UNIVERSIDAD NACIONAL DE CÓRDOBA

FACULTAD DE MATEMÁTICA ASTRONOMÍA, FÍSICA Y COMPUTACIÓN.

ARQUITECTURA DE COMPUTADORAS

TEÓRICOS: PABLO A. FERREYRA

PRÁCTICOS:
DELFINA VELEZ
AGUSTÍN LAPROVITA
GONZALO VODANOVICK

T2: HARDWARE DESCRIPTION LANGUAGES (A3. Structural Modeling Structural Model of a 4 bit 4:1 Multiplexer)

Example A.17 Structural Model of 4:1 Multiplexer

SystemVerilog

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

  logic [3:0] low, high;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 highmux(d2, d3, s[0], high);
  mux2 finalmux(low, high, s[1], y);
endmodule
```

The three mux2 instances are called `lowmux`, `highmux`, and `finalmux`. The mux2 module must be defined elsewhere in the SystemVerilog code.

T2: HARDWARE DESCRIPTION LANGUAGES (A3. Structural Modeling Structural Model of a 4 bit 4:1 Multiplexer)

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4 is
  port(d0, d1,
      d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
      s:      in  STD_LOGIC_VECTOR(1 downto 0);
      y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux4 is
  component mux2
     port(d0,
       d1: in  STD_LOGIC_VECTOR(3 downto 0);
       s:  in  STD_LOGIC;
       y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal low, high: STD_LOGIC_VECTOR(3 downto 0);
begin
  lowmux:    mux2 port map(d0, d1, s(0), low);
  highmux:   mux2 port map(d2, d3, s(0), high);
  finalmux:  mux2 port map(low, high, s(1), y);
end;
```

T2: HARDWARE DESCRIPTION LANGUAGES (A3. Structural Modeling Structural Model of a 4 bit 4:1 Multiplexer)

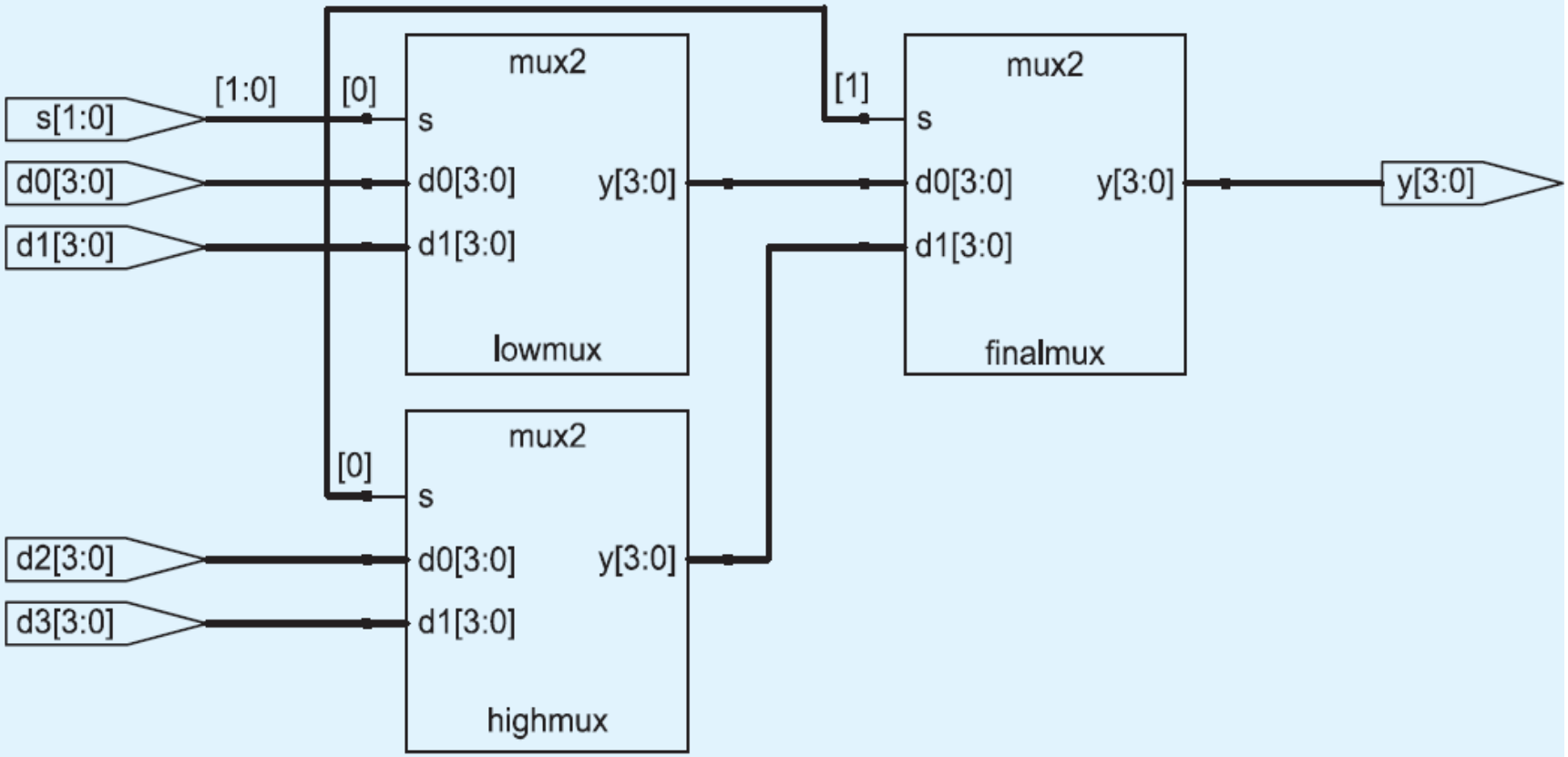


FIGURE A.14 mux4

T2: HARDWARE DESCRIPTION LANGUAGES (A3. Structural Modeling Structural Model of a 4 bit 4:1 Multiplexer)

Example A.18 Structural Model of 2:1 Multiplexer

SystemVerilog

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output tri   [3:0] y);

  tristate t0(d0, ~s, y);
  tristate t1(d1, s, y);
endmodule
```

In SystemVerilog, expressions such as `~s` are permitted in the port list for an instance. Arbitrarily complicated expressions are legal, but discouraged because they make the code difficult to read.

Note that `y` is declared as `tri` rather than `logic` because it has two drivers.

T2: HARDWARE DESCRIPTION LANGUAGES (A3. Structural Modeling Structural Model of a 4 bit 4:1 Multiplexer)

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  port(d0, d1:  in STD_LOGIC_VECTOR(3 downto 0);
       s:       in  STD_LOGIC;
       y:       out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux2 is
  component tristate
    port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
         en: in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal sbar: STD_LOGIC;
begin
  sbar <= not s;
  t0: tristate port map(d0, sbar, y);
  t1: tristate port map(d1, s, y);
end;
```

T2: HARDWARE DESCRIPTION LANGUAGES (A3. Structural Modeling Structural Model of a 4-bit 4:1 Multiplexer)

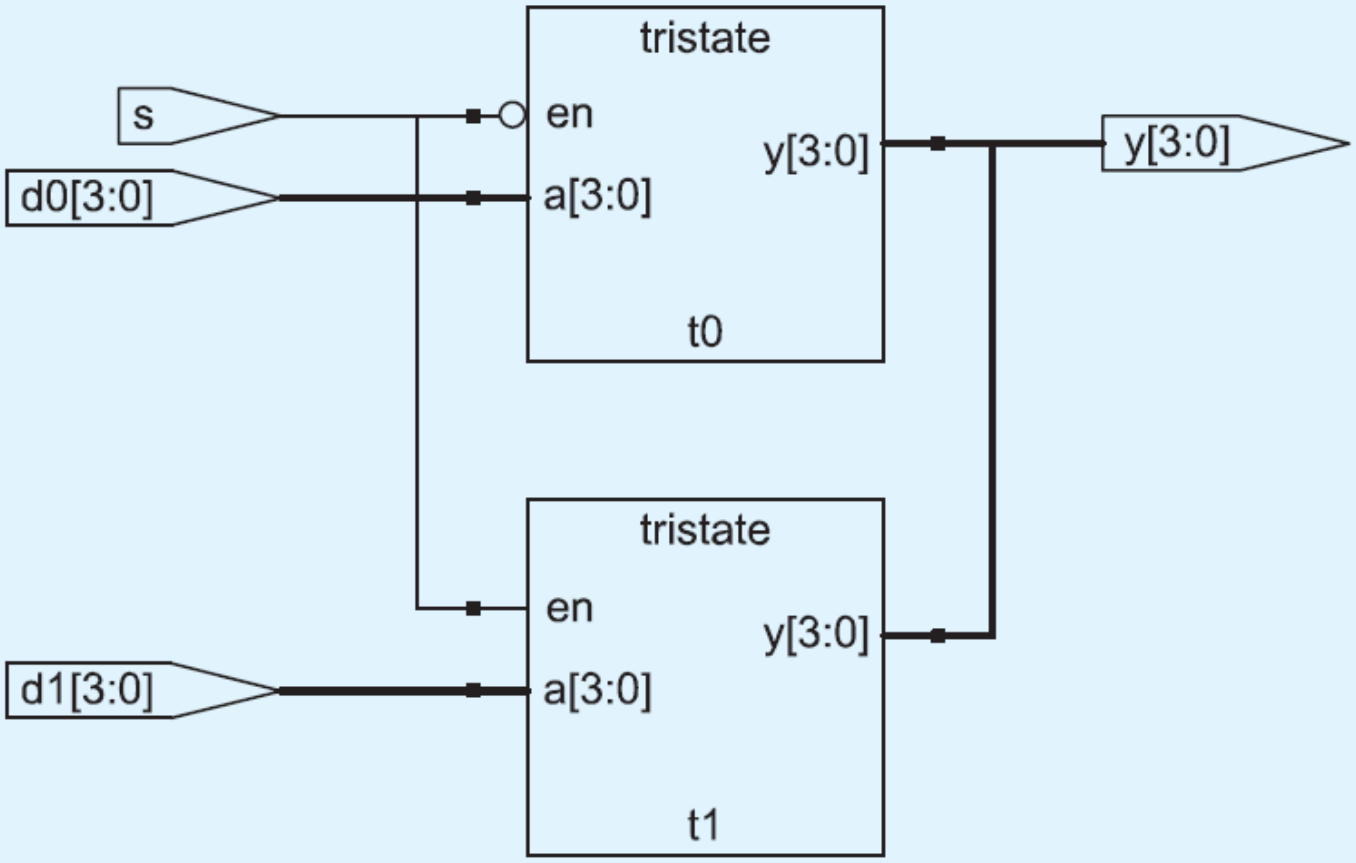


FIGURE A.15 mux2

T2: HARDWARE DESCRIPTION LANGUAGES (A3. Structural Modeling – Accessing Part of Buses)

Example A.19 Accessing Parts of Busses

SystemVerilog

```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A3. Structural Modeling Accessing Part of Buses)

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2_8 is
  port(d0, d1:in  STD_LOGIC_VECTOR(7 downto 0);
       s:     in  STD_LOGIC;
       y:     out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture struct of mux2_8 is
  component mux2
    port(d0, d1: in  STD_LOGIC_VECTOR(3
                                 downto 0);
         s:  in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
begin
```

T2: HARDWARE DESCRIPTION LANGUAGES (A3. Structural Modeling Accessing Part of Buses)

```
architecture struct of mux2_8 is
  component mux2
    port(d0, d1: in  STD_LOGIC_VECTOR(3
                                      downto 0);
         s:  in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
begin

  lsbmux:  mux2
    port map(d0(3 downto 0), d1(3 downto 0),
             s, y(3 downto 0));
  msbhmux: mux2
    port map(d0(7 downto 4), d1(7 downto 4),
             s, y(7 downto 4));
end;
```

T2: HARDWARE DESCRIPTION LANGUAGES (A3. Structural Modeling Accessing Part of Buses)

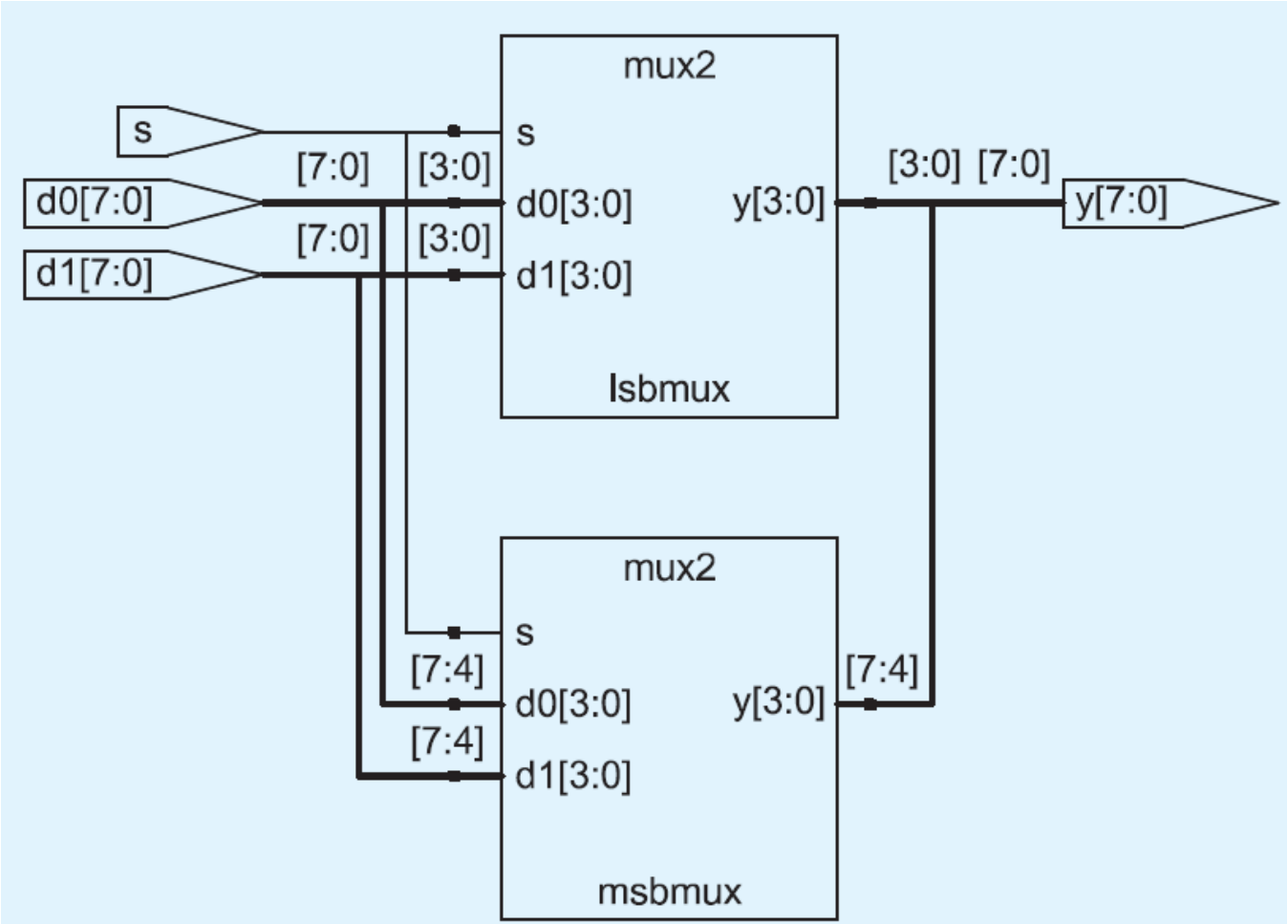


FIGURE A.16 mux2_8

Example A.20 Register

SystemVerilog

```
module flop(input  logic        clk,
            input  logic [3:0] d,
            output logic [3:0] q);

  always_ff @(posedge clk)
    q <= d;
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A4. Sequential Logic - Registers)

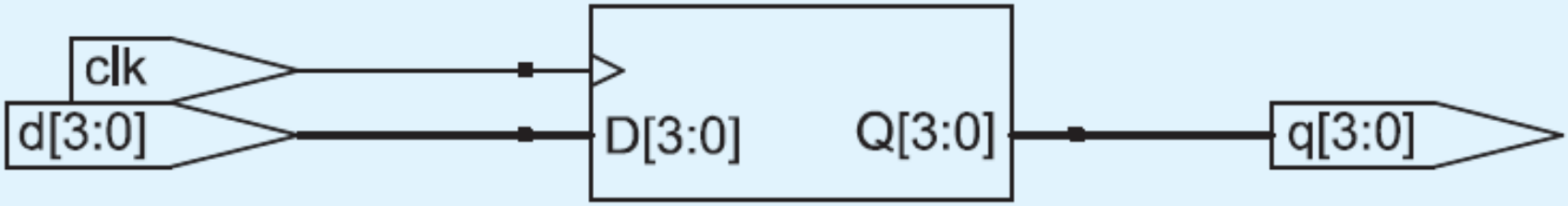


FIGURE A.17 flop

T2: HARDWARE DESCRIPTION LANGUAGES (A4. Sequential Logic - Reseteable Register)

Example A.21 Resettable Register

SystemVerilog

```
module flopr(input  logic        clk,
             input  logic        reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // synchronous reset
  always_ff @(posedge clk)
     if (reset) q <= 4'b0;
     else       q <= d;
endmodule

module flopr(input  logic        clk,
             input  logic        reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A4. Sequential Logic - Reseteable Register)

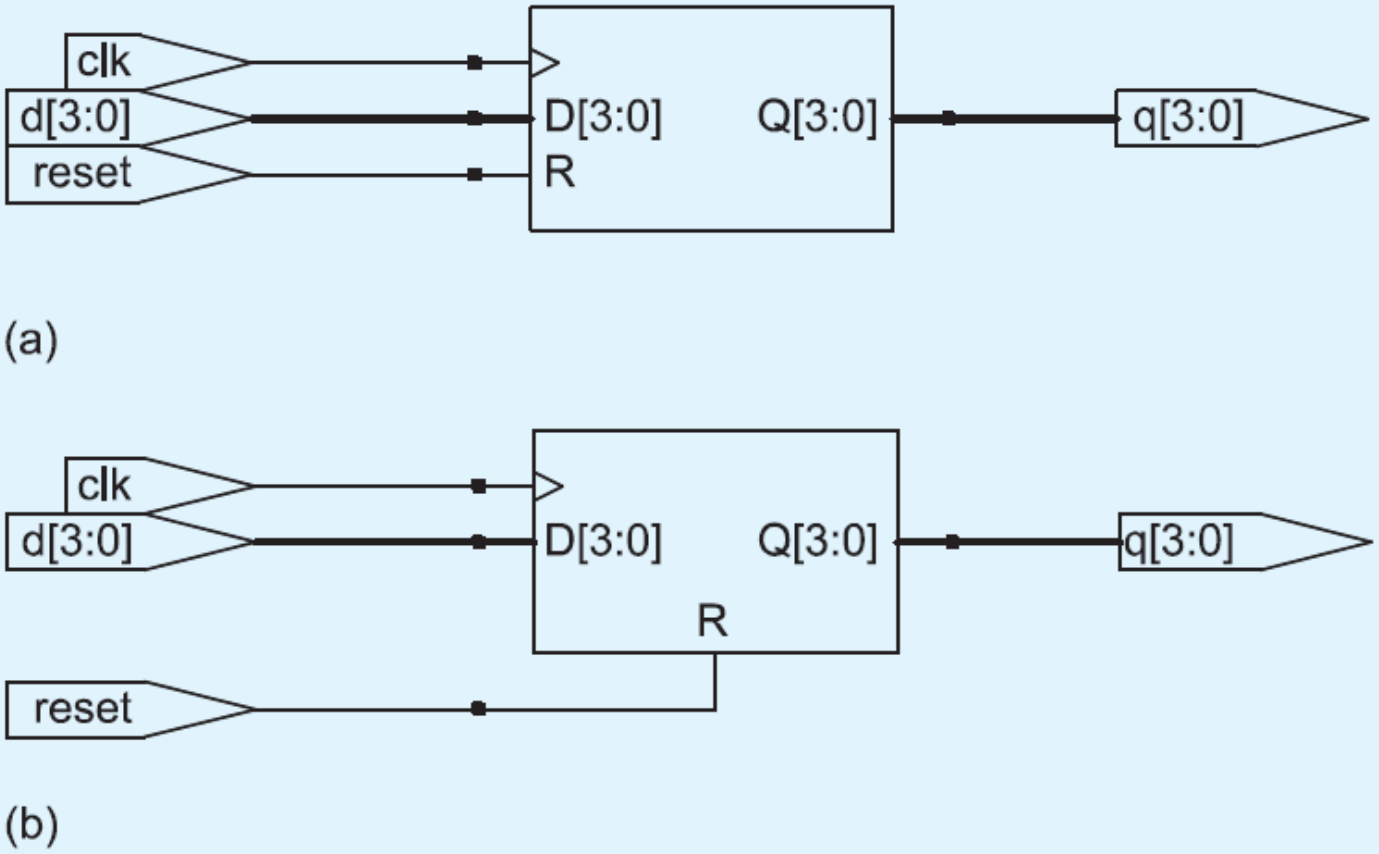


FIGURE A.18 flopr (a) synchronous reset, (b) asynchronous reset

T2: HARDWARE DESCRIPTION LANGUAGES (A4. Sequential Logic – Enabled Registers)

SystemVerilog

```
module flopenr(input  logic        clk,
               input  logic        reset,
               input  logic        en,
               input  logic [3:0]  d,
               output logic [3:0]  q);

  // synchronous reset
  always_ff @(posedge clk)
    if      (reset) q <= 4'b0;
    else if (en)    q <= d;
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A4. Sequential Logic)

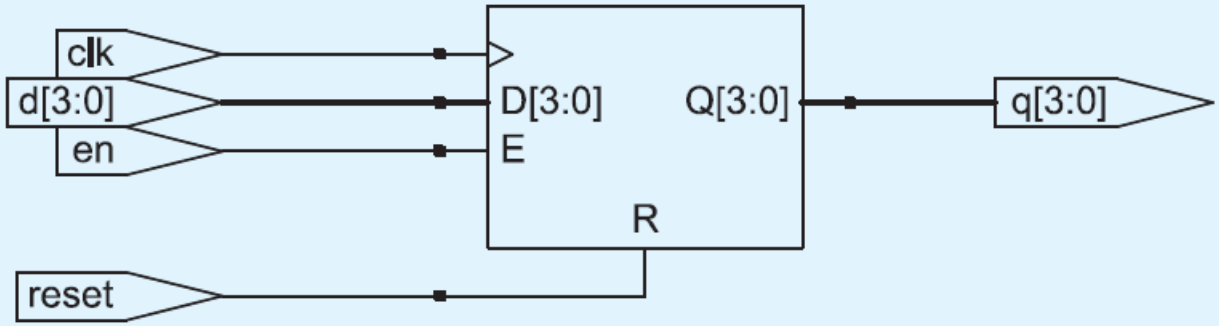


FIGURE A.19 flopenr

T2: HARDWARE DESCRIPTION LANGUAGES (A4. Sequential Logic – Multiple Registers)

Example A.23 Synchronizer

SystemVerilog

```
module sync(input  logic clk,
            input  logic d,
            output logic q);

  logic n1;

  always_ff @(posedge clk)
    begin
      n1 <= d;
      q <= n1;
    end
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A4. Sequential Logic - Multiple Registers)

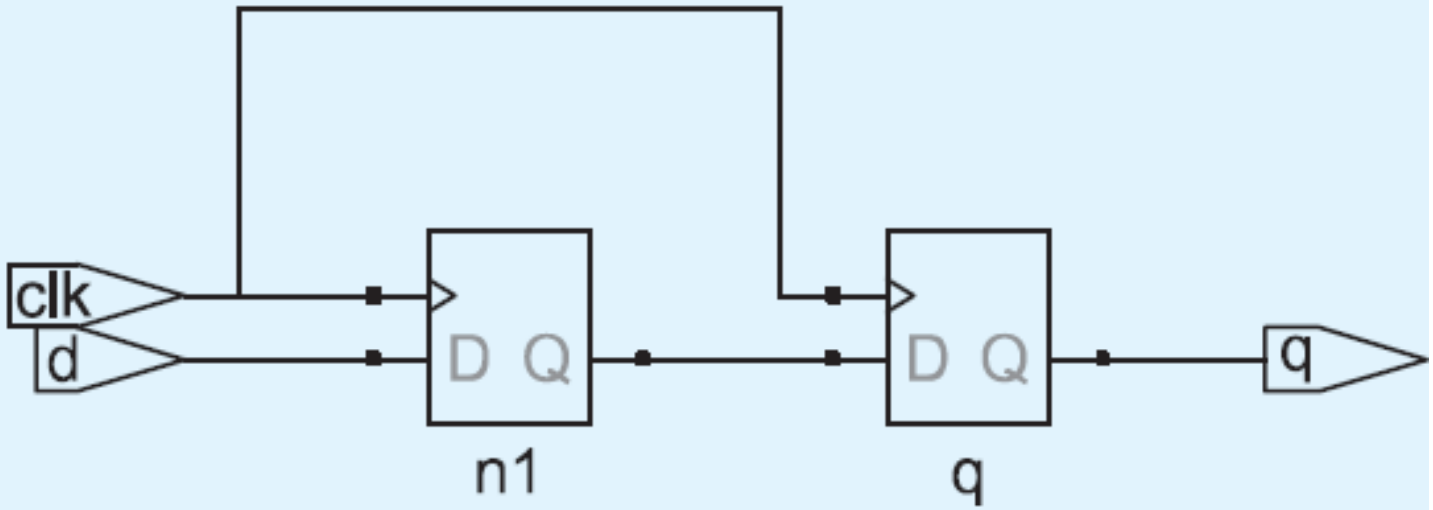


FIGURE A.21 sync

T2: HARDWARE DESCRIPTION LANGUAGES (A4. Sequential Logic - Latches)

Example A.24 D Latch

SystemVerilog

```
module latch(input  logic        clk,
             input  logic [3:0] d,
             output logic [3:0] q);

  always_latch
    if (clk) q <= d;
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A4. Sequential Logic – Latches)

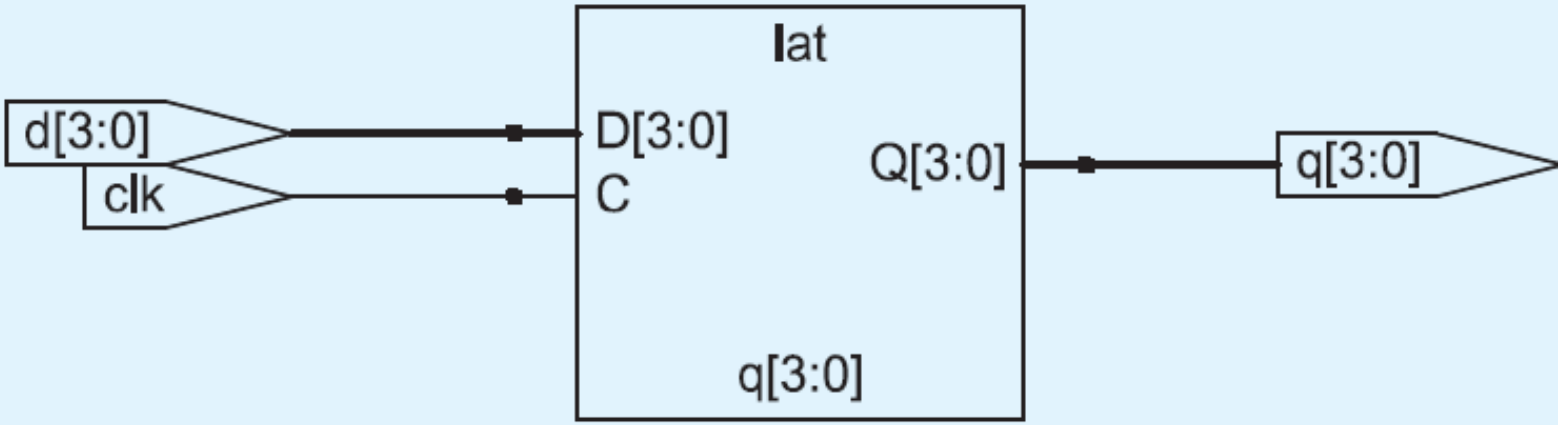


FIGURE A.22 latch

T2: HARDWARE DESCRIPTION LANGUAGES (A4. Sequential Logic – Counters – Behavioral Style)

Example A.25 Counter (Behavioral Style)

SystemVerilog

```
module counter(input  logic        clk,
               input  logic        reset,
               output logic [3:0] q);

  always_ff @(posedge clk)
    if (reset) q <= 4'b0;
    else       q <= q+1;
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A4. Sequential Logic – Counters – Behavioral Style)

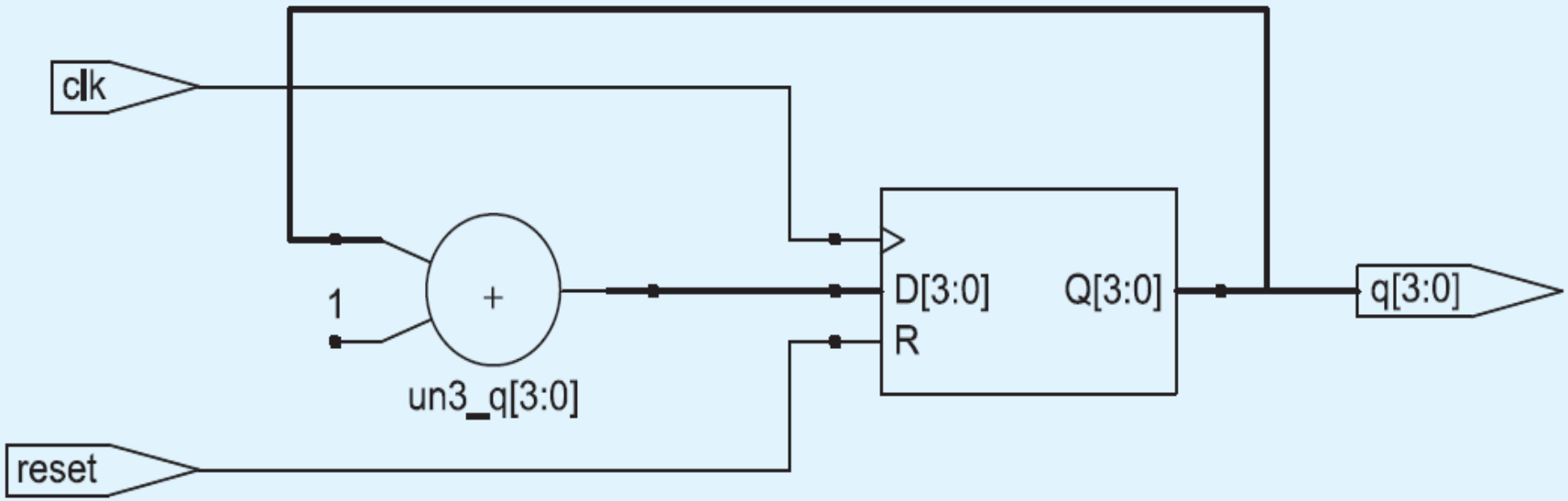


FIGURE A.23 Counter (behavioral)

T2: HARDWARE DESCRIPTION LANGUAGES (A4. Sequential Logic – Counters – Structural Style)

Example A.26 Counter (Structural Style)

SystemVerilog

```
module counter(input  logic        clk,
               input  logic        reset,
               output logic [3:0] q);

   logic [3:0] nextq;

   flopr qflop(clk, reset, nextq, q);
   adder inc(q, 4'b0001, nextq);
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A4. Sequential Logic – Counters – Structural Style)

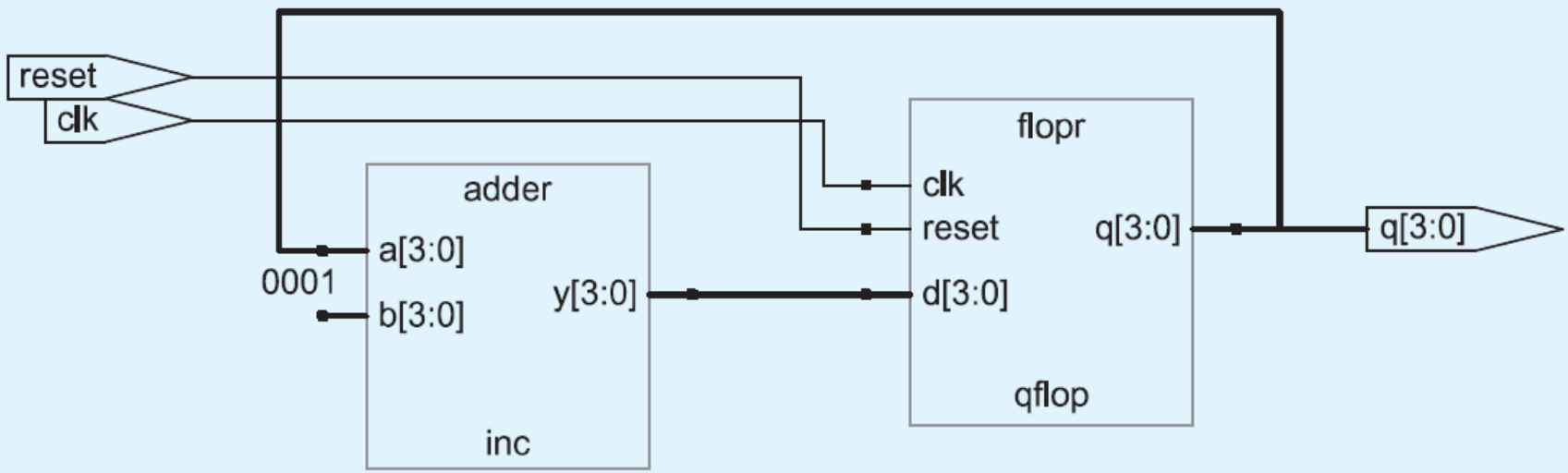


FIGURE A.24 Counter (structural)

T2: HARDWARE DESCRIPTION LANGUAGES (A4. Sequential Logic – Shift Registers)

Example A.27 Shift Register with Parallel Load

SystemVerilog

```
module shiftreg(input  logic        clk,
                input  logic        reset, load,
                input  logic        sin,
                input  logic [3:0]  d,
                output logic [3:0]  q,
                output logic        sout);

  always_ff @(posedge clk)
    if (reset)      q <= 0;
    else if (load)  q <= d;
    else            q <= {q[2:0], sin};

  assign sout = q[3];
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A4. Sequential Logic - Shift Registers)

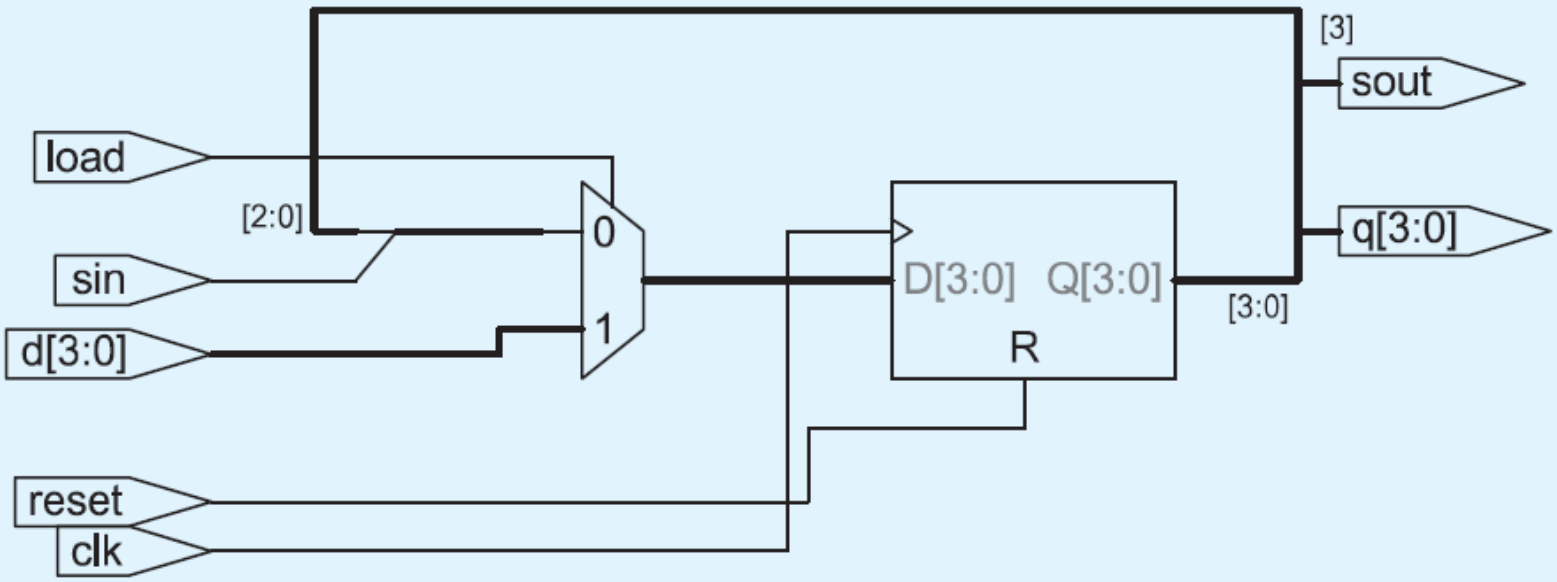


FIGURE A.25 Synthesized shiftreg

T2: HARDWARE DESCRIPTION LANGUAGES (A5. Combinational With Always)

Example A.28 Inverter (Using `always / process`)

SystemVerilog

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  always_comb
    y = ~a;
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A5. Combinational With Always)

SystemVerilog

In an `always` statement, = indicates a blocking assignment and <= indicates a nonblocking assignment.

Do not confuse either type with continuous assignment using the `assign` statement. `assign` statements are normally used outside `always` statements and are also evaluated concurrently.

T2: HARDWARE DESCRIPTION LANGUAGES (A5. Combinational With Always)

Example A.29 Full Adder (Using `always / process`)

SystemVerilog

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);

  logic p, g;

  always_comb
    begin
      p = a ^ b;   // blocking
      g = a & b;   // blocking

      s = p ^ cin;
      cout = g | (p & cin);
    end
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A5. Combinational With Always - Case Statements)

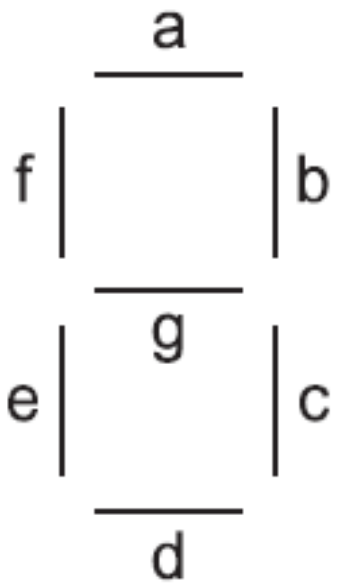


FIGURE A.26
7-segment display

T2: HARDWARE DESCRIPTION LANGUAGES (A5. Combinational With Always - Case Statements)

Example A.30 Seven-Segment Display Decoder

SystemVerilog

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);

  always_comb
    case (data)
      //                  abc_defg
      0: segments = 7'b111_1110;
      1: segments = 7'b011_0000;
      2: segments = 7'b110_1101;
      3: segments = 7'b111_1001;
      4: segments = 7'b011_0011;
      5: segments = 7'b101_1011;
      6: segments = 7'b101_1111;
      7: segments = 7'b111_0000;
      8: segments = 7'b111_1111;
      9: segments = 7'b111_1011;
      default: segments = 7'b000_0000;
    endcase
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A5. Combinational With Always - Case Statements)

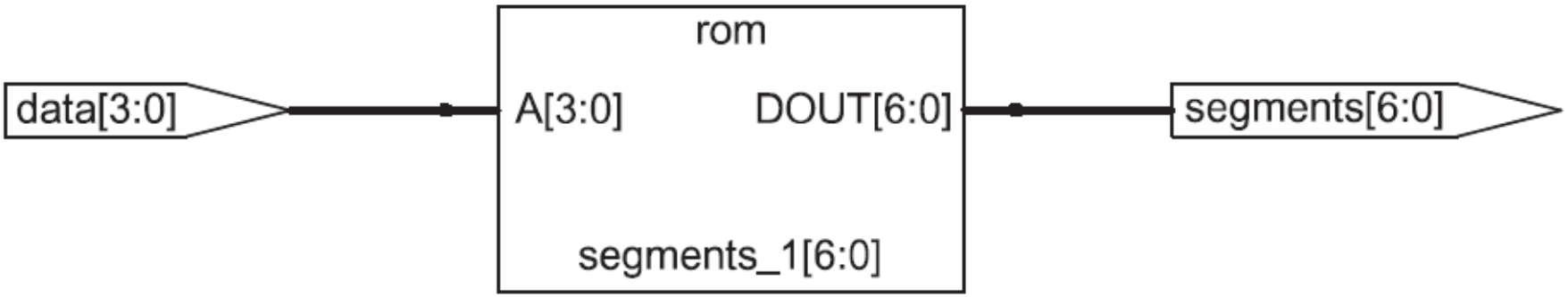


FIGURE A.27 sevenseg

T2: HARDWARE DESCRIPTION LANGUAGES (A5. Combinational With Always - Case Statements)

Example A.31 3:8 Decoder

SystemVerilog

```
module decoder3_8(input  logic [2:0] a,
                  output logic [7:0] y);

  always_comb
    case (a)
      3'b000: y = 8'b00000001;
      3'b001: y = 8'b00000010;
      3'b010: y = 8'b00000100;
      3'b011: y = 8'b00001000;
      3'b100: y = 8'b00010000;
      3'b101: y = 8'b00100000;
      3'b110: y = 8'b01000000;
      3'b111: y = 8'b10000000;
    endcase
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A5. Combinational With Always - Case Statements)

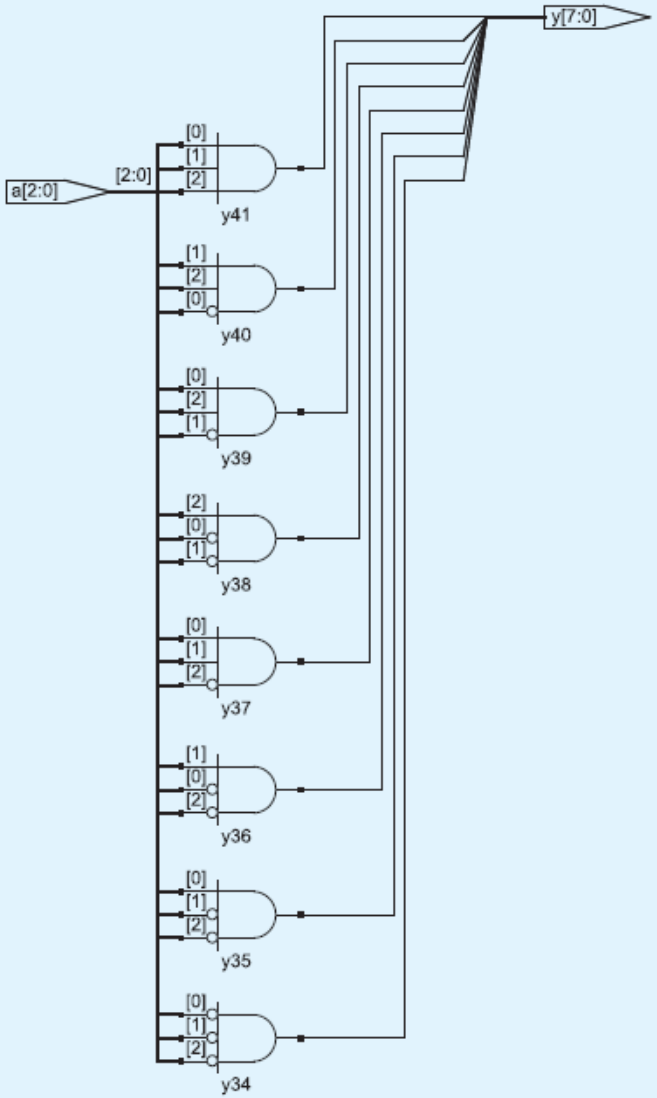
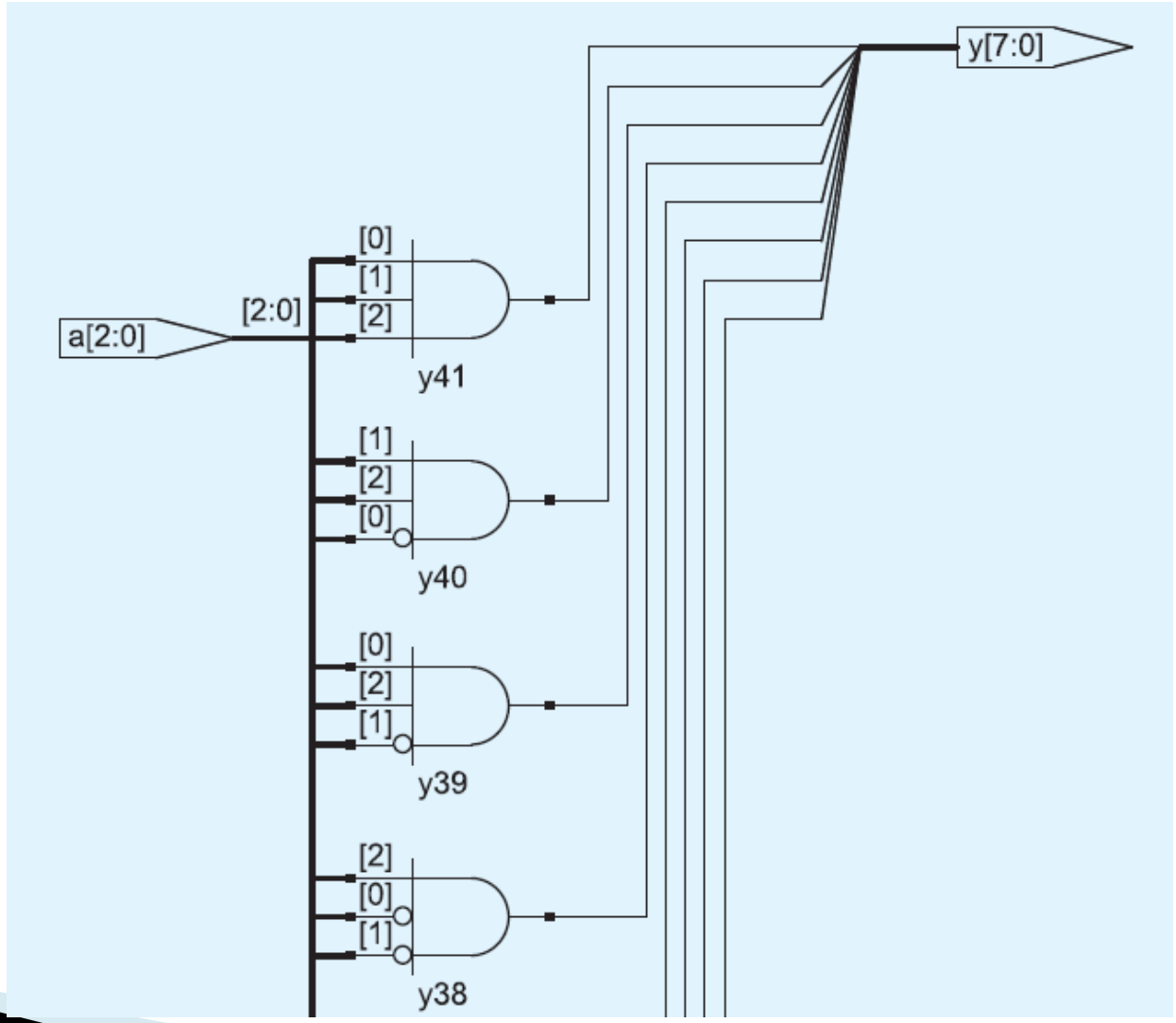


FIGURE A.28 3:8 decoder

T2: HARDWARE DESCRIPTION LANGUAGES (A5. Combinational With Always - Case Statements)



T2: HARDWARE DESCRIPTION LANGUAGES (A5. Combinational With Always - Case Statements)

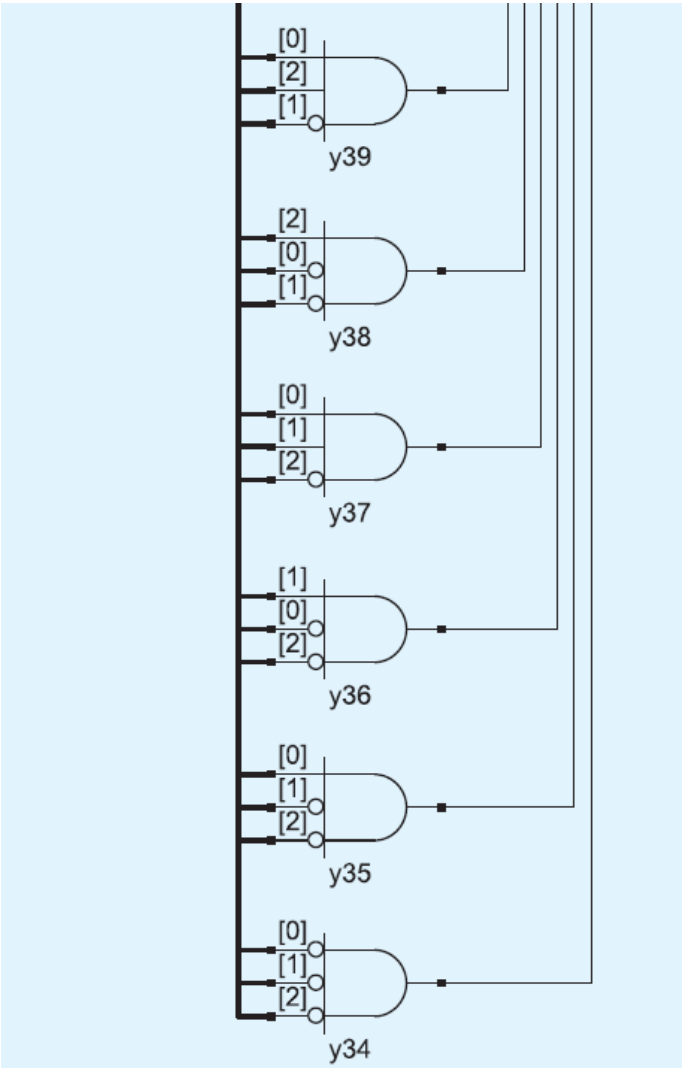


FIGURE A.28 3:8 decoder

T2: HARDWARE DESCRIPTION LANGUAGES (A5. Combinational With Always – If Statements)

Example A.32 Priority Circuit

SystemVerilog

```
module priorityckt(input  logic [3:0] a,
                   output logic [3:0] y);

  always_comb
    if      (a[3]) y = 4'b1000;
    else if (a[2]) y = 4'b0100;
    else if (a[1]) y = 4'b0010;
    else if (a[0]) y = 4'b0001;
    else           y = 4'b0000;
endmodule
```

In SystemVerilog, `if` statements must appear inside `always` statements.

T2: HARDWARE DESCRIPTION LANGUAGES (A5. Combinational With Always – If Statements)

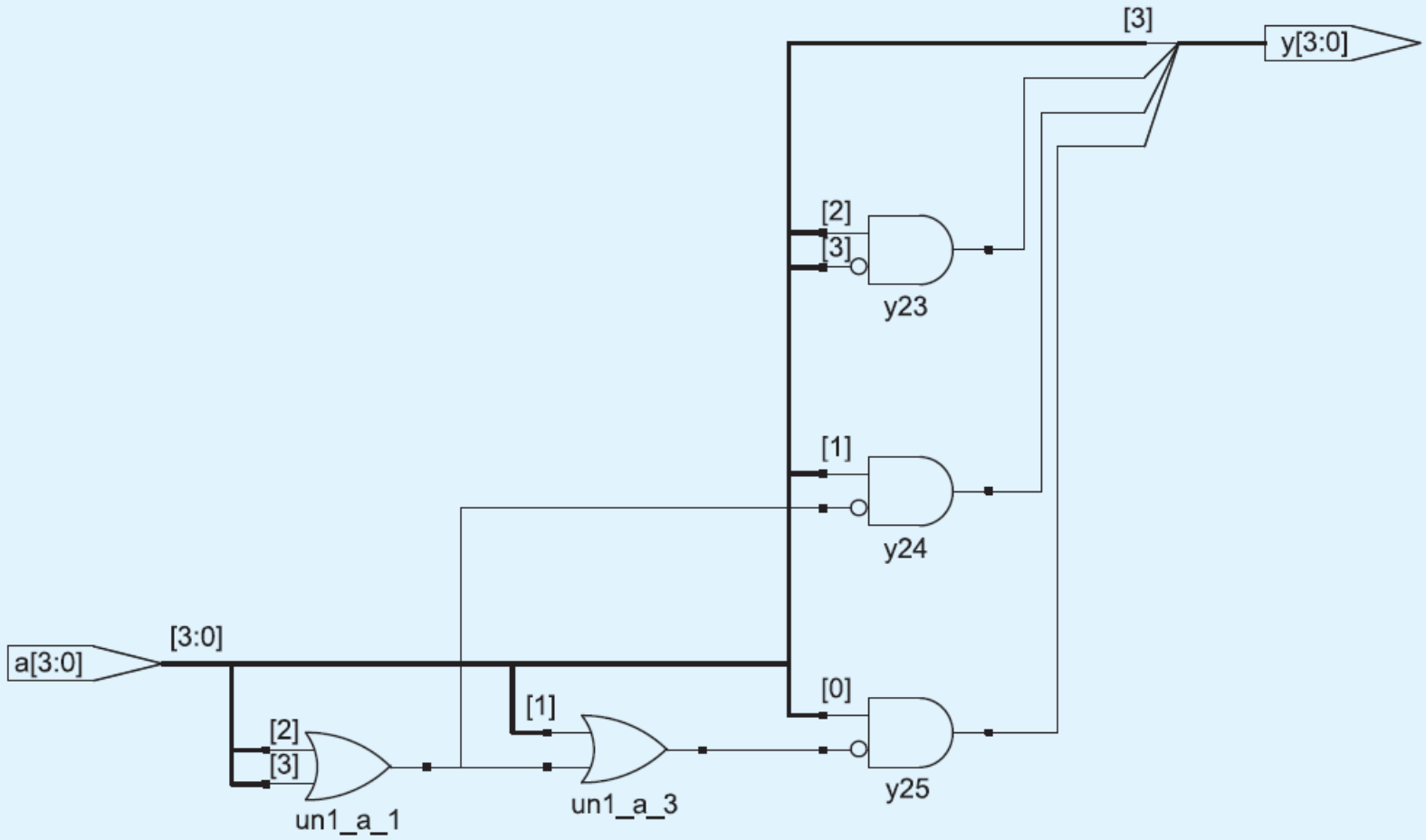


FIGURE A.29 Priority circuit

T2: HARDWARE DESCRIPTION LANGUAGES (A5. Combinational With Always – Casez Statement)

Example A.33 Priority Circuit Using `casez`

SystemVerilog

```
module priority_casez(input  logic [3:0] a,
                      output logic [3:0] y);

  always_comb
    casez(a)
      4'b1???: y = 4'b1000;
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
   endcase
endmodule
```

Synplify Pro synthesizes a slightly different circuit for this module, shown in Figure A.30, than it did for the priority circuit in Figure A.29. However, the circuits are logically equivalent.

T2: HARDWARE DESCRIPTION LANGUAGES (A5. Combinational With Always – Casez Statement)

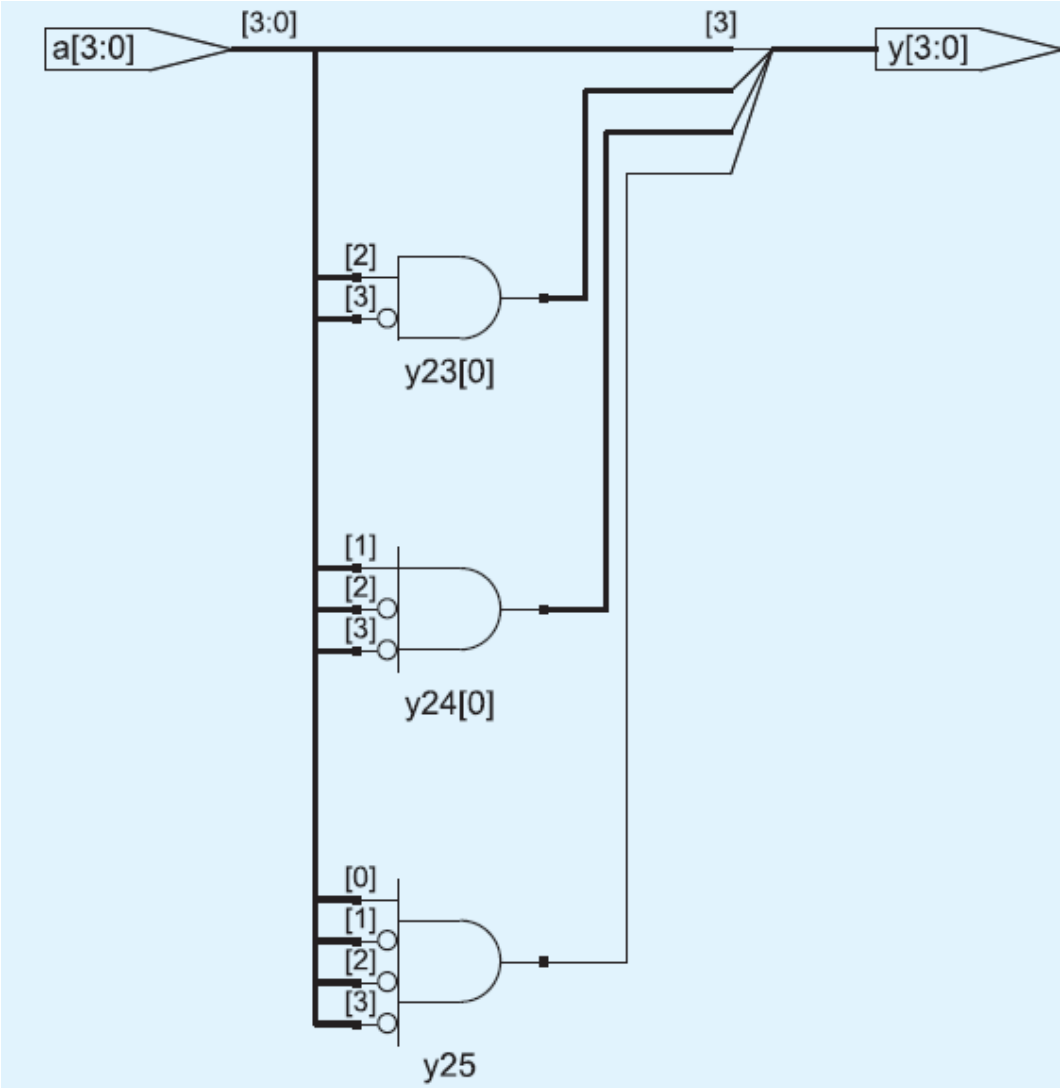


FIGURE A.30 priority_casez

T2: HARDWARE DESCRIPTION LANGUAGES (A6 Finite State Machines _ Mealy and Moore)

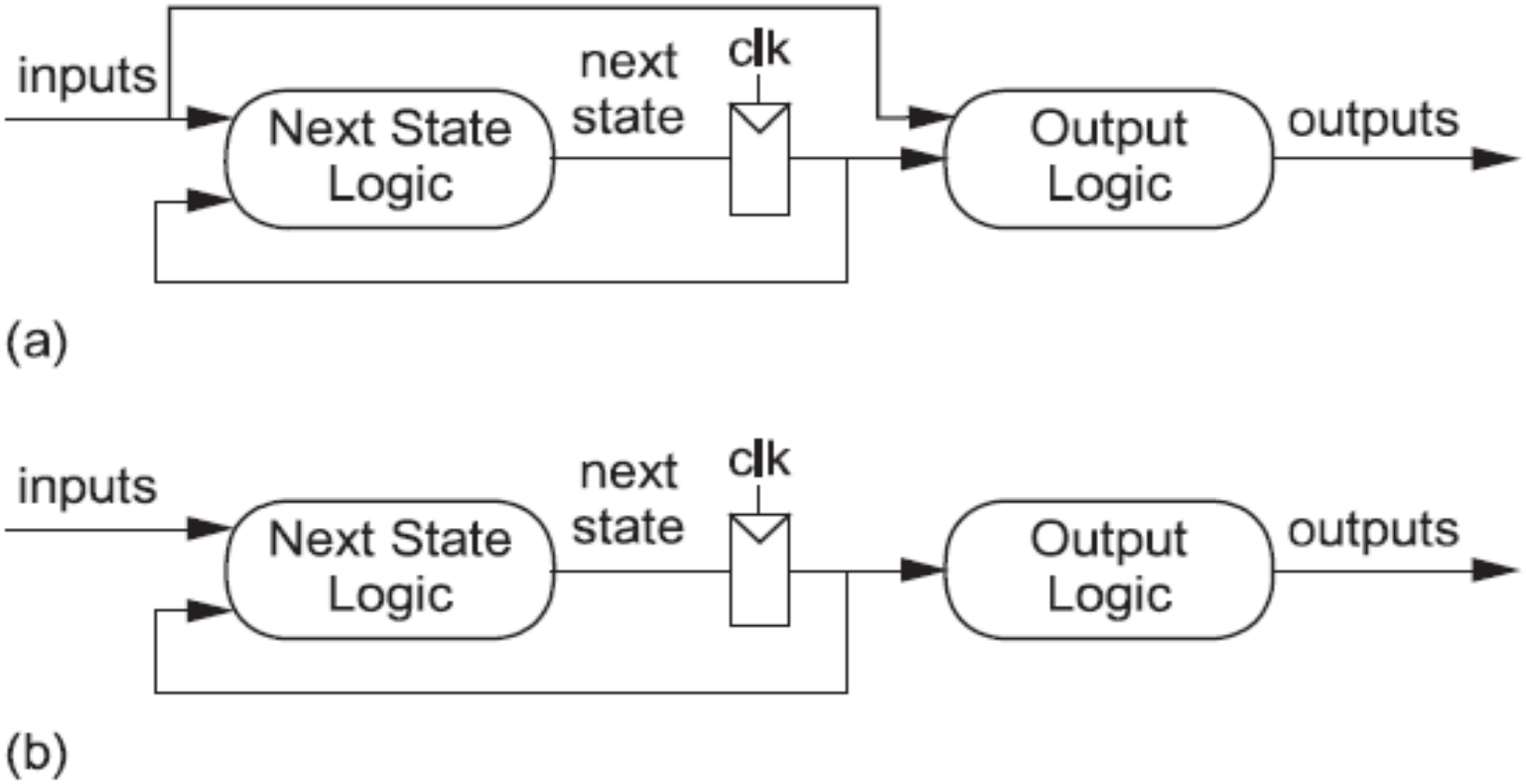


FIGURE A.32 Mealy and Moore machines

T2: HARDWARE DESCRIPTION LANGUAGES (A6 Finite State Machines – Divide by 3 example)

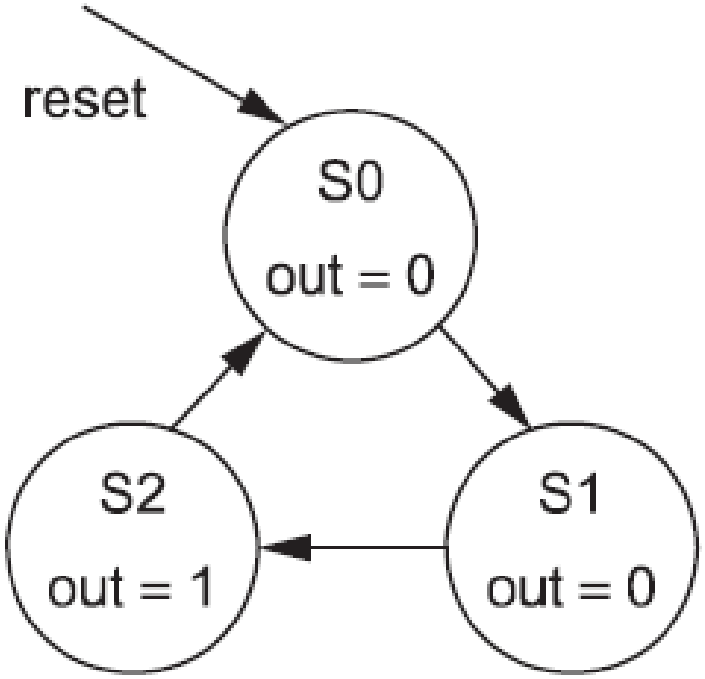


FIGURE A.33 Divide-by-3 counter state transition diagram

T2: HARDWARE DESCRIPTION LANGUAGES (A6 Finite State Machines – Divide by 3 example)

Example A.36 Divide-by-3 Finite State Machine

SystemVerilog

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic y);

  logic [1:0] state, nextstate;

  // State Register
  always_ff @(posedge clk)
    if (reset) state <= 2'b00;
    else       state <= nextstate;

  // Next State Logic
  always_comb
    case (state)
      2'b00: nextstate = 2'b01;
      2'b01: nextstate = 2'b10;
      2'b10: nextstate = 2'b00;
      default: nextstate = 2'b00;
    endcase
```

T2: HARDWARE DESCRIPTION LANGUAGES (A6 Finite State Machines – Divide by 3 example)

SystemVerilog (continued)

```
// Output Logic
  assign y = (state == 2'b00);
endmodule
```

Notice how a `case` statement is used to define the state transition table. Because the next state logic should be combinational, a default is necessary even though the state `11` should never arise.

The output `y` is 1 when the state is `00`. The *equality comparison* `a == b` evaluates to 1 if `a` equals `b` and 0 otherwise. The *inequality comparison* `a != b` does the inverse, evaluating to 1 if `a` does not equal `b`.

T2: HARDWARE DESCRIPTION LANGUAGES (A6 Finite State Machines – Divide by 3 example)

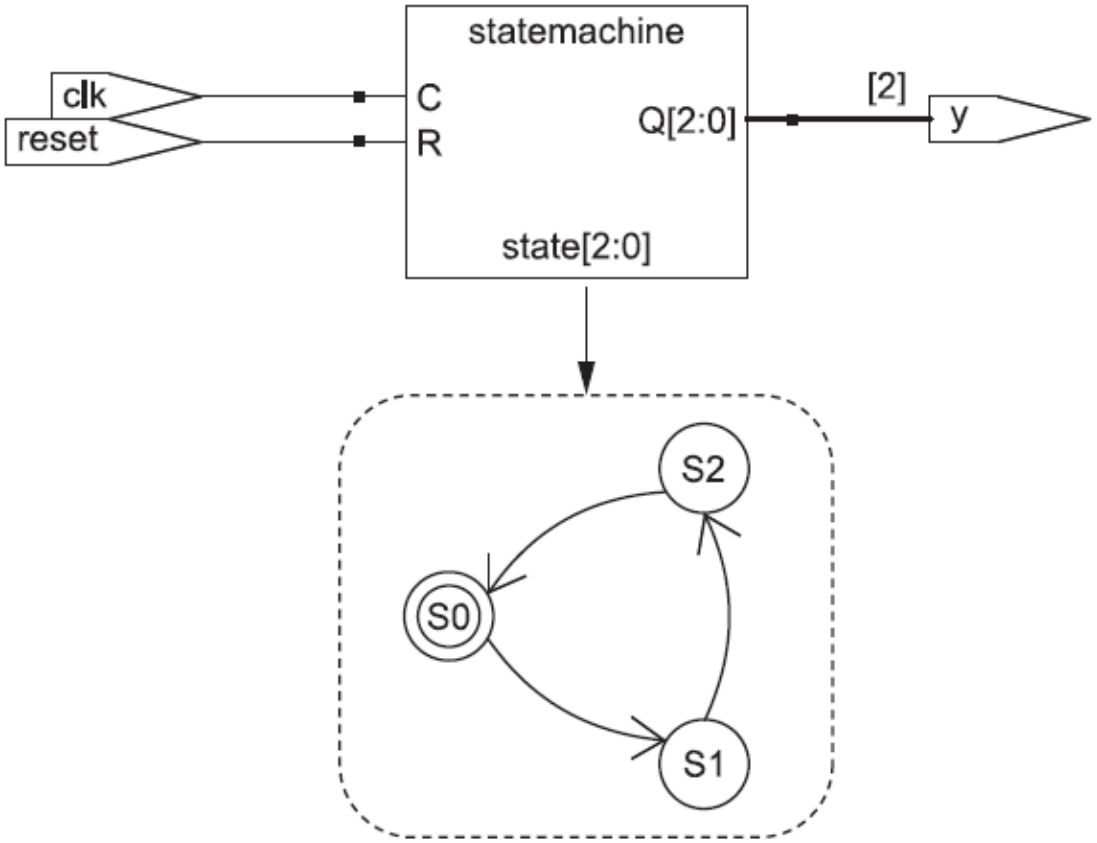


FIGURE A.34 divideby3fsm

T2: HARDWARE DESCRIPTION LANGUAGES (A6 Finite State Machines – State Enumeration)

Example A.37 State Enumeration

SystemVerilog

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic y);

  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype state, nextstate;

  // State Register
  always_ff @(posedge clk)
    if (reset) state <= S0;
    else       state <= nextstate;

  // Next State Logic
  always_comb
    case (state)
      S0: nextstate = S1;
      S1: nextstate = S2;
      S2: nextstate = S0;
      default: nextstate = S0;
    endcase

  // Output Logic
  assign y = (state == S0);
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A6 Finite State Machines – State Enumeration)

```
typedef enum logic [2:0] {S0 = 3'b001,
                          S1 = 3'b010,
                          S2 = 3'b100} statetype;
```

T2: HARDWARE DESCRIPTION LANGUAGES (A6 Finite State Machines – State Enumeration)

SystemVerilog

```
// Output Logic
assign y = (state == S0 | state == S1);
```

T2: HARDWARE DESCRIPTION LANGUAGES (A6 Finite State Machines – FSM With Inputs)

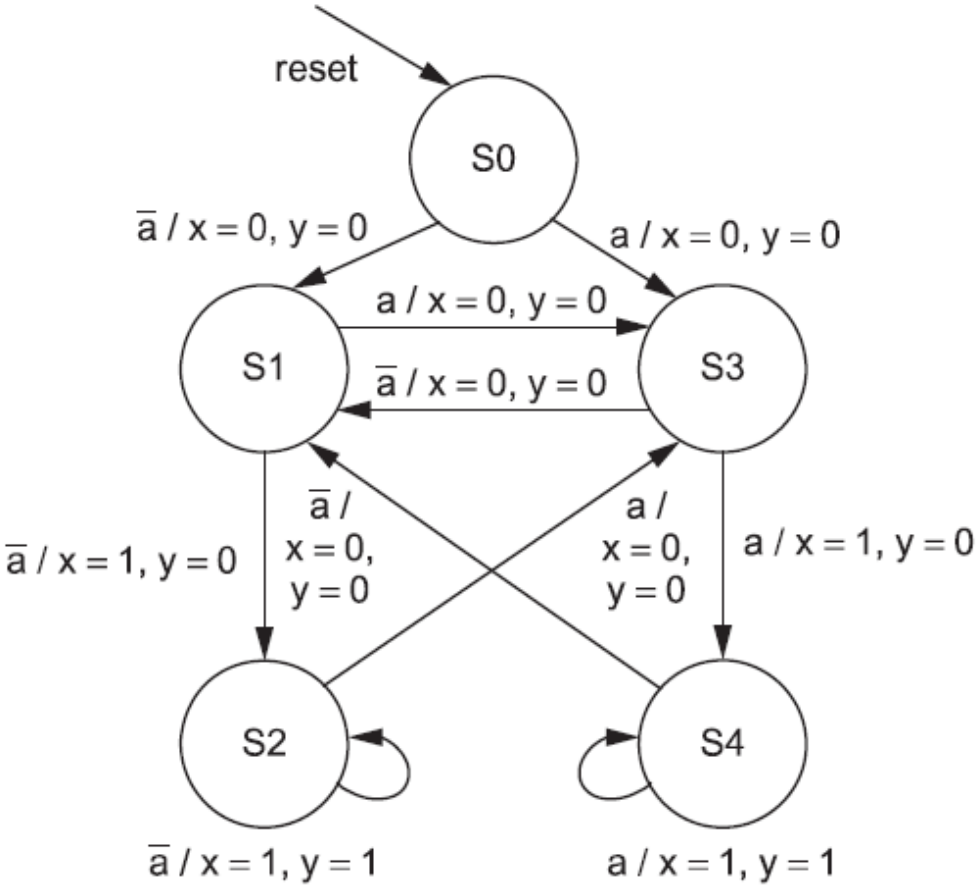


FIGURE A.35 History FSM state transition diagram

T2: HARDWARE DESCRIPTION LANGUAGES (A6 Finite State Machines – FSM With Inputs)

Example A.38 History FSM

SystemVerilog

```
module historyFSM(input  logic clk,
                  input  logic reset,
                  input  logic a,
                  output logic x, y);

  typedef enum logic [2:0]
    {S0, S1, S2, S3, S4} statetype;
  statetype state, nextstate;

  // State Register
  always_ff @(posedge clk)
    if (reset) state <= S0;
    else       state <= nextstate;

  // Next State Logic
  always_comb
    case (state)
      S0: if (a) nextstate = S3;
          else   nextstate = S1;
      S1: if (a) nextstate = S3;
          else   nextstate = S2;
      S2: if (a) nextstate = S3;
          else   nextstate = S2;
      S3: if (a) nextstate = S4;
```

T2: HARDWARE DESCRIPTION LANGUAGES (A6 Finite State Machines - FSM With Inputs)

```
// Next State Logic
always_comb
  case (state)
    S0: if (a) nextstate = S3;
        else   nextstate = S1;
    S1: if (a) nextstate = S3;
        else   nextstate = S2;
    S2: if (a) nextstate = S3;
        else   nextstate = S2;
    S3: if (a) nextstate = S4;
        else   nextstate = S1;
    S4: if (a) nextstate = S4;
        else   nextstate = S1;
    default:   nextstate = S0;
  endcase
```

T2: HARDWARE DESCRIPTION LANGUAGES (A6 Finite State Machines – FSM With Inputs)

SystemVerilog (continued)

```
// Output Logic
assign x = ((state == S1 | state == S2) & ~a) |
           ((state == S3 | state == S4) & a);
assign y = (state == S2 & ~a) | (state == S4 & a);
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A6 Finite State Machines – FSM With Inputs)

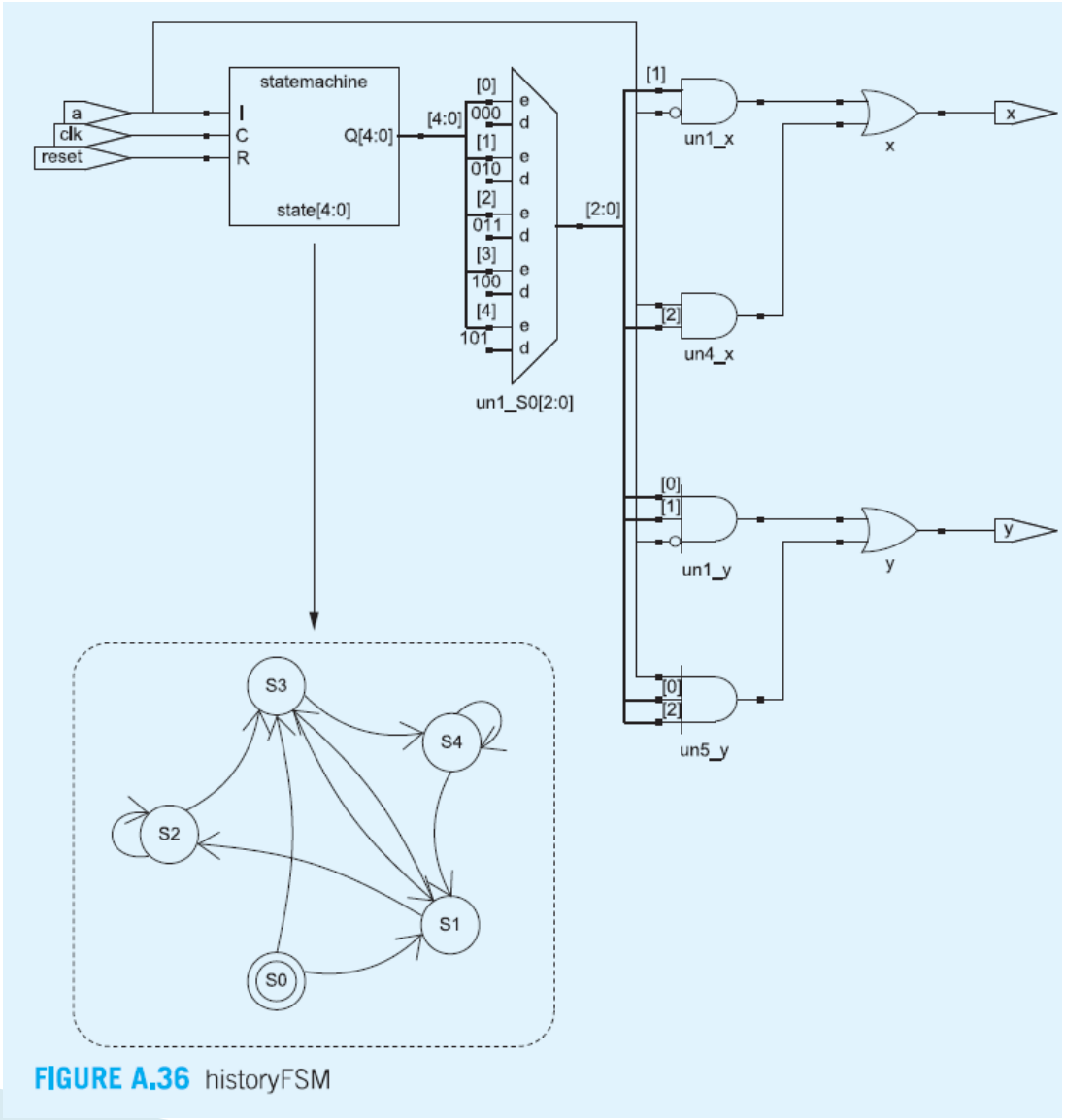
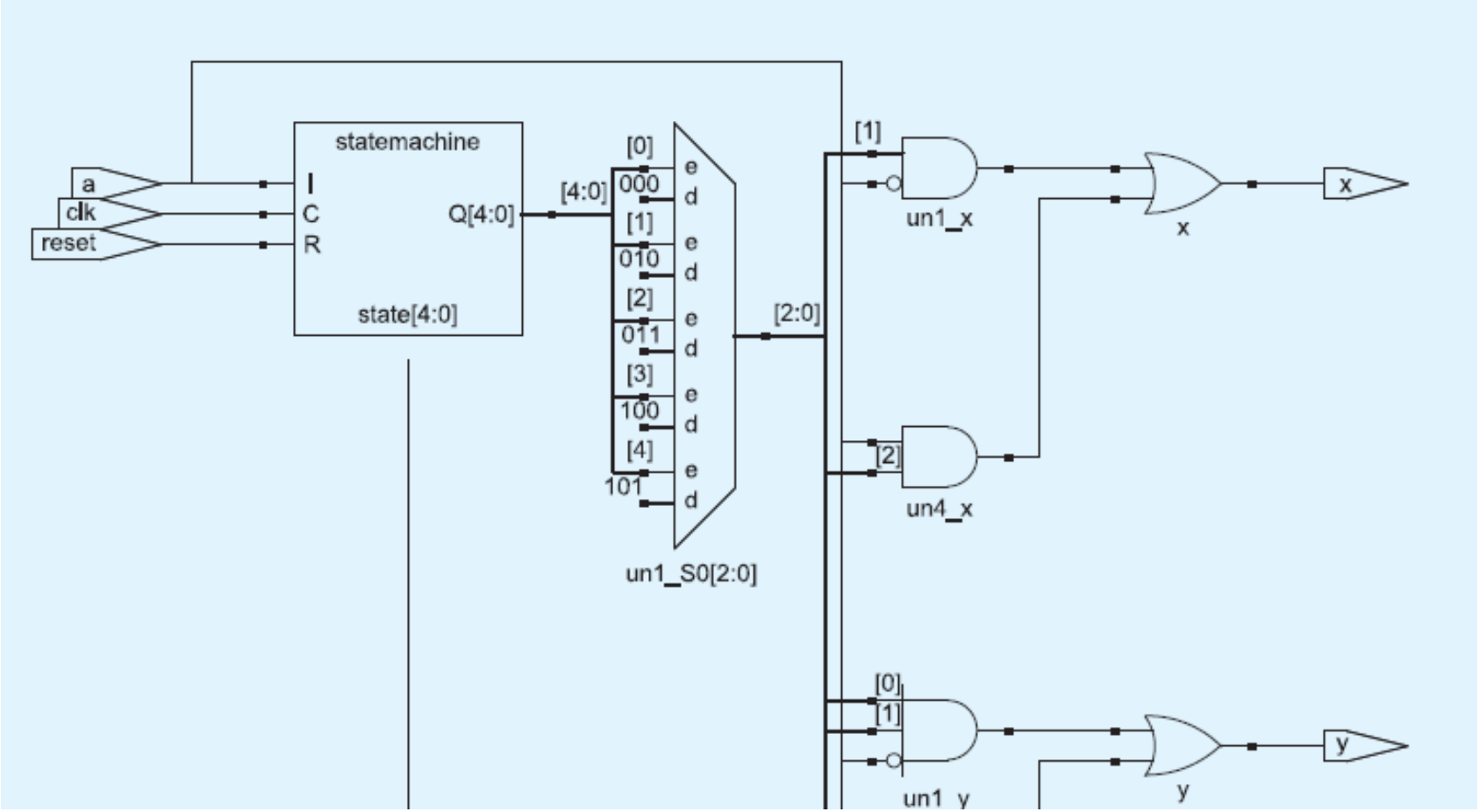


FIGURE A.36 historyFSM

T2: HARDWARE DESCRIPTION LANGUAGES (A6 Finite State Machines – FSM With Inputs)



T2: HARDWARE DESCRIPTION LANGUAGES (A6 Finite State Machines)

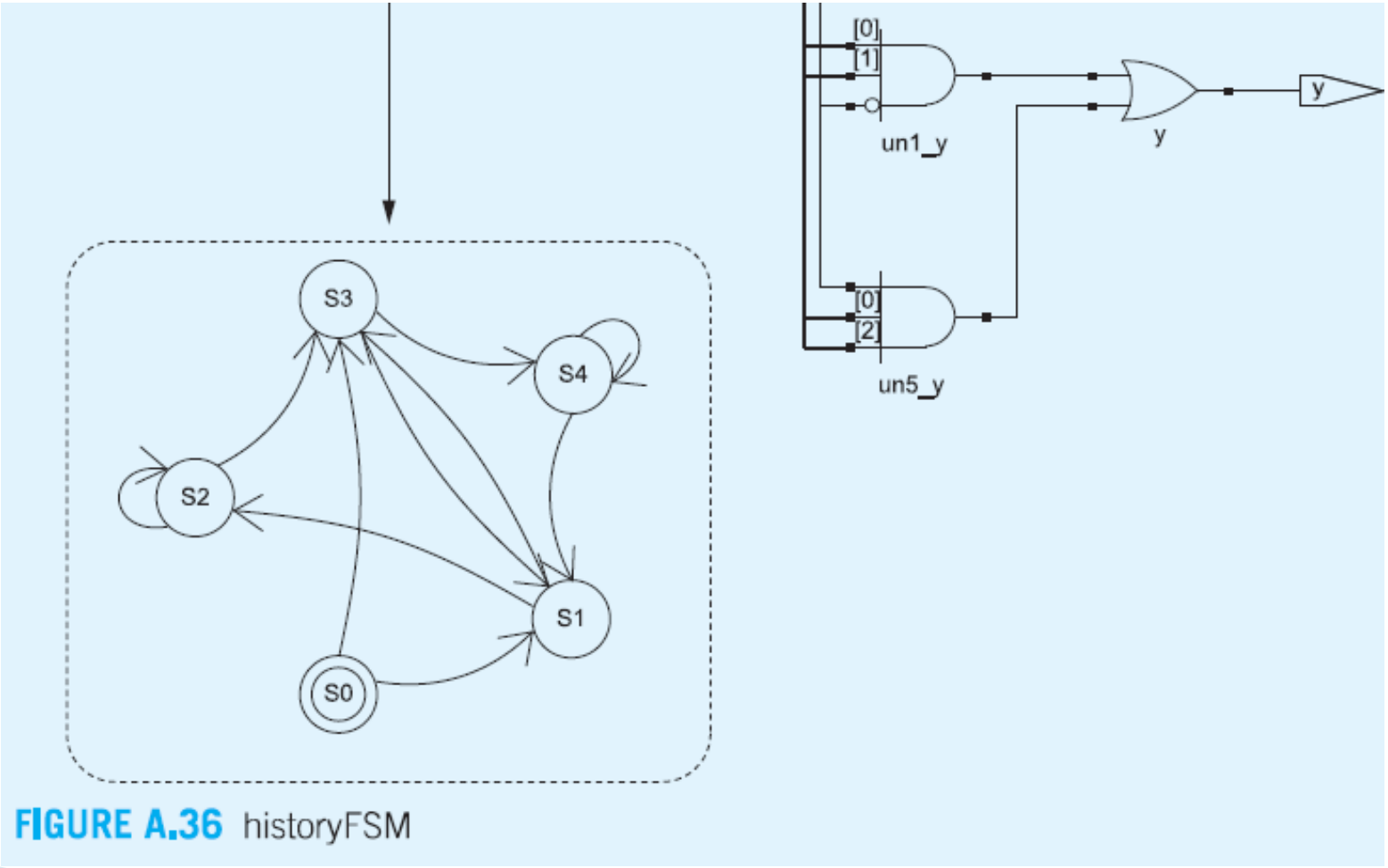


FIGURE A.36 historyFSM

T2: HARDWARE DESCRIPTION LANGUAGES (A7 A7_TypeIdyosincracies)

SystemVerilog

Standard Verilog primarily uses two types: `reg` and `wire`. Despite its name, a `reg` signal might or might not be associated with a register. This was a great source of confusion for those learning the language. SystemVerilog introduced the `logic` type and relaxed some requirements to eliminate the confusion; hence, the examples in this appendix use `logic`. This section explains the `reg` and `wire` types in more detail for those who need to read legacy Verilog code.

T2: HARDWARE DESCRIPTION LANGUAGES (A7 A7_TypeIdyosincracies)

In Verilog, if a signal appears on the left-hand side of <= or = in an `always` block, it must be declared as `reg`. Otherwise, it should be declared as `wire`. Hence, a `reg` signal might be the output of a flip-flop, a latch, or combinational logic, depending on the sensitivity list and statement of an `always` block.

Input and output ports default to the `wire` type unless their type is explicitly specified as `reg`. The following example shows how a flip-flop is described in conventional Verilog. Notice that `clk` and `d` default to `wire`, while `q` is explicitly defined as `reg` because it appears on the left-hand side of <= in the `always` block.

T2: HARDWARE DESCRIPTION LANGUAGES (A7 A7_TypeIdyosincracies)

```
module flop(input              clk,
            input      [3:0] d,
            output reg [3:0] q);

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

T2: HARDWARE DESCRIPTION LANGUAGES (A7 A7_TypeIdyosincracies)

SystemVerilog introduces the `logic` type. `logic` is a synonym for `reg` and avoids misleading users about whether it is actually a flip-flop. Moreover, SystemVerilog relaxes the rules on `assign` statements and hierarchical port instantiations so `logic` can be used outside `always` blocks where a `wire` traditionally would be required. Thus, nearly all SystemVerilog signals can be `logic`. The exception is that signals with multiple drivers (e.g., a tristate bus) must be declared as a net, as described in Example A.11. This rule allows SystemVerilog to generate an error message rather than an `x` value when a `logic` signal is accidentally connected to multiple drivers.

T2: HARDWARE DESCRIPTION LANGUAGES (A7 A7_TypeIdyosincracies)

The most common type of net is called a **`wire`** or **`tri`**. These two types are synonymous, but **`wire`** is conventionally used when a single driver is present and **`tri`** is used when multiple drivers are present. Thus, **`wire`** is obsolete in SystemVerilog because **`logic`** is preferred for signals with a single driver.

When a **`tri`** net is driven to a single value by one or more drivers, it takes on that value. When it is undriven, it floats (**`z`**). When it is driven to different values (0, 1, or **`x`**) by multiple drivers, it is in contention (**`x`**).

There are other net types that resolve differently when undriven or driven by multiple sources. The other types are rarely

T2: HARDWARE DESCRIPTION LANGUAGES (A7 A7_TypeIdyosincracies)

SystemVerilog (continued)

used, but can be substituted anywhere a `tri` net would normally appear (e.g., for signals with multiple drivers). Each is described in Table A.7:

TABLE A.7 net resolution

Net Type	No Driver	Conflicting Drivers
tri	z	x
triand	z	0 if any are 0
trior	z	1 if any are 1
trireg	previous value	x
tri0	0	x
tri1	1	x

T2: HARDWARE DESCRIPTION LANGUAGES (A7 A7_TypeIdyosincracies)

Most operations such as addition, subtraction, and Boolean logic are identical whether a number is signed or unsigned. However, magnitude comparison, multiplication and arithmetic right shifts are performed differently for signed numbers.

In Verilog, nets are considered unsigned by default. Adding the *signed* modifier (e.g., `logic signed a [31:0]`) causes the net to be treated as signed.

T2: HARDWARE DESCRIPTION LANGUAGES (A7 A7_TypeIdyosincracies)

Example A.39 8:1 Multiplexer with Type Conversion

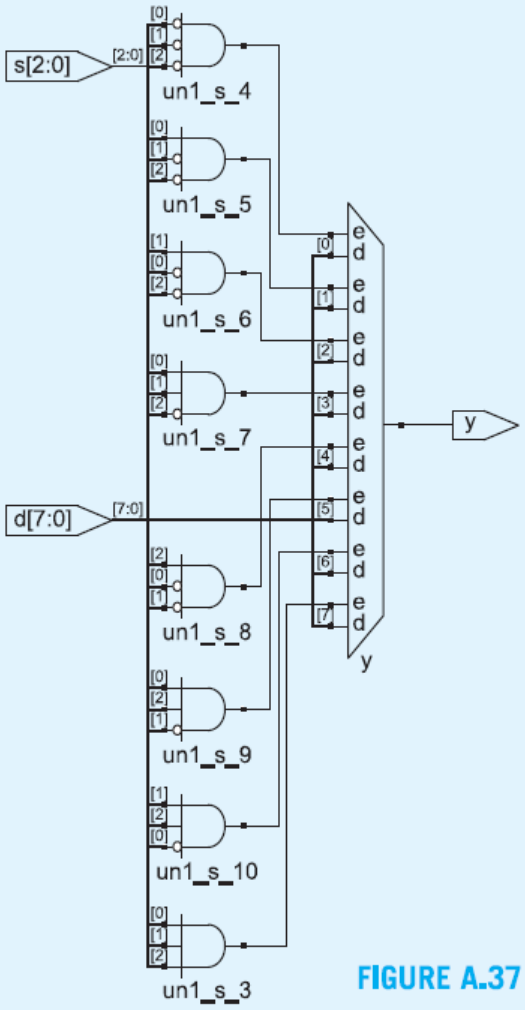


FIGURE A.37 mux8

T2: HARDWARE DESCRIPTION LANGUAGES (A7 A7_TypeIdyosincracies)

Example A.39 8:1 Multiplexer with Type Conversion (continued)

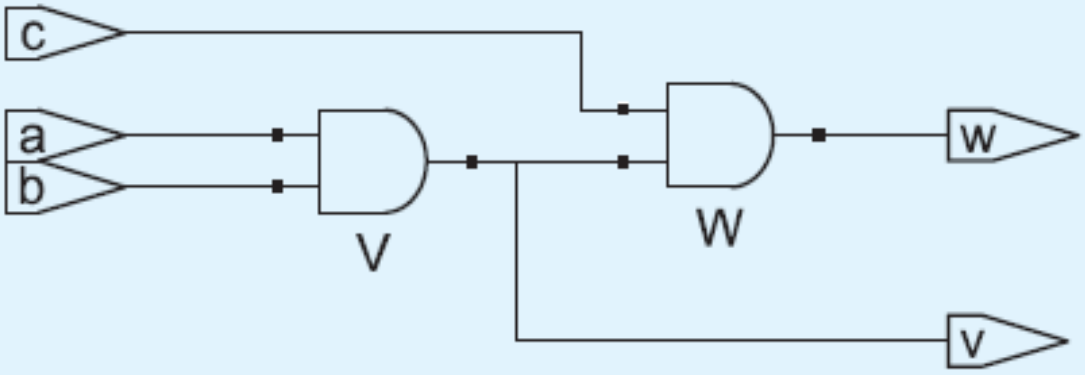


FIGURE A.38 and23

T2: HARDWARE DESCRIPTION LANGUAGES (A8 Parametrized Modules – N-bit Multiplexers)

Example A.40 Parameterized *N*-bit Multiplexers

SystemVerilog

```
module mux2
  #(parameter width = 8)
   (input  logic [width-1:0] d0, d1,
    input  logic             s,
    output logic [width-1:0] y);

   assign y = s ? d1 : d0;
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A8 Parametrized Modules – N-bit Multiplexers)

SystemVerilog (continued)

```
module mux4_8(input  logic [7:0] d0, d1, d2, d3,
              input  logic [1:0] s,
              output logic [7:0] y);

  logic [7:0] low, hi;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 himux(d2, d3, s[0], hi);
  mux2 outmux(low, hi, s[1], y);
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A8 Parametrized Modules – N-bit Multiplexers)

```
module mux4_12(input logic [11:0] d0, d1, d2, d3,
               input logic [1:0]  s,
               output logic [11:0] y);

  logic [11:0] low, hi;

  mux2 #(12) lowmux(d0, d1, s[0], low);
  mux2 #(12) himux(d2, d3, s[0], hi);
  mux2 #(12) outmux(low, hi, s[1], y);
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A8 Parametrized Modules – N-bit Multiplexers)

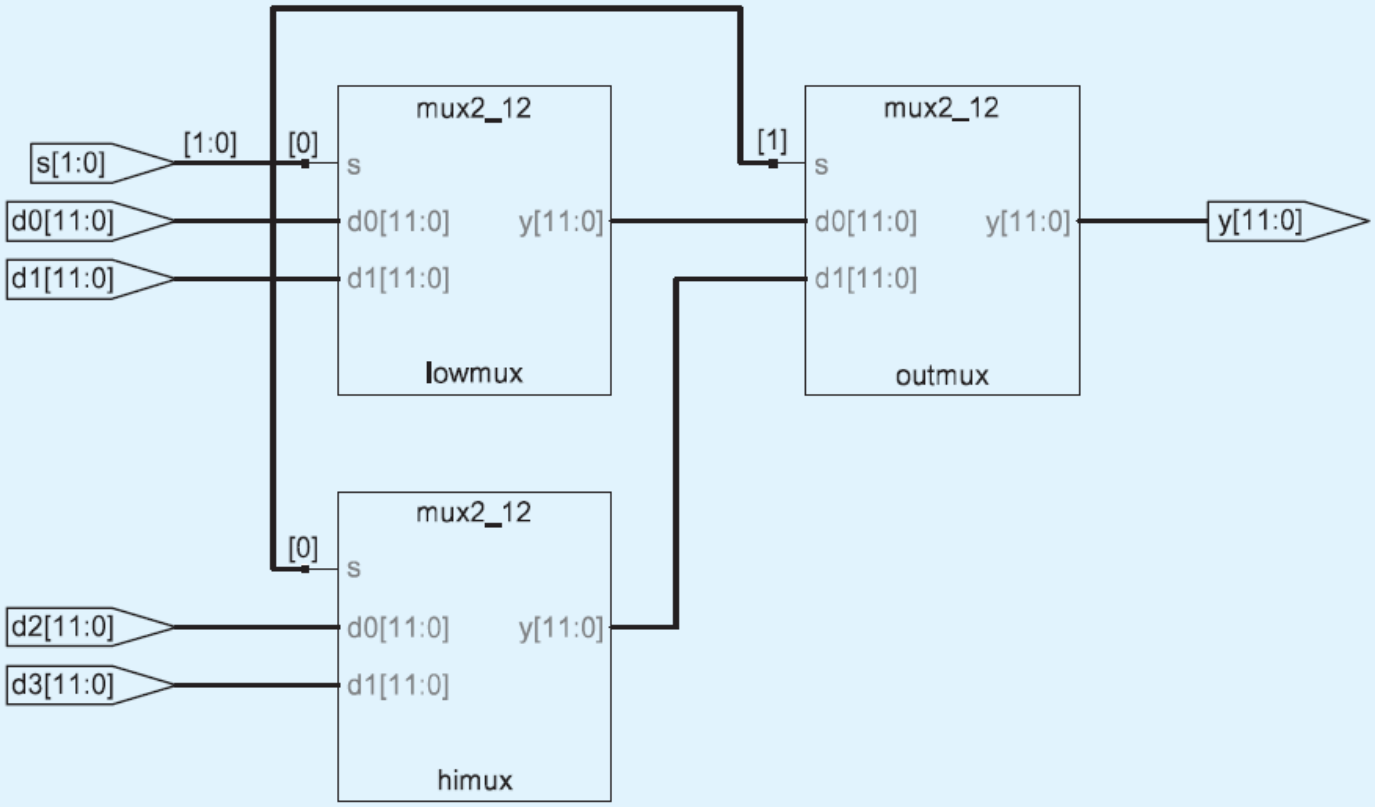


FIGURE A.39 mux4_12

T2: HARDWARE DESCRIPTION LANGUAGES (A8 Parametrized Modules – N-bit Decoder)

Example A.41 Parameterized N:2^N Decoder

SystemVerilog

```
module decoder #(parameter N = 3)
               (input  logic [N-1:0]    a,
                output logic [2**N-1:0] y);

  always_comb
    begin
      y = 0;
      y[a] = 1;
    end
endmodule
```

`2**N` indicates 2^N.

T2: HARDWARE DESCRIPTION LANGUAGES (A8 Parametrized Modules – N-bit AND remember Reductor Operators)

Example A.42 Parameterized *N*-input AND Gate

SystemVerilog

```
module andN
  #(parameter width = 8)
   (input  logic [width-1:0] a,
    output logic             y);

  genvar i;
  logic [width-1:1] x;

  generate
    for (i=1; i<width; i=i+1) begin:forloop
      if (i == 1)
        assign x[1] = a[0] & a[1];
      else
        assign x[i] = a[i] & x[i-1];
    end
  endgenerate
  assign y = x[width-1];
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A8 Parametrized Modules N-bit AND remember Reductor Operators)

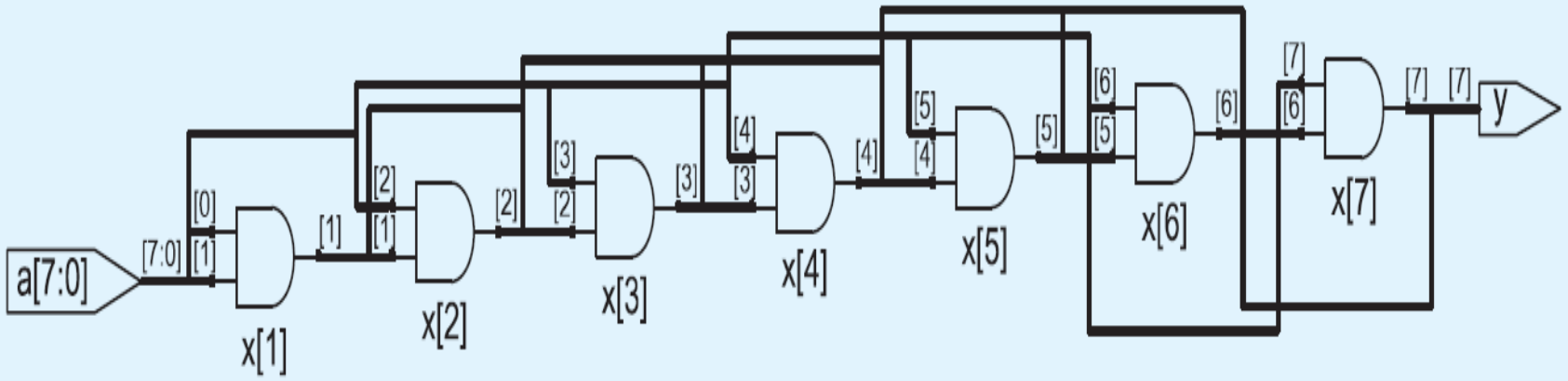


FIGURE A.40 andN

T2: HARDWARE DESCRIPTION LANGUAGES (A9 Memories – RAM With Separate Din and Dout)

Example A.43 RAM

SystemVerilog

```
module ram #(parameter N = 6, M = 32)
            (input  logic          clk,
             input  logic          we,
             input  logic [N-1:0]  adr,
             input  logic [M-1:0]  din,
             output logic [M-1:0]  dout);

  logic [M-1:0] mem[2**N-1:0];

  always @(posedge clk)
    if (we) mem[adr] <= din;

  assign dout = mem[adr];
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A9 Memories – RAM With Separate Din and Dout)

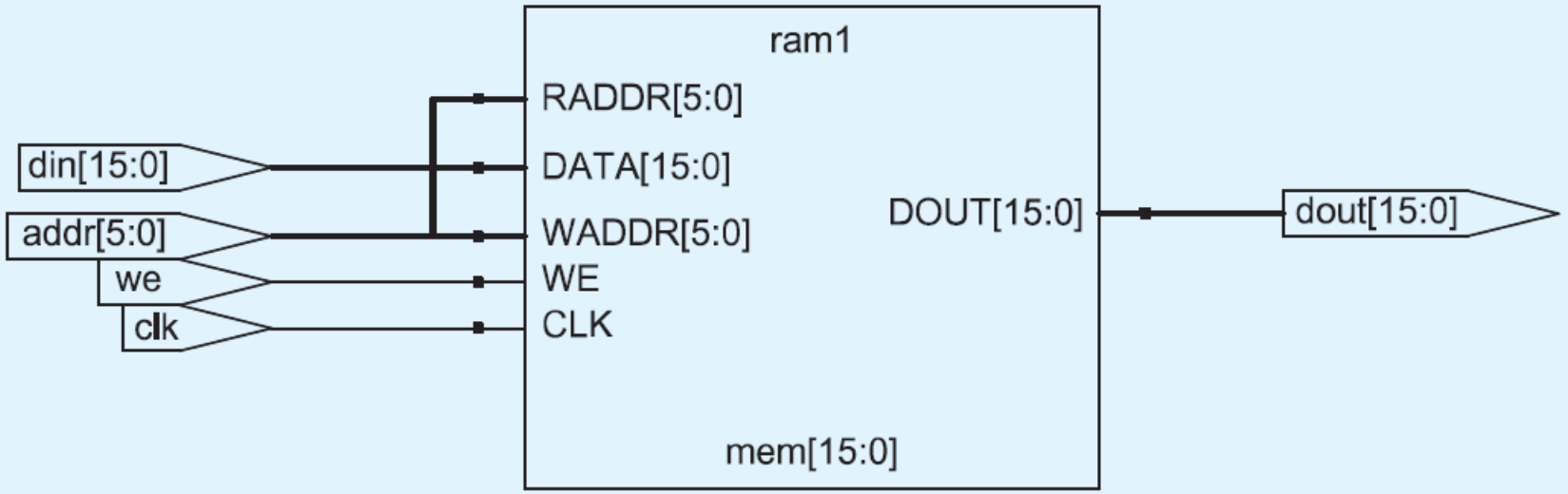


FIGURE A.41 Synthesized ram

T2: HARDWARE DESCRIPTION LANGUAGES (A9 Memories RAM With Bidirectional Data Bus)

Example A.44 RAM with Bidirectional Data Bus

SystemVerilog

```
module ram #(parameter N = 6, M = 32)
            (input  logic          clk,
             input  logic          we,
             input  logic [N-1:0]  adr,
             inout  tri   [M-1:0]  data);

  logic [M-1:0] mem[2**N-1:0];

  always @(posedge clk)
    if (we) mem[adr] <= data;

  assign data = we ? 'z : mem[adr];
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A9 Memories RAM With Bidirectional Data Bus)

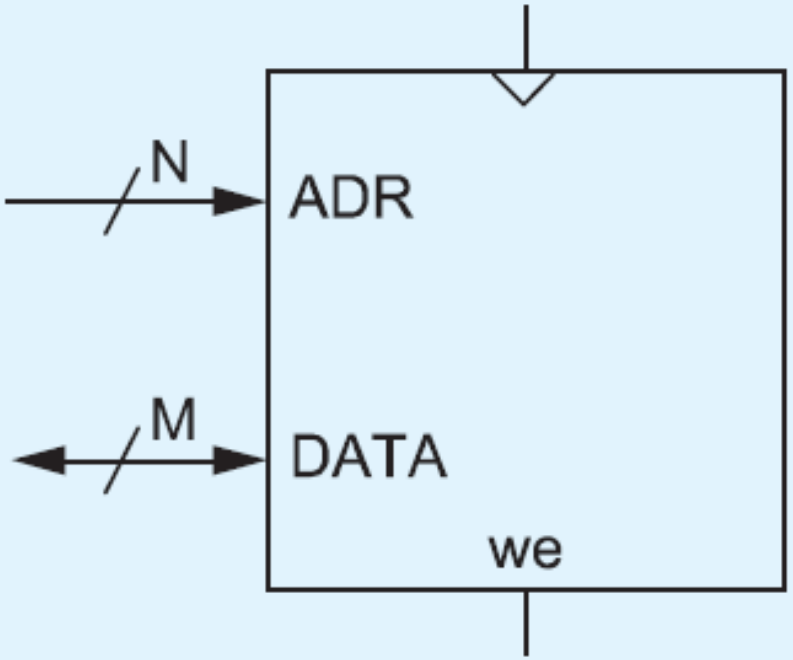


FIGURE A.42 Synthesized ram with bidirectional data bus

T2: HARDWARE DESCRIPTION LANGUAGES (A9 Memories – Three Ported Registers Files)

Example A.45 Three-Ported Register File

SystemVerilog

```
module ram3port #(parameter N = 6, M = 32)
                 (input  logic          clk,
                  input  logic          we3,
                  input  logic [N-1:0] a1, a2, a3,
                  output logic [M-1:0] d1, d2,
                  input  logic [M-1:0] d3);

  logic [M-1:0] mem[2**N-1:0];

  always @(posedge clk)
    if (we3) mem[a3] <= d3;

  assign d1 = mem[a1];
  assign d2 = mem[a2];
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A9 Memories – Three Ported RegistersFiles)

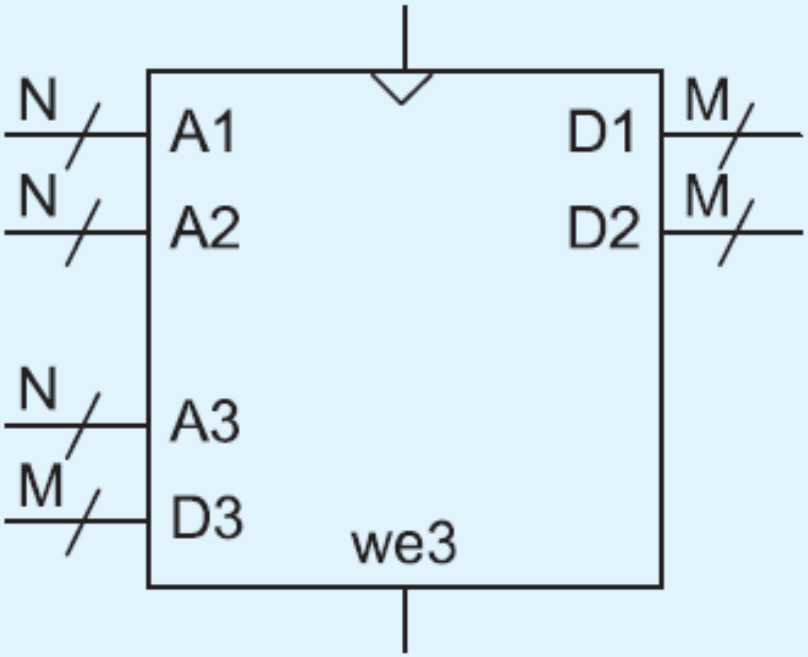


FIGURE A.43
Three-ported register file

T2: HARDWARE DESCRIPTION LANGUAGES (A9 Memories – ROMs)

Example A.46 ROM

SystemVerilog

```
module rom(input  logic [1:0] adr,
           output logic [2:0] dout);

  always_comb
    case(adr)
      2'b00: dout = 3'b011;
      2'b01: dout = 3'b110;
      2'b10: dout = 3'b100;
      2'b11: dout = 3'b010;
    endcase
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A10 Test Benches -Basic Tests)

Example A.47 Testbench

SystemVerilog

```
module testbench1();
  logic a, b, c;
  logic y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // apply inputs one at a time

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

T2: HARDWARE DESCRIPTION LANGUAGES (A10 Test Benches – Self Ch)

Example A.48 Self-Checking Testbench

SystemVerilog

```
module testbench2();
  logic a, b, c;
  logic y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // apply inputs one at a time
  // checking results

  initial begin
    a = 0; b = 0; c = 0; #10;
    assert (y === 1) else $error("000 failed.");
    c = 1;               #10;
    assert (y === 0) else $error("001 failed.");
    b = 1; c = 0;        #10;
    assert (y === 0) else $error("010 failed.");
    c = 1;               #10;
    assert (y === 0) else $error("011 failed.");
    a = 1; b = 0; c = 0; #10;
    assert (y === 1) else $error("100 failed.");
    c = 1;               #10;
    assert (y === 1) else $error("101 failed.");
    b = 1; c = 0;        #10;
    assert (y === 0) else $error("110 failed.");
```

T2: HARDWARE DESCRIPTION LANGUAGES (A10 Test Benches – Self Checking)

```
    assert (y === 0) else $error("010 failed.");
    c = 1;                  #10;
    assert (y === 0) else $error("011 failed.");
    a = 1; b = 0; c = 0; #10;
    assert (y === 1) else $error("100 failed.");
    c = 1;                  #10;
    assert (y === 1) else $error("101 failed.");
    b = 1; c = 0;           #10;
    assert (y === 0) else $error("110 failed.");
    c = 1;                  #10;
    assert (y === 0) else $error("111 failed.");
  end
endmodule
```

T2: HARDWARE DESCRIPTION LANGUAGES (A10 Test Benches)

Example A.49 Testbench with Test Vector File

SystemVerilog

```
module testbench3();
  logic        clk, reset;
  logic        a, b, c, yexpected;
  logic        y;
  logic [31:0] vectornum, errors;
  logic [3:0]  testvectors[10000:0];

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // generate clock
  always
    begin
      clk = 1; #5; clk = 0; #5;
    end

  // at start of test, load vectors
  // and pulse reset
```

T2: HARDWARE DESCRIPTION LANGUAGES (A10 Test Benches – Automatic File Based)

```
// at start of test, load vectors
// and pulse reset
initial
  begin
    $readmemb("example.tv", testvectors);
    vectornum = 0; errors = 0;
    reset = 1; #27; reset = 0;
  end

// apply test vectors on rising edge of clk
always @(posedge clk)
  begin
    #1; {a, b, c, yexpected} =
          testvectors[vectornum];
  end
```

T2: HARDWARE DESCRIPTION LANGUAGES (A10 Test Benches – Automatic File Based)

SystemVerilog (continued)

```
// check results on falling edge of clk
   always @(negedge clk)
    if (~reset) begin // skip during reset
      if (y !== yexpected) begin
        $display("Error: inputs = %b", {a, b, c});
        $display("  outputs = %b (%b expected)",
                 y, yexpected);
        errors = errors + 1;
      end
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 'bx) begin
        $display("%d tests completed with %d
                  errors", vectornum, errors);
        $finish;
      end
    end
endmodule
```