Verilog Introduction

Naehyuck Chang
Dept. of EECS/CSE
Seoul National University
naehyuck@snu.ac.kr

Seoul National University

Verilog – Introduction

- Verilog Hardware Description Language (HDL)
  - Behavioral level
  - Register Transfer Level (RTL)
  - Gate level
  - Transistor/Switch level
- Developed in 1983 by Gateway Design Automation Inc.
  - Most popular HDL of the time
  - Traditional computer languages such as C
- IEEE Standard:
  - Verilog 95: IEEE Std. 1364-1995
  - Verilog 2001: IEEE Std. 1364-2001
  - SystemVerilog: IEEE Std 1800-2005

Verilog – Data Types

Name	Description	Example
Value set	0 - Logic zero, false condition 1 - Logic one, true condition x - Unknown logic value z - High impedance, floating state	
Nets	Declared by the predefined word **wire** Represent connections between hardware elements Values continuously driven on them by the outputs of devices	wire sum; wire S1 = 1'b0;
Registers	Declared by the predefined word **reg** Represent data storage elements Retain value until another value is placed onto them	reg Sum_total;
Vectors	Declared by brackets [] Represent multiple bits of net or register	wire [3:0] a = 4'b1010; reg [7:0] total = 8'd12;
Integers	Declared by the predefined word integer	integer no_bits;
Real	Declared by the predefined word real Represent real (floating-point) numbers	real weight;
Parameters	Declared by the predefined word parameter Represent global constants	parameter N=4; parameter M=3;
Arrays	None predefined word Registers and integers can be written as arrays	reg [M:0] b [0:N] integer sum [0:N]

Verilog – Operators

- Bit-select operator
  - []
- Parenthesis
  - ()
- Negations operators
  - ! (logical), ~ (bit-wise)
- Unary arithmetic operators
  - +, - (sign)
- Concatenation
  - {a, b[2:1], c}
- Replication
  - {n{m}} (m n times)
- Binary arithmetic operators
  - *, /, % (mod)
  - +, -
- Shift operators
  - <<. >>
- Relational operators
  - >, <, >=, <=
  - ==, !=
  - ===, !== (including x and z)
- Bitwise logical operators
  - & (AND)
  - ^ (XOR), ^~ or ~^ (XNOR)
  - | (OR)
- Boolean logical operators
  - && (AND)
  - || (OR)
- Conditional operator
  - Cond. Exp. ? True Exp. : False Exp.

Verilog – Two Main Components of Verilog

- Concurrent, event-triggered processes (behavioral)
  - Initial and Always blocks
  - Imperative code that can perform standard data manipulation tasks (assignment, if-then, case)
  - Processes run until they delay for a period of time or wait for a triggering event
- Structure (Plumbing)
  - Verilog program build from modules with I/O interfaces
  - Modules may contain instances of other modules
  - Modules contain local signals, etc.
  - Module configuration is static and all run concurrently

Verilog – Modules and Instances

- Basic structure of a Verilog module:

```
module mymod(output1, output2, … input1, input2);
output output1;
output [3:0] output2;
input input1;
input [2:0] input2;
…
endmodule
```

- Instances of
  - module mymod(y, a, b);

- look like

```
mymod mm1(y1, a1, b1);          // Connect-by-position
mymod mm2(.a(a2), .b(b2), .y(c2)); // Connect-by-name
```

Seoul National University

Verilog – Initial and Always Blocks

- Basic components for behavioral modeling

```
initial
  begin
    … imperative statements …
  end
```

Runs when simulation starts
Terminates when control reaches the end
Good for providing stimulus endmodule

```
always
  begin
    … imperative statements …
  end
```

Runs when simulation starts
Restarts when control reaches the end
Good for modeling/specifying hardware

Verilog – Initial and Always

- Run until they encounter a delay

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

- or a wait for an event

```
always @(posedge clk) q = d;
always begin wait(i); a = 0; wait(~i); a = 1; end
```

Verilog – Procedural Assignment

- Inside an initial or always block:
  - sum = a + b + cin;
- Just like in C: RHS evaluated and assigned to LHS before next statement executes
- RHS may contain wires and regs
  - Two possible sources for data
- LHS must be a reg
  - Primitives or cont. assignment may set wire values

Verilog – Blocking vs. Non-blocking

- Verilog has two types of procedural assignment

- Fundamental problem:
  - In a synchronous system, all flip-flops sample simultaneously
  - In Verilog, always @(posedge clk) blocks run in some undefined sequence

Verilog – A Flawed Shift Register

- This doesn't work as you'd expect:

```
reg d1, d2, d3, d4;

always @(posedge clk) d2 = d1;
always @(posedge clk) d3 = d2;
always @(posedge clk) d4 = d3;
```

- These run in some order, but you don't know which

Verilog – Non-blocking Assignments

- This version does work:

```
reg d1, d2, d3, d4;

always @(posedge clk) d2 <= d1;
always @(posedge clk) d3 <= d2;
always @(posedge clk) d4 <= d3;
```

Non-blocking rule:

RHS evaluated when assignment runs

LHS updated only after all events for the current instant have run

Verilog – Non-blocking Can Behave Oddly

- A sequence of non-blocking assignments don't communicate

```
a = 1;
b = a;
c = b;
```

```
a <= 1;
b <= a;
c <= b;
```

Blocking assignment:

a = b = c = 1

Non-blocking assignment:

a = 1

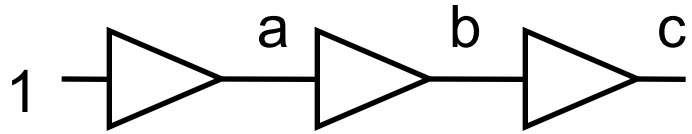
b = old value of a

c = old value of b

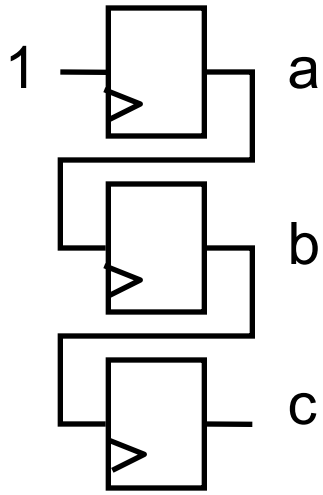
Verilog – Non-blocking Looks Like Latches

- RHS of blocking taken from wires
- RHS of non-blocking taken from latches

```
a = 1;
b = a;
c = b;
```




```
a <= 1;
b <= a;
c <= b;
```



Verilog – IF Statement

if (Boolean Expression)

begin

- statement 1; /*if only one statement, begin and end can be omitted */
- statement 2;
- begin
  - ……
- end

end

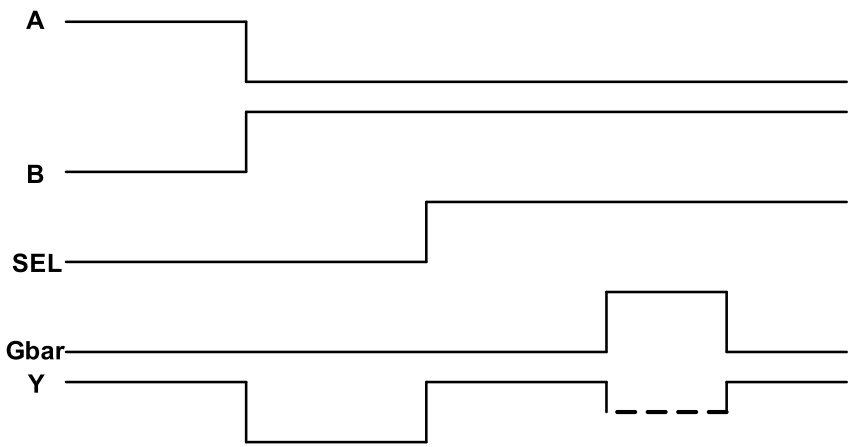
else if (Boolean Expression)

- statement a; /*if only one statement, begin and end can be omitted */

Verilog – IF Statement example

- 2x1 Multiplexer

```
module MUXBH (A, B, SEL, Gbar, Y);
input A, B, SEL, Gbar;
output Y;
reg Y;                          /* since Y is an output and appears inside always,
                                   Y has to be declared as register) */

always @ (SEL, A, B, Gbar)
begin
    if (Gbar == 0 & SEL == 1)
    begin
    Y = B;
    end
    else if (Gbar == 0 & SEL == 0)
    Y = A;
    else
    Y = 1'bz;
    end
endmodule
```



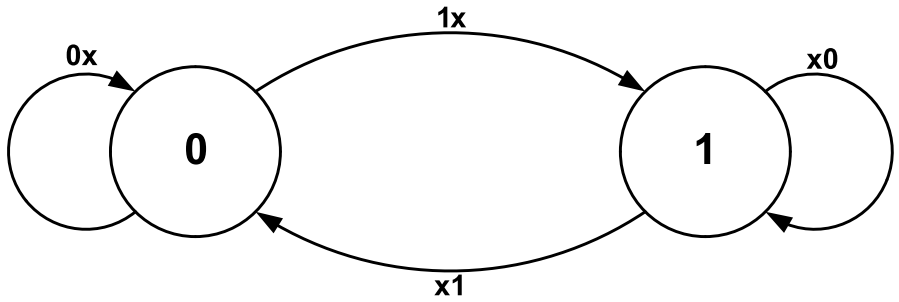
Verilog – CASE Statement

```
case (control expression)
   test value1:
      begin
        statement1;
      end
   test value2:
      …….
   default:
      default statements
endcase
```

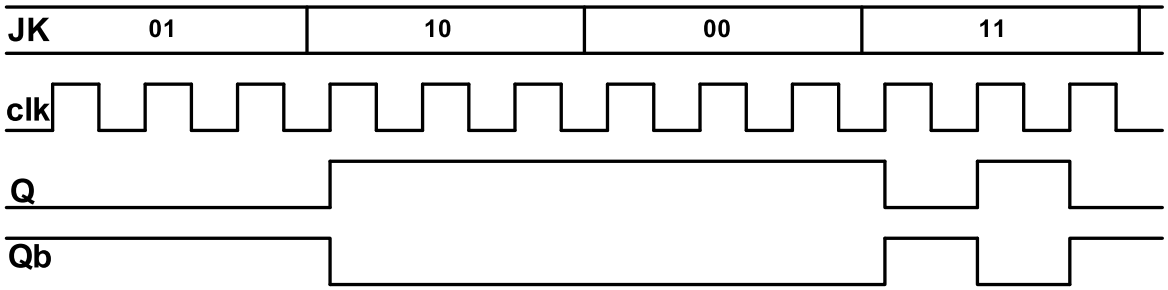
Verilog – CASE Statement example

- Positive Edge-Triggered JK Flip-Flop

```
module JK_FF (JK, clk, q, qb);
input [1:0] JK;
input clk;
output q, qb;
reg q, qb;

always @(posedge clk)
begin
    case (JK)
        2'd0: q = q;
        2'd1: q = 0;
        2'd2: q = 1;
        2'd3: q = ~q;
    endcase
qb = ~q;
end

endmodule
```



State diagram



Simulation waveform of a positive edge -triggered JK flip-flop

Verilog – FSM from a Single Always Block

```
module FSM(o, a, b);
output o;
reg o;
input a, b;
reg [1:0] state;

always @(posedge clk or reset)
  if (reset) state <= 2'b00;
  else case (state)
    2'b00: begin
       state <= a ? 2'b00 : 2'b01;
       o <= a & b;
    end
    2'b01: begin state <= 2'b10; o <= 0; end
 endcase
```

Expresses Moore machine behavior:

Outputs are latched

Inputs only sampled at clock edges

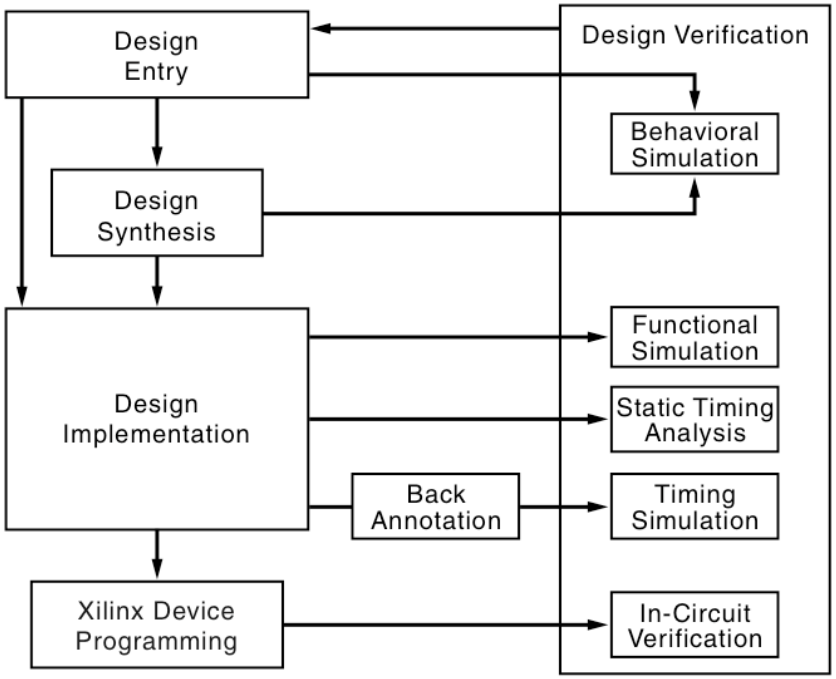
Non-blocking assignments used throughout to ensure coherency.

RHS refers to values calculated in previous clock cycle

Verilog - Design flow

- Verilog can be synthesized using various tools
  - Xilinx ISE (for Xilinx FPGAs)

- Verilog files (.v)
- Netlists (logical)
  - Logical Synthesis
- Physical programming file
  - Place & Route
  - Assign package pins



Synthesizable Verilog Codes

- Behavioral simulation is not enough!
- Verilog codes must be synthesizable
- Do not code Verilog like C

```
module verilog_top(
    input clk,
    input a,
    input b,
    output reg c
     );


    always @(posedge clk or a)
    begin
       c <= clk & a & b;
    end

endmodule
```

```
=========================================================================
*                            HDL Analysis                               *
=========================================================================
Analyzing top module <verilog_top>.
ERROR:Xst:902 - "verilog_top.v" line 29: Unexpected a event in always block sensitivity list.
```

Synthesizable Verilog Codes

- Behavioral simulation is not enough!
- Verilog codes must be synthesizable
- Do not code Verilog like C

```
module verilog_top(
    input clk,
    input a,
    input b,
    output reg c
     );


    always @(posedge clk or negedge a)
    begin
       c <= clk & a & b;
    end

endmodule
```

```
=========================================================================
*                            HDL Analysis                               *
=========================================================================
Analyzing top module <verilog_top>.
ERROR:Xst:902 - "verilog_top.v" line 29: Unexpected a event in always block sensitivity list.
```

Synthesizable Verilog Codes

- Behavioral simulation is not enough!
- Verilog codes must be synthesizable
- Do not code Verilog like C

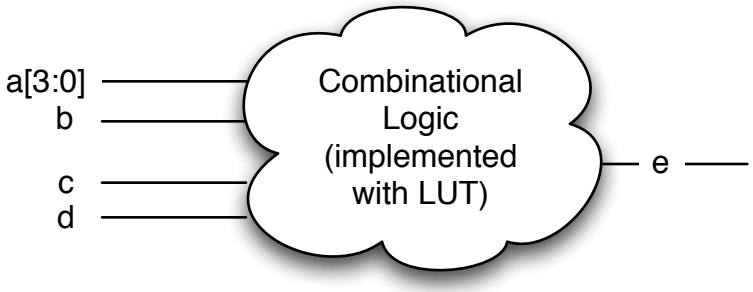
```
module verilog_top(
    input clk,
    input a,
    input b,
    output reg c
     );


    always @(posedge clk or negedge a)
    begin
       c <= clk & a & b;
    end

endmodule
```

```
=====                                                ================================
=========================================================================
*                            HDL Analysis                               *
=========================================================================
Analyzing top module <verilog_top>.
ERROR:Xst:899 - "verilog_top.v" line 31: The logic for <c> does not match a known FF or Latch template
```

Combinational Logic

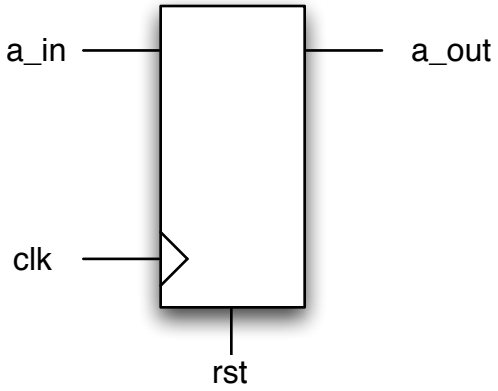
- All inputs must be in the sensitivity list
- Must describe the output for all kinds of inputs
  - Otherwise a latch will be generated!
  - Latches are not preferred in logic designs
  - Be careful when using 'if' statements or 'case' statements

```
always @(a or b or c or d)
begin
    if ( a == 4'b1001 )
        e <= b & c & d;
    else
        e <= 1'b0;
end
```



Sequential Logic

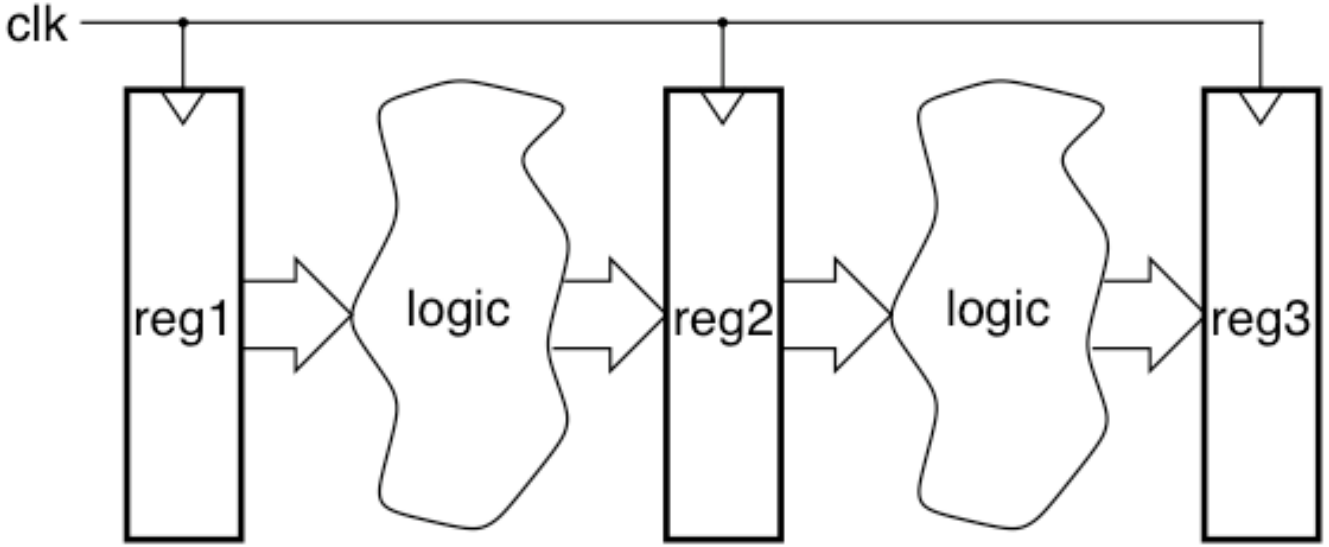
- 'posedge' or 'negedge' must be included in the sensitivity list
- FF with asynchronous reset
- FF with synchronous reset




```
always @(posedge clk or negedge rst)
begin
    if (~rst)
        a_out <= 0;
    else
        a_out <= a_in;
end

always @(posedge clk)
begin
    if (~rst)
        a_out <= 0;
    else
        a_out <= a_in;
end
```

Synthesizable Verilog code

- Always think about the hardware architecture first and then describe it in Verilog
- RTL (Register Transfer Level) Coding

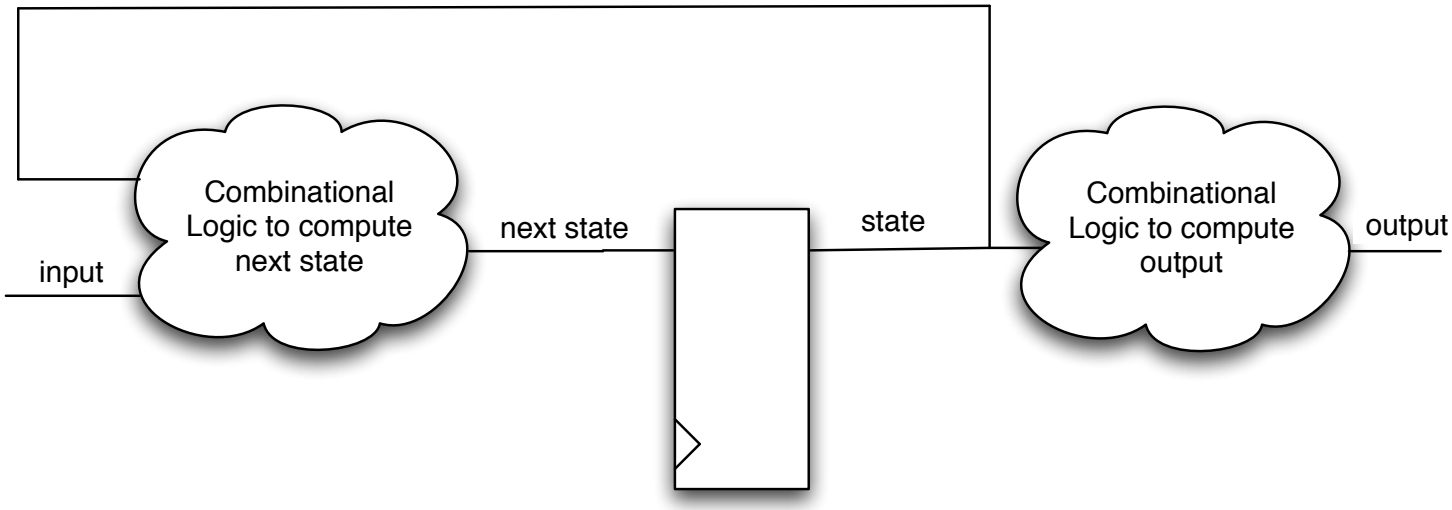


Modeling FSMs Behaviorally

- There are many ways to do it
- Define the next-state logic combinationally and define the state-holding latches explicitly

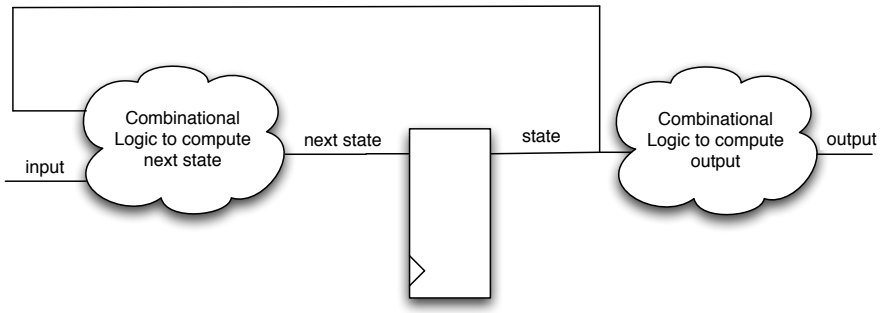
Finite State Machines

- Moore Machine



Finite State Machines

- Moore Machine
  - 3 always blocks




```
always @(posedge clk or negedge rst)
begin
    if ( ~rst)
        state <= 0;
    else
        state <= next_state;
end

always @( state or /*inputs*/ )
begin
    ...
    next_state <= ..
end

always @( state )
begin
    ...
    /*outputs*/ <= ..
end
```

Finite State Machines

```
module FSM(o, a, b, reset);
output o;
reg o;
input a, b, reset;
reg [1:0] state, nextState;

always @(a or b or state)
 case (state)
    2'b00: begin
       nextState = a ? 2'b00 : 2'b01;
       o = a & b;
    end
    2'b01: begin nextState = 2'b10; o = 0; end
 endcase

 ......
```

Output o is declared a reg because it is assigned procedurally, not because it holds state

Combinational block must be sensitive to any change on any of its inputs

(Implies state-holding elements otherwise)

Finite State Machines

```
always @(a or b or state)
 case (state)
    2'b00: begin
       nextState = a ? 2'b00 : 2'b01;
       o = a & b;
    end
    2'b01: begin nextState = 2'b10; o = 0; end
 endcase

always @(posedge clk or reset)
  if (reset)
    state <= 2'b00;
  else
    state <= nextState;
```

This is a Mealy machine because the output is directly affected by any change on the input

Latch implied by sensitivity to the clock or reset only

Synthesis Result

Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	4,262	138,240	3%	
Number used as Flip Flops	4,238			
Number used as Latches	24			
Number of Slice LUTs	8,032	138,240	5%	
Number used as logic	7,664	138,240	5%	
Number using O6 output only	7,250			
Number using O5 output only	177			
Number using O5 and O6	237			
Number used as Memory	336	36,480	1%	
Number used as Dual Port RAM	133			

Synthesis Result

```
Synthesizing Unit <NOR_FSM>.
    Related source file is "NOR_ctrl/NOR_FSM.v".
WARNING:Xst:646 - Signal <MEM_MASK> is assigned but never used. This unconnected sig
    Found finite state machine <FSM_6> for signal <state>.
    -----------------------------------------------------------------------
    | States             | 5                                              |
    | Transitions        | 15                                             |
    | Inputs             | 8                                              |
    | Outputs            | 5                                              |
    | Clock              | HCLK (rising_edge)                             |
    | Reset              | HRESETn (negative)                             |
    | Reset type         | synchronous                                    |
    | Reset State        | 00000000000                                    |
    | Power Up State     | 00000000000                                    |
    | Encoding           | automatic                                      |
    | Implementation     | LUT                                            |
    -----------------------------------------------------------------------
    Found 1-bit register for signal <ready>.
    Found 32-bit register for signal <CFIFO_ADDR_1>.
    Found 32-bit adder for signal <CFIFO_ADDR_1$share0000> created at line 125.
    Found 5-bit register for signal <CFIFO_LEN_1>.
    Found 5-bit subtractor for signal <CFIFO_LEN_1$share0000> created at line 125.
    Found 11-bit up counter for signal <cnt>.
    Found 4-bit register for signal <DELAY_CNT>.
    Found 4-bit subtractor for signal <DELAY_CNT$addsub0000>.
    Summary:
        inferred   1 Finite State Machine(s).
        inferred   1 Counter(s).
        inferred  42 D-type flip-flop(s).
        inferred   3 Adder/Subtractor(s).
Unit <NOR_FSM> synthesized.
```

Synthesis Result

```
Timing Summary:
---------------
Speed Grade: -2

   Minimum period: 11.888ns (Maximum Frequency: 84.117MHz)
   Minimum input arrival time before clock: 10.698ns
   Maximum output required time after clock: 3.921ns
   Maximum combinational path delay: 3.246ns
```

RTL coding

- Can you implement it in Verilog?