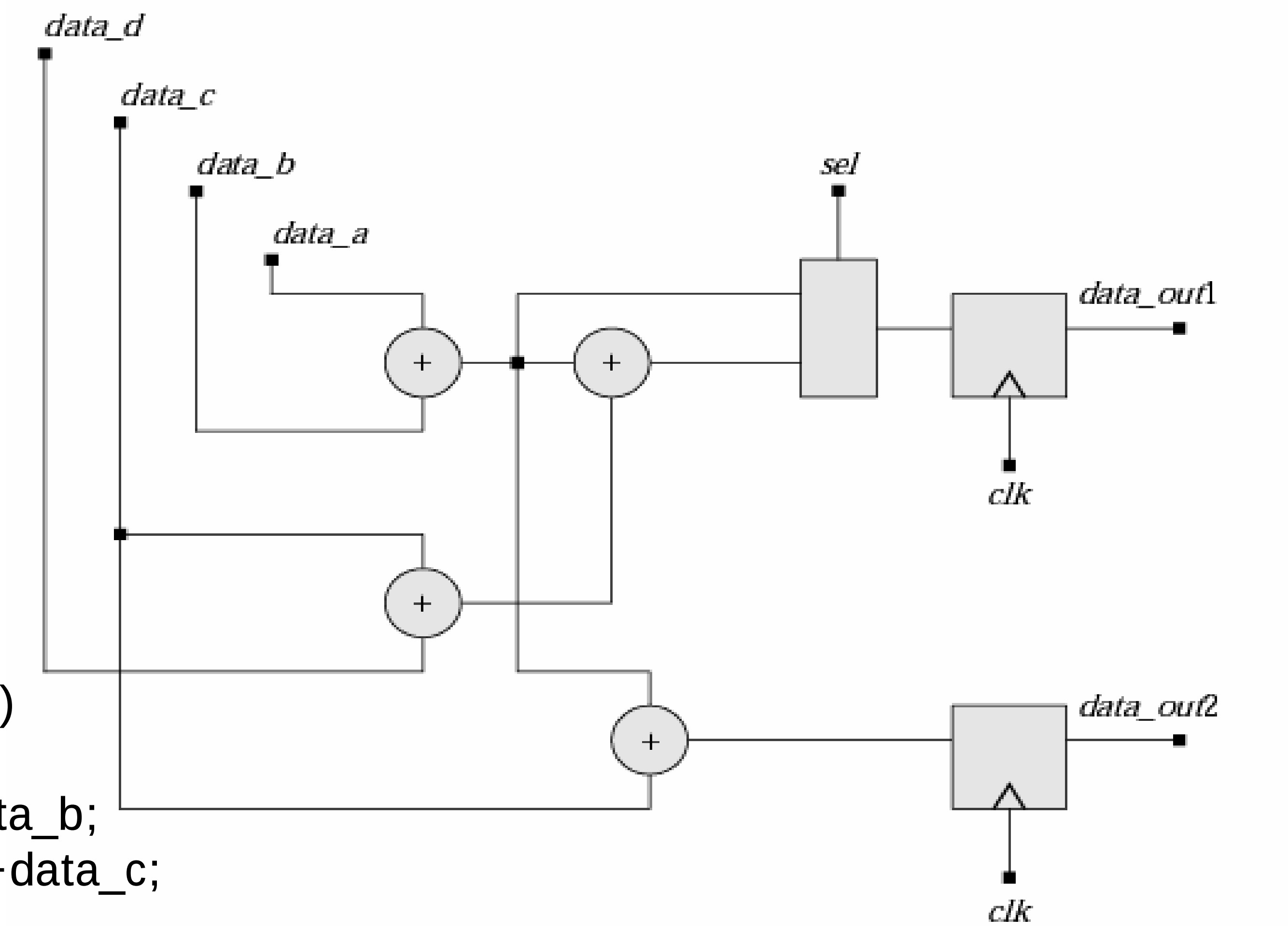
6.12.3 Expression substitution

- Synthesis tools perform expression substitution to determine the outcome of a sequence of procedural (blocking) assignments in a behavior
- If it is necessary for procedural assignments to be used, be aware that expression substitution will affect the result

Example 6.36

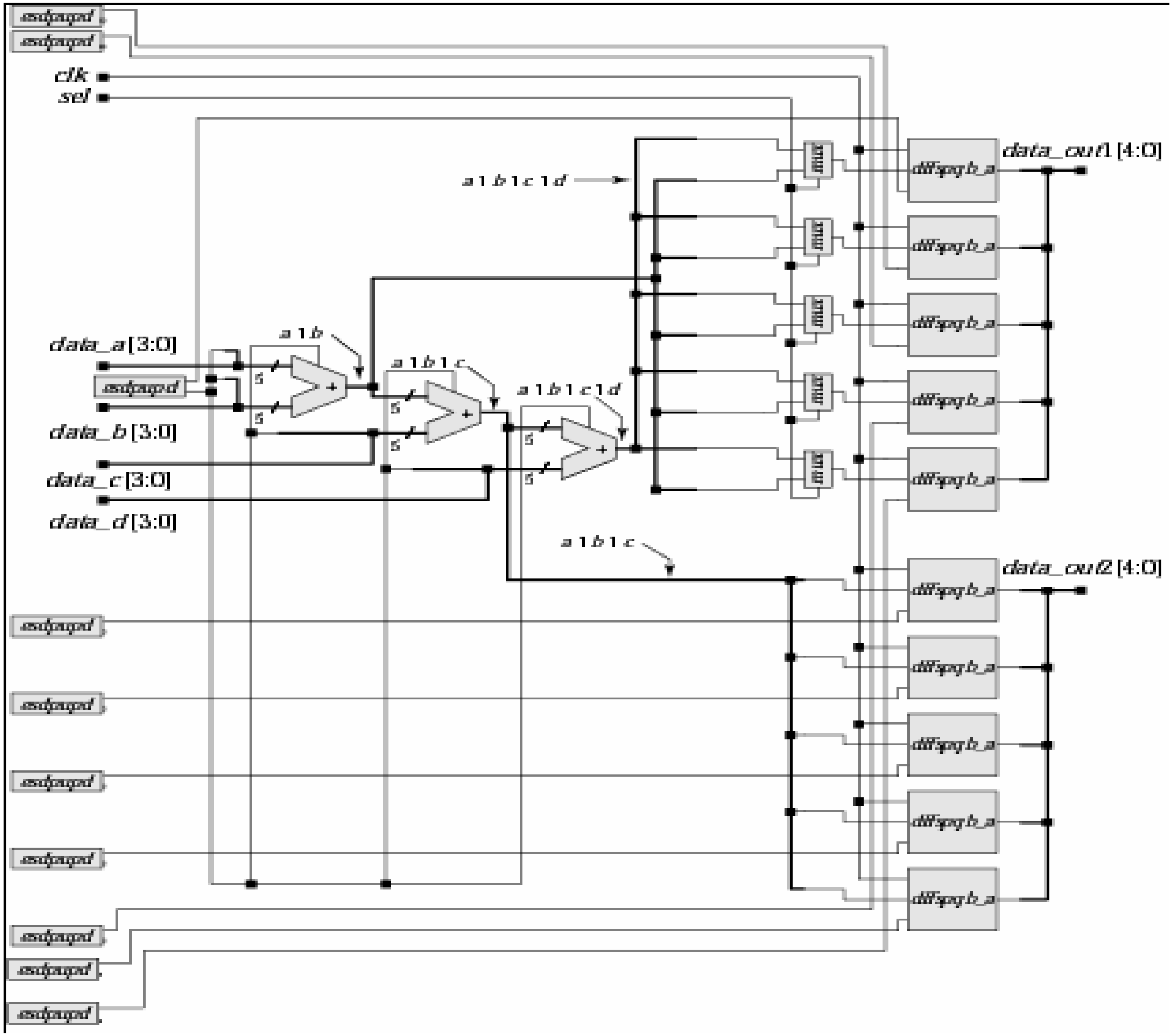
```
module multiple_reg_assign
        (data_out1, data_out2, data_a, data_b, data_c, data_d, sel, clk);
    output [4:0] data_out1, data_out2;
    input [3:0] data_a, data_b, data_c, data_d;
    input sel, clk;
    reg data_out1, data_out2;
    always @ (posedge clk)
      begin
            data_out1=data_a+data_b;
            data_out2=data_out1+data_c;
            if (sel==1’b0) data_out1=data_out2+data_d;
      end
endmodule
```

Dataflow structure



```
always @ (posedge clk)
 begin
   data_out1=data_a+data_b;
   data_out2=data_out1+data_c;
    if (sel==1'b0)
    data_out1=data_out2+data_d;
end
```

Synthesize to the circuit



```
module expression_sub
 (data_out1, data_out2, data_ a, data_b, data_c,
               data_d, sel, clk);
    output [4:0] data_out1, data_out2;
    input [3:0] data_a, data_b, data_c, data_d;
    input sel, clk;
    reg [4:0] data_out1, data_out2;
    always @ (posedge clk)
      begin
        data_out2=data_a+data_b+data_c;
           if (sel==1’b0)
               data_out1=data_a+data_b+data_c+data_d;
           else
               data_out1=data_a+data_b;
       end
endmodule
```

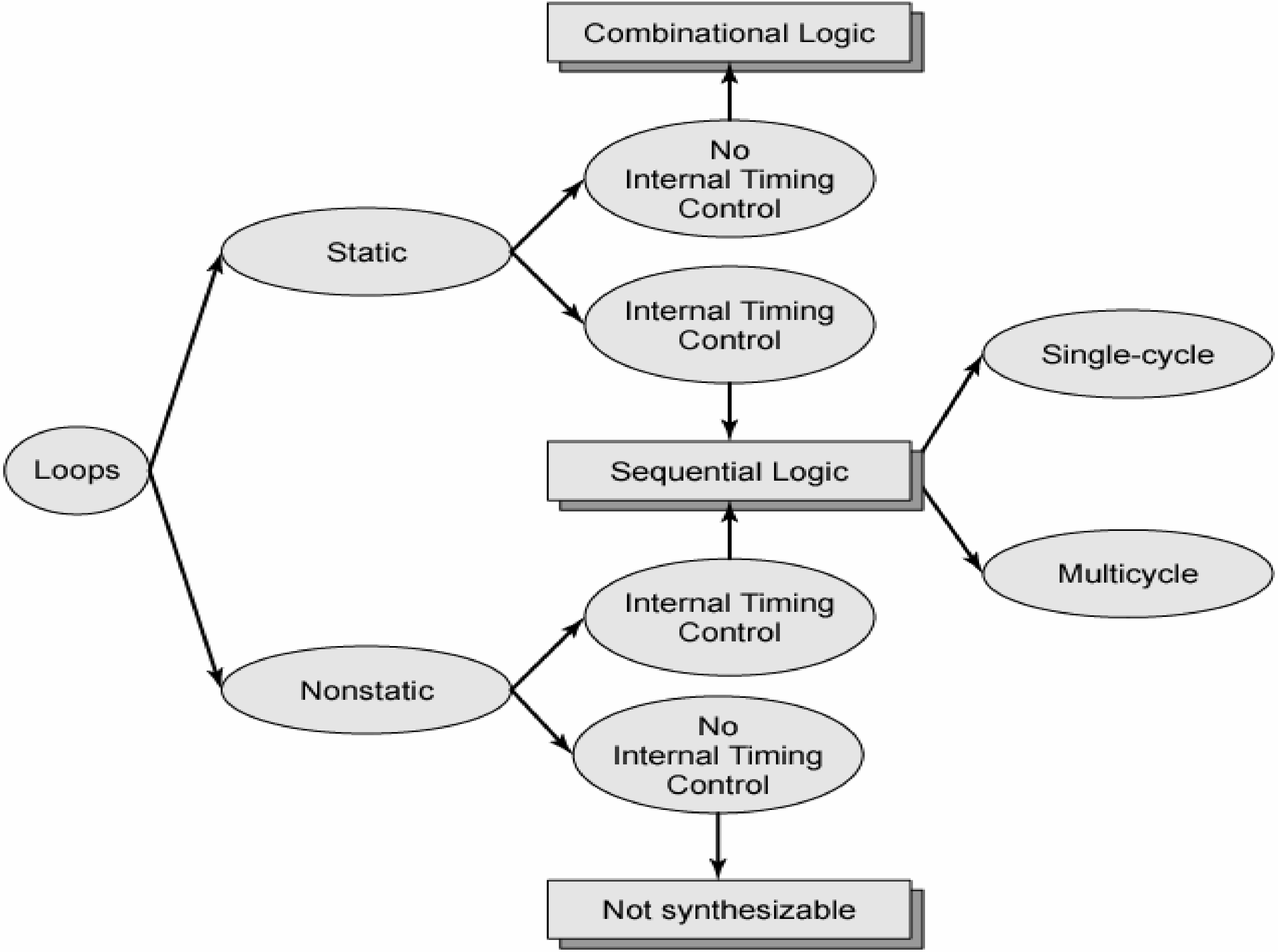
a recommended style which implements equivalent logic with the nonblocking operator

```
module expression_sub_nb
      (data_out1nb, data_out2nb, data_a, data_b, data_c, data_d, sel, clk);
    output [4:0] data_out1nb, data_out2nb;
    input [3:0] data_a, data_b, data_c, data_d;
    input sel, clk;
    reg [4:0] data_out1, data_out2;
    always @ (posedge clk)
      begin
        data_out2nb<=data_a+data_b+data_c;
            if (sel==1'b0)
              data_out1nb<=data_a+data_b+data_c+data_d;
            else
              data_out1nb<=data_a+data_b;
      end
endmodule
```

6.13 Synthesis of loops

- A loop is ***data-independent (static),*** if the number of its iterations can be determined by the compiler before simulation
- A loop is **data-dependent** if the number of iterations depends on some variable during operation
- A data-dependent loop may have a dependency on embedded timing controls

Possible loop structures



In principle

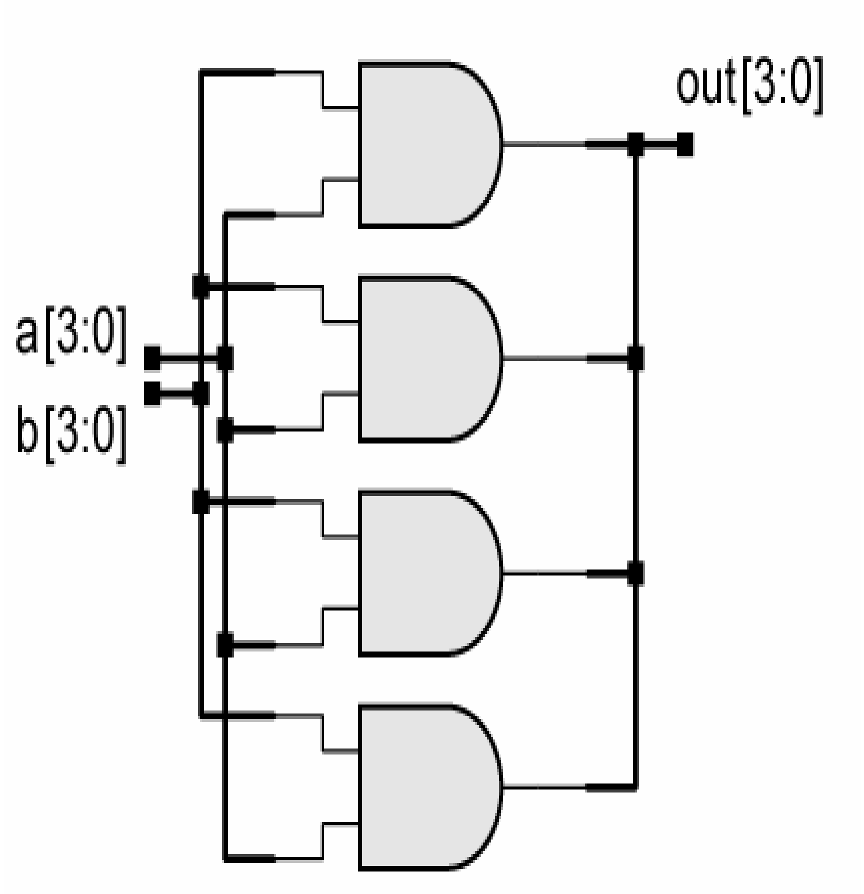
- Static loops can be synthesized using ***repeat***, ***for***, ***while***, and ***forever*** loop constructs
- A given vendor might choose to confine the descriptive style of a static loop to a particular construct
- Non-static loops that do not have internal timing controls cannot be synthesized

6.13.1 Static loops without embedded timing controls

- If a loop has **no internal timing controls and no data dependencies**, its computational activity is implicitly **combinational**
- The iterative computational sequence has a non-iterative counterpart obtained by unrolling the loop

Example 6.37

```
module for_and_loop_comb (out, a, b);
    output [3:0] out;
    input [3:0] a, b;
    reg [2:0] i;
    reg [3:0] out;
    wire [3:0] a, b;
    always @ (a or b)
      begin
        for (i=0;i<=3;i=i+1)
           out[i]=a[i] & b[i];
      end
endmodule
```



The unrolled loop is equivalent to:

```
always @ (a or b)
     begin
       for (i=0;i<=3;i=i+1)  out[i]=a[i] & b[i];
     end
```

```
out[0]=a[0] & b[0];
out[1]=a[1] & b[1];
out[2]=a[2] & b[2];
out[3]=a[3] & b[3];
```

For loop and *repeat* loop

- A ***for*** loop with static range is equivalent to a ***repeat*** loop with the same range
- **Some tools support only the *for* loop**

Example 6.38

```
module count_ones_a (bit_count, data, clk, reset);
   parameter data_width=4;
   parameter count_width=3;
   output [count_width-1:0] bit_count;
   input [data_width-1;0] data;
   input clk, reset;
   reg [count_width-1:0] count, bit_count, index;
   reg [data_width-1:0] temp;
   always @ (posedge clk)
    if (reset) begin count=0; bit_count=0; end
     else begin
      count=0; bit_count=0; temp=data;
        for (index = 0; index < data_width; index = index + 1)
         begin count=count+temp[0];  temp=temp >>1;  end
       bit_count=count;
      end
endmodule
```

6.13.2 Static loops with embedded timing controls

- If a static loop has an embedded edge-sensitive event-control expression, the computational activity of the loop is synchronized and distributed over one or more cycles of the clock
- As a result, **the behavior is that of an implicit state machine** in which each iteration of the loop occurs after a clock edge

Ex 6.39

```
module count_ones_b0 (bit_count, data, clk, reset);
    parameter  data_width = 4;
    parameter  count_width = 3;
    output     [count_width-1: 0]    bit_count;
    input      [data_width-1: 0]     data;
    input      clk, reset;
    reg     [count_width-1: 0]       count, bit_count;
    reg    [data_width-1: 0]         temp;
    integer                          index;
    always begin: wrapper_for_synthesis
    @ (posedge clk) begin: machine
    if (reset) begin bit_count = 0;  disable machine; end
      else
    count = 0; bit_count = 0; index = 0; temp = data;
    forever  @ (posedge clk)
     if (reset) begin bit_count = 0; disable machine; end
      else if (index < data_width-1)
               begin
                  count = count + temp[0];
                  temp = temp >> 1;
                  index = index + 1;
                 end
               else begin   //index=3,no shift
                        bit_count = count + temp[0];
                         disable machine;
                        end
    end // machine
  end // wrapper_for_synthesis
endmodule
```

Alternative approach-1

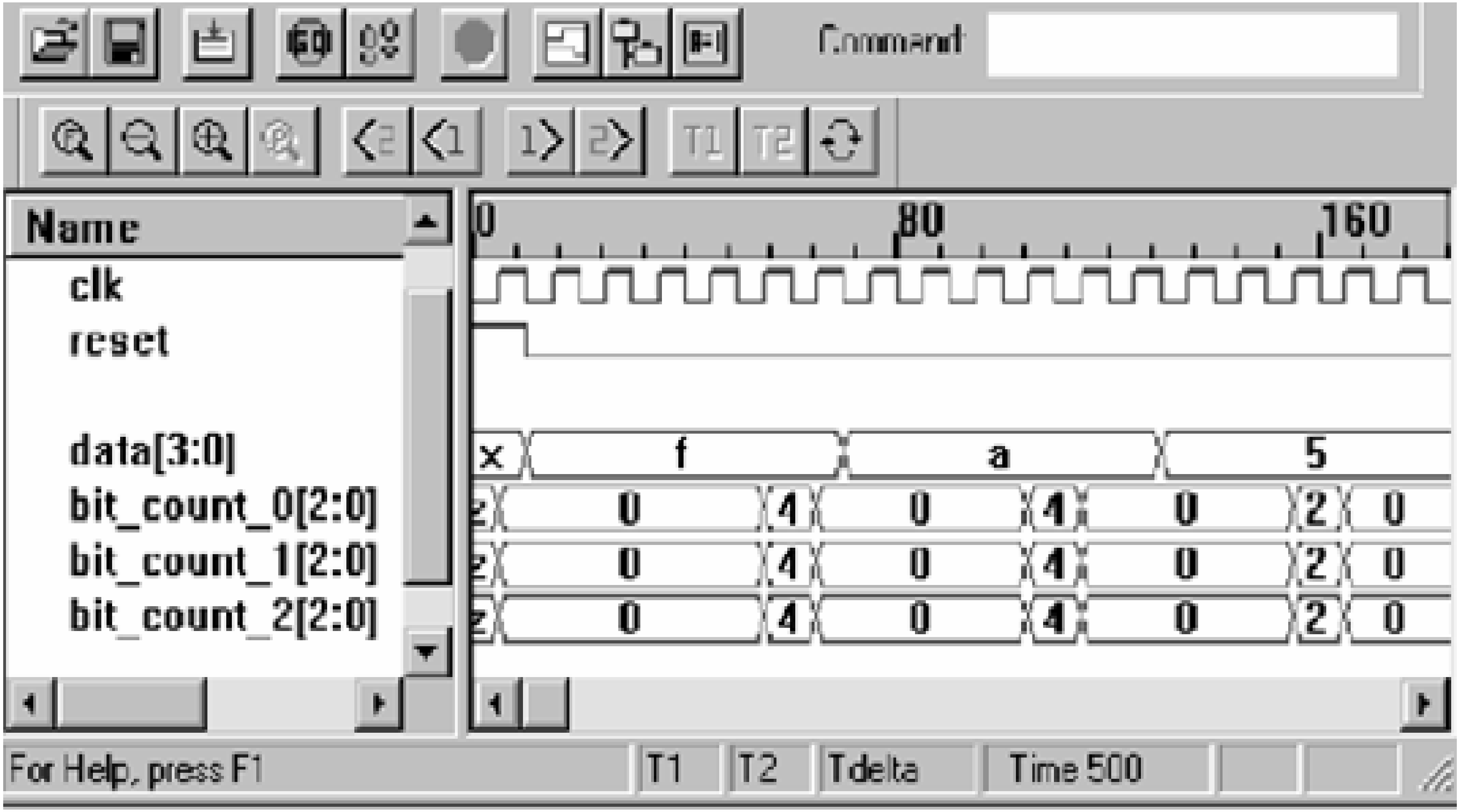
```
module count_ones_b1 (bit_count, data, clk, reset);
  parameter data_width=4;
  parameter count_width=3;
  output [count_width-1:0] bit_count;
  input [data_width-1;0] data;
  input clk, reset;
  reg [count_width-1:0] count, bit_count, index;
  reg [data_width-1:0] temp;
  integer index;
  always begin: wrapper_for_synthesis
   @ (posedge clk)
     begin: machine
     if (reset) begin bit_count=0; disable machine; end
     else  begin count=0; bit_count=0; index=0; temp=data;
           while (index<data_width-1) begin
             if (reset) begin bit_count=0; disable machine; end
             else if ((index<data_width) && (temp[0]))
               count=count+1;
                temp=temp>>1;
                index=index+1;
               @ (posedge clk)
              end
              if (reset) begin bit_count=0; disable machine; end
                  else bit_count=count;
                     disable machine;
           end
    end  // machine
  end  // wrapper_for_synthesis
endmodule
```

Alternative approach-2

```
module count_ones_b2 (bit_count, data, clk,
   reset);
   parameter data_width=4;
   parameter count_width=3;
   output [count_width-1:0] bit_count;
   input [data_width-1;0] data;
   input clk, reset;
   reg [count_width-1:0] count, bit_count, index;
   reg [data_width-1:0] temp;
   integer index;
   always begin: machine
```

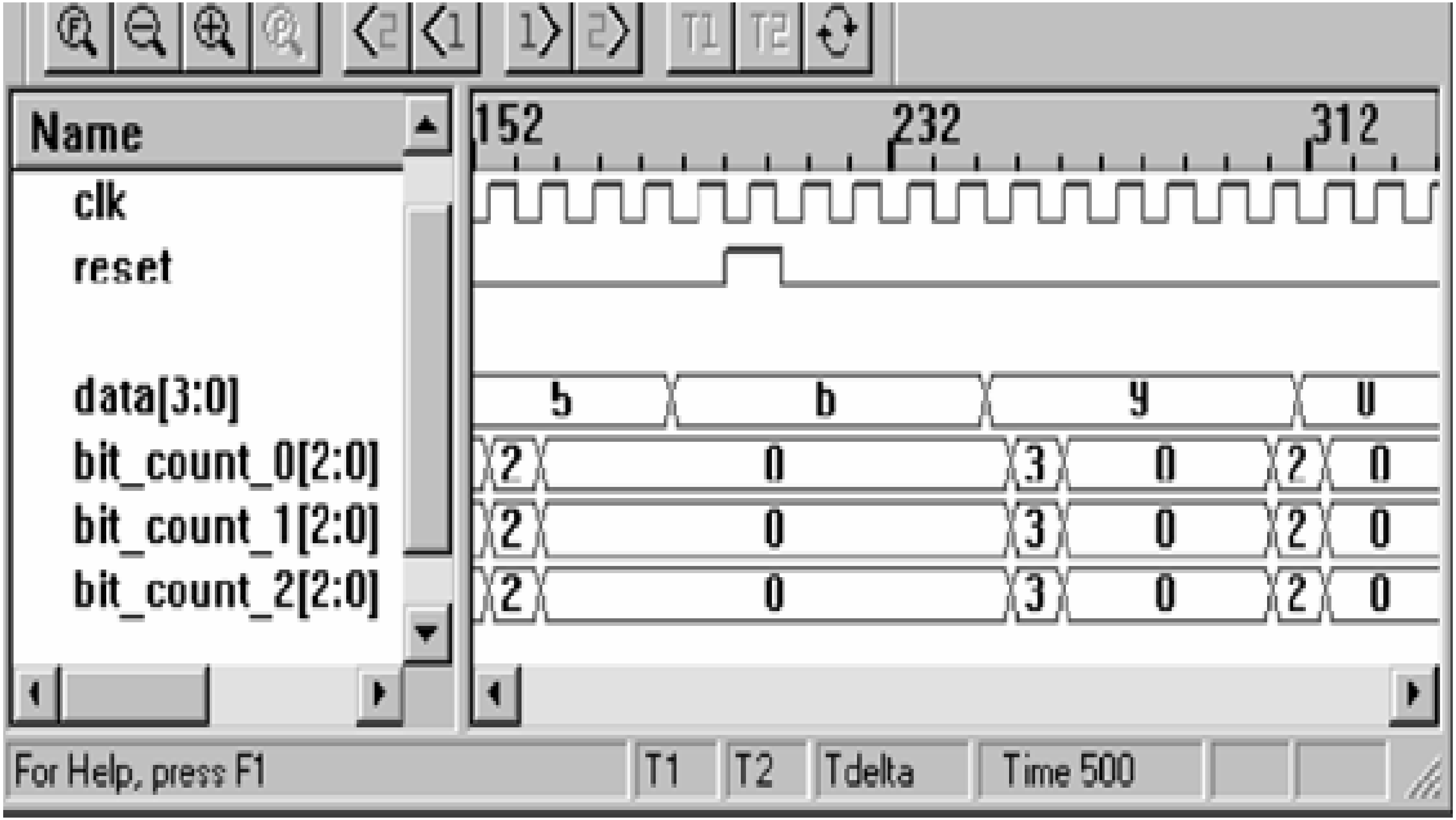
```
  for (index=0; index<=data_width; index=index+1)
    begin
     @ (posedge clk)
      if (reset) begin bit_count=0; disable machine; end
        else if (index==0) begin
         count=0; bit_count=0; index=0; temp=data; end
         else if (index<data_width) begin
                    count=count+temp[0];
                    temp=temp>>1; end
                  else bit_count=count+temp[0];
      end
    end  // machine
endmodule
```

Simulation result (reset begin)

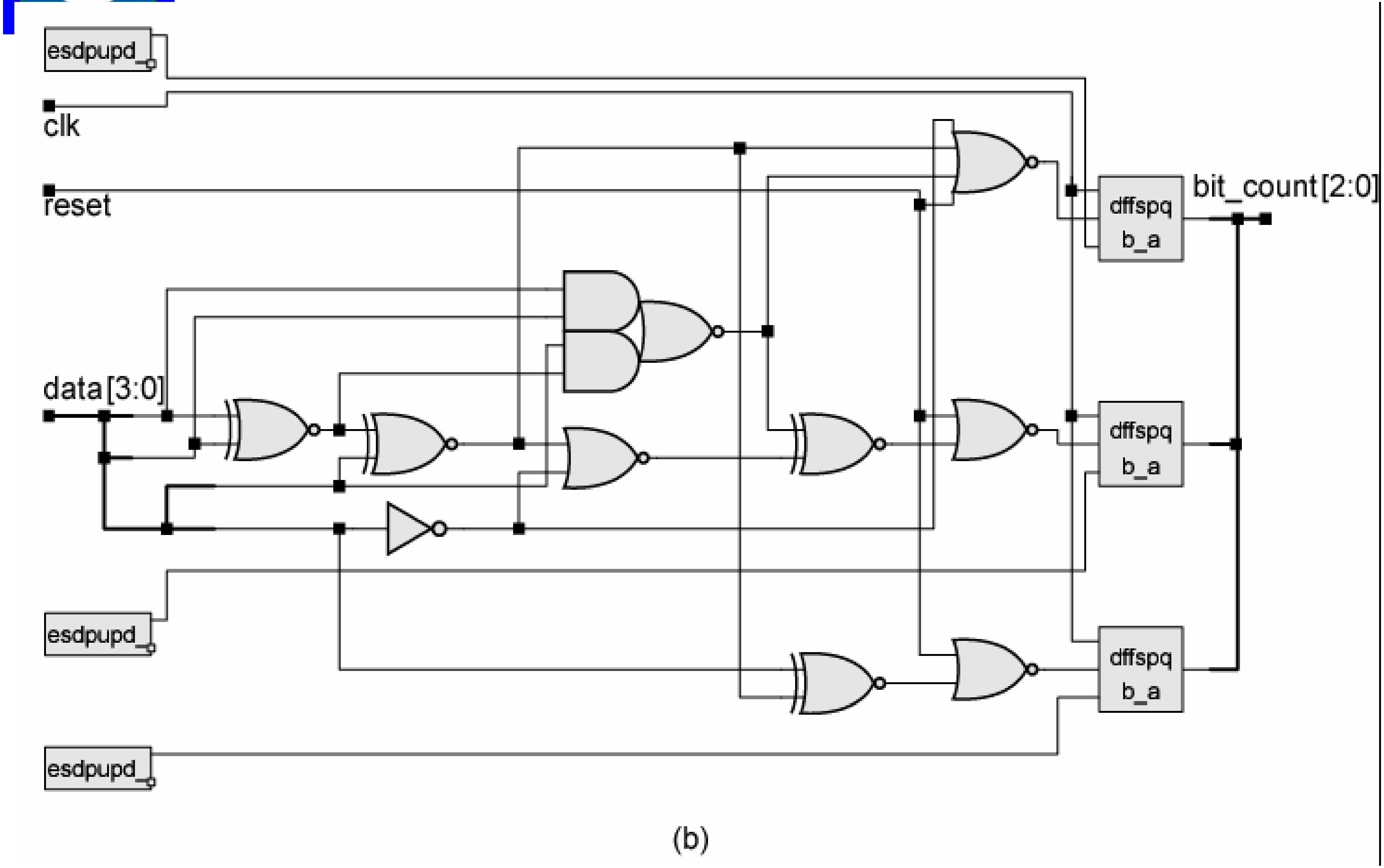


(a)

Simulation result
(reset at the random position)



The synthesized circuit



(b)

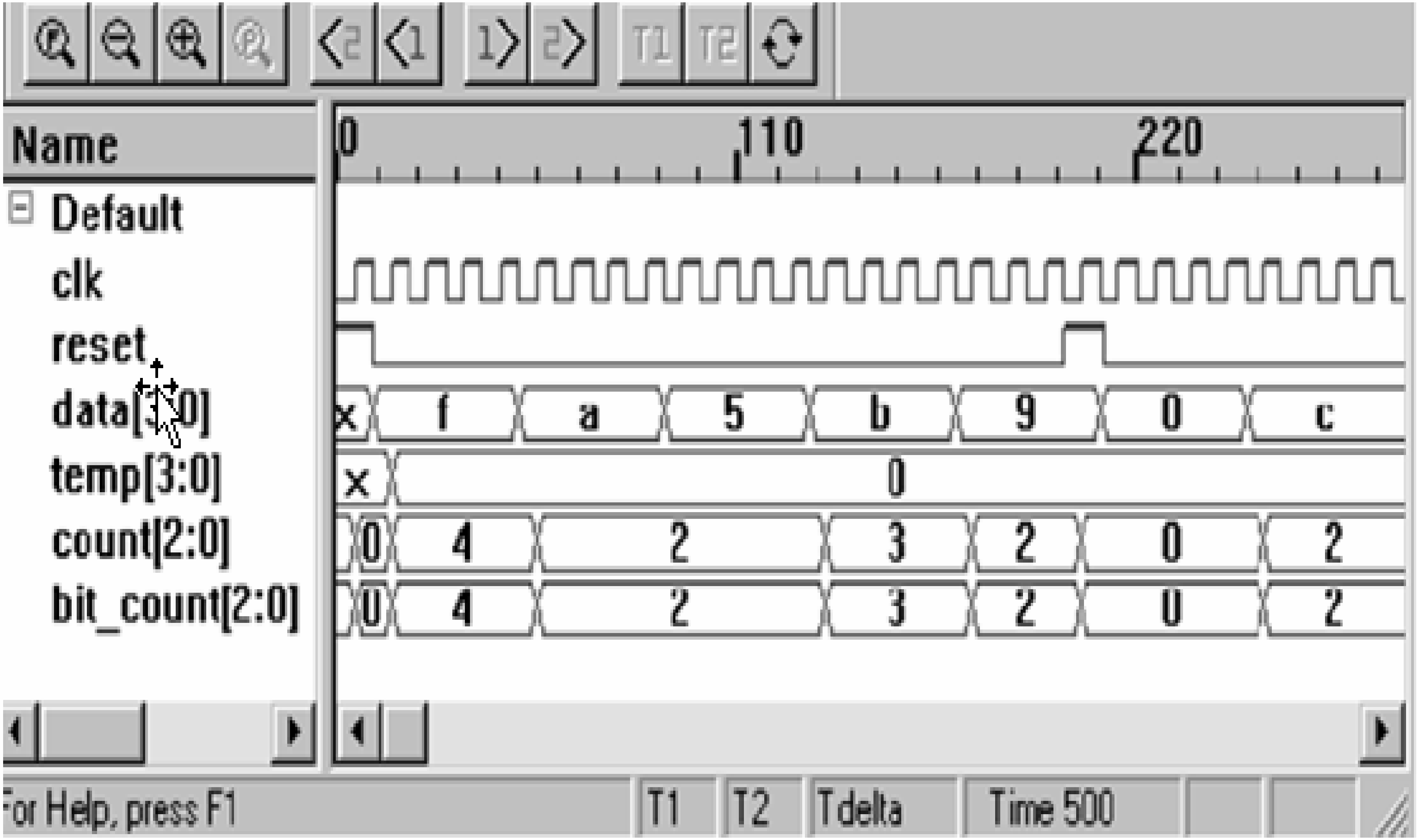
6.13.3 Nonstatic loops without embedded timing controls

- **If the loop does not have embedded timing control, the behavior can be simulated, but it cannot be synthesized**
- Under the action of a simulator, the behavior is virtually sequential and can be simulated
- **But hardware cannot execute** the computation of the loop in a single cycle of the clock

Example 6.40

```
module count_ones_c (bit_count, data, clk, reset);
    parameter data_width=4;
    parameter count_width=3;
    output [count_width-1:0] bit_count;
    input [data_width-1;0] data;
    input clk, reset;
    reg [count_width-1:0] count, bit_count, index;
    reg [data_width-1:0] temp;
    always @ (posedge clk)
      if (reset) begin count=0; bit_count=0; end
      else begin
        count=0;   temp=data;
          for (index=0; | temp; index=index+1)
            begin
              if (temp[0]) count=count+1;  temp=temp>>1;
            end
         bit_count=count;
     end
endmodule
```

Simulation result



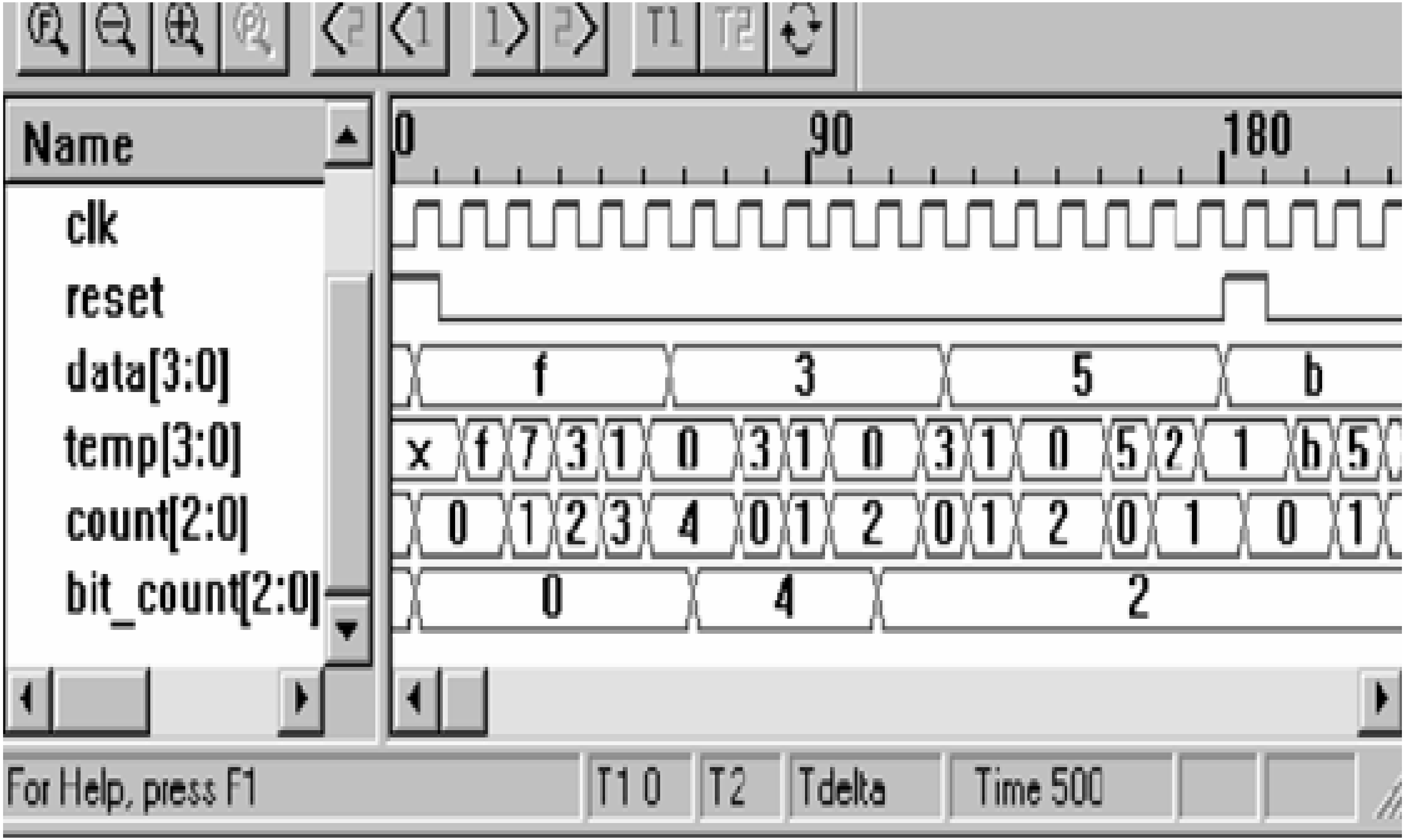
6.13.4 Nonstatic loops with embedded timing controls

- A nonstatic loop may implement a multicycle operation
- The data dependency alone is not a barrier to synthesis because the activity of the loop can be distributed over multiple cycles of the clock
- However, **the iterations of a non-static loop must be separated by a synchronizing edge-sensitive event control expression in order to be synthesized.**

Example 6.41

```
module count_ones_d (bit_count, data, clk, reset);
  parameter data_width=4;
  parameter count_width=3;
  output [count_width-1:0] bit_count;
  input [data_width-1;0] data;
  input clk, reset;
  reg [count_width-1:0] count, bit_count, index;
  reg [data_width-1:0] temp;
  always begin: wrapper_for_synthesis
  @ (posedge clk)
    if (reset) begin count=0; bit_count=0; end
    else begin: bit_counter
       count=0;
       temp=data;
       while (temp)
        @ (posedge clk)
           if (reset) begin count=2’b0;
                   disable bit_counter; end
           else begin
                     count=count+temp[0];
                      temp=temp>>1;
                   end
             @ (posedge clk)
              if (reset) begin count=0;
                   disable bit_counter; end
              else bit_count=count;
   end    // bit_counter
 end // wrapper_for_synthesis
endmodule
```

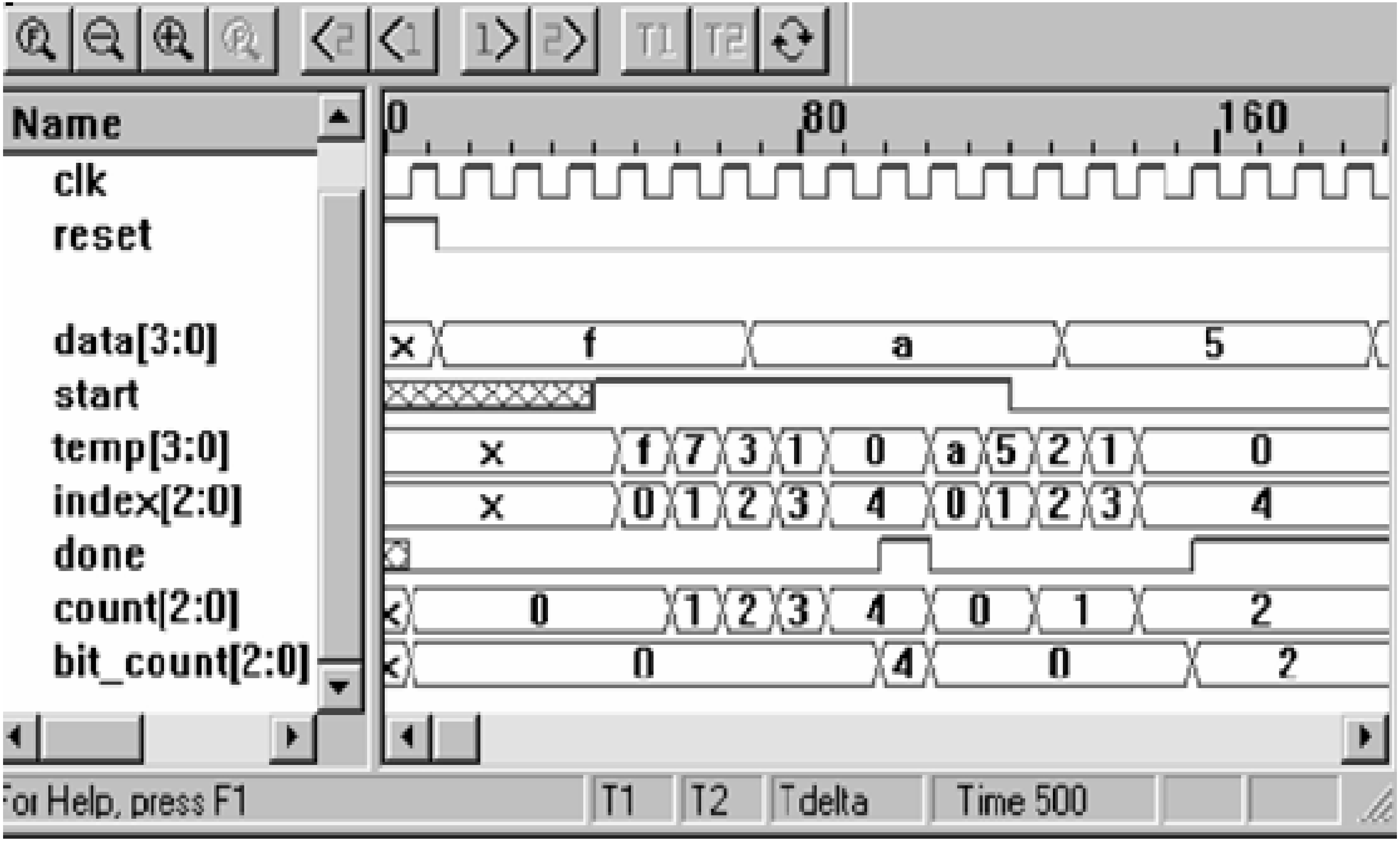
Simulation result



Another version

```
module count_ones_SD (bit_count, done, data, start, clk, reset);
    parameter data_width=4;
    parameter count_width=3;
    output [count_width-1:0] bit_count;
    output done;
    input [data_width-1;0] data;
    input start, clk, reset;
    reg [count_width-1:0] count, bit_count, index;
    reg [data_width-1:0] temp;
    reg done, start;
    always@ (posedge clk) begin: bit_counter
     if (reset) begin count=0; bit_count=0; done=0; end
    else if start begin
       done=0;
       count=0;
       bit_count=0;
       temp=data;
      for (index=0; index<data_width; index=index+1)
    @ (posedge clk)
     if (reset) begin count=0; bit_count=0; done=0;
                   disable bit_counter; end
            else begin
                   count=count+temp[0];
                   temp=temp>>1; end
    @ (posedge clk)   // Required for final register transfer
     if (reset) begin count=0; bit_count=0; done=0;
                   disable bit_counter; end
    else begin bit_count=count; done=1; end  //output
   end
 end
endmodule
```

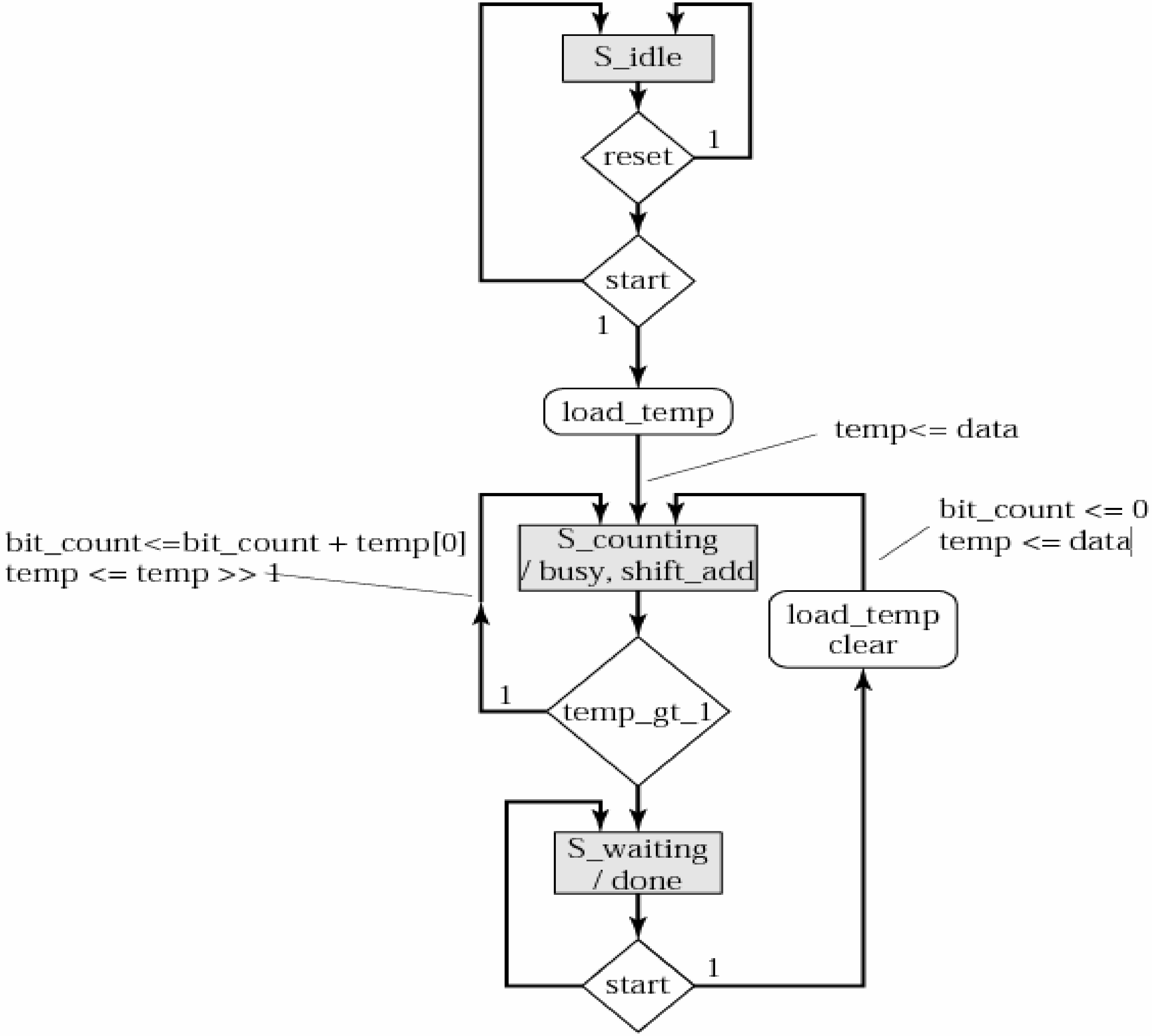
Simulation result



6.13.5 State-machine replacements for unsynthesizable loops

- Synthesis tools **do not support non-static loops** that do not have embedded timing controls
- Such machines are not directly synthesizable
- But their loop structures can be **replaced by equivalent synthesizable sequential behavior**.

Ex 6.42 ASMD chart of count_ones_SM



```
module count_ones_SM (bit_count, busy, done, data, start, clk,
    reset);
    parameter counter_size=3;
    parameter word_size=4;
    output [counter_size-1:0] bit_count;
    output busy, done;
    input [word_size-1:0] data;
    input start, clk, reset;
    wire load_temp, shift_add, clear;
    wire temp_0, temp_gt_1;
    controller M0 (load_temp, shift_add, clear, busy, done, start,
    temp_gt_1, clk, reset);
    datapath M1 (temp_gt_1, temp_0, data, load_temp, shift_add,
    clk, reset);
    bit_counter_unit M2 (bit_count, temp_0, clear, clk, reset);
endmodule
```

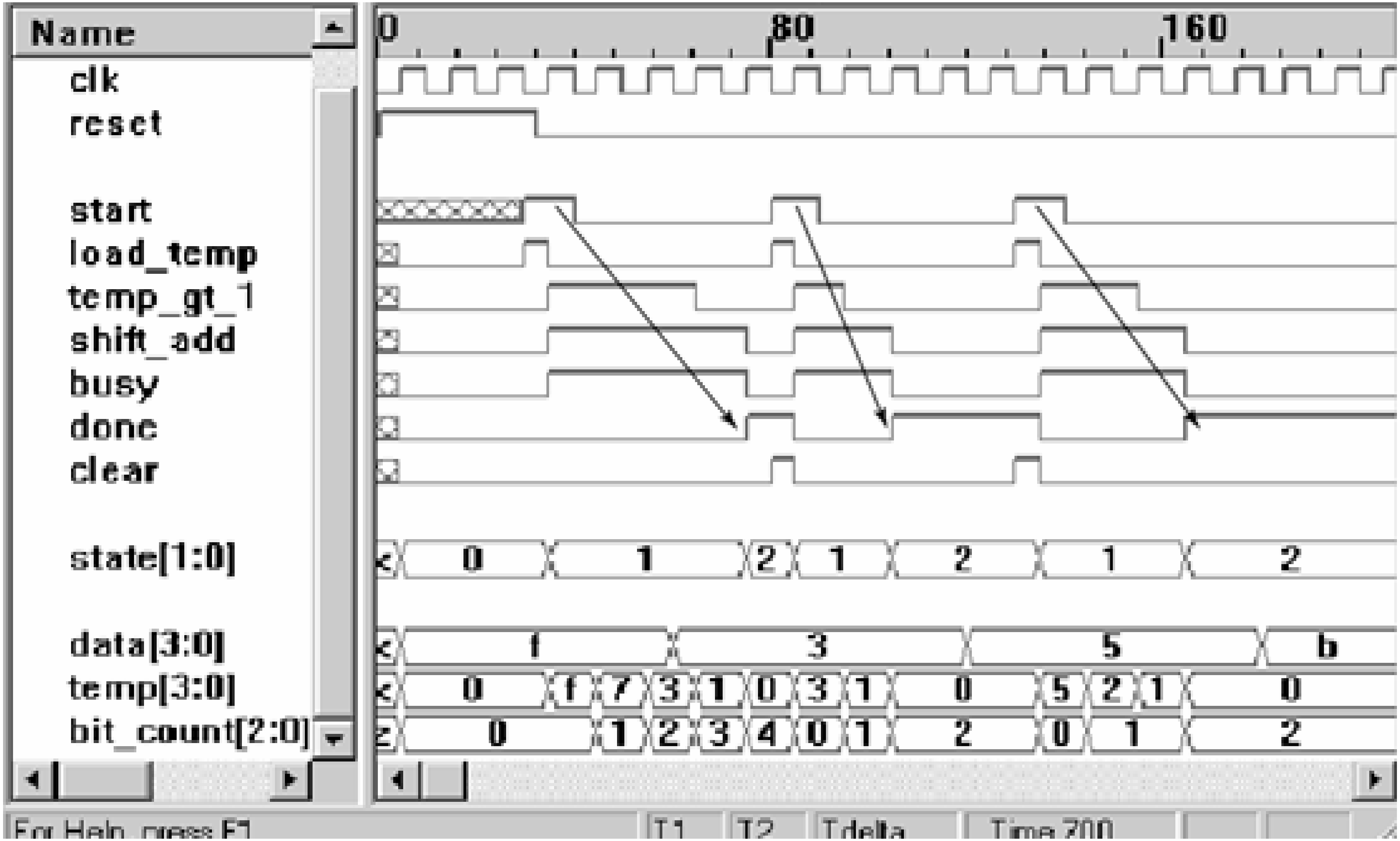
```
module controller (load_temp, shift_add, clear, busy, done,
    start, temp_gt_1, clk, reset);
    parameter state_size=2;
    parameter S_idle=0;
    parameter S_counting=1;
    parameter S_waiting=2;
    output load_temp, shift_add, clear, busy, done;
    input start, temp_gt_1, clk, reset;
    reg bit_count;
    reg [state_size-1:0] state, next_state;
    reg load_temp, shift_add, busy, done, clear;
    always @ (state or start or temp_gt_1)
      begin
      load_temp=0;
      shift_add=0;
      done=0;
      busy=0;
      clear=0;
      next_state=S_idle;
```

```
case (state)
  S_idle: if (start) begin next_state=D_counting; load_temp=1;
                           end
  S_counting: begin busy=1; if (temp_gt_1)
                       begin   next_state=S_counting; shift_add=1; end
                       else begin next_state=S_waiting; shift_add=1; end
                     end
  S_waiting: begin
                      done=1;
                      if (start) begin next_state=S_counting; load_temp=1;
                         clear=1; end
                      else next_state=S_waiting; end
                        default: begin clear=1; next_state=S_idle;   end
endcase
end
   always @ (posedge clk)    // state transitions
    if (reset)
       state<=S_idle;
    else state<=next_state;
endmodule
```

```
module datapath (temp_gt_1, temp_0, data, load_temp,
    shift_add, clk, reset);
    parameter word_size=4;
    output temp_gt_1, temp_0;
    input [word_size-1:0] data;
    input load_temp, shift_add, clk, reset;
    reg [word_size-1:0] temp;
    wire temp_gt_1=(temp>1);
    wire temp_0=temp[0];
    always @ (posedge clk) // state ,loading, shiftting
      if (reset) begin   temp<=0; end
      else begin
                if (load_temp)  temp<=data;
                if (shift_add)  begin temp<= temp >>1; end
              end
endmodule
```

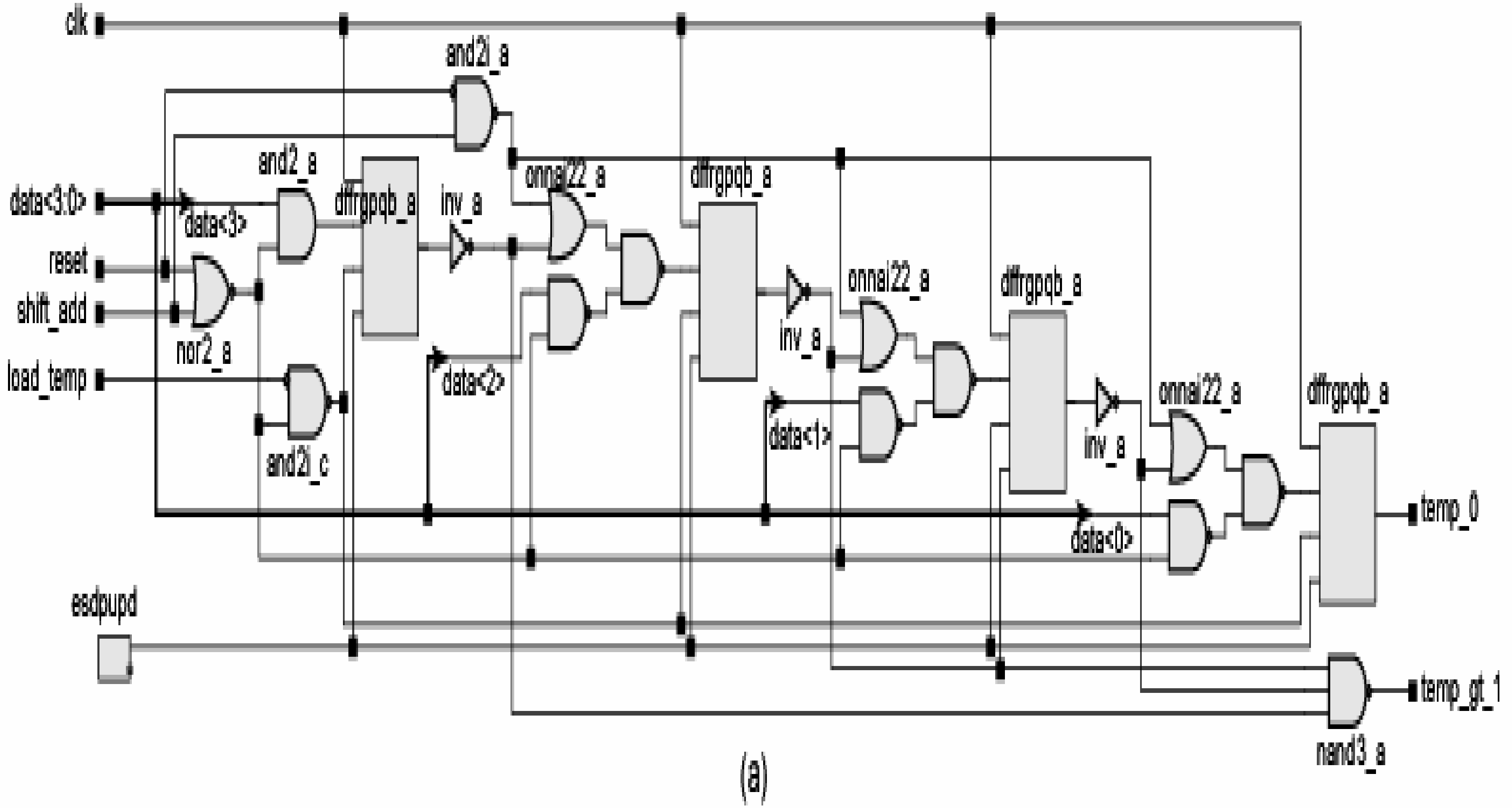
```
module bit_counter_unit (bit_count, temp_0,
    clear, clk, reset);
    parameter counter_size=3;
    output [counter_size-1:0] bit_count;
    input temp_0;
    input clear, clk, reset;
    reg bit_count;
    always @ (posedge clk)
      if (reset || clear)
          bit_count<=0;
      else bit_count<=bit_count+temp_0;
endmodule
```

Simulation



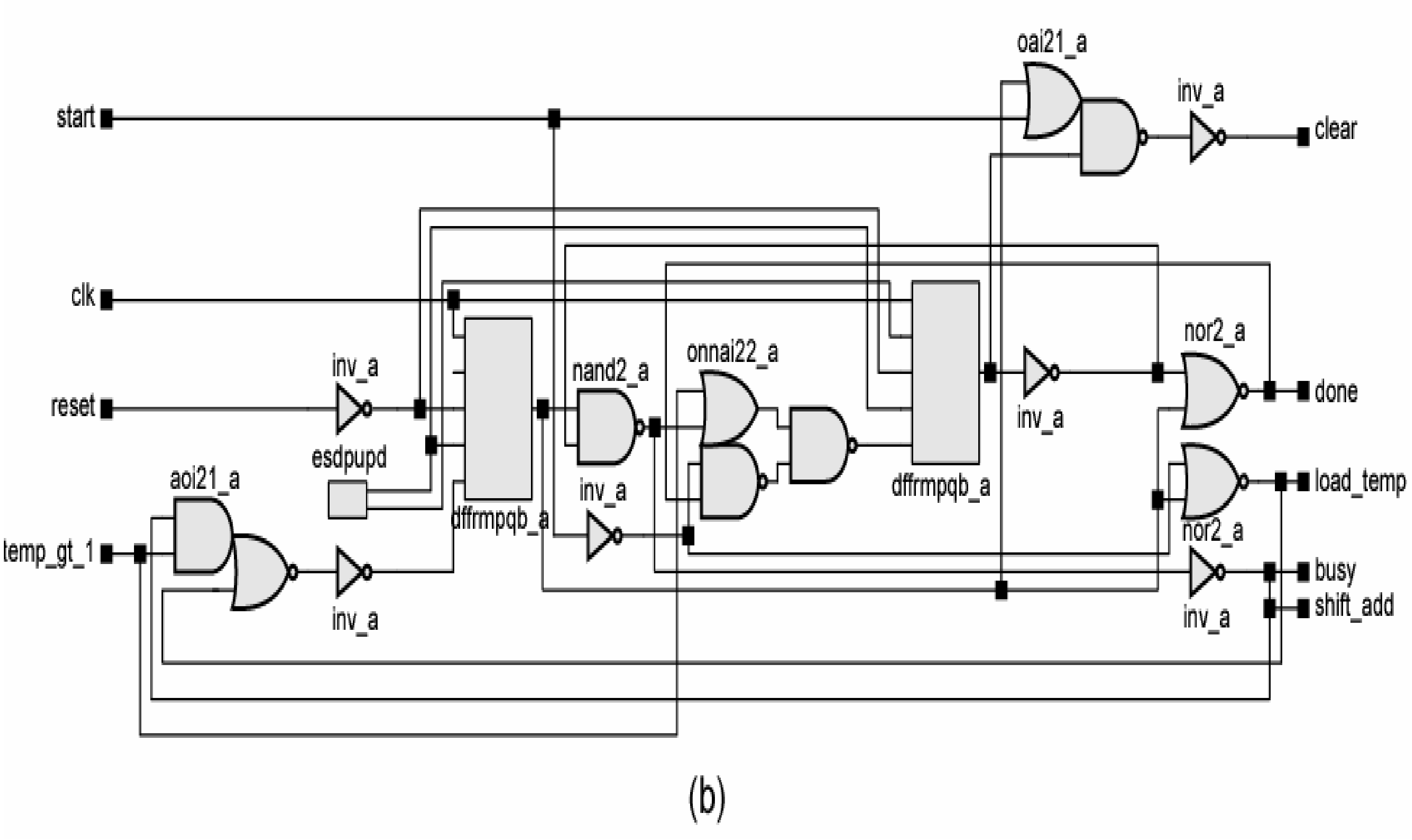
For Help, press F1

Synthesis circuit of datapath unit



(a)

Synthesis circuit of control unit



(b)

6.14 To avoid design traps

- In general, **avoid referencing the same variable in more than one cyclic (always) behavior.**
- When variables are referenced in more than one behavior, there can **be races in the software**, and the post-synthesis simulated behavior may not match the pre-synthesis behavior.
- Never assign value to the same variable in multiple behaviors.

6.15 Divide and conquer: partitioning a design

- VLSI circuits commonly contain more gates than a synthesis tool can accommodate
- It is standard practice **to partition such circuits hierarchically into small functional units**, which have a manageable complexity
- This partitioning is done **top-down, across one or more layers of hierarchy**
- Synthesis tools commonly synthesize circuits with about 10,000 to 100,000 gates, but beyond that the return diminishes as the run-time becomes large
- It is easier to modify the lower-level modules to make the design more amenable to synthesis, then to work with large modules.

Partitioning a design

partitioning is not done randomly, Skill and experience play a role

- A design is partitioned along functional lines into smaller functional units:
- each with a common clock domain
- each of which is **to be verified separately**
- functionally related logic should be grouped within a partition
- **a module contain no more than one state machine**
- the partition of a design should group registers and their logic
- module boundaries are preserved in synthesis (i.e., optimized separately)

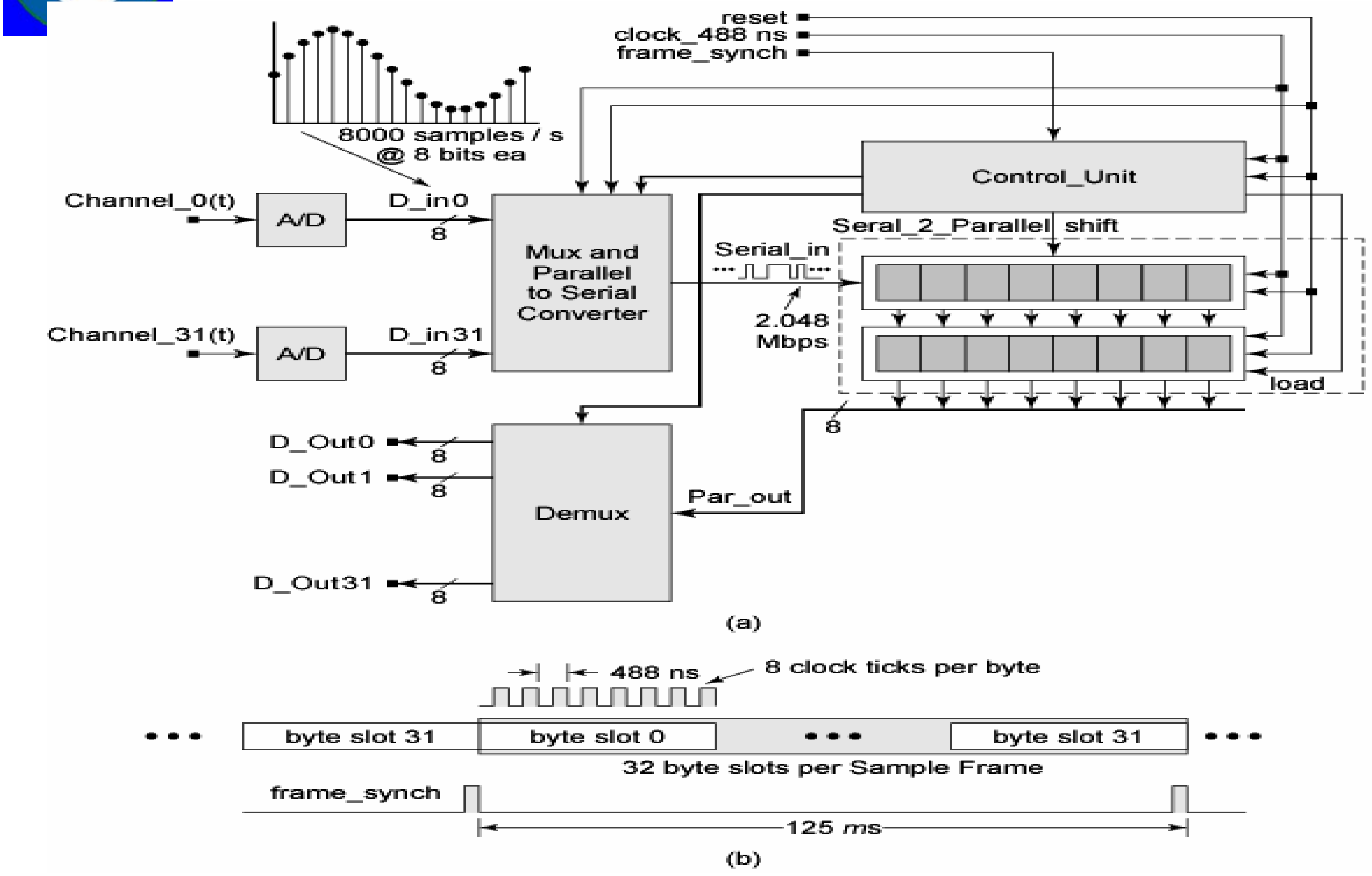
Problems

- 6-20. Synthesize a circuit that will detect an illegal BCD-encoded word.
- 6-24. Under what conditions will a synthesis tool create combinational logic?
- 6-25. Under what conditions will a synthesis tool create a circuit that implements a transparent latch?
- 6-26. Under what conditions will a synthesis tool create an edge-triggered sequential circuit?
- 6-27. Discuss how to describe a synchronous reset condition using Verilog.

problem 6-21

- Develop, verify (before and after synthesis), and synthesize a Verilog module that encapsulates the functionality shown in **Fig – P621 (a)**, where the outputs of the A/D converters are inputs to a module that interleaves the sample bytes, with separate sub-modules for the control unit, the mux, the demux, the parallel to serial converter, and the serial to parallel converter.
- Define additional interface signals as needed to complete the design.
- Model the multiplexer so that its outputs will be registered. Carefully document your work

Fig – P621



reset
clock_488 ns
frame_synch
8000 samples / s
@ 8 bits ea
Channel_0(t)
A/D
D_in0
8
Channel_31(t)
A/D
D_in31
8
Mux and Parallel to Serial Converter
Control_Unit
Seral_2_Parallel shift
Serial_in
2.048 Mbps
load
8
D_Out0
8
D_Out1
8
D_Out31
8
Demux
Par_out
(a)
488 ns
8 clock ticks per byte
byte slot 31
byte slot 0
byte slot 31
32 byte slots per Sample Frame
frame_synch
125 ms
(b)

电话系统

公用电话交换网PSTN

(Public Switched Telephone Network)

-端到端的数据传输率：10^4 b/s

-干线的数据传输率：10^{10} b/s

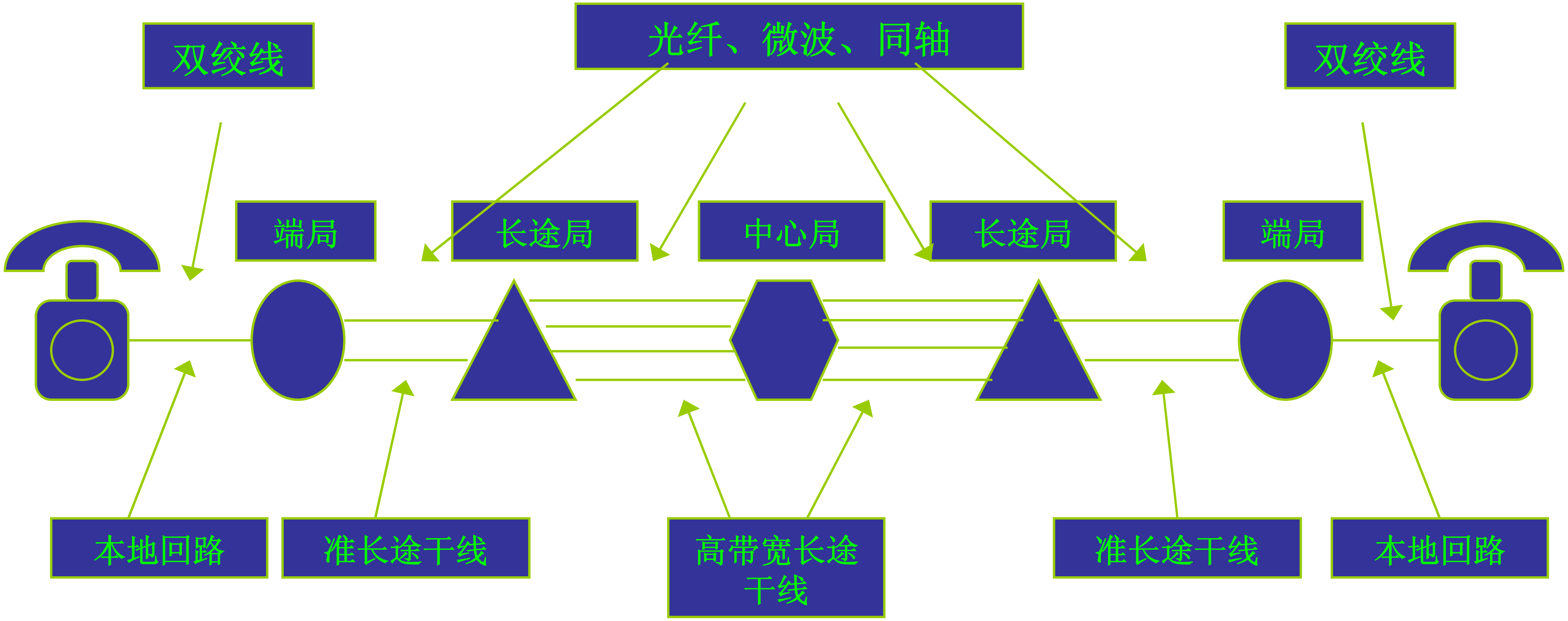
-总体误码率：$<10^{-5}$

-通信干线采用光纤或微波数字传输系统

。

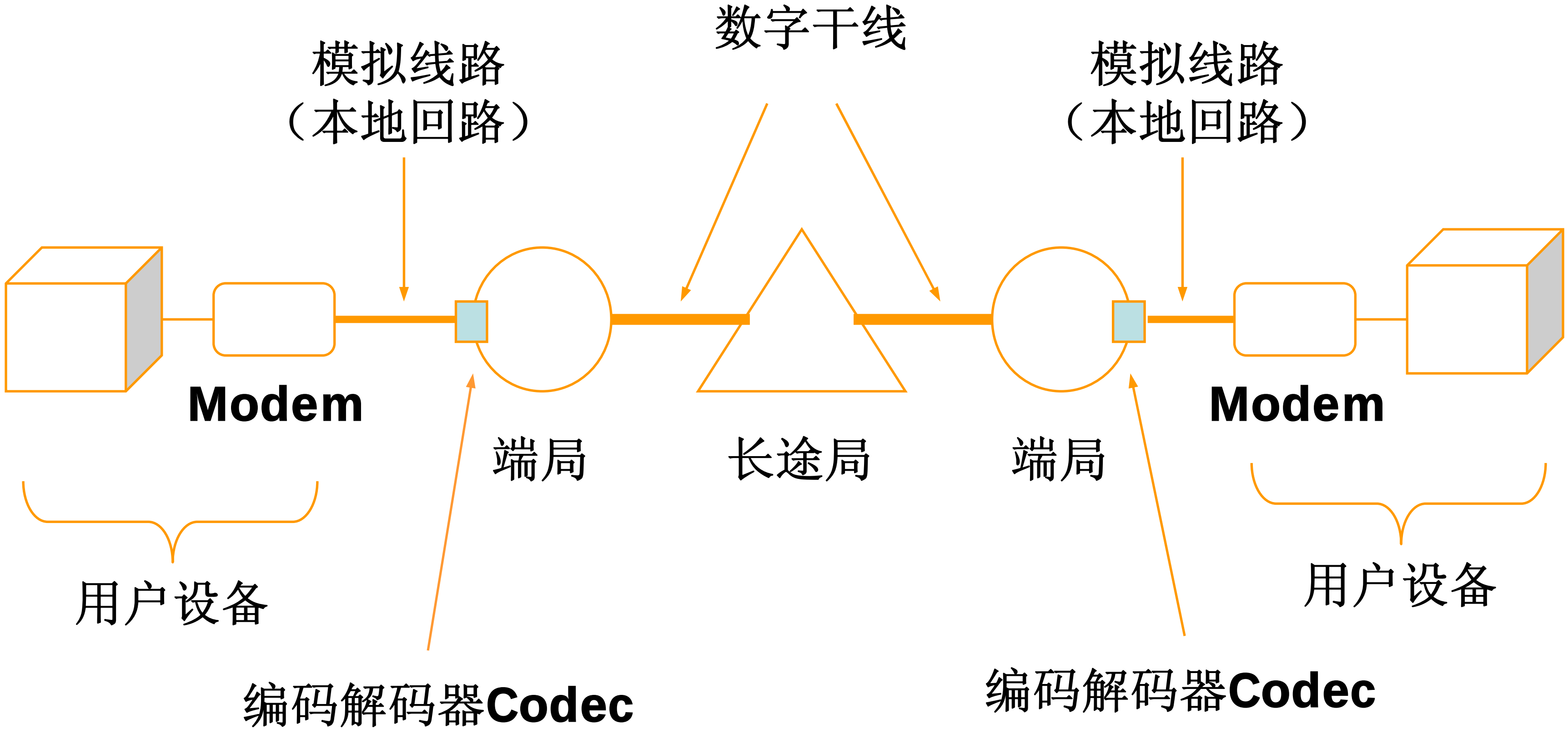
优点：出错率低、可传输语音/数据/图像、速率高、成本低、维护容易

构成：本地回路、交换局、干线



本地回路

- 绝大多数为模拟线路
- 计算机间传输：以模拟为主，信号要经过多次变换：数字→模拟→数字→模拟→数字

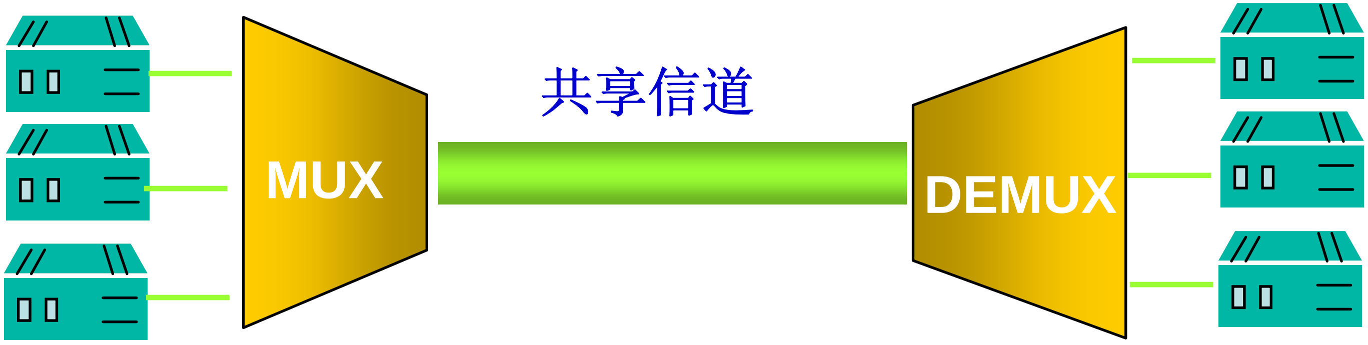


数据复用技术

- 复用——多路不同信号共享一个公共信道

为何要复用？

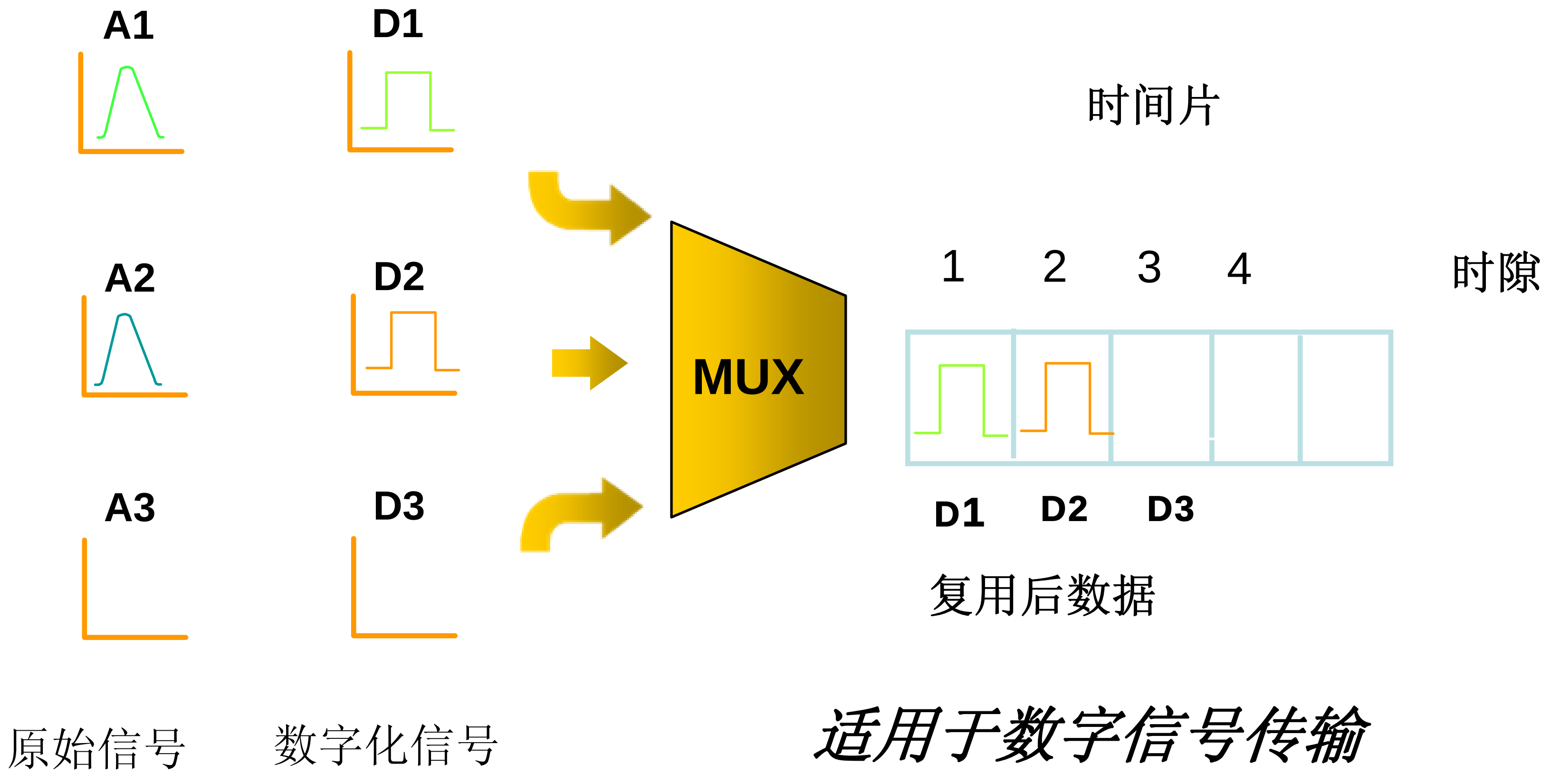
线路成本问题，能提高通信线路的利用率



复用器　　解复用器

Time Division Multiplexing

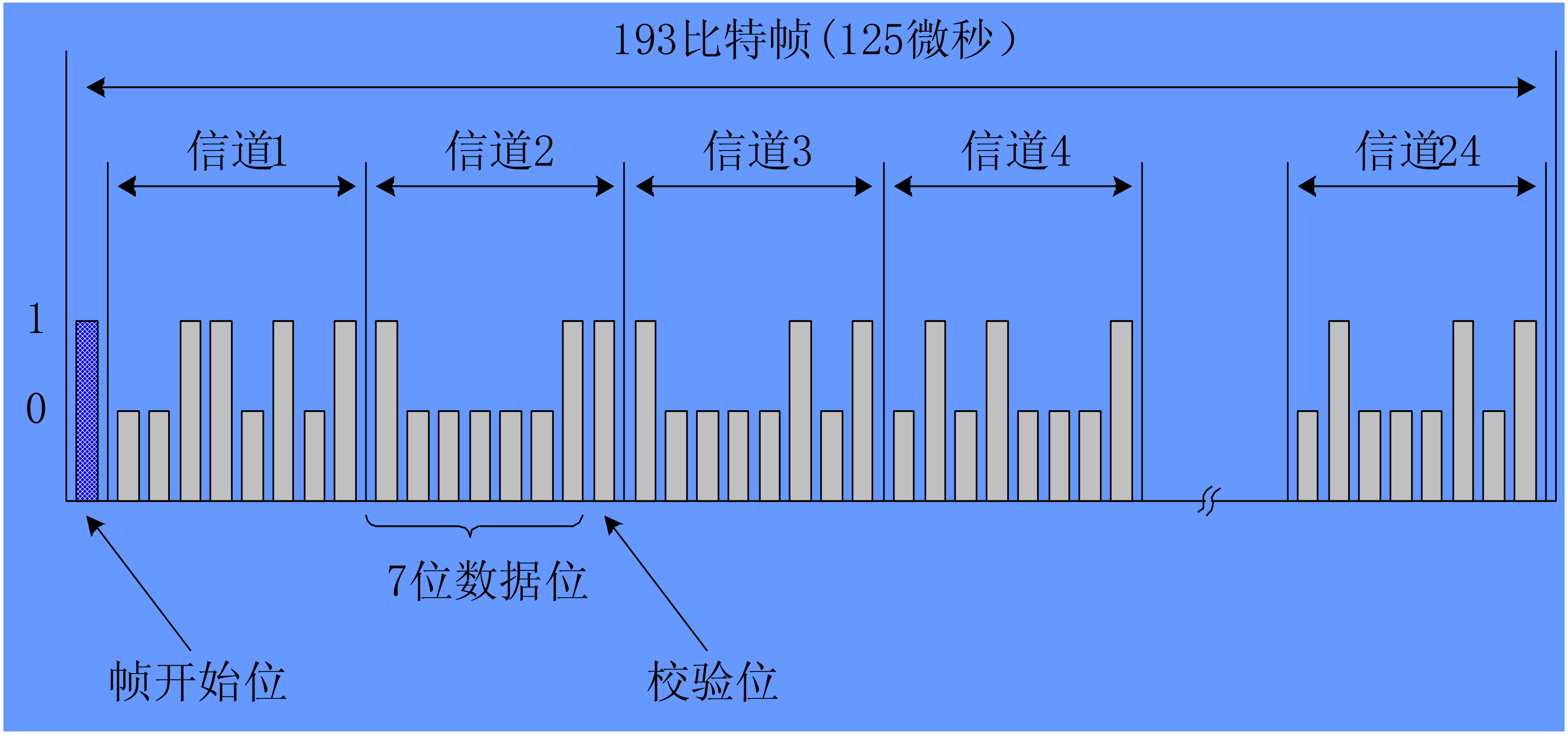
原理：把时间分割成小的时间片，每个时间片分为若干个通道（时隙），每个用户占用一个通道传输数据。



适用于数字信号传输

Time Division Multiplexing

- 时分多路复用是将信道用于传输的时间划分为若干个时间；
- 每个用户分得一个时间片；
- 在其占有的时间片内，用户使用通信信道的全部带宽。



E1标准

每125us为一个时间片，每时间片分为32个通道（供32个用户轮流使用）。

每通道占用125 us /32=3.90625 us

每通道一次传送8位二进制数据，即每个二进制位占用0.48828125 us，所以

E1速率 = 1/0.48828125=2.048Mb/s

也可以这样计算E1速率

E1速率 = (32x8bit)/125 μs = 2.048 Mb/s

时分复用（TDM）原理

① $T_s = \frac{1}{f_s} = 125$ μs

$T_c = {T_s}/{K} = 125 \div 32 = 3.9$ μs

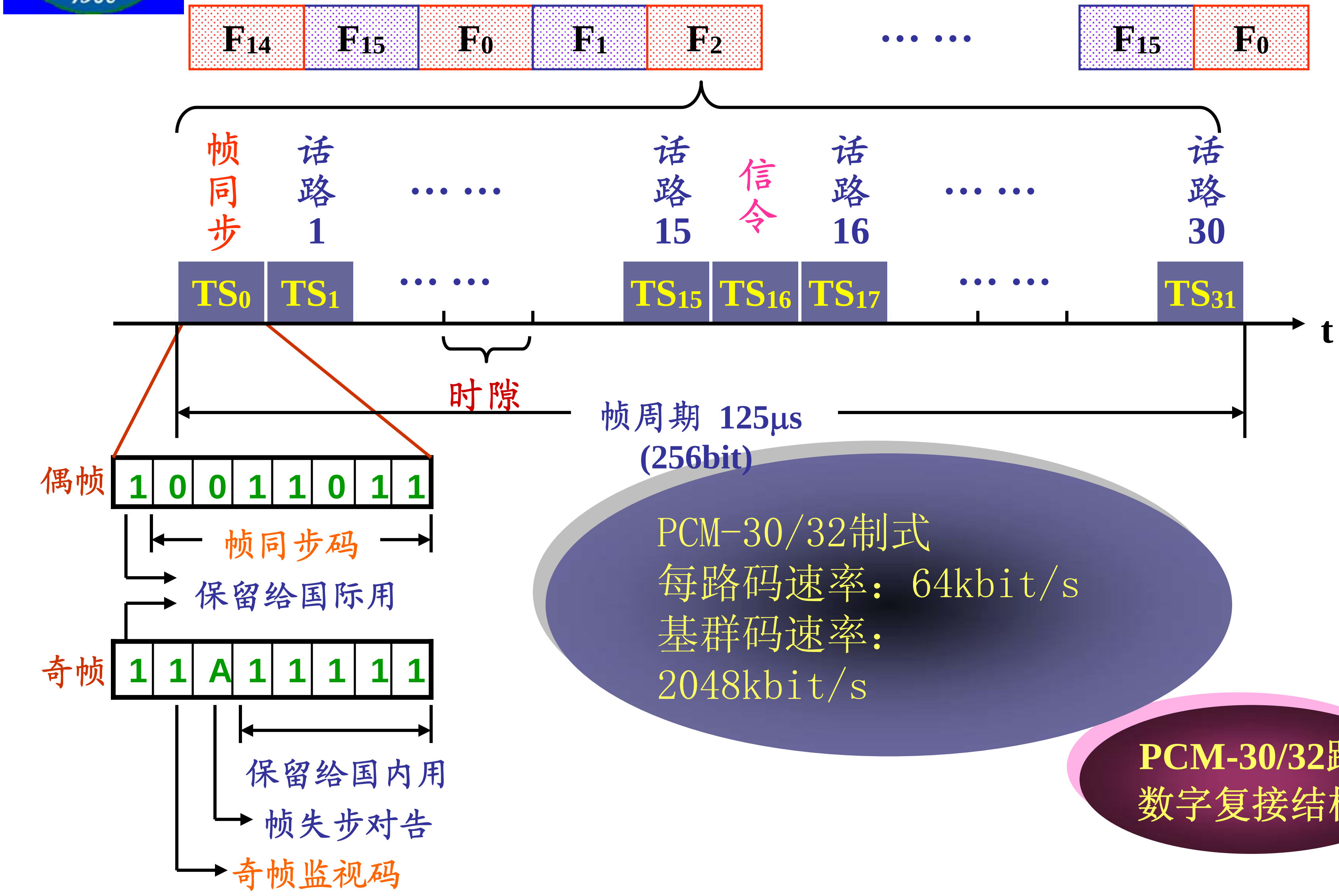
② $f_B = nKf_s = 8 \times 32 \times 8000 = 2048\ Kb/s$

③ $B_{\min} = {f_B}/{2} = 1024\ \ KHZ$

④ 平均每路电话占用带宽：

1024KHZ÷32 = 32KHZ

时分多路复用



F_{14} F_{15} F_0 F_1 F_2 …… F_{15} F_0

帧同步 话路1 …… 话路15 信令 话路16 …… 话路30

TS_0 TS_1 …… TS_{15} TS_{16} TS_{17} …… TS_{31} t

时隙

帧周期 125μs (256bit)

偶帧 1 0 0 1 1 0 1 1

帧同步码

保留给国际用

奇帧 1 1 A 1 1 1 1 1

保留给国内用

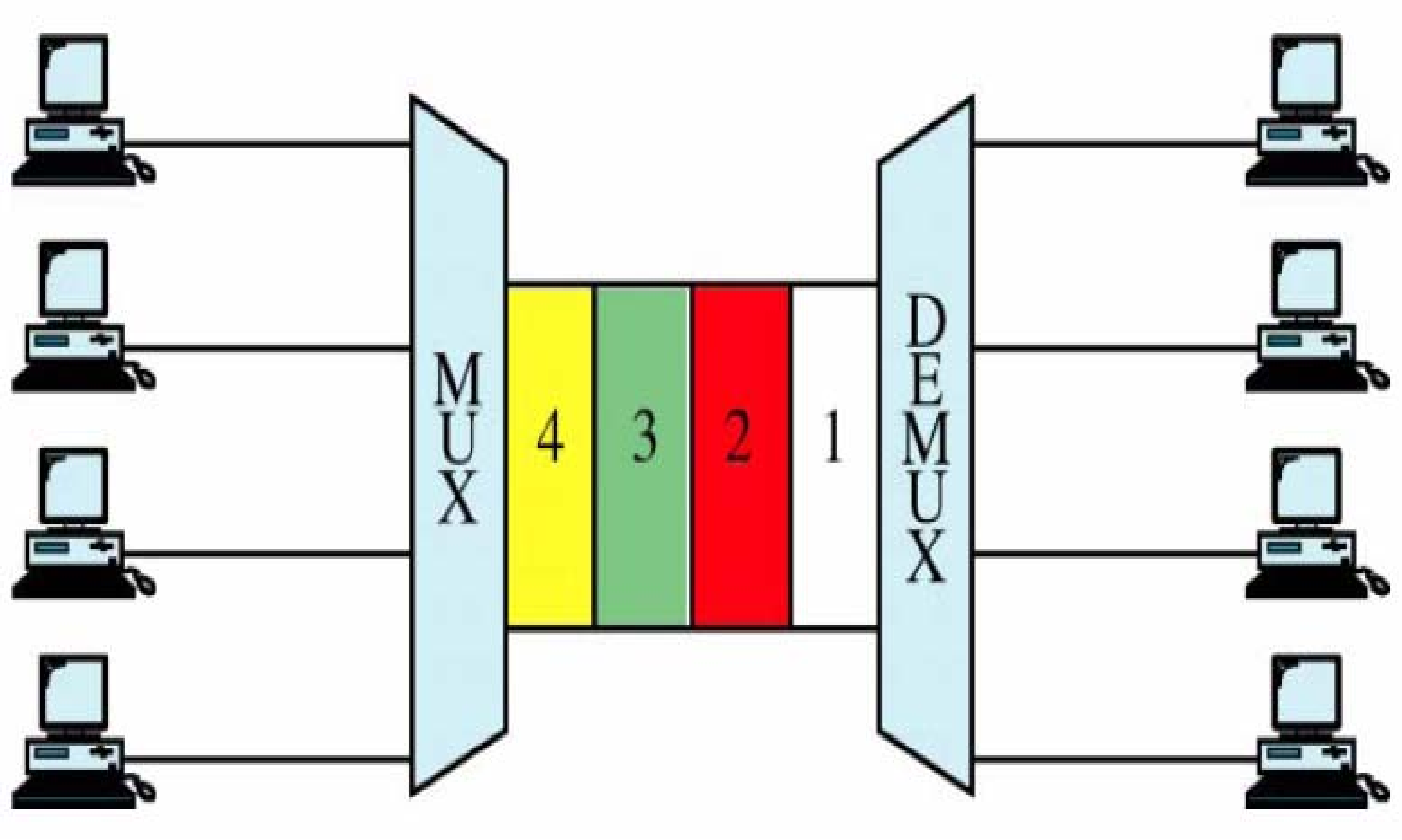
帧失步对告

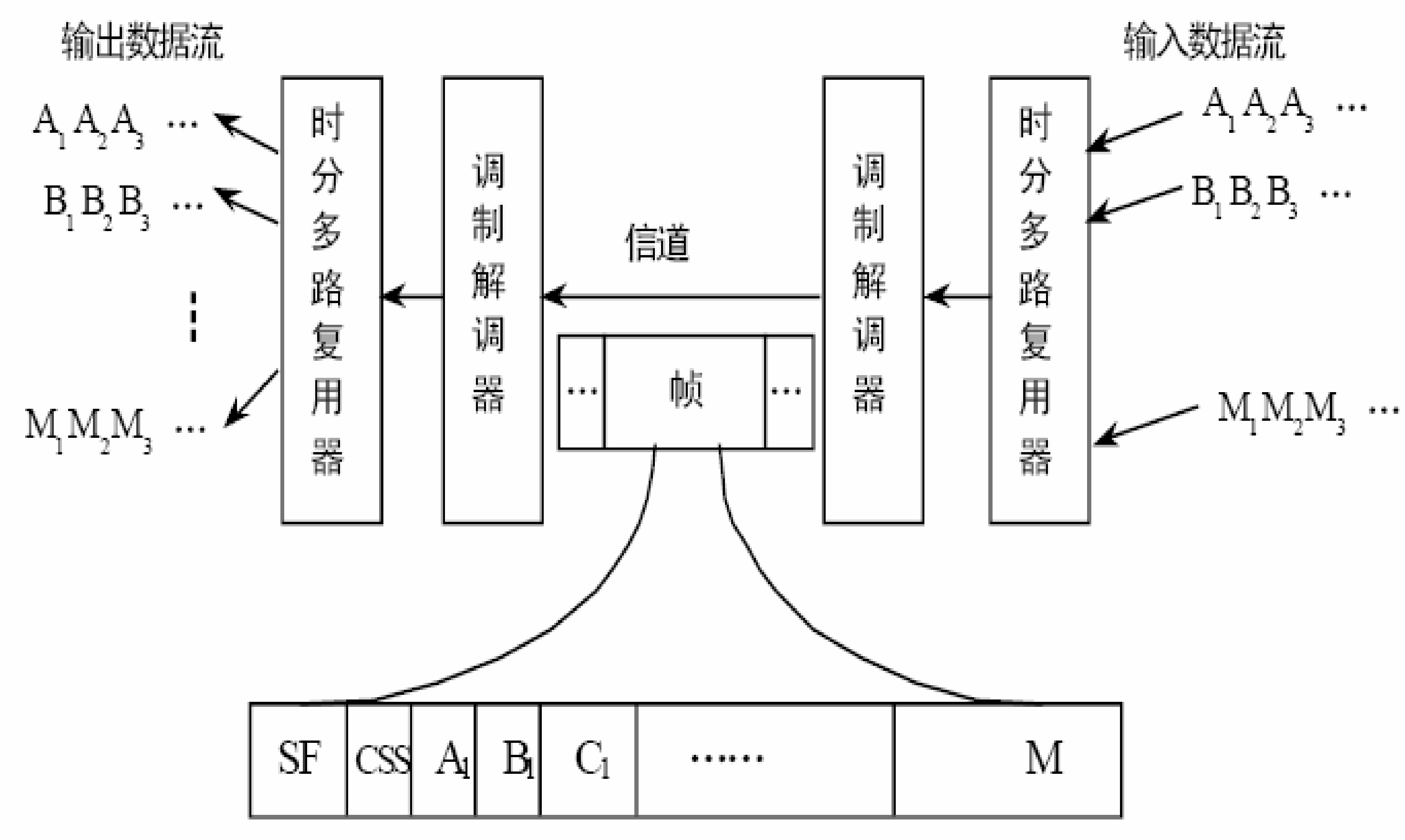
奇帧监视码

PCM-30/32制式
每路码速率：64kbit/s
基群码速率：
2048kbit/s

PCM-30/32路
数字复接结构

MUX/DEMUX





输出数据流

$A_1A_2A_3$ …

$B_1B_2B_3$ …

⋮

$M_1M_2M_3$ …

时分多路复用器

调制解调器

信道

… 帧 …

调制解调器

时分多路复用器

输入数据流

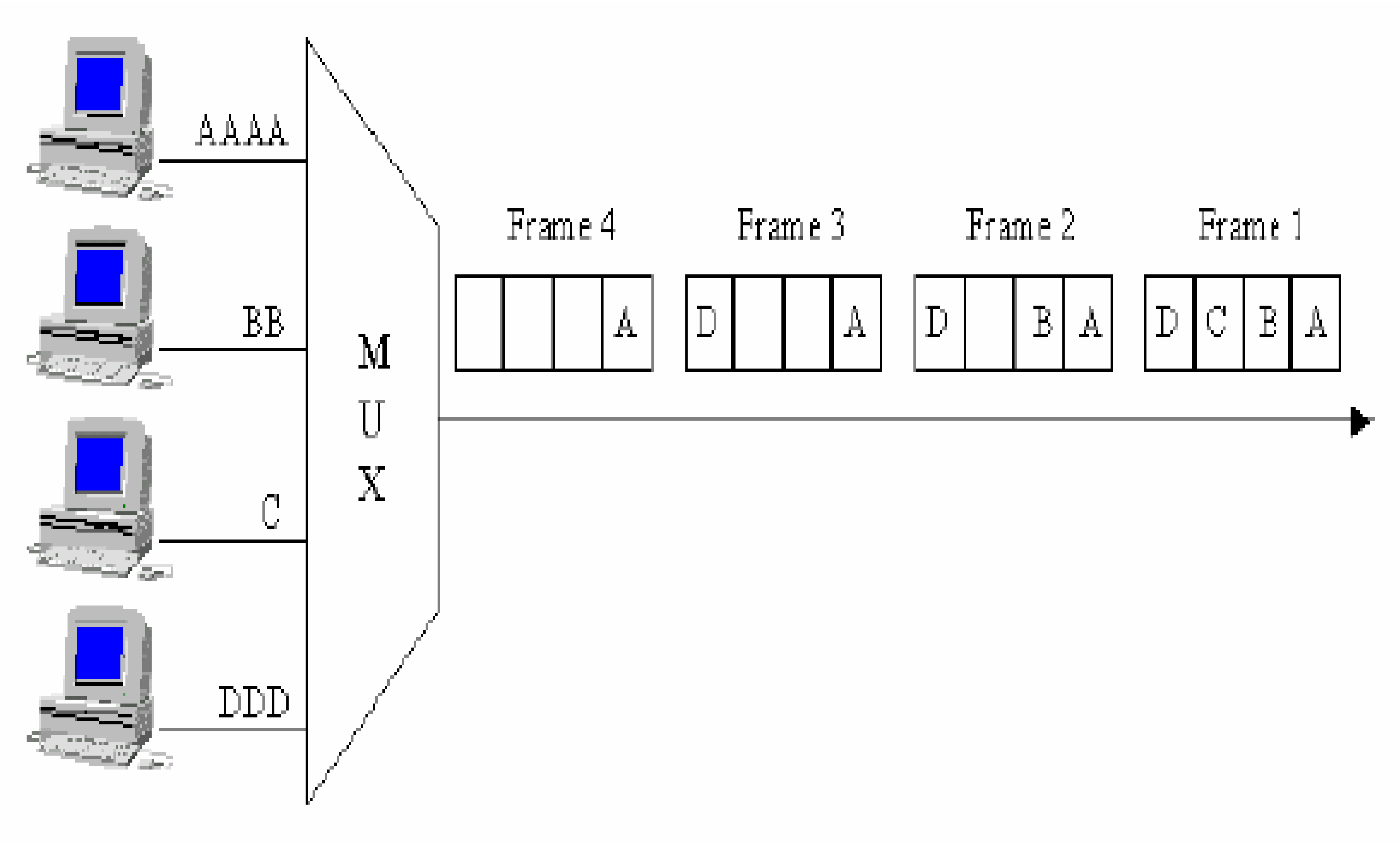
$A_1A_2A_3$ …

$B_1B_2B_3$ …

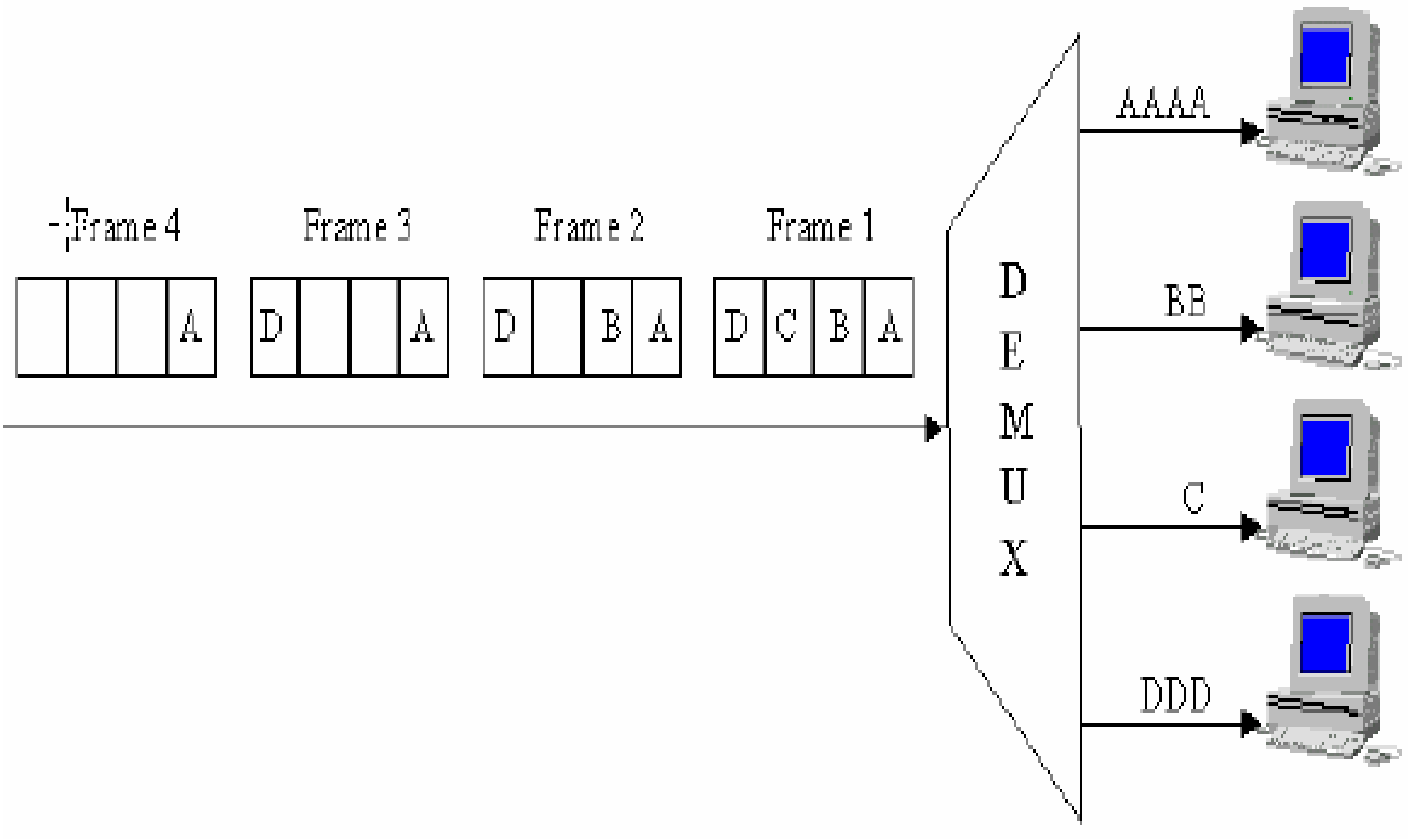
$M_1M_2M_3$ …

SF	CSS	A_1	B_1	C_1	……	M

Multiplexing Process



Demultiplexing process


Frame 4
Frame 3
Frame 2
Frame 1
A
D
A
D
B
A
D
C
B
A
DEMUX
AAAA
BB
C
DDD