Lecture 4 RTL Design with Verilog HDL

Outline

4.1 Design Statements in always and initial: if-else, case, for/while/repeat/forever loop

4.2 Blocking and Non-Blocking

4.3 Asynchronous and Synchronous Reset

4.4 Hierarchical Design and Instantiation

4.5 RTL Design Rules with Verilog HDL

Outline

4.1 Design Statements in always and initial: if-else, case, for/while/repeat/forever loop

4.2 Blocking and Non-Blocking

4.3 Asynchronous and Synchronous Reset

4.4 Hierarchical Design and Instantiation

4.5 RTL Design Rules with Verilog HDL

4.1.1 if-else statement

- if-else Statement

```
1  if (expression1) begin
2    statement1;
3  end else if (expression2) begin
4    statement2;
5  end else begin //can be omitted if it is assigned to itself
6    statement3;
7  end
```

  - Missing-else Design on a Latch
    - Uses the if-else statement in a level-trigger *always* block
    - LHS signal ``q'' must be declared as *reg*
    - All the signals able to trigger the design circuit must be included in the trigger list

```
1  //Ex 1: A Latch Design Using if-else Statement
2  module example_latch (input      rst, en, a, b,
3                        output reg q            );
4  always @(rst, en, a, b) begin
5    if(~rst) begin
6      q <= 1'b0 ;
7    end else if(en) begin
8      q <= a&b  ;
9    end else begin
10     q <= q    ;
11   end
12 end
13 endmodule
```

4.1.1 if-else statement

- if-else Statement
  - Missing-else Design on a Latch
    - A recommended coding style:
      - First, the circuit design is divided into a combinational logic AND using an *assign* block (line 15) and a sequential latch using an *always* block (lines 16-23).
      - In addition, a missing-else design is employed in the if-else statement. A missing-else design is very common to describe latches that is highly recommended in this book.

```
1  //Ex 1: A Latch Design Using if-else Statement
2  module example_latch (input      rst, en, a, b,
3                        output reg q            );
4  always @(rst, en, a, b) begin
5    if(~rst) begin
6      q <= 1'b0 ;
7    end else if(en) begin
8      q <= a&b  ;
9    end else begin
10     q <= q    ;
11   end
12 end
13 endmodule
```

```
15 //Recommended Design for a Latch
16 module example_rec_latch (input      rst, en, a, b,
17                           output reg q            );
18 wire nxt_q = a&b; //combinational AND gate
19 always @(rst, en, nxt_q) begin
20   if(~rst) begin
21     q <= 1'b0 ;
22   end else if(en) begin
23     q <= nxt_q;
24   end //last else (q<=q;) is omitted
25 end
26 endmodule
```

4.1.1 if-else statement

- if-else Statement
  - Missing-else Design on a Register
    - Two ``negedge rst'' and ``posedge clk'', are listed in the sensitivity list
    - A recommended coding style:
      - A combinational logic AND using an *assign* block (line 15) and a sequential latch using an *always* block (lines 16-24).
      - A missing-else design is employed.

```
1  //Ex 2: A Register Design Using if-else Statement
2  module example_reg (input      rst, clk, en, a, b,
3                      output reg q                 );
4  always @(negedge rst, posedge clk) begin
5    if(~rst) begin
6      q <= 1'b0 ;
7    end else if(en) begin
8      q <= a&b  ;
9    end else begin
10     q <= q    ;
11   end
12 end
13 endmodule
```

```
15 //Recommended Design for a Register
16 module example_rec_reg (input      rst, clk, en, a, b,
17                         output reg q                 );
18 wire nxt_q = a&b; //combinational AND gate
19 always @(negedge rst, posedge clk) begin
20   if(~rst) begin
21     q <= 1'b0 ;
22   end else if(en) begin
23     q <= nxt_q;
24   end //last else (q<=q;) is omitted
25 end
26 endmodule
```

4.1.1 if-else statement

- if-else Statement
  - Using Logic Operators in if-else Statement
    - A recommended design
      - Using a simple nested conditional assignment, the same functions are realized.
      - *assign* blocks are recommended for combinational circuit and latch designs, due to its simplicity and readability

```
1  //Ex 3: Using Logic Operators in if-else Statement
2  module example_logic_op_ifelse (input [31:0] a,
3                                  output reg   b);
4  always @(a) begin
5    if(~|a) begin          // if(a==32'h0)
6      b <= 1'b0;
7    end else if(&a) begin // if(a==32'hffff_ffff)
8      b <= 1'b1;
9    end
10 end
11 endmodule
12
13 //Recommended Design for EX3
14 module example_rec_ex3 (input [31:0] a,
15                          output       b);
16 assign b = ~|a ? 1'b0 :
17                 &a ? 1'b1 : b ;
18 endmodule
```

4.1.1 if-else statement

- if-else Statement
  - Nested if-else Statement
    - A recommended design
      - A concurrent *assign* block shown in 16-17.

```
1  //Ex 4: Nested if-else Statement
2  module example_nested_ifelse (input      a, b,
3                                output reg c   );
4  always @(a,b) begin
5    if(a) begin
6      if(b) begin
7        c <= 1'b1;
8      end else begin
9        c <= 1'b0;
10     end
11   end else begin
12     c <= 1'b0;
13   end
14 end
15 endmodule
16
17 //Recommended Design for EX4
18 module example_rec_ex4 (input  a, b,
19                         output c   );
20 assign c = a&b ;
21 endmodule
```

Design Rule - if-else Statement

- It is allowed to omit the last *else* in an *if−else* statement. However, designers must keep in mind that omitting the last *else* in a level-triggered *always* block will generate latches to sustain the value for the LHS signals, and omitting the last *else* in an edge-triggered *always* block will generate registers to preserve the value for the LHS signals.
- Within a level-triggered *always* block, confine your designs solely to the associated signals and logic. Any designs pertaining to unrelated signals and logic should be segregated into separate level-triggered *always* blocks or arranged within *assign* blocks.
- When using edge-triggered *always* blocks, it is advisable to use them exclusively for describing registers. Combinational circuits, on the other hand, should be defined within other edge-triggered *always* blocks or in *assign* blocks.

4.1.2 case statement

- case Statement

```
case(expression)
  item1     : begin item statement1    end
  item2     : begin item statement2    end
  default   : begin DEFAULT statement  end
endcase
```

  - A Design on a 4-to-1 Multiplexer Using case-endcase Statement

```
// Ex1: A Design Example Using case-endcase Statement
module example_case1 (input       [1:0] a, b, c, sel,
                      output reg  [1:0] d);
always @ (a, b, c, sel) begin
  case (sel)
    2'h0    : d <= a;
    2'h1    : d <= b;
    2'h2    : d <= c;
    default : begin
                d <= 2'b0;
                $display("Mismatch: sel = 2'h3, X, or Z");
              end
  endcase
end
endmodule
```

4.1.2 case statement

- case Statement
  - A Design on a 4-to-1 Multiplexer Using case-endcase Statement
    - An equivalent design

```
1  // Ex1: A Design Example Using case-endcase Statement
2  module example_case1 (input       [1:0] a, b, c, sel,
3                        output reg [1:0] d);
4  always @ (a, b, c, sel) begin
5    case (sel)
6      2'h0     : d <= a;
7      2'h1     : d <= b;
8      2'h2     : d <= c;
9      default  : begin
10                  d <= 2'b0;
11                  $display("Mismatch: sel = 2'h3, X, or Z");
12                end
13   endcase
14 end
15 endmodule
```

```
17 // An Equivalent Design to Ex1
18 module example_case2 (input    [1:0] a, b, c, sel,
19                        output   [1:0] d);
20 assign d = (sel==2'h0) ? a :
21                 (sel==2'h1) ? b :
22                     (sel==2'h2) ? c : 2'h0;
23 endmodule
```

4.1.2 case statement

- case Statement
  - Missing Default in case-endcase
    - An unwanted latch may be generated.
    - For simulation, the signal ``d'' will sustain its previous value when ``sel=2'h3'' occurs.
    - Missing *default* is not recommended.

```
1  //Ex 2: Missing default Causes Unwanted Latch!
2  module example_case_missing_item (input       [1:0] a, b, c,
3                                    input       [1:0] sel,
4                                    output reg  [1:0] d);
5  always @ (a, b, c, sel) begin
6    case (sel)
7      2'h0 : d <= a;
8      2'h1 : d <= b;
9      2'h2 : d <= c;
10   endcase
11 end
12 endmodule
```

Design Rule - case-endcase Statement

- Using $case-endcase$ statements is highly recommended over a long $if-else$ statement in level-triggered *always* blocks. This approach enhances the clarity and efficiency of the design code, especially when dealing with a large number of listing items.
- It is crucial to ensure that all items in a $case-endcase$ statement are completely listed. In cases where a $default$ item is missing, the design may inadvertently generate unwanted latches or registers. This can lead to unintended behavior and potential simulation failures.

4.1.2 case statement

- case Statement
  - X and Z in case-endcase

```
1  //Ex 3: An Example Using X and Z in case-endcase Statement
2  module example_case1_x_z (input  [3:0]  sel);
3  //--- Design Code---//
4  always @(sel) begin
5    case (sel)
6      4'b0000 :   $display ("item0 matches");
7      4'b0zzz :   $display ("item1 matches");
8      4'b0xxx :   $display ("item2 matches");
9      4'b1010 :   $display ("item3 matches");
10     default :   $display ("nothing matches");
11   endcase
12 end
13
14 //--- Testbench ---//
15 initial begin
16   sel = 4'b0000;          // Printed log: # item0 matches
17   #100 sel = 4'b0xxx;     // Printed log: # item2 matches
18   #100 sel = 4'b0zzz;     // Printed log: # item1 matches
19   #100 sel = 4'b1111;     // Printed log: # nothing matches
20 end
21 endmodule
```

4.1.2 case statement

- case Statement
  - X and Z in case-endcase

```
1  //Ex 4: An Example Using X and Z in case-endcase Statement
2  module example_case2_x_z (input [1:0] sel);
3  //--- Design Code---//
4  always @(sel) begin
5   case (sel)
6     2'b00: $display("%d ns, sel=%b, 2'b00 sel", $time,sel);
7     2'b01: $display("%d ns, sel=%b, 2'b01 sel", $time,sel);
8     2'b1x: $display("%d ns, sel=%b, 2'b1x sel", $time,sel);
9     2'b1z: $display("%d ns, sel=%b, 2'b1z sel", $time,sel);
10    default: $display("%d ns, sel=%b, def sel", $time,sel);
11  endcase
12 end
13
14 //--- Testbench ---//
15 initial begin
16   sel = 2'b00;      //Printed log: # 0 ns, sel=00, 2'b00 sel
17   #10 sel = 2'b01; //Printed log: # 10 ns, sel=01, 2'b01 sel
18   #10 sel = 2'b10; //Printed log: # 20 ns, sel=10, def sel
19   #10 sel = 2'b1x; //Printed log: # 30 ns, sel=1x, 2'b1x sel
20   #10 sel = 2'b1z; //Printed log: # 40 ns, sel=1z, 2'b1z sel
21 end
22 endmodule
```

4.1.3 for/while/repeat/forever loop

- for/while/repeat/forever loop
  - Verilog HDL supports loop statements in *always* and *initial* blocks.

Design Rule - for/while/repeat/forever Loop Statement

- The loop statement serves as an efficient approach for simulation and testbenches. However, it is generally not recommended in synthesizable designs due to uncertain hardware implementations and performance.
- To ensure reliable hardware implementation, consider using alternative constructs such as sequential circuits for the iterative control, which are more appropriate for synthesis purposes.

  - for loop
    - 1) initializes a variable that controls the number of loops executed;
    - 2) evaluates an expression that exits when the result is FALSE and executes its statement when the result is TRUE;
    - 3) executes a step assignment to modify the value of the loop-control variable.

```
1 for (Initial assignment; expression; step assignment) begin
2   statement;
3 end
```

4.1.3 for/while/repeat/forever loop

- for/while/repeat/forever loop
  - for loop
    - One of the practical applications using a *for* loop is to refresh a memory block or register array.
    - A register array is declared:
      - The depth of the register array: ``mem[0:15]''. Notice that in the bracket it starts with the minimum 0 and ends with the maximum 15 which is different from other declarations
      - The width of each memory cell: ``reg [7:0]''.

```
1  reg [7:0] mem[0:15];
2  integer i;
3  initial begin
4    for (i=0; i<16; i=i+1) begin
5      mem[i] = i;
6      $display ("%d register is initialized as %h", i, i);
7    end
8  end
```

4.1.3 for/while/repeat/forever loop

- for/while/repeat/forever loop
  - while loop
    - Executes a statement until the expression becomes FALSE.
- for/while/repeat/forever loop
  - repeat loop
    - Executes the statement as the fixed number of times.

```
1  while (expression) begin
2    statement;
3  end
```

```
1  integer a;
2  initial begin
3    a=0;
4    while (a<4) begin
5      #10 a = a + 1;
6    end
7  end
```

```
1  repeat (number) begin
2    statement;
3  end
```

```
1  reg a;
2  initial begin
3    a=1'b0;
4    repeat (4) begin
5      #10 a = ~a;
6    end
7  end
```

4.1.3 for/while/repeat/forever loop

- for/while/repeat/forever loop
  - repeat loop

```
parameter MEM_SIZE = 16;
reg [3:0] addr;
reg [7:0] mem[0:MEM_SIZE-1];

initial  begin
  addr = 4'h0;
  repeat (MEM_SIZE) begin
    mem [addr] = 8'h0;
    addr = addr + 4'h1;
  end
end
```

  - forever loop

```
forever begin
  statement;
end
```

```
reg clk;
initial begin
  clk = 1'b0;
  forever begin
    #5 clk = ~clk;
  end
end
```

Outline

4.1 Design Statements in always and initial: if-else, case, for/while/repeat/forever loop

4.2 Blocking and Non-Blocking

4.3 Asynchronous and Synchronous Reset

4.4 Hierarchical Design and Instantiation

4.5 RTL Design Rules with Verilog HDL

4.2 Blocking and Non-Blocking

- Differences Between Blocking and Non-Blocking
  - Blocking assignment
    - Uses the equals (=) character
    - Used in assign block and initial block
    - Be carried out in sequence.
  - Nonblocking assignment
    - Uses the less-than-equals (<=) character.
    - Used in always block and initial block
    - Be executed in parallel

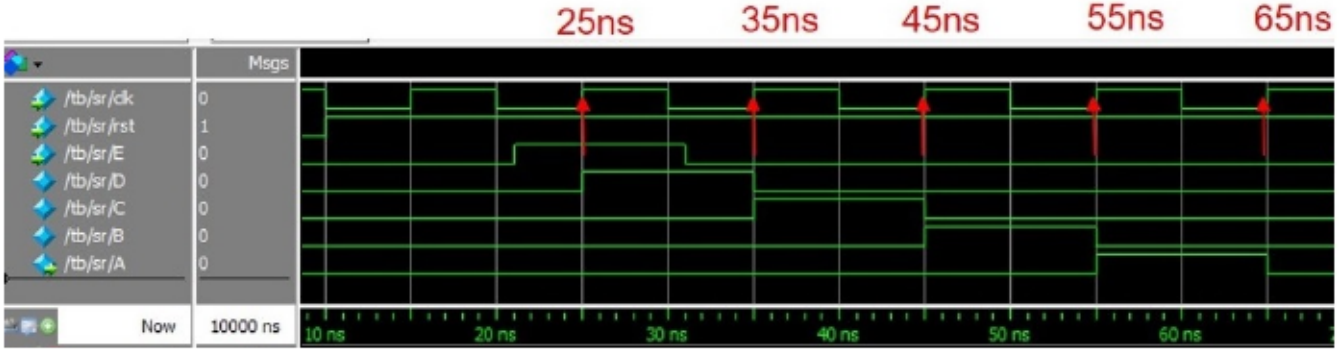
4.2 Blocking and Non-Blocking

- 4.2.1 Examples of Blocking Designs

```
1  // An Example of Blocking Design #1
2  module example_blk1 (input       e, clk, rst,
3                       output reg a          );
4  reg b, c, d;
5  always @ (posedge clk, negedge rst) begin
6    if (~rst) begin
7      a = 1'b0; b = 1'b0; c = 1'b0; d = 1'b0;
8    end else begin
9      a = b;
10     b = c;
11     c = d;
12     d = e;
13   end
14 end
15 endmodule
```

```
17 // Another Example of Blocking Design #2
18 module example_blk2 (input       e, clk, rst,
19                       output reg a          );
20 reg b, c, d;
21 always @ (posedge clk, negedge rst) begin
22   if (~rst) begin
23     a = 1'b0; b = 1'b0; c = 1'b0; d = 1'b0;
24   end else begin
25     d = e;
26     c = d;
27     b = c;
28     a = b;
29   end
30 end
31 endmodule
```

- 4.2.1 Examples of Blocking Designs
  - Simulation Results of Blocking Designs



(a) Simulation Result of Blocking Design #1

```
10      a = b;
11      b = c;
12      c = d;
13      d = e;
```

(b) Simulation Result of Blocking Design #2

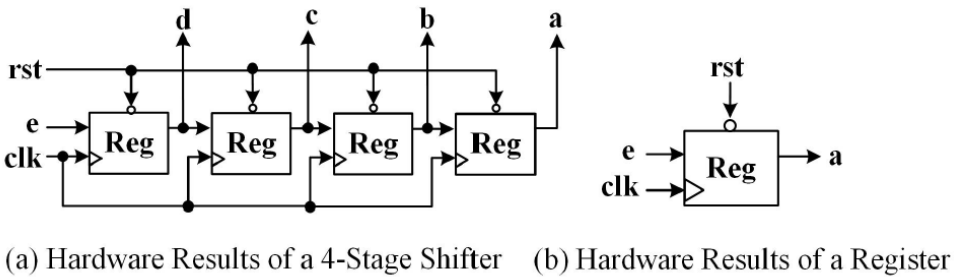
```
27      d = e;
28      c = d;
29      b = c;
30      a = b;
```

FIGURE 4.2
Simulation Results of Blocking Design.

- 4.2.1 Examples of Blocking Designs
  - Hardware Results

```
1  // An Example of Blocking Design #1
2  module example_blk1 (input      e, clk, rst,
3                       output reg a          );
4  reg b, c, d;
5  always @ (posedge clk, negedge rst) begin
6    if (~rst) begin
7      a = 1'b0; b = 1'b0; c = 1'b0; d = 1'b0;
8    end else begin
9      a = b;
10     b = c;
11     c = d;
12     d = e;
13   end
14 end
15 endmodule
```

```
17 // Another Example of Blocking Design #2
18 module example_blk2 (input      e, clk, rst,
19                       output reg a          );
20 reg b, c, d;
21 always @ (posedge clk, negedge rst) begin
22   if (~rst) begin
23     a = 1'b0; b = 1'b0; c = 1'b0; d = 1'b0;
24   end else begin
25     d = e;
26     c = d;
27     b = c;
28     a = b;
29   end
30 end
31 endmodule
```

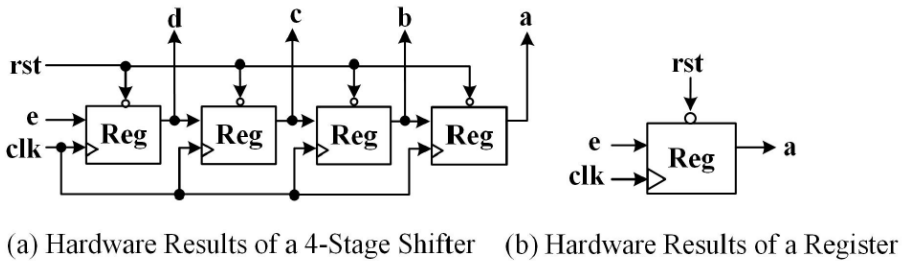


(a) Hardware Results of a 4-Stage Shifter (b) Hardware Results of a Register

FIGURE 4.3
Hardware Results of Blocking Designs

- 4.2.2 Examples of Non-Blocking Designs

```
1  // Non-Blocking Design on a 4-Stage Shift Register
2  module example_nonblk1 (input       e, clk, rst,
3                          output reg a          );
4  reg b, c, d;
5  always @ (posedge clk, negedge rst) begin
6    if (~rst) begin
7      a <= 1'b0; b <= 1'b0; c <= 1'b0; d
8    end else begin
9      a <= b;
10     b <= c;
11     c <= d;
12     d <= e;
13   end
14 end
15 endmodule
16
17 // Non-Blocking Design on a Register
18 module example_nonblk2 (input       e, clk, rst,
19                         output reg a          );
20 always @ (posedge clk, negedge rst) begin
21   if (~rst) begin
22     a <= 1'b0;
23   end else begin
24     a <= e   ;
25   end
26 end
27 endmodule
```



(a) Hardware Results of a 4-Stage Shifter (b) Hardware Results of a Register

FIGURE 4.3
Hardware Results of Blocking Designs

Outline

4.1 Design Statements in always and initial: if-else, case, for/while/repeat/forever loop

4.2 Blocking and Non-Blocking

4.3 Asynchronous and Synchronous Reset

4.4 Hierarchical Design and Instantiation

4.5 RTL Design Rules with Verilog HDL

4.3 Asynchronous and Synchronous Reset in Verilog

- 4.3.1 Asynchronous reset
  - An asynchronous reset activates as soon as the reset signal is asserted.
  - In sensitivity list
- 4.3.2 Synchronous reset
  - A synchronous reset activates on the active clock edge when the reset signal is asserted.
  - Not in sensitivity list

```
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
12
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   else begin
20     q <= d:
21   end
22 end
23 endmodule
```

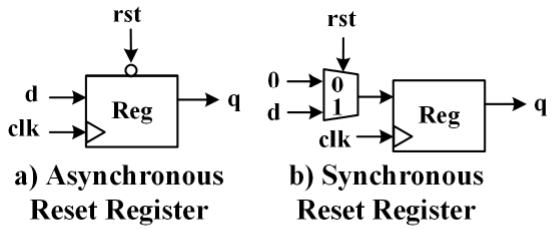
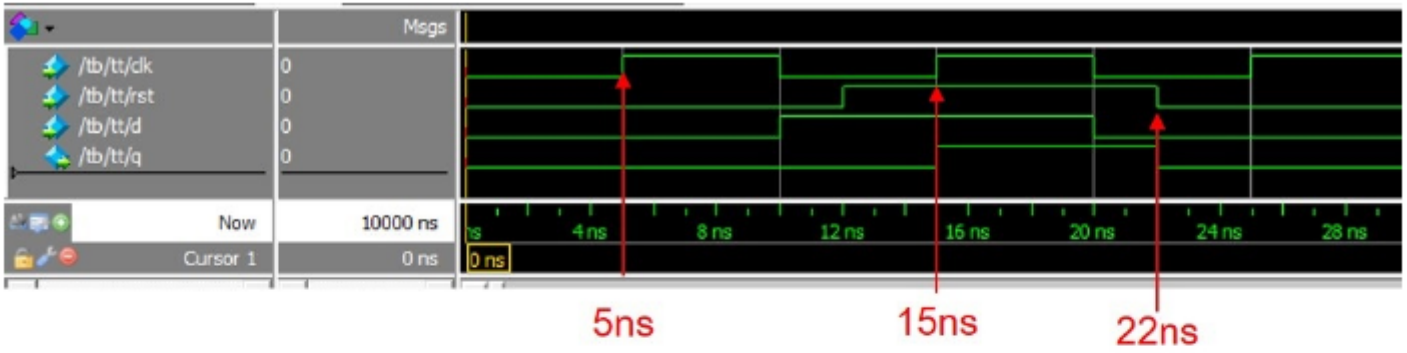


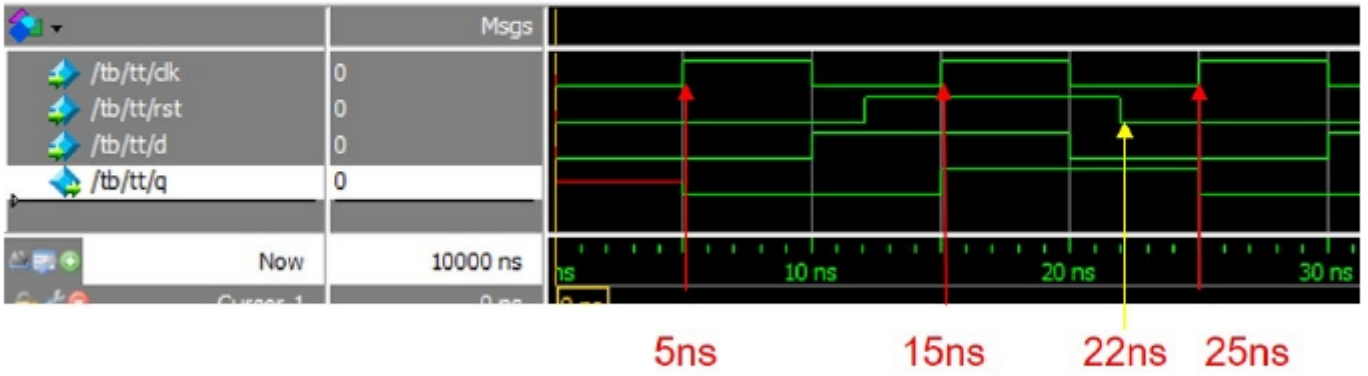
FIGURE 4.4
Hardware Results of Asynchronous and Synchronous Reset

4.3 Asynchronous and Synchronous Reset in Verilog

- Simulation Waveform of Asynchronous and Synchronous Reset



(a) Simulation Result of Asynchronous Reset



(b) Simulation Result of Synchronous Reset

FIGURE 4.4
Simulation Results of Asynchronous and Synchronous Reset.

Outline

4.1 Design Statements in always and initial: if-else, case, for/while/repeat/forever loop

4.2 Blocking and Non-Blocking

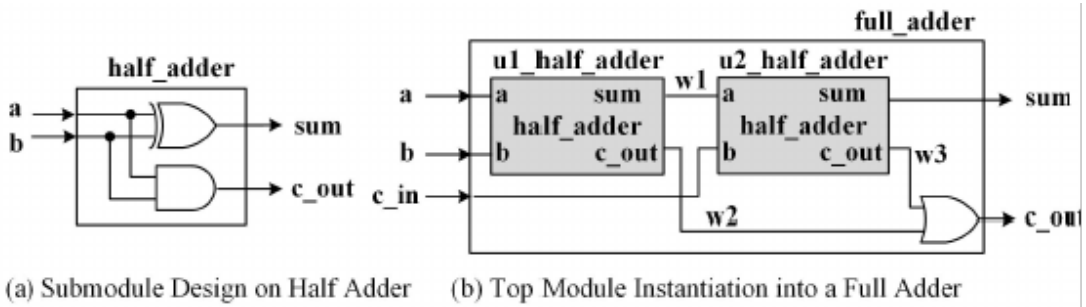
4.3 Asynchronous and Synchronous Reset

4.4 Hierarchical Design and Instantiation

4.5 RTL Design Rules with Verilog HDL

4.4 Hierarchical Design and Instantiation

- 4.4.1 Verilog Design Example of Hierarchical Design
  - full_adder
    - u1_half_adder
      - XOR
      - AND
    - u2_half_adder
      - XOR
      - AND
    - OR gate



(a) Submodule Design on Half Adder (b) Top Module Instantiation into a Full Adder

FIGURE 4.5
Hierarchical Design and Instantiation

```
1  module half_adder (output sum, c_out,
2                     input  a, b      );
3  // Behavioral model level design:
4  // assign {c_out,sum} = a + b;
5  assign sum   = a ^ b;
6  assign c_out = a & b;
7  endmodule
```

4.4 Hierarchical Design and Instantiation

- 4.4.1 Verilog Design Example of Hierarchical Design
  - full_adder
    - u1_half_adder
      - XOR
      - AND
    - u2_half_adder
      - XOR
      - AND
    - OR gate

```
9  // Behavioral model level design:
10 // assign {c_out,sum} = a + b + c_in;
11 module full_adder (output sum, c_out,
12                     input  a, b, c_in);
13 wire              w1, w2, w3;
14
15 half_adder u1_half_adder (.sum   (w1)   ,
16                           .c_out(w2)   ,
17                           .a    (a )   ,
18                           .b    (b )   );
19
20 half_adder u2_half_adder (.sum   (sum)  ,
21                           .c_out(w3 )  ,
22                           .a    (w1 )  ,
23                           .b    (c_in));
24
25 assign c_out = w2 | w3;
26 endmodule
```

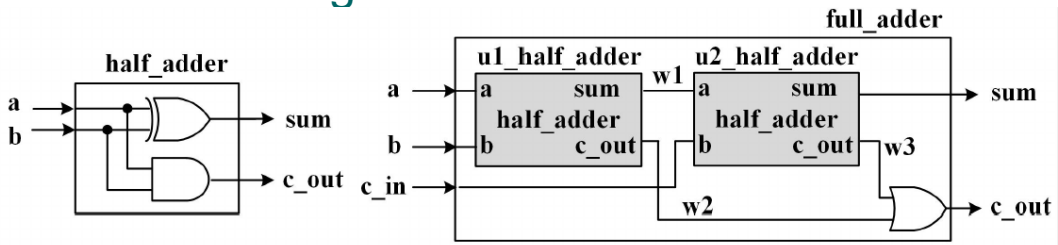


FIGURE 4.5
Hierarchical Design and Instantiation

4.4 Hierarchical Design and Instantiation

- 4.4.2 Verilog Testbench Example of Instantiation
  - tb_full_adder
    - u_full_adder
      - u1_half_adder
      - u2_half_adder
      - OR

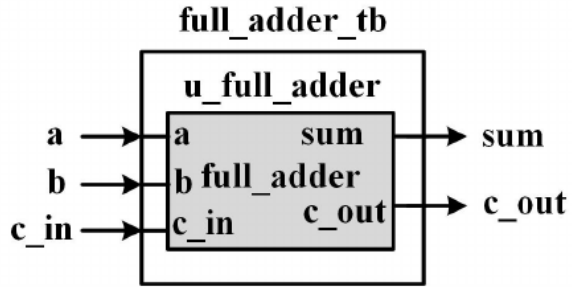


FIGURE 4.6
Design Instantiation in Testbench

```
1  module full_adder_tb ();
2  reg  a, b, c_in;
3  wire sum, c_out;
4  // Design-Under-Test Instantiation
5  full_adder u_full_adder (.sum   (sum   ),
6                           .c_out(c_out),
7                           .a     (a     ),
8                           .b     (b     ),
9                           .c_in  (c_in  ));
10 endmodule
```

Outline

4.1 Design Statements in always and initial: if-else, case, for/while/repeat/forever loop

4.2 Blocking and Non-Blocking

4.3 Asynchronous and Synchronous Reset

4.4 Hierarchical Design and Instantiation

4.5 RTL Design Rules with Verilog HDL

4.5 RTL Design Rules with Verilog HDL

Verilog Design Rules

1. **RTL Programming:**

   1) Avoid creating **design loops** within combinational circuits that can lead to unpredictable behavior in hardware.

   2) Since RTL designs rely on clock-edge-triggered registers, generating half-cycle signals is not feasible.

   3) Align the bit numbers of signals within signal assignments to avoid bit-mismatching issues.

   4) In edge-triggered *always* blocks, include only clock edge and reset edge in the sensitivity list. In level-triggered *always* blocks, include all signals that can trigger the LHS signals inside the sensitivity list.

   5) Separate combinational and sequential circuit descriptions into different design blocks, in order to promote clarity in design code and simplify debugging and modifications.

   6) Avoid using **deep logic** to maintain concise and readable design code.

   7) Organize your code by grouping functionally related design statements within the same *always* block, and place functionally unrelated statements in separate design blocks.

   8) For synthesizable RTL designs, avoid unknown statuses for signals and IOs during design and simulation. Initialize all inputs of the design-under-test in the test cases.

4.5 RTL Design Rules with Verilog HDL

2. **RTL Design:**

1) Use the **hierarchical structure** to divide complex circuit designs into different levels of submodules, promoting independent design and simulation.

2) Implement a **global reset** for the entire system to enhance system stability and predictability during startup or error conditions.

3) Minimize the use of **multiple clocks** to avoid complexity and clock domain crossing issues.

4) For data transfer between different clock domains, use a **data buffer** or **synchronizer** to ensure proper synchronization.

5) Follow the **register-in register-out** rule for RTL designs to create a clear pipeline between registers and avoid unnecessary timing issues.

6) Establish consistent register designs across various scenarios involving low/high valid reset, synchronous/asynchronous reset, and rising/falling clock edge.

7) Avoid using the division operator provided by Verilog HDL for both ASIC and FPGA designs. Use arithmetic IP modules instead.

4.5 RTL Design Rules with Verilog HDL

3. **FPGA Design and Verification:**

   1) For FPGA verification within the ASIC design flow, strive to maximize the resemblance between the FPGA design code and the ASIC design code.

   2) FPGAs may automatically reset all signals when powered on, which may not apply to ASICs. In ASIC designs, ensure that all signals are reset in the initial stage.

4. **Design for Testing and Remediation Circuitry:**

   1) Integrate testing circuits to verify proper circuit functionality.

   2) Incorporate Build-in-Self-Test (BIST) modules for memory blocks, enabling the identification and containment of errors.

   3) Develop remediation circuits to rectify identified errors.

5. **Consideration of Timing:**

   1) When developing RTL code, it's crucial to take timing considerations into account. Practical outcomes might be different from RTL simulation results due to timing issues.

   2) During design specification, identify the anticipated operational clock frequency and the longest critical path.

6. **Memory:** Avoid reading memory before writing to it. Some FPGAs may reset all memory blocks to zeros when powered up.