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

Outline

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

Outline

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.

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

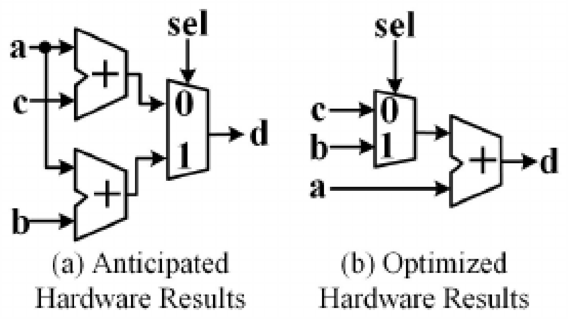

sel
a
c
b
+
0
1
d
(a) Anticipated Hardware Results
sel
c
b
a
0
1
+
d
(b) Optimized Hardware Results

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.

```
always @(sel) begin
  if (sel & en) begin
    a <= #1 1'b1;
  end else if(sel==1'bx) begin
    a <= #1 1'bx;
  end else begin
    a <= #1 1'b0;
  end
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.

```
// Concurrent assignment
assign d = a & b | c; // combinational logic

// Conditional assignment
assign b = enable ? a : 1'bz; // tri-state buffer
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.

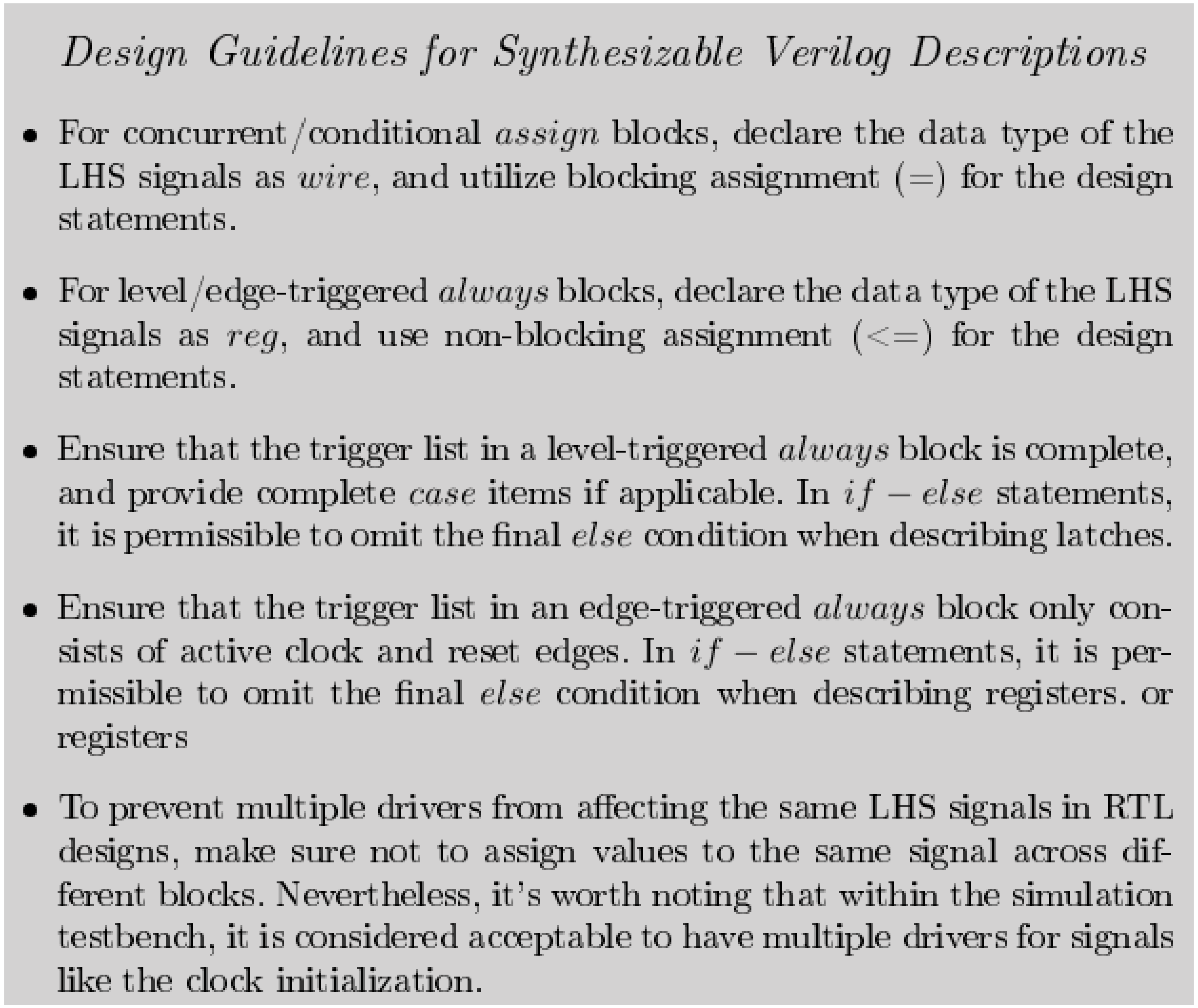
6.1.3 Synthesizable Verilog HDL

- 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.

```
// Level-trigger always (if-else, case, loop):
always @(a, b, en)  if (en) c <= a & b;  //latch

// Edge-trigger always (if-else, case, loop):
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.

Outline

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

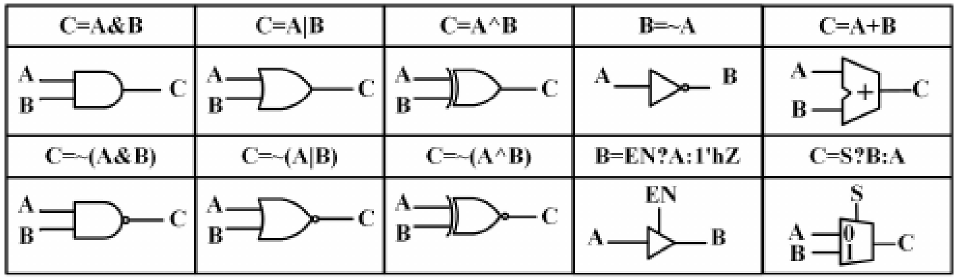
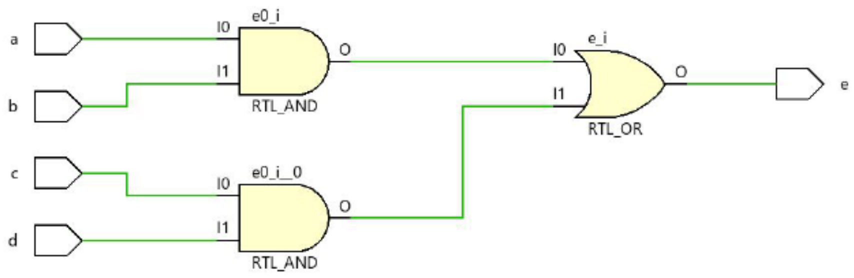


FIGURE 6.2
Basic Logic Descriptions Using Concurrent/Conditional Assignments

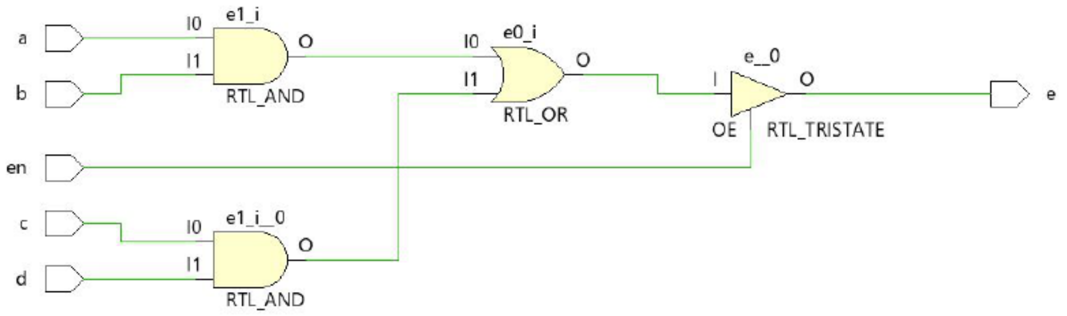
```
// Example #1: Design AND-OR Using Concurrent Assign
module example1_and_or (output e          ,
                        input  a, b, c, d );
assign e = (a & b) | (c & d);
endmodule

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

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

6.2.1. Fundamental Combinational Logic

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

// Example #2: Design AND-OR Using Level-Trigger Always
module example2_and_or (output reg e          ,
                        input      a, b, c, d );
always @ (a, b, c, d) begin
  e = (a & b) | (c & d);
end
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

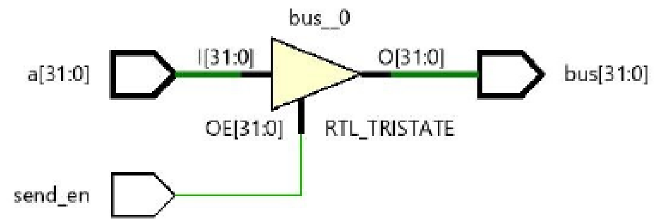
FIGURE 6.3
Synthesis Results of AND-OR-Tri Description.

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

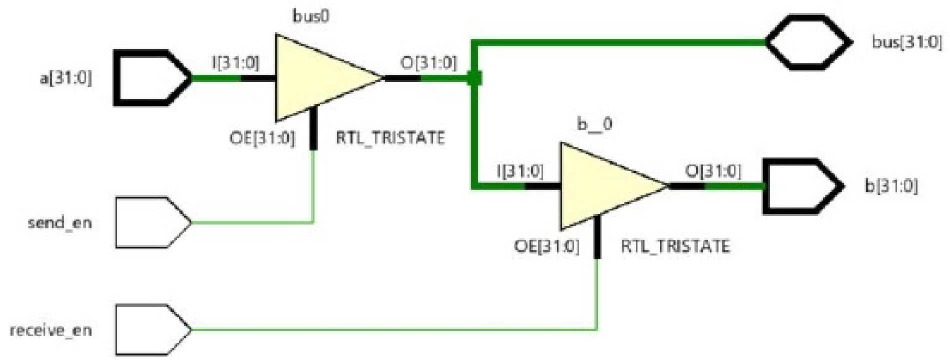
6.2.2 Uni-Directional and Bi-Directional Buses

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



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

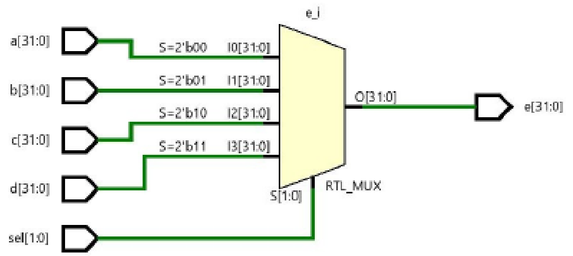


(b) Example #2 : Design Bi-Directional Bus

FIGURE 6.4
Synthesis Results of Uni- and Bi-Directional Buses

6.2.3 Multiplexer

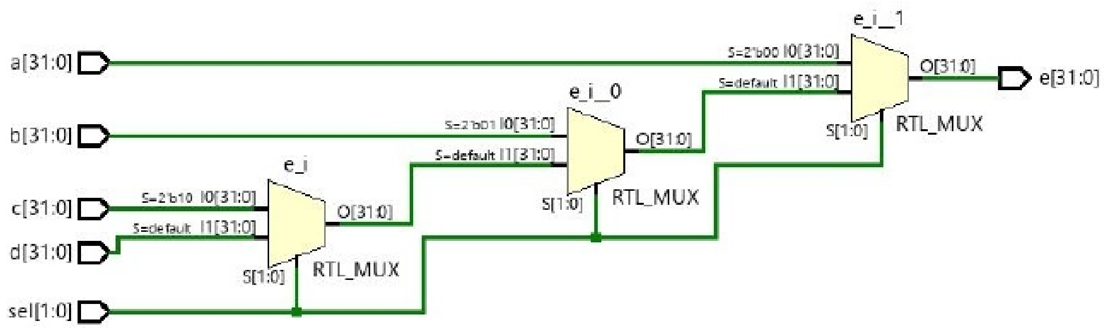
- Multiplexer



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

```
// Example #1: Multiplexer Design Using case-endcase
module example1_mux (output reg [31: 0] e          ,
                     input      [31: 0] a, b, c, d,
                     input      [1 : 0] sel        );
always @ (a, b, c, d, sel) begin
  case (sel)
    2'h0    : e <= a    ;
    2'h1    : e <= b    ;
    2'h2    : e <= c    ;
    2'h3    : e <= d    ;
    default: e >= a     ;
  endcase
end
endmodule
```

```
// Example #2: Multiplexer Design Using if-else
module example2_mux (output reg [31: 0] e          ,
                     input      [31: 0] a, b, c, d,
                     input      [1 : 0] sel        );
always @ (a, b, c, d, sel) begin
  if (sel == 2'h0) begin
    e <= a;
  end else if (sel == 2'h1) begin
    e <= b;
  end else if (sel == 2'h2) begin
    e <= c;
  end else begin
    e <= d;
  end
end
endmodule
```



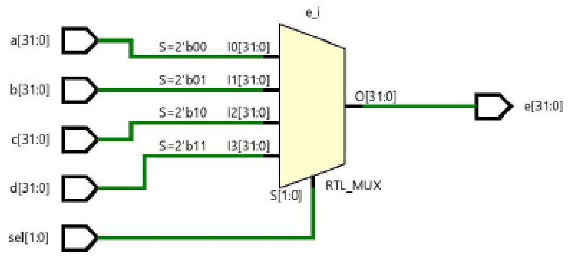
(b) Example #2: Design a Multiplexer with Different Priority Using if-else in always or conditional assign

FIGURE 6.5
Synthesis Results of Different Designs with Multiplexer

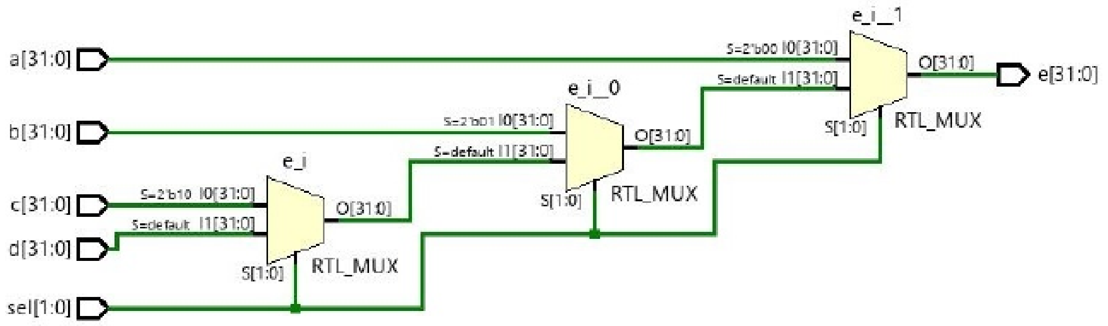
6.2.3 Multiplexer

- Multiplexer

```
// Example #3: Multiplexer Design Using Conditional Assign
module example3_mux (output [31: 0] e          ,
                     input  [31: 0] a, b, c, d,
                     input  [1 : 0] sel        );
assign e = (sel == 2'h0) ? a :
               (sel == 2'h1) ? b :
                   (sel == 2'h2) ? c : d;
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

Outline

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

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

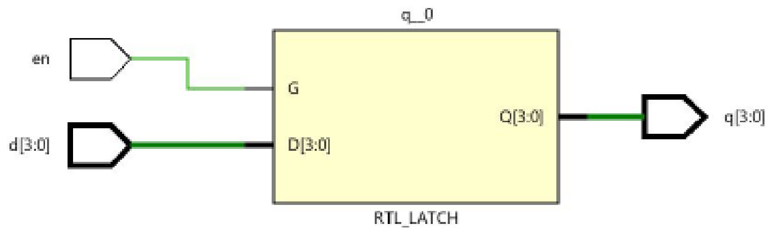
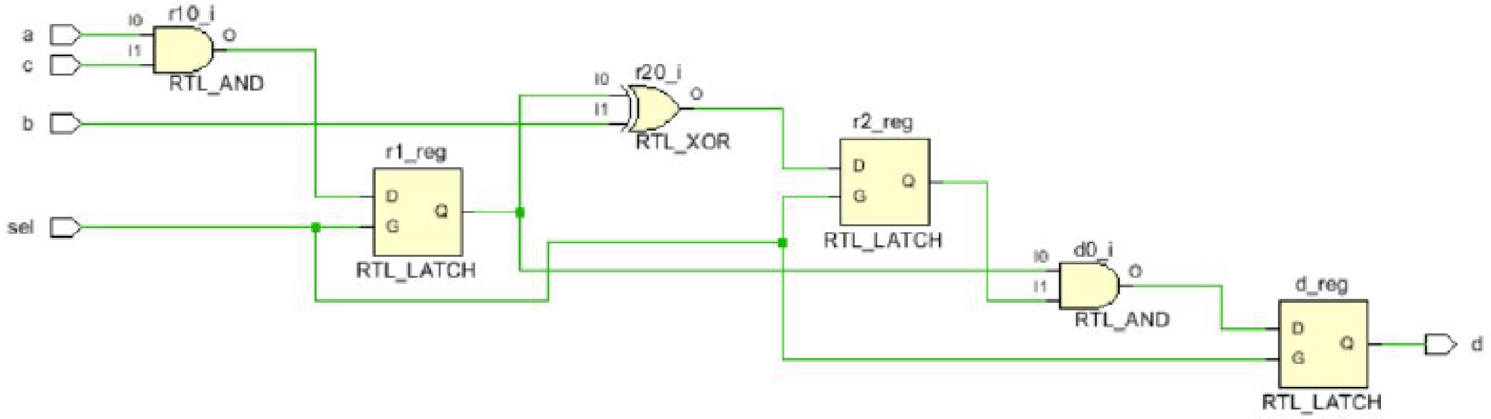


FIGURE 6.6
Synthesis Results of Latch

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

```
// Example #1: Design Combinational Logic and Latches
module example1_comb_latch (input        a, b, c, sel,
                            output reg d           );
reg r1, r2;

always @ (a, b, c, sel) begin
  if (sel) begin
    r1 <= a & c;
    r2 <= r1 ^ b;
    d  <= r1 & r2;
  end
end
endmodule
```

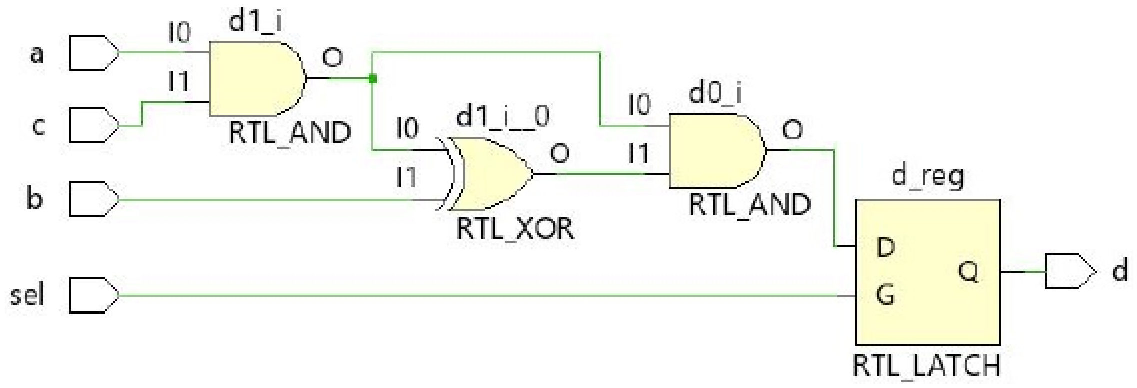


(a) Example #1: Intential Design of Laches

6.3.1 Intentional Latches Design with Verilog

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

```
// Example #2: An Alternative Way to the Design of
// Combinational Logic and Latches
module example2_comb_latch (input        a, b, c, sel,
                            output reg d            );
wire w1, w2, w3;
assign w1 = a & c;
assign w2 = w1 ^ b;
assign w3 = w1 & w2;

always @(w3, sel) begin
  if (sel) begin
    d  <= w3;
  end
end
endmodule
```



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

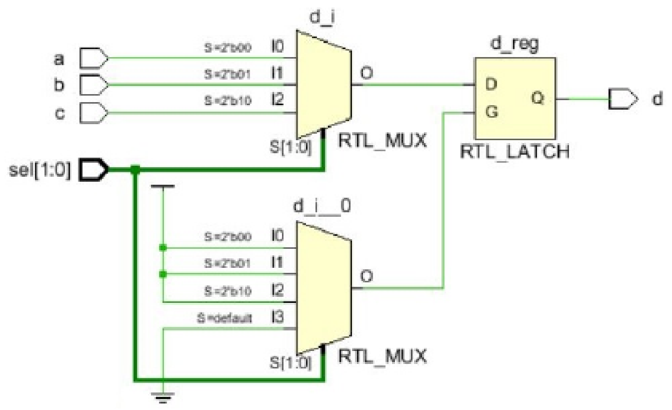
FIGURE 6.7
Synthesis Results of Intentional Latches Design

6.3.2 Accidental Design Latches with Verilog

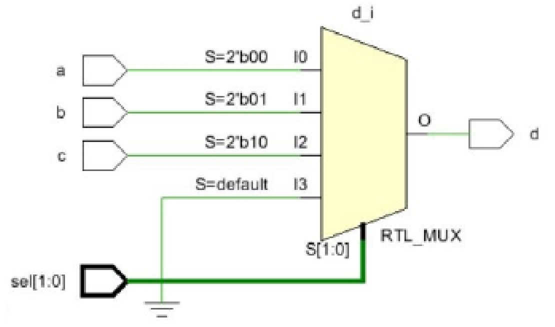
- Accidental Design Latches with Verilog
  - Incomplete case-endcase

```
// Example #1: Accidental Latches with Incomplete Case
module example1_acc_latch_co_case (input        a, b, c,
                                   input  [1:0] sel     ,
                                   output reg   d       );
always @ (a, b, c, sel) begin
  case(sel)
    2'b00  : d <= a   ;
    2'b01  : d <= b   ;
    2'b10  : d <= c   ;
  endcase
end
endmodule
```

```
// Example #2: Design with Complete Case
module example2_acc_latch_co_case (input        a, b, c,
                                   input  [1:0] sel     ,
                                   output reg   d       );
always @ (a, b, c, sel) begin
  case(sel)
    2'b00  : d <= a   ;
    2'b01  : d <= b   ;
    2'b10  : d <= c   ;
    default: d <= 1'b0;
  endcase
end
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 *always* block during simulation.
    - Synthesis: the synthesis tool may automatically analyze the design code and complete the sensitivity list to generate the desired hardware.

```
// Example #1: Accidental Latches with Incomplete List
module example1_acc_latch_inc_list (input        a, b, c,
                                    input  [1:0] sel     ,
                                    output reg   d       );

always @(a, b, sel) begin
  case(sel)
    2'b00   : d <= a;
    2'b01   : d <= b;
    2'b10   : d <= c;
    default: d <= 1'b0;
  endcase
end
endmodule
```

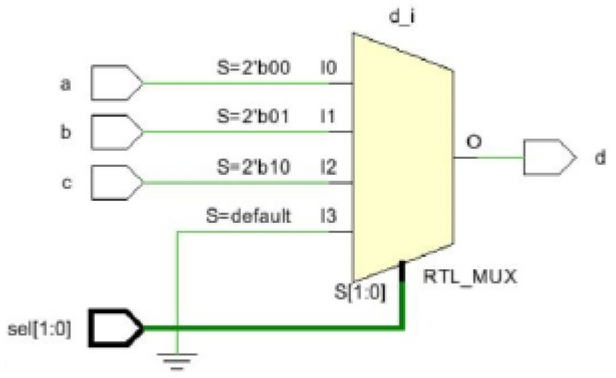


FIGURE 6.9
Synthesis Result of Incomplete Sensitivity List

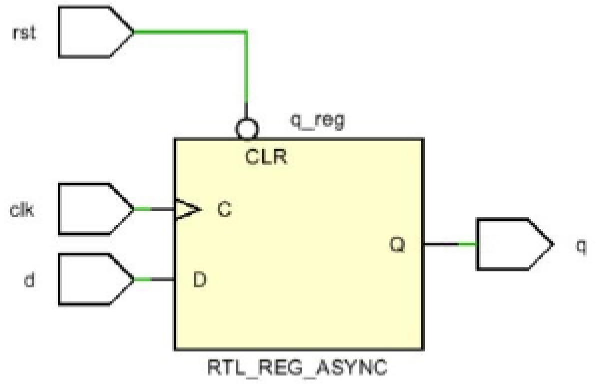
Outline

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

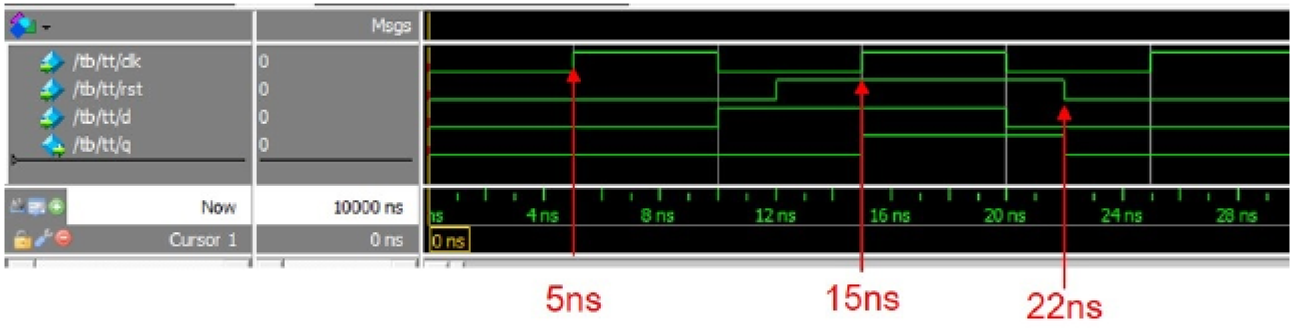
6.4.1 Single-Bit Register with Asynchronous and Synchronous Reset

- Single-Bit Register with Asynchronous Reset

```verilog
// A Design Example of Asynchronous Reset
module example_async_reset (input  d, clk, rst,
                            output reg q       );
always @ (posedge clk, negedge rst) begin
  if (~rst) begin
    q <= 1'b0;
  end else begin
    q <= d;
  end
end
endmodule
```



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

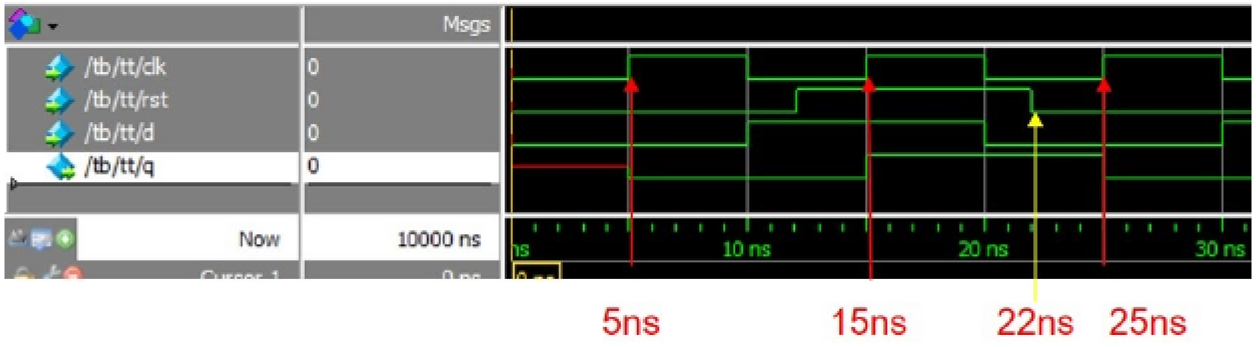


(a) Simulation Result of Asynchronous Reset

6.4.1 Single-Bit Register with Asynchronous and Synchronous Reset

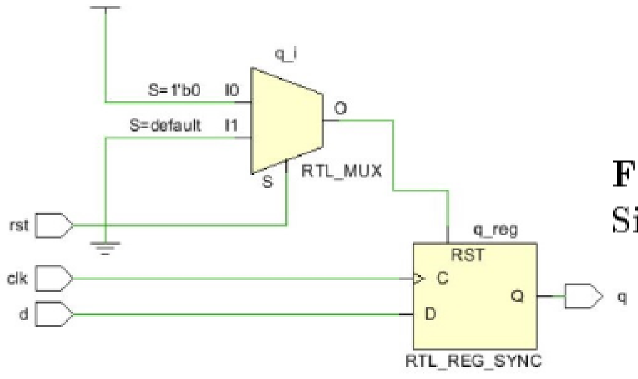
- Single-Bit Register with Synchronous Reset

```
// A Design Example of Synchronous Reset
module example_sync_reset (input  d, clk, rst,
                           output reg q     );
always @ (posedge clk) begin
  if (~rst) begin
    q <= 1'b0;
  end else begin
    q <= d;
  end
end
endmodule
```



(b) Simulation Result of Synchronous Reset

FIGURE 4.4
Simulation Results of Asynchronous and Synchronous Reset

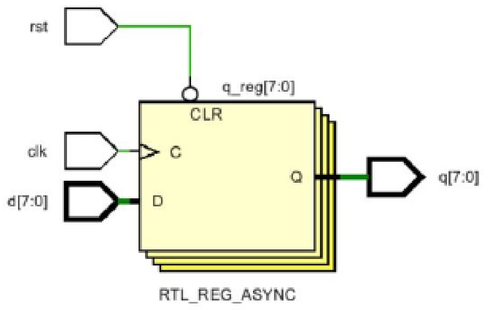


(b) Example #2: Single-Bit Register Design with Synchronous Reset

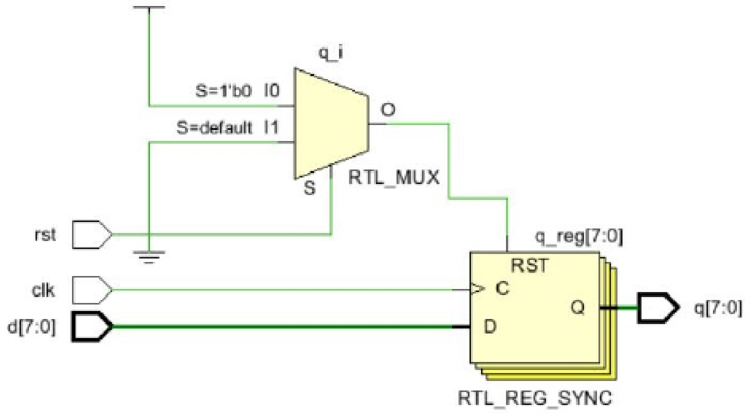
FIGURE 6.10
Synthesis Results of Single-Bit Register Design with Asynchronous and Synchronous Reset

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 Synchronouse Reset

```
// Example #1: 8-bit Register with Asynchrnous Reset
module reg_async_rst (input              rst, clk,
                      input        [7:0] d      ,
                      output reg   [7:0] q      );
always @ (negedge rst, posedge clk) begin
  if (~rst) begin
    q<=8'h0;
  end else begin
    q<=d   ;
  end
end
endmodule

// Example #2: 8-bit Register with Synchrnous Reset
module reg_sync_rst (input              rst, clk,
                     input        [7:0] d      ,
                     output reg   [7:0] q      );
always @ (posedge clk) begin
  if (~rst) begin
    q<=8'h0;
  end else begin
    q<=d   ;
  end
end
endmodule
```

FIGURE 6.11
Synthesis Results of 8-bit Register Design with Asynchronous and Synchronous Reset

6.4.3 Shift Register

- Shift Register
  - Applications:
    - Serial data reception and the introduction of clock cycle delays for data processing. Typically, the shift register design allows for the propagation of input data over multiple clock cycles. The number of clock cycle delays is determined by the number of registers connected in series.
  - Big-endian input and little-endian input:
    - In the big-endian input mode, the data stream begins with the input of the most-significant bit (MSB), followed by the successive single-bit data in sequential order until reaching the least-significant bit (LSB) of the data stream.
    - Conversely, in the little-endian input mode, the process starts with the input of the LSB, followed by the input of the remaining data in sequential order.

6.4.3 Shift Register

- Shift Register
  - 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.
  - B. Verilog Design and Synthesis

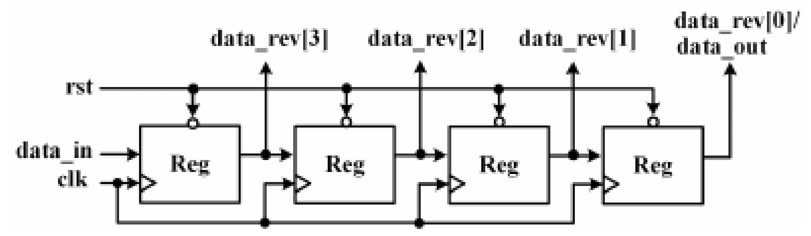


FIGURE 1.12
Block Diagram of Shift Register (Little-Endian Input)

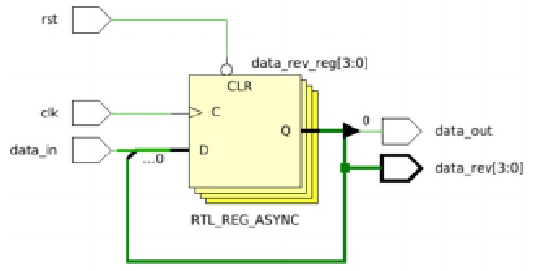


FIGURE 1.13
Synthesis Result of Shift Register

```
// Shift Register Design
module shift_reg (input              rst     ,
                  input              clk     ,
                  input              data_in ,
                  output             data_out ,
                  output reg [3:0]   data_rev);

assign data_out = data_rev[0];
always @ (negedge rst, posedge clk) begin
  if (~rst) begin
    data_rev  <= 4'h0                        ;
  end else begin
    data_rev  <= {data_in, data_rev[3:1]};
  end
end
endmodule
```

Outline

6.1 Introduction to Synthesis

6.2 Synthesis of Combinational Logic

6.3 Synthesis of Sequential Latches

6.4 Synthesis of Sequential Registers

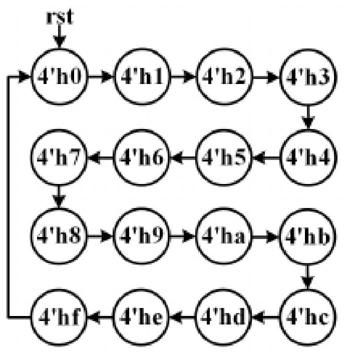
6.5 Synthesis of Counter and Timer

6.5.1 Counter 0-f

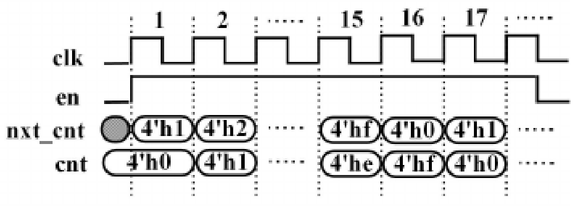
- Counter 0-f
  - Application:
    - Counters play a critical role in controlling timing and sequencing across multiple clock cycles.
      - For instance: serial buses like I2C, SPI, SDIO, GPIO, UART, etc., as well as numerical designs involving long datapaths such as vector-vector multiplications.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

TABLE 6.1
Counter IOs Description

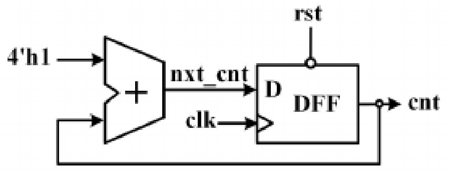
Name	Direction	Bit Width	Description
clk	Input	1	Clock
rst	Input	1	Asynchronous reset, 0 valid
en	Input	1	Enable signal, 1 valid
cnt	Output	4	Counter output



(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

```
// 0-f Counter Design
module cnt_0_f (input                 rst, clk, en,
                output reg [3:0] cnt           );

wire [3:0] nxt_cnt = en ? cnt+4'h1 : cnt;

always @ (negedge rst, posedge clk) begin
  if (~rst) begin
    cnt <= 4'h0     ;
  end else begin
    cnt <= nxt_cnt ;
  end
end
endmodule
```

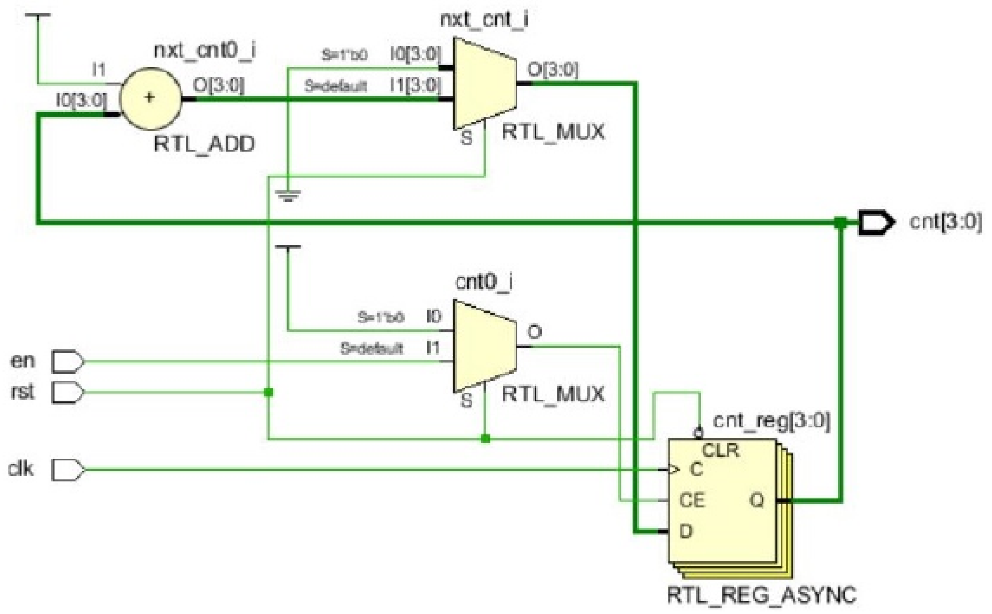


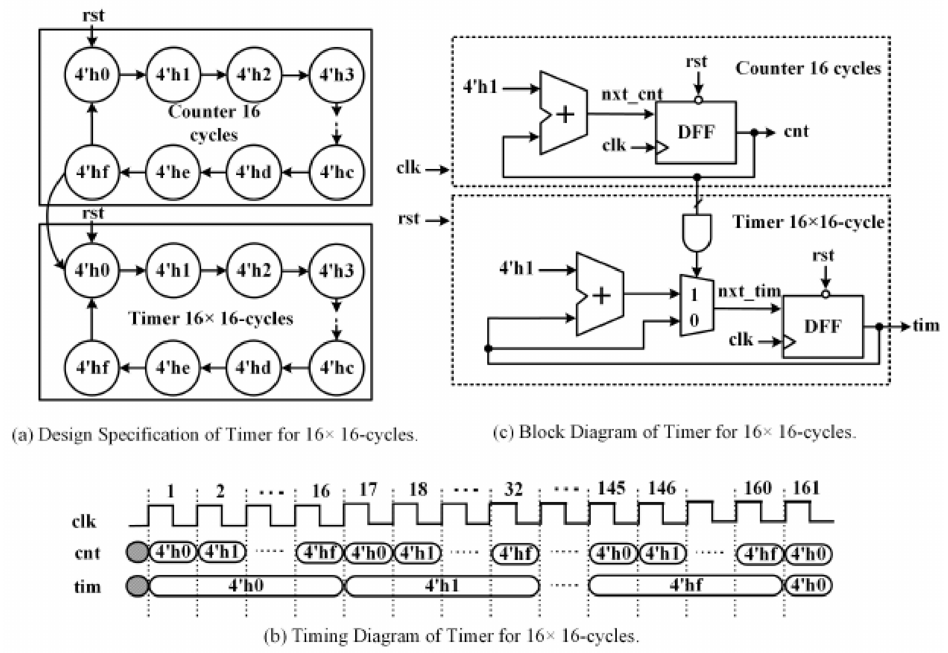
FIGURE 6.15
Synthesis Result of Counter from 0 to f

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



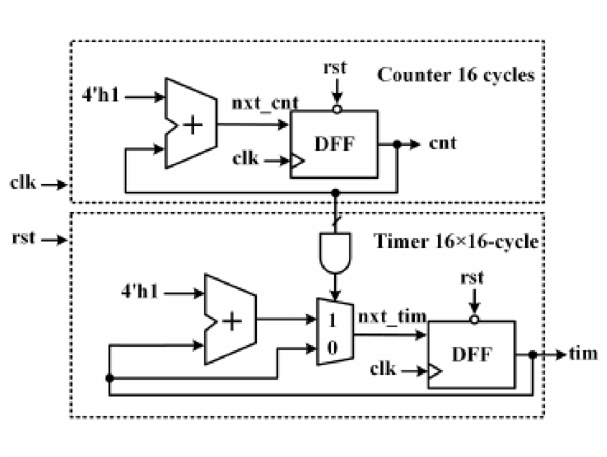
(a) Design Specification of Timer for 16× 16-cycles.
(c) Block Diagram of Timer for 16× 16-cycles.
(b) Timing Diagram of Timer for 16× 16-cycles.

FIGURE 6.16
Design Specification of Timer for 16× 16-cycles

6.5.2 Timer 0-f

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

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



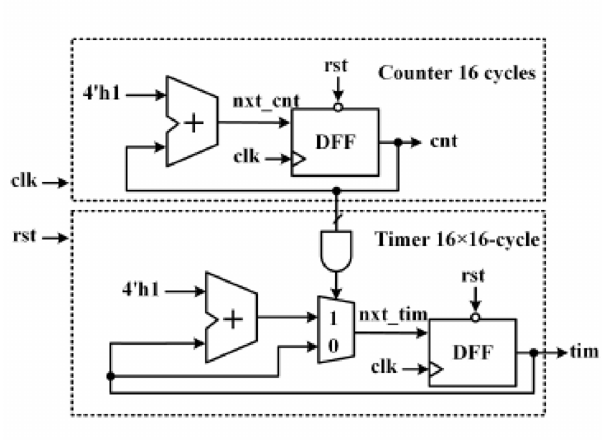
:les.   (c) Block Diagram of Timer for 16× 16-cycles.

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

6.5.2 Timer 0-f

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

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

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



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

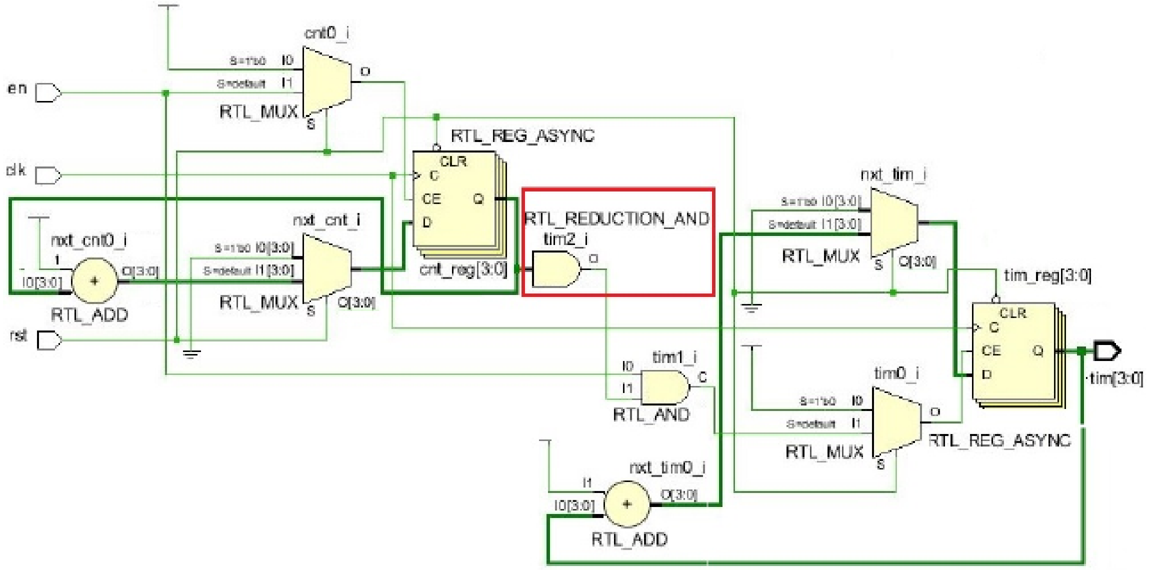


FIGURE 6.17
Synthesis Result of Timer for 16×16-cycle