
Lecture 6: Synthesis - Matching Verilog HDL with Basic Combinational and Sequential Circuit

- 6.1 Introduction to Synthesis
- 6.2 Synthesis of Combinational Logic
- 6.3 Synthesis of Sequential Latches
- 6.4 Synthesis of Sequential Registers
- 6.5 Synthesis of Counter and Timer

6.1 Introduction to Synthesis

6.2 Synthesis of Combinational Logic

6.3 Synthesis of Sequential Latches

6.4 Synthesis of Sequential Registers

6.5 Synthesis of Counter and Timer

6.1.1 What is Synthesis?

- What is synthesis?
 - Conversion
 - Convert register-transfer level (RTL) descriptions into gate-level representations suitable for implementation on ASICs or FPGAs.
 - The goal is to maintain equivalence between the RTL design and the synthesized netlist.
 - Optimization
 - Improve various design aspects, including area reduction, power consumption minimization, and timing constraint optimization.

```
1  assign d = sel ? (a+b) : (a+c);
```

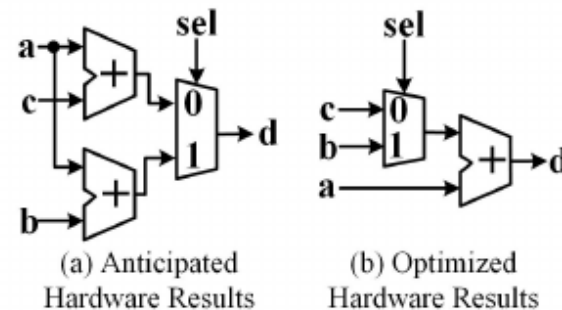


FIGURE 6.1

Synthesis Results of `assign d = sel ? (a+b) : (a+c);`

6.1.2 Mismatches Between Simulation and Synthesis

- Mismatches Between Simulation and Synthesis
 - Incomplete statements
 - Simulation: may cause simulation failures since the changes of missing signals cannot trigger the associated statements.
 - Synthesis: synthesis tools can automatically analyze the design code and complete the sensitivity list.

```
1  always @(sel) begin
2      if (sel & en) begin
3          a <= #1 1'b1;
4      end else if (sel==1'b1) begin
5          a <= #1 1'b1;
6      end else begin
7          a <= #1 1'b0;
8      end
9  end
```

- Delay statements
- Logic comparison
 - Hardware can only distinguish logic levels of zeros and ones, rendering comparisons to unknown values and high impedance states in simulation irrelevant for synthesis.

6.1.3 Synthesizable Verilog HDL

- Synthesizable Verilog HDL
 - Verilog HDL is a large and comprehensive IEEE standard however, most of them are unsynthesizable, meaning they cannot be translated into hardware by synthesis tools.
 - For instance, delays, system tasks, display statements.
 - RTL description
 - The subset of Verilog that is considered synthesizable is commonly referred to as RTL description.
 - RTL code describes the flow of data between registers and is more hardware-oriented, making it suitable for synthesis.
 - RTL designers need to adhere to the subset of constructs and coding styles supported by synthesis tools.
 - Summary
 - Synthesizable Verilog descriptions encompass a small subset constructs that are suitable for hardware synthesis, including
 - concurrent *assign*, conditional *assign*, level-triggered *always* blocks, and edge-triggered *always* blocks.

6.1.3 Synthesizable Verilog HDL

- Concurrent assignment
 - Describe combinational circuits.
- Conditional assignment
 - Describe various combinational circuits.

```
1 // Concurrent assignment
2 assign d = a & b | c; // combinational logic
3
4 // Conditional assignment
5 assign b = enable ? a : 1'bz; // tri-state buffer
6 assign c = enable ? a : b;    // multiplexer
```

- Level-triggered always blocks
 - Describe both combinational and sequential circuits.
- Edge-triggered always blocks
 - Describe registers
 - Should typically include only ``posedge clock" and ``posedge/negedge reset" in the sensitivity list.

6.1.3 Synthesizable Verilog HDL

- Concurrent assignment
 - Describe combinational circuits.
- Conditional assignment
 - Describe various combinational circuits.
- Level-triggered always blocks
 - Describe both combinational and sequential circuits.
- Edge-triggered always blocks
 - Describe registers
 - Should typically include only ``posedge clock" and ``posedge/negedge reset" in the sensitivity list.

```
8 // Level-trigger always (if-else, case, loop):
9 always @(a, b, en) if (en) c <= a & b; //latch
10
11 // Edge-trigger always (if-else, case, loop):
12 always @(posedge clk) b <= nxt_b; //register
```


6.1.3 Synthesizable Verilog HDL

- Design Guidelines for Synthesizable Verilog Descriptions

Design Guidelines for Synthesizable Verilog Descriptions

- For concurrent/conditional *assign* blocks, declare the data type of the LHS signals as *wire*, and utilize blocking assignment (=) for the design statements.
- For level/edge-triggered *always* blocks, declare the data type of the LHS signals as *reg*, and use non-blocking assignment (<=) for the design statements.
- Ensure that the trigger list in a level-triggered *always* block is complete, and provide complete *case* items if applicable. In *if – else* statements, it is permissible to omit the final *else* condition when describing latches.
- Ensure that the trigger list in an edge-triggered *always* block only consists of active clock and reset edges. In *if – else* statements, it is permissible to omit the final *else* condition when describing registers or registers
- To prevent multiple drivers from affecting the same LHS signals in RTL designs, make sure not to assign values to the same signal across different blocks. Nevertheless, it's worth noting that within the simulation testbench, it is considered acceptable to have multiple drivers for signals like the clock initialization.

6.1 Introduction to Synthesis

6.2 Synthesis of Combinational Logic

6.3 Synthesis of Sequential Latches

6.4 Synthesis of Sequential Registers

6.5 Synthesis of Counter and Timer

6.2.1. Fundamental Combinational Logic

- Fundamental Combinational Logic
 - Fundamental logic descriptions using concurrent *assign* and conditional *assign* blocks

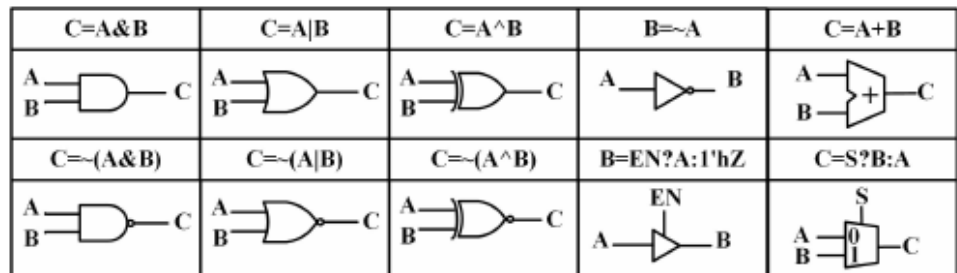


FIGURE 6.2

Basic Logic Descriptions Using Concurrent/Conditional Assignments

```

1  // Example #1: Design AND-OR Using Concurrent Assign
2  module example1_and_or (output e
3                          input  a, b, c, d );
4  assign e = (a & b) | (c & d);
5  endmodule
6
7  // Example #2: Design AND-OR Using Level-Trigger Always
8  module example2_and_or (output reg e
9                          input  a, b, c, d );
10 always @ (a, b, c, d) begin
11     e = (a & b) | (c & d);
12 end
13 endmodule
14
15 // Example #3: Design AND-OR with Tri-State Buffer
16 // Using Conditional Assign
17 module example3_and_or_tri (output e
18                             input  a, b, c, d, en );
19 assign e = en ? (a & b) | (c & d) : 1'bz;
20 endmodule
    
```

6.2.1. Fundamental Combinational Logic

```

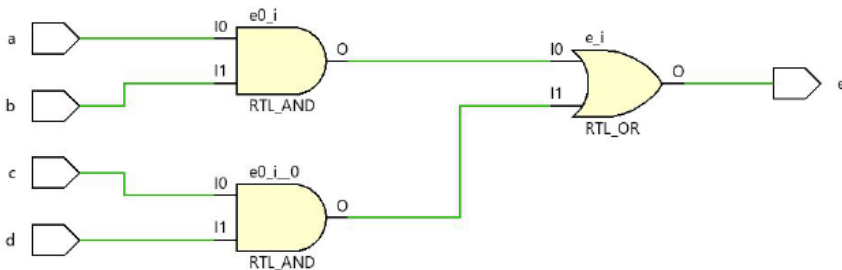
1 // Example #1: Design AND-OR Using Concurrent Assign
2 module example1_and_or (output e
3                          input a, b, c, d );
4 assign e = (a & b) | (c & d);
5 endmodule

```

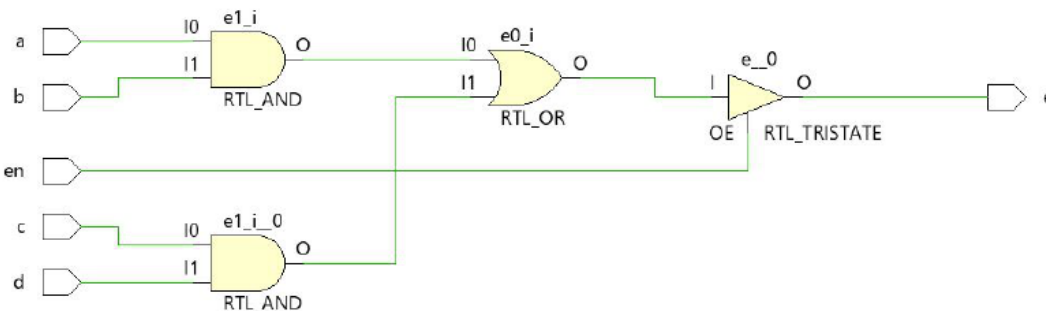
```

6
7 // Example #2: Design AND-OR Using Level-Trigger Always
8 module example2_and_or (output reg e
9                          input a, b, c, d );
10 always @ (a, b, c, d) begin
11     e = (a & b) | (c & d);
12 end
13 endmodule

```



(a) Example #1 and #2: Design AND-OR Gates Using Continuous Assign and Level-Trigger Always



(b) Example #3: Design AND-OR Gates with Tri-State Buffer Using Conditional Assign

FIGURE 6.3
Synthesis Results of AND-OR-Tri Description.

```

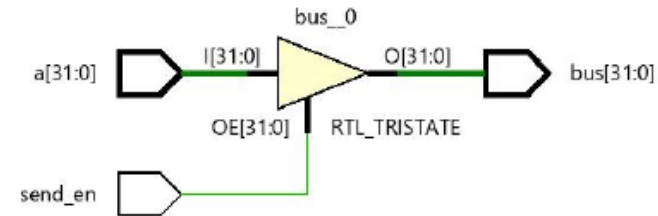
15 // Example #3: Design AND-OR with Tri-State Buffer
16 // Using Conditional Assign
17 module example3_and_or_tri (output e
18                             input a, b, c, d, en );
19 assign e = en ? (a & b) | (c & d) : 1'bz;
20 endmodule

```

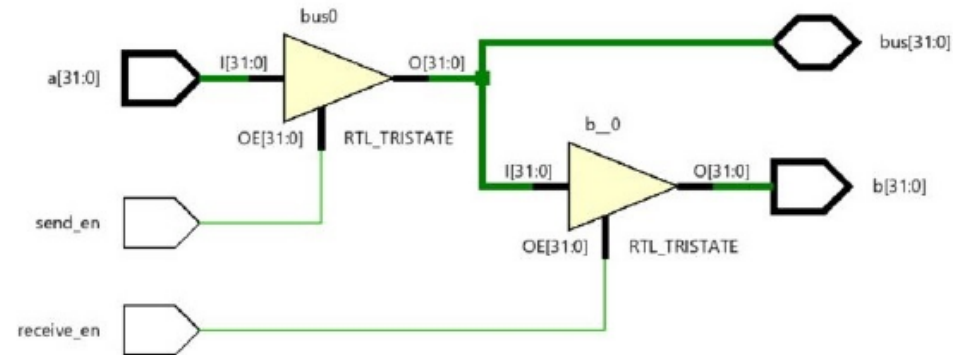
6.2.2 Uni-Directional and Bi-Directional Buses

- Fundamental Combinational Logic
 - Uni-directional bus (lines 1-6),
 - Bi-directional bus (lines 8-16).

```
1 // Example #1: Design Uni-Direction Bus
2 module uni_dir_bus (output [31:0] bus
3                     input  [31:0] a
4                     input          send_en
5 );
6   assign bus = send_en ? a : 32'bz;
7   endmodule
8
9 // Example #2: Design Bi-Direction Bus
10 module bi_dir_bus (inout  [31:0] bus
11                   input  [31:0] a
12                   output [31:0] b
13                   input          send_en
14                   input          receive_en
15 );
16   assign b = receive_en ? bus : 32'bz;
17   assign bus = send_en ? a : 32'bz;
18   endmodule
```



(a) Example #1 : Design Uni-Directional Bus



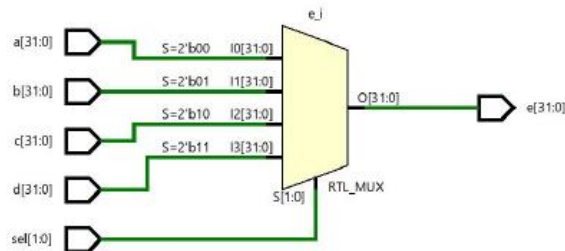
(b) Example #2 : Design Bi-Directional Bus

FIGURE 6.4

Synthesis Results of Uni- and Bi-Directional Buses

6.2.3 Multiplexer

• Multiplexer



(a) Example #1: Design a Multiplexer Using Always case

```

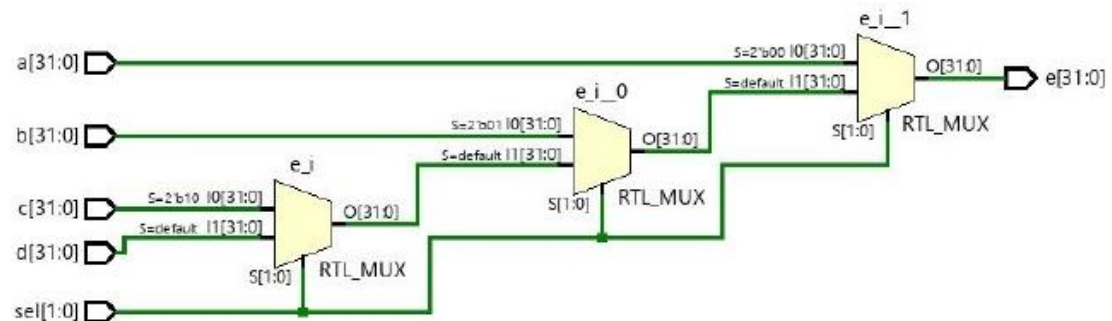
1 // Example #1: Multiplexer Design Using case-endcase
2 module example1_mux (output reg [31: 0] e
3                       input          [31: 0] a, b, c, d,
4                       input          [1 : 0] sel
5 );
6 always @ (a, b, c, d, sel) begin
7     case (sel)
8         2'h0   : e <= a      ;
9         2'h1   : e <= b      ;
10        2'h2   : e <= c      ;
11        2'h3   : e <= d      ;
12        default: e >= a      ;
13    endcase
14 end
15 endmodule

```

```

16 // Example #2: Multiplexer Design Using if-else
17 module example2_mux (output reg [31: 0] e
18                       input          [31: 0] a, b, c, d,
19                       input          [1 : 0] sel
20 );
21 always @ (a, b, c, d, sel) begin
22     if (sel == 2'h0) begin
23         e <= a;
24     end else if (sel == 2'h1) begin
25         e <= b;
26     end else if (sel == 2'h2) begin
27         e <= c;
28     end else begin
29         e <= d;
30     end
31 end
32 endmodule

```



(b) Example #2: Design a Multiplexer with Different Priority Using if-else in always or conditional assign

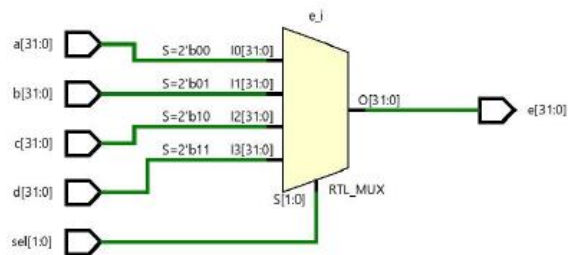
FIGURE 6.5

Synthesis Results of Different Designs with Multiplexer

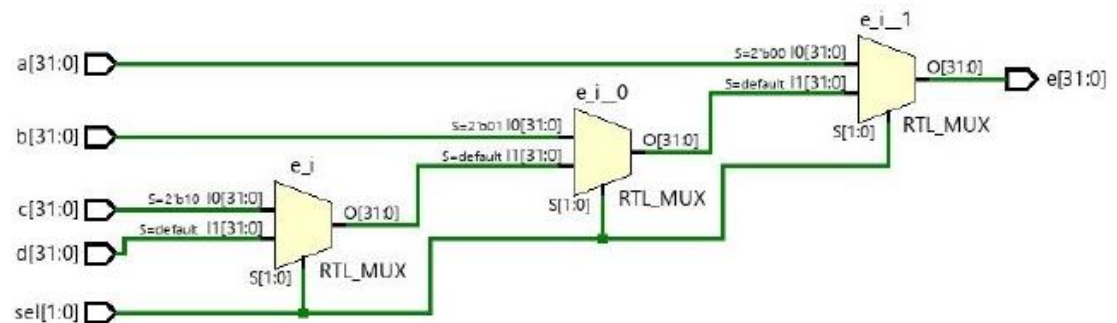
6.2.3 Multiplexer

- Multiplexer

```
33 // Example #3: Multiplexer Design Using Conditional Assign
34 module example3_mux (output [31: 0] e
35                       input  [31: 0] a, b, c, d,
36                       input  [1 : 0] sel
37 );
38   assign e = (sel == 2'h0) ? a :
39             (sel == 2'h1) ? b :
40             (sel == 2'h2) ? c : d;
41 endmodule
```



(a) Example #1: Design a Multiplexer Using Always case



(b) Example #2: Design a Multiplexer with Different Priority Using if-else in always or conditional assign

FIGURE 6.5

Synthesis Results of Different Designs with Multiplexer

6.1 Introduction to Synthesis

6.2 Synthesis of Combinational Logic

6.3 Synthesis of Sequential Latches

6.4 Synthesis of Sequential Registers

6.5 Synthesis of Counter and Timer

6.3.1 Intentional Latches Design with Verilog

- Intentional Latch Design with Verilog

```
1 // Example: Design Latches Using Omitted else
2 module example2_latch_always (output reg [3:0] q ,
3                               input  [3:0] d ,
4                               input      en );
5 always @(en, d) begin
6     if (en) q <= d;
7 end
8 endmodule
```

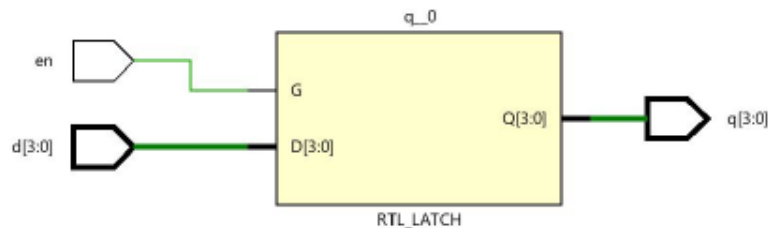
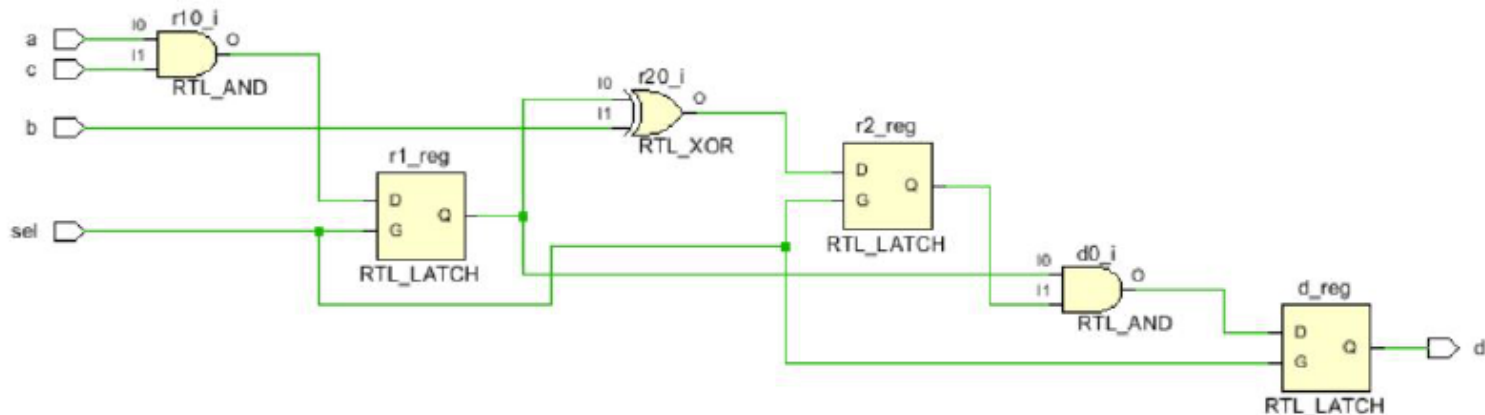


FIGURE 6.6
Synthesis Results of Latch

6.3.1 Intentional Latches Design with Verilog

- Intentional Latches Design with Verilog
 - Combination of the combinational logic and sequential latches includes the incorporation of an omitted else statement

```
1 // Example #1: Design Combinational Logic and Latches
2 module example1_comb_latch (input      a, b, c, sel,
3                               output reg d
4                               );
5
6 always @ (a, b, c, sel) begin
7     if (sel) begin
8         r1 <= a & c;
9         r2 <= r1 ^ b;
10        d  <= r1 & r2;
11    end
12 end
13 endmodule
```

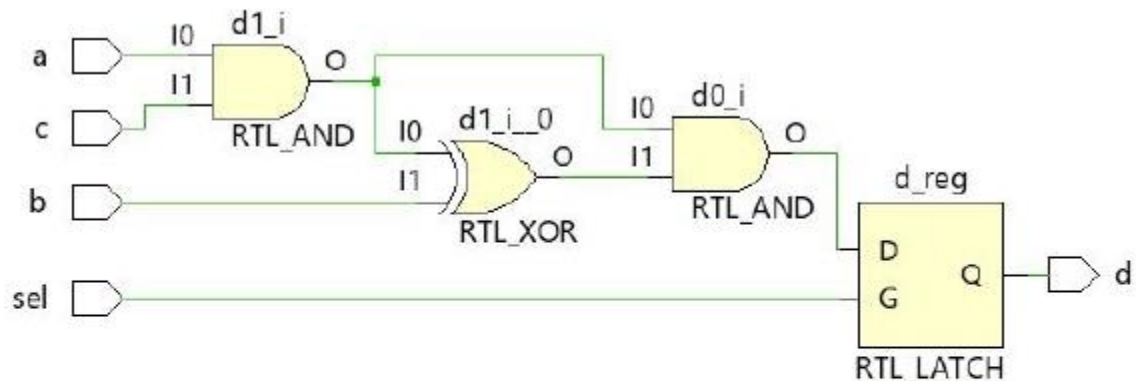


(a) Example #1: Intentional Design of Latches

6.3.1 Intentional Latches Design with Verilog

- Intentional Latches Design with Verilog
 - Divides the design of combinational logic from the sequential latch

```
15 // Example #2: An Alternative Way to the Design of
16 // Combinational Logic and Latches
17 module example2_comb_latch (input      a, b, c, sel,
18                             output reg d
19                             );
19 wire w1, w2, w3;
20 assign w1 = a & c;
21 assign w2 = w1 ^ b;
22 assign w3 = w1 & w2;
23
24 always @(w3, sel) begin
25     if (sel) begin
26         d <= w3;
27     end
28 end
29 endmodule
```



(b) Example #2: An Alternative Way to the Design of Combinational Logic and Latches

FIGURE 6.7

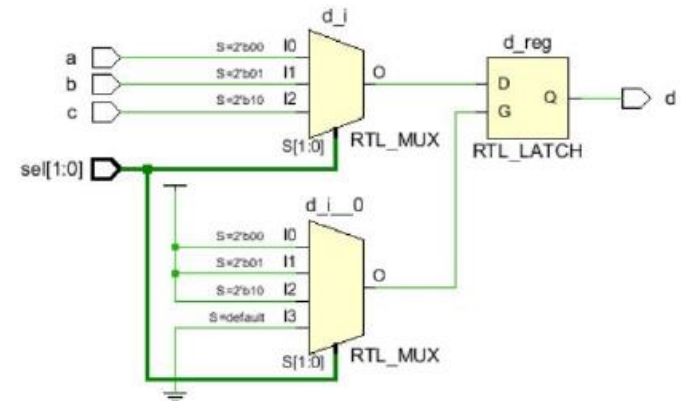
Synthesis Results of Intentional Latches Design

6.3.2 Accidental Design Latches with Verilog

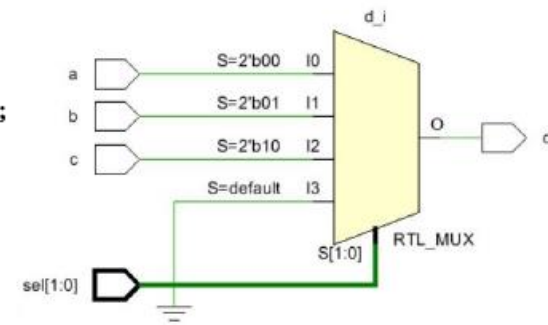
- Accidental Design Latches with Verilog
 - Incomplete case-endcase

```
1 // Example #1: Accidental Latches with Incomplete Case
2 module example1_acc_latch_co_case (input      a, b, c,
3                                     input [1:0] sel,
4                                     output reg  d
5                                     );
6     always @ (a, b, c, sel) begin
7         case(sel)
8             2'b00 : d <= a    ;
9             2'b01 : d <= b    ;
10            2'b10 : d <= c    ;
11        endcase
12    end
13 endmodule
```

```
14 // Example #2: Design with Complete Case
15 module example2_acc_latch_co_case (input      a, b, c,
16                                     input [1:0] sel,
17                                     output reg  d
18                                     );
19     always @ (a, b, c, sel) begin
20         case(sel)
21             2'b00 : d <= a    ;
22             2'b01 : d <= b    ;
23             2'b10 : d <= c    ;
24             default: d <= 1'b0;
25         endcase
26     end
27 endmodule
```



(a) Example #1: Accidental Design of Latches with Incomplete case-endcase



(b) Example #2: Design Results with Complete case-endcase

FIGURE 6.8

Synthesis Results of Incomplete and Complete case-endcase

6.3.2 Accidental Design Latches with Verilog

- Accidental Design Latches with Verilog
 - B. Incomplete Sensitivity List
 - Simulation: any modifications to the signal ``c" will not activate the *a/ways* block during simulation.
 - Synthesis: the synthesis tool may automatically analyze the design code and complete the sensitivity list to generate the desired hardware.

```
1 // Example #1: Accidental Latches with Incomplete List
2 module example1_acc_latch_inc_list (input      a, b, c,
3                                     input [1:0] sel,
4                                     output reg  d );
5     always @(a, b, sel) begin
6         case(sel)
7             2'b00 : d <= a;
8             2'b01 : d <= b;
9             2'b10 : d <= c;
10            default: d <= 1'b0;
11        endcase
12    end
13 endmodule
```

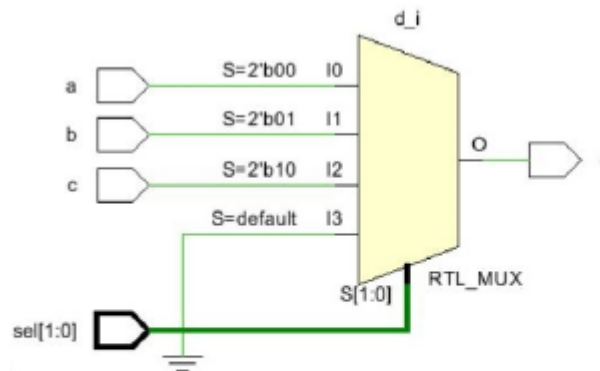


FIGURE 6.9

Synthesis Result of Incomplete Sensitivity List CENG5534

6.1 Introduction to Synthesis

6.2 Synthesis of Combinational Logic

6.3 Synthesis of Sequential Latches

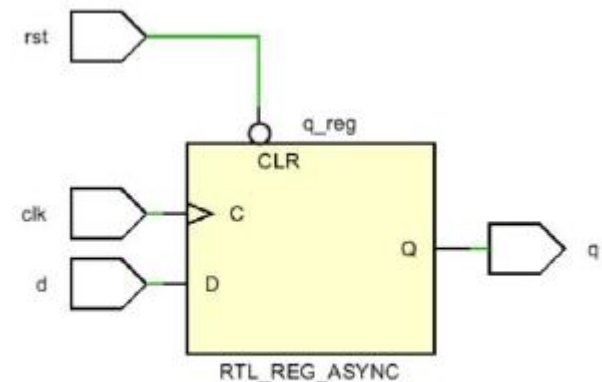
6.4 Synthesis of Sequential Registers

6.5 Synthesis of Counter and Timer

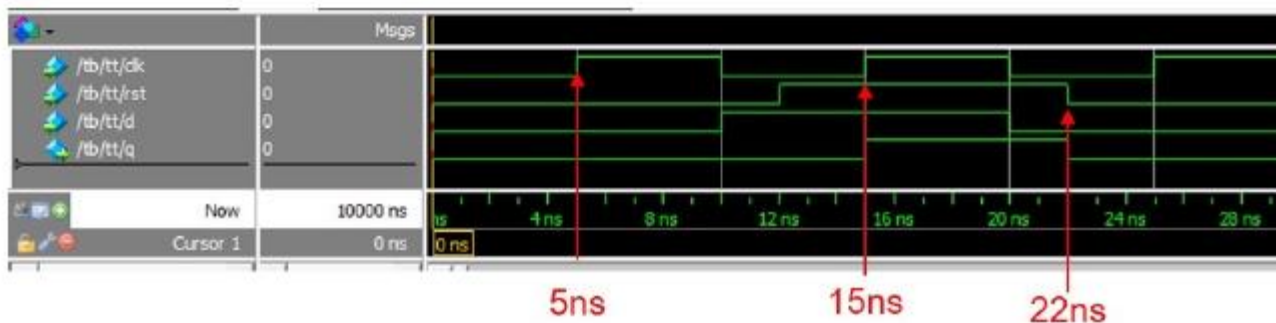
6.4.1 Single-Bit Register with Asynchronous and Synchronous Reset

- Single-Bit Register with Asynchronous Reset

```
1 // A Design Example of Asynchronous Reset
2 module example_async_reset (input d, clk, rst,
3                             output reg q);
4 always @ (posedge clk, negedge rst) begin
5     if (~rst) begin
6         q <= 1'b0;
7     end else begin
8         q <= d;
9     end
10 end
11 endmodule
```



(a) Example #1: Single-Bit Register Design with Asynchronous Reset

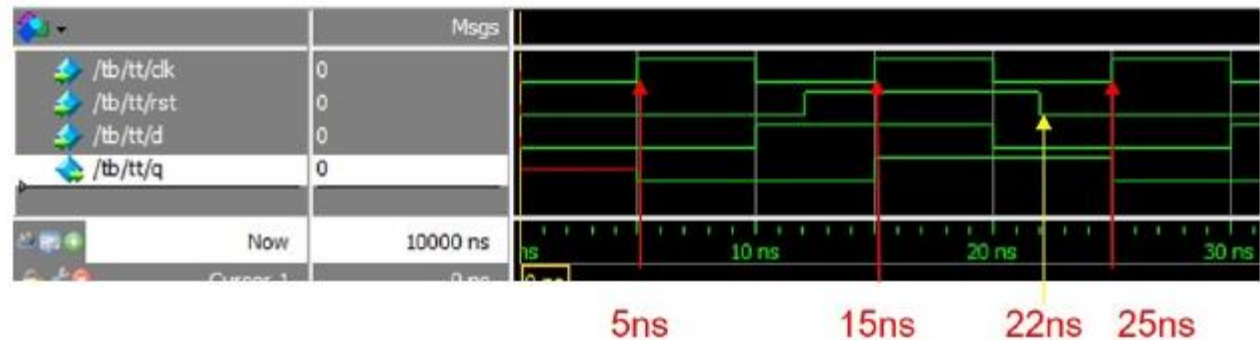


(a) Simulation Result of Asynchronous Reset

6.4.1 Single-Bit Register with Asynchronous and Synchronous Reset

- Single-Bit Register with Synchronous Reset

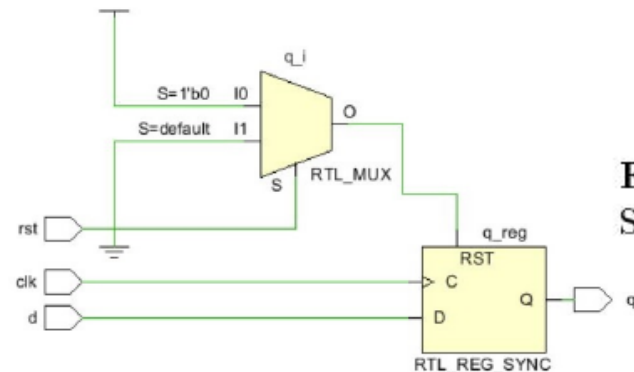
```
13 // A Design Example of Synchronous Reset
14 module example_sync_reset (input d, clk, rst,
15                             output reg q      );
16 always @ (posedge clk) begin
17     if (~rst) begin
18         q <= 1'b0;
19     end else begin
20         q <= d;
21     end
22 end
23 endmodule
```



(b) Simulation Result of Synchronous Reset

FIGURE 4.4

Simulation Results of Asynchronous and Synchronous Reset



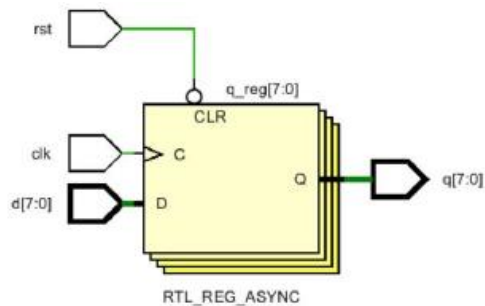
(b) Example #2: Single-Bit Register Design with Synchronous Reset

FIGURE 6.10

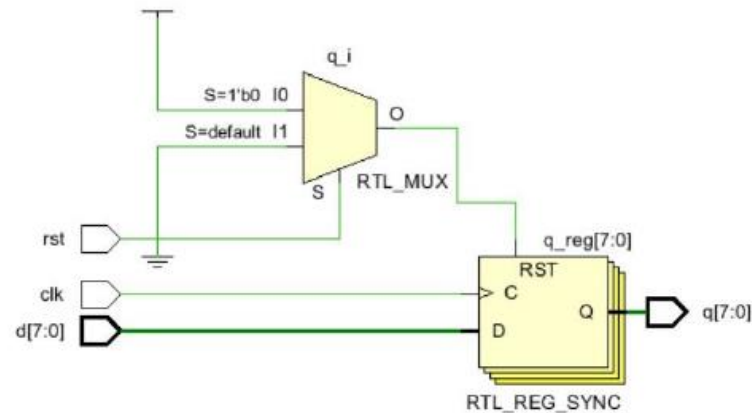
Synthesis Results of Single-Bit Register Design with Asynchronous and Synchronous Reset

6.4.2 Multi-Bit Register with Asynchronous and Synchronous Reset

- Multi-Bit Register with Asynchronous and Synchronous Reset



(a) Example #1: 8-bit Register Design with Asynchronous Reset



(b) Example #2: 8-bit Register Design with Synchronous Reset

```

1  // Example #1: 8-bit Register with Asynchronous Reset
2  module reg_async_rst (input          rst, clk,
3                        input          [7:0] d          ,
4                        output reg [7:0] q              );
5  always @ (negedge rst, posedge clk) begin
6      if (~rst) begin
7          q<=8'h0;
8      end else begin
9          q<=d      ;
10     end
11 end
12 endmodule
13
14 // Example #2: 8-bit Register with Synchronous Reset
15 module reg_sync_rst (input          rst, clk,
16                     input          [7:0] d          ,
17                     output reg [7:0] q              );
18 always @ (posedge clk) begin
19     if (~rst) begin
20         q<=8'h0;
21     end else begin
22         q<=d      ;
23     end
24 end
25 endmodule

```

FIGURE 6.11

Synthesis Results of 8-bit Register Design with Asynchronous and Synchronous Reset

6.4.3 Shift Register

- Shift Register

- Applications:

- For instance, series data input and collection, waiting for other data input, etc.

- A. Design Specification

- A 4-bit shift register, or 4-bit shifter, consists of four single-bit registers connected in series.
 - Allows the data input to be shifted out by four clock cycles, with each cycle moving the data to the next register in the series.
 - The 4-bit signal ``data_reg" represents the entire 4-bit output of the four registers, with the Most Significant Bit (MSB) on the left and the Least Significant Bit (LSB) on the right.

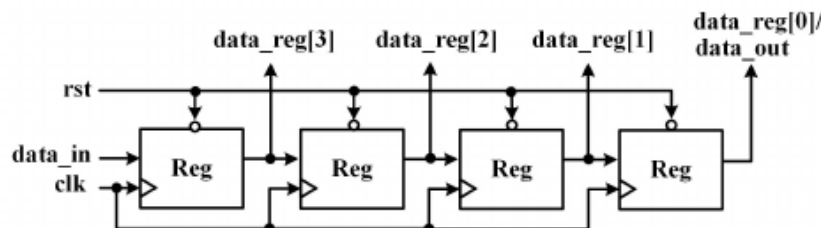


FIGURE 6.12
Block Diagram of Shift Register

6.4.3 Shift Register

- Shift Register
 - B. Verilog Design and Synthesis

```
1 // Shift Register Design
2 module shift_reg (input          rst      ,
3                   input          clk      ,
4                   input          data_in  ,
5                   output         data_out);
6 reg [3:0] data_reg;
7
8 assign data_out = data_reg[0];
9 always @ (negedge rst, posedge clk) begin
10     if (~rst) begin
11         data_reg <= 4'h0      ;
12     end else begin
13         data_reg <= {data_in, data_reg[3:1]};
14     end
15 end
16 endmodule
```

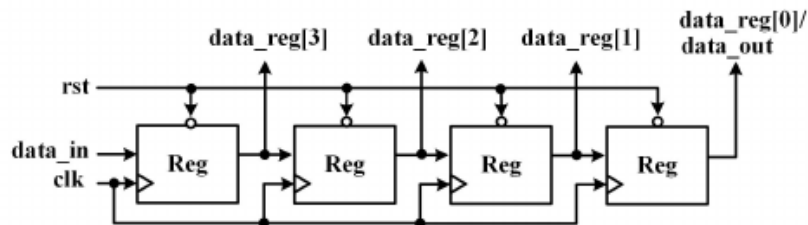


FIGURE 6.12
Block Diagram of Shift Register

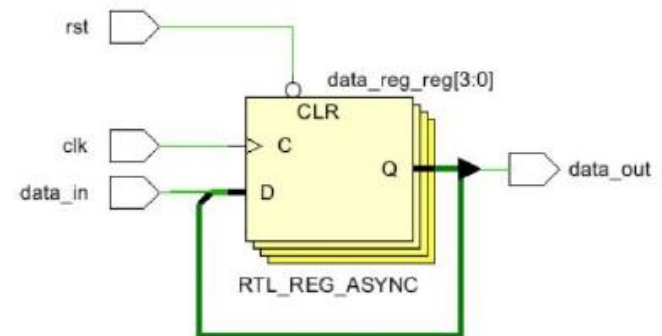


FIGURE 6.13
Synthesis Result of Shift Register

6.1 Introduction to Synthesis

6.2 Synthesis of Combinational Logic

6.3 Synthesis of Sequential Latches

6.4 Synthesis of Sequential Registers

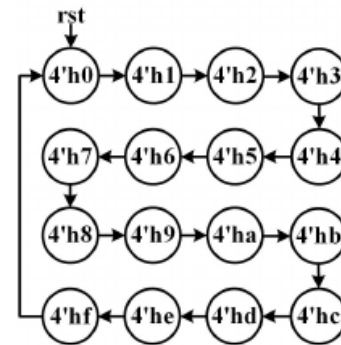
6.5 Synthesis of Counter and Timer

6.5.1 Counter 0-f

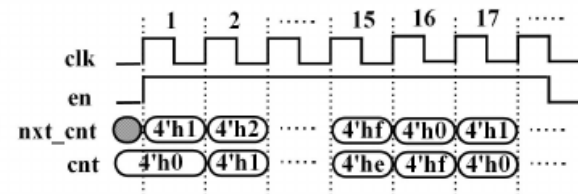
- Counter 0-f
 - Application:
 - Counters play a critical role in controlling timing and sequencing across multiple clock cycles.
 - For instance: serial buses like I2C, SPI, SDIO, GPIO, UART, etc., as well as numerical designs involving long datapaths such as vector-vector multiplications.
 - A. Design Specification
 - A counting loop from hexadecimal 4'h0 to 4'hf

TABLE 6.1
Counter IOs Description

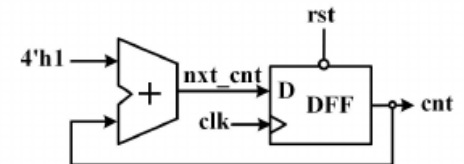
Name	Direction	Bit Width	Description
clk	Input	1	Clock
rst	Input	1	Asynchronous reset, 0 valid
en	Input	1	Enable signal, 1 valid
cnt	Output	4	Counter output



(a) Design Specification of Counter for 16 cycles.



(b) Timing Diagram of Counter for 16 cycles.



(c) Block Diagram of Counter for 16 cycles.

FIGURE 6.14

Design Specification of Counter for 16 Cycles

6.5.1 Counter 0-f

- Counter 0-f
 - B. Verilog Design and Synthesis

```
1 // 0-f Counter Design
2 module cnt_0_f (input          rst, clk, en,
3                 output reg [3:0] cnt
4                 );
5 wire [3:0] nxt_cnt = en ? cnt+4'h1 : cnt;
6
7 always @ (negedge rst, posedge clk) begin
8     if (~rst) begin
9         cnt <= 4'h0 ;
10    end else begin
11        cnt <= nxt_cnt ;
12    end
13 end
14 endmodule
```

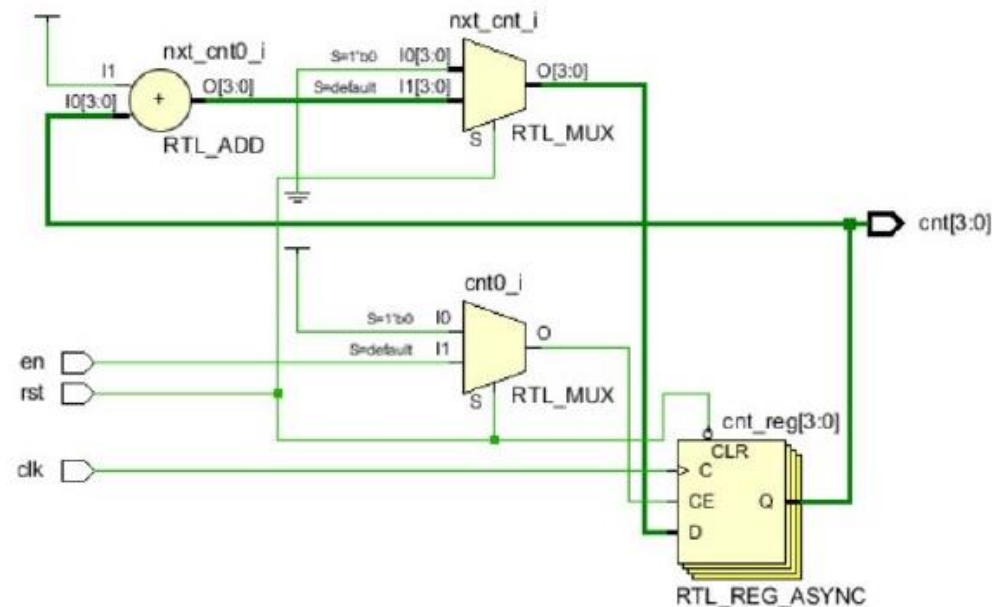


FIGURE 6.15

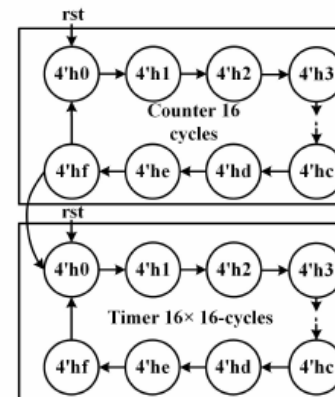
Synthesis Result of Counter from 0 to f

6.5.2 Timer 0-f

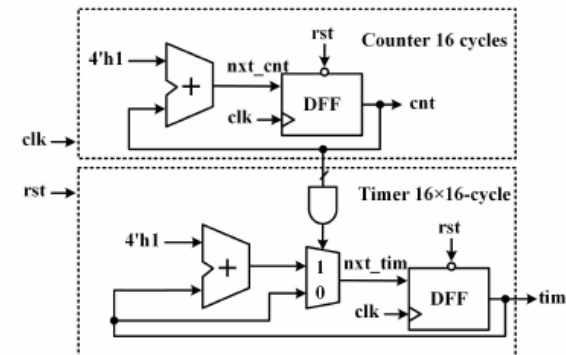
- Timer 0-f
 - A. Design Specification
 - The first level consists of a counter that counts 16 clock cycles, from hexadecimal 4'h0 to 4'hf.
 - In the second level, the timer counts the number of units, with each unit comprising 16 clock cycles. Every 16 cycles, the timer increments by one, starting from hexadecimal 4'h0 and progressing up to the maximum value of hexadecimal 4'hf.

TABLE 6.2
Timer IOs Description

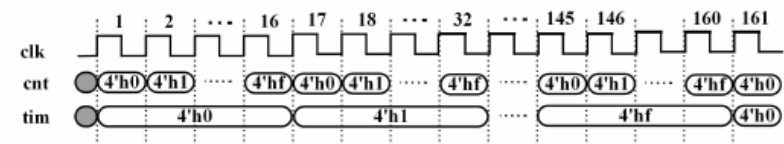
Name	Direction	Bit Width	Description
clk	Input	1	Clock
rst	Input	1	Asynchronous reset, 0 valid
en	Input	1	Enable signal, 1 valid
tim	Output	4	Timer output



(a) Design Specification of Timer for 16x16-cycles.



(c) Block Diagram of Timer for 16x16-cycles.



(b) Timing Diagram of Timer for 16x16-cycles.

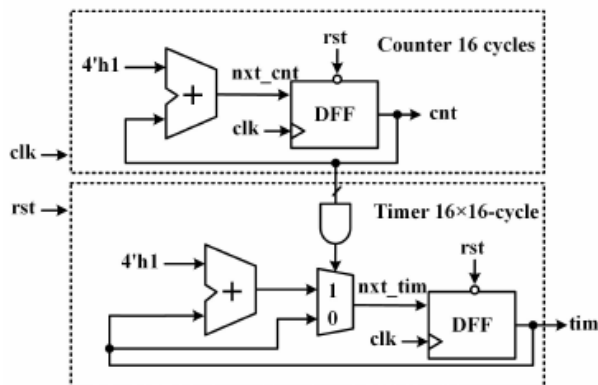
FIGURE 6.16

Design Specification of Timer for 16x16-cycles

6.5.2 Timer 0-f

- Timer 0-f
 - B. Verilog Design and Synthesis

```
1 // Design on Timer 16x16-cycle
2 module timer_0_f_16_cycles (input    rst, clk, en,
3                             output reg [3:0] tim);
4 // The first level counter
5 reg [3:0] cnt;
6 wire [3:0] nxt_cnt = en ? cnt+4'h1 : cnt;
7 always @(posedge clk, negedge rst) begin
8     if(~rst) begin
9         cnt<=4'h0 ;
10    end else begin
11        cnt<=nxt_cnt;
12    end
13 end
```



```
15 // The second level timer
16 wire [3:0] nxt_tim = (en & cnt) ? tim+4'h1 : tim;
17 always @(posedge clk, negedge rst) begin
18     if(~rst) begin
19         tim<=4'h0 ;
20     end else begin
21         tim<=nxt_tim;
22     end
23 end
24 endmodule
```

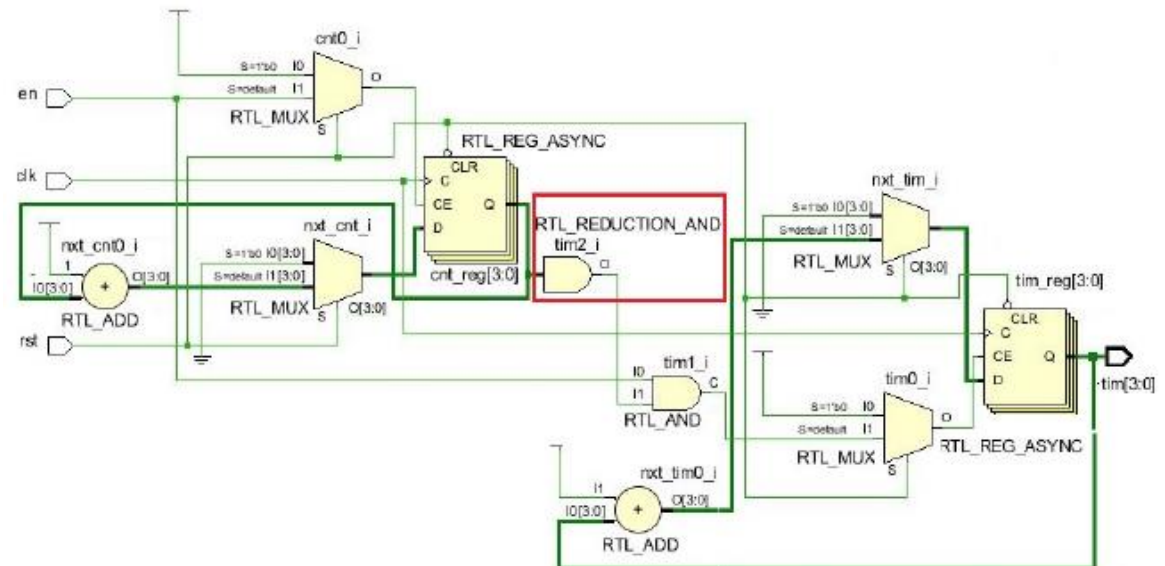
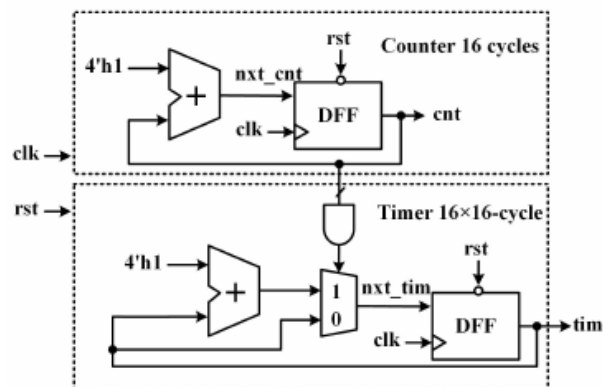

6.5.2 Timer 0-f

- Timer 0-f
 - B. Verilog Design and Synthesis

```

1 // Design on Timer 16x16-cycle
2 module timer_0_f_16_cycles (input    rst, clk, en
3                             output reg [3:0] tim)
4 // The first level counter
5 reg [3:0] cnt;
6 wire [3:0] nxt_cnt = en ? cnt+4'h1 : cnt;
7 always @(posedge clk, negedge rst) begin
8     if(~rst) begin
9         cnt<=4'h0 ;
10    end else begin
11        cnt<=nxt_cnt;
12    end
13 end
14
15 // The second level timer
16 wire [3:0] nxt_tim = (en & &cnt) ? tim+4'h1 : tim;
17 always @(posedge clk, negedge rst) begin
18     if(~rst) begin
19         tim<=4'h0 ;
20     end else begin
21         tim<=nxt_tim;
22     end
23 end
24 endmodule

```



des.

(c) Block Diagram of Timer for 16×16-cycles.

FIGURE 6.17
Synthesis Result of Timer for 16×16-cycle