Verilog Synthesis

Franco Fummi
Alessia Bozzini

UNIVERSITÀ di VERONA
Dipartimento di INFORMATICA

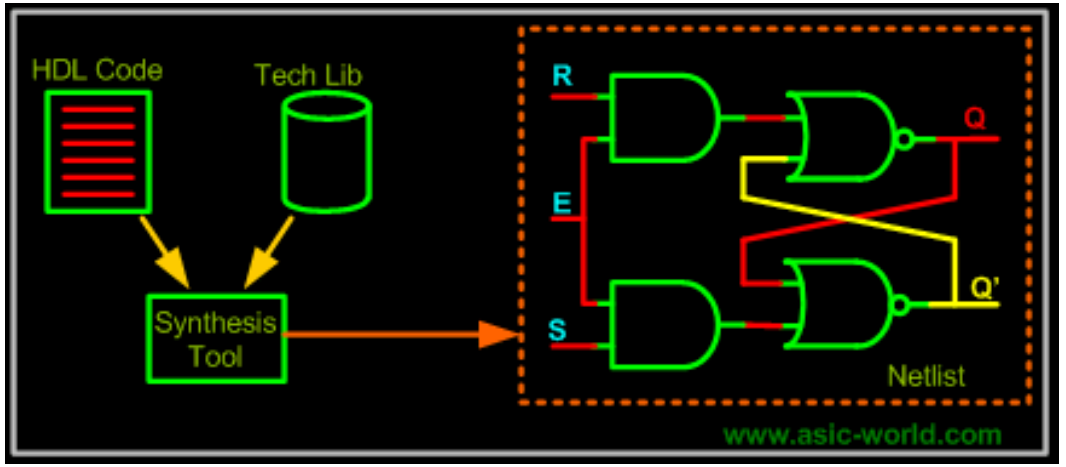
Version 1.1

Contents

Synthesis Process

- Logic synthesis is the process of converting a high-level description of design into an optimized gate-level representation
- Logic synthesis uses a standard cell library which have simple cells, such as basic logic gates like and, or, and nor, or macro cells, such as adder, muxes, memory, and flip-flops
- Standard cells put together are called technology library. Normally the technology library is known by the transistor size (0.18u, 90nm)
- The designer should first understand the architectural description. Then he should consider design constraints such as
  - Timing
  - Area
  - Testability
  - Power

Usual importance order:
1) Area → small circuit
2) Timing → Reduce the delay
3) Power
4) Testability



Synthesis Process

- High-level design is done without significant concern about design constraints
- Conversion from high-level design to gates is done by synthesis tools, using various algorithms to optimize the design as a whole. This *removes the problem with varied designer styles* for the different blocks in the design and suboptimal designs
- Logic synthesis tools allow technology independent design
- Design reuse is possible for technology-independent descriptions

Constructs Not Supported in Synthesis

Construct Type	Notes
`initial`	Used only in test benches.
events	Events make more sense for syncing test bench components.
`real`	Real data type not supported.
`time`	Time data type not supported.
`force` and `release`	Force and release of data types not supported.
assign and deassign	assign and deassign of reg data types is not supported. But assign on wire data type is supported.
`fork join`	Use nonblocking assignments to get same effect.
primitives	Only gate level primitives are supported.
table	UDP and tables are not supported.

Any code that contains the above constructs are not synthesizable, but within synthesizable constructs, bad coding could cause synthesis issues.

Constructs Not Supported in Synthesis

- Delay
  - Example: `a = #10 b;`
  - This code is useful only for simulation purpose
  - Synthesis tool normally ignores such constructs, and just assumes that there is no `#10` in above statement, thus treating above code as

```
a = b;
```

Constructs Not Supported in Synthesis

- Comparison to X and Z are always ignored
  - Normally, there is a tendency to compare variables with X and Z, but in practice it is the worst thing to do, so please avoid comparing with X and Z

➥ Limit design to two states, 0 and 1. Use tri-state only at chip IO pads level

```
module synthesis_compare_xz (a,b);
output a;
input b;
reg a;

always @ (b)
begin
   if ((b == 1'bz) || (b == 1'bx)) begin
      a = 1;
   end else begin
      a = 0;
   end
end

endmodule
```

Constructs Supported in Synthesis

Construct Type	Keyword or Description	Notes
ports	input, inout, output	Use inout only at IO level.
parameters	parameter	This makes design more generic
module definition	module	
signals and variables	wire, reg, tri	Vectors are allowed
instantiation	module instances / primitive gate instances	E.g.- nand (out,a,b), bad idea to code RTL this way.
function and tasks	function , task	Timing constructs ignored
procedural	always, if, else, case, casex, casez	initial is not supported
procedural blocks	begin, end, named blocks, disable	Disabling of named blocks allowed
data flow	assign	Delay information is ignored
named Blocks	disable	Disabling of named block supported.
loops	for, while, forever	While and forever loops must contain @(posedge clk) or @(negedge clk)

4

Operators and their Effect

Operator Type	Operator Symbol	Operation Performed
Arithmetic	*	Multiply
	/	Division
	+	Add
	-	Subtract
	%	Modulus
	+	Unary plus
	-	Unary minus
Logical	!	Logical negation
	&&	Logical AND
	||	Logical OR
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal
	<=	Less than or equal

Operator Type	Operator Symbol	Operation Performed
Equality	==	Equality
	!=	inequality
Reduction	&	Bitwise AND
	~&	Bitwise NAND
	|	Bitwise OR
	~|	Bitwise NOR
	^	Bitwise XOR
	^~ ~^	Bitwise XNOR
Shift	>>	Right shift
	<<	Left shift
Concatenation	{ }	Concatenation
Conditional	?	conditional

Logic Circuit Modeling

- There could be only two types of digital circuits:
  1. combinational circuits
  2. sequential circuits

Logic Circuit Modeling: Combinational circuits

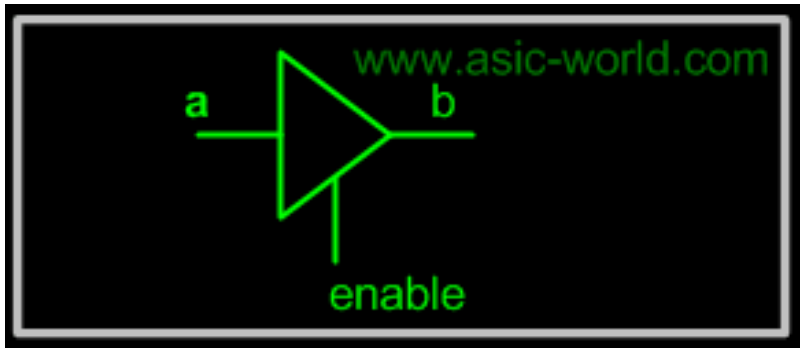
- Combinational circuits modeling in Verilog can be done using assign and always blocks
- Writing simple combinational circuits using assign statements is very straightforward
- Example

```
assign y = (a&b) | (c^d);
```

Logic Circuit Modeling: Combinational circuits

- Tri-state buffer
  - Example

```
module tri_buf (a,b,enable);
    input a;
    output b;
    input enable;
    wire b;

    assign b = (enable) ? a : 1'bz;

 endmodule
```

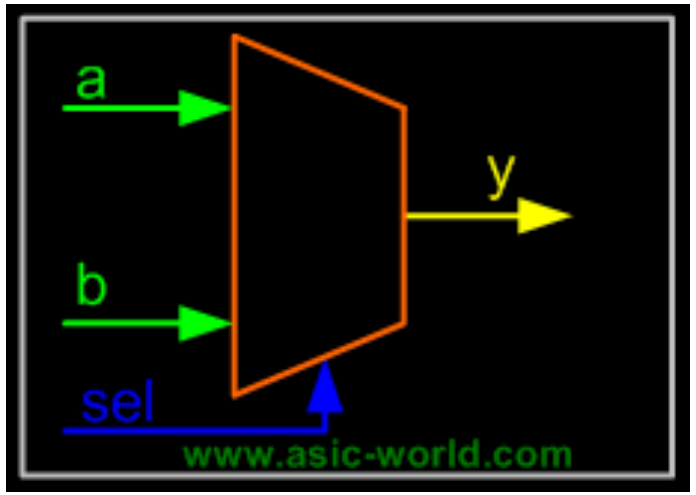


Logic Circuit Modeling: Combinational circuits

- Mux
  - Example

1

```
module mux_21 (a,b,sel,y);
    input a, b;
    output y;
    input sel;
    wire y;

    assign y = (sel) ? b : a;

endmodule
```



2

```
reg out;
    always @(a or b or sel)
        begin
            case(sel)
                1'b0: out = b;
                1'b1: out = a;
        endcase
    end
```

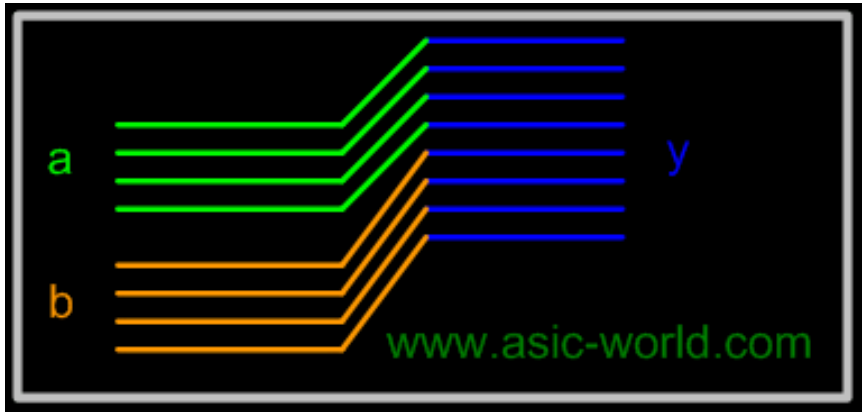
3

```
reg out;
    always @(a or b or sel)
        if (sel)
            out = a;
        else
            out = b;
```

Logic Circuit Modeling: Combinational circuits

- Simple Concatenation
  - Example

```
module bus_con (a,b);
   input [3:0] a, b;
   output [7:0] y;
   wire [7:0] y;

   assign y = {a,b};

endmodule
```



Logic Circuit Modeling: Combinational circuits

- 1 bit adder with carry
  - Example

```
module addbit (
   a ,   // first input
   b ,   // Second input
   ci ,  // Carry input
   sum,  // sum output
   co    // carry output
   );
   //Input declaration
   input a;
   input b;
   input ci;

   //Ouput declaration
   output sum;
   output co;

   //Port Data types
   wire a;
   wire b;
   wire ci;
   wire sum;
   wire co;

   //Code starts here
   assign {co,sum} = a + b + ci;

   endmodule // End of Module addbit
```

Logic Circuit Modeling: Combinational circuits

- Multiply by 2
  - Example

```
module muliply (a,product);
   input [3:0] a;
   output [4:0] product;
   wire [4:0] product;

   assign product = a << 1;

endmodule
```

- 3 is to 8 decoder
  - Example

```
module decoder (in,out);
   input [2:0] in;
   output [7:0] out;
   wire [7:0] out;

   assign out = (in == 3'b000 ) ? 8'b0000_0001 :
   (in == 3'b001 ) ? 8'b0000_0010 :
   (in == 3'b010 ) ? 8'b0000_0100 :
   (in == 3'b011 ) ? 8'b0000_1000 :
   (in == 3'b100 ) ? 8'b0001_0000 :
   (in == 3'b101 ) ? 8'b0010_0000 :
   (in == 3'b110 ) ? 8'b0100_0000 :
   (in == 3'b111 ) ? 8'b1000_0000 : 8'h00;

endmodule
```

Logic Circuit Modeling: Sequential circuits

- Sequential logic can be modeled only using always blocks
- Non-blocking assignments for sequential circuits are normally used

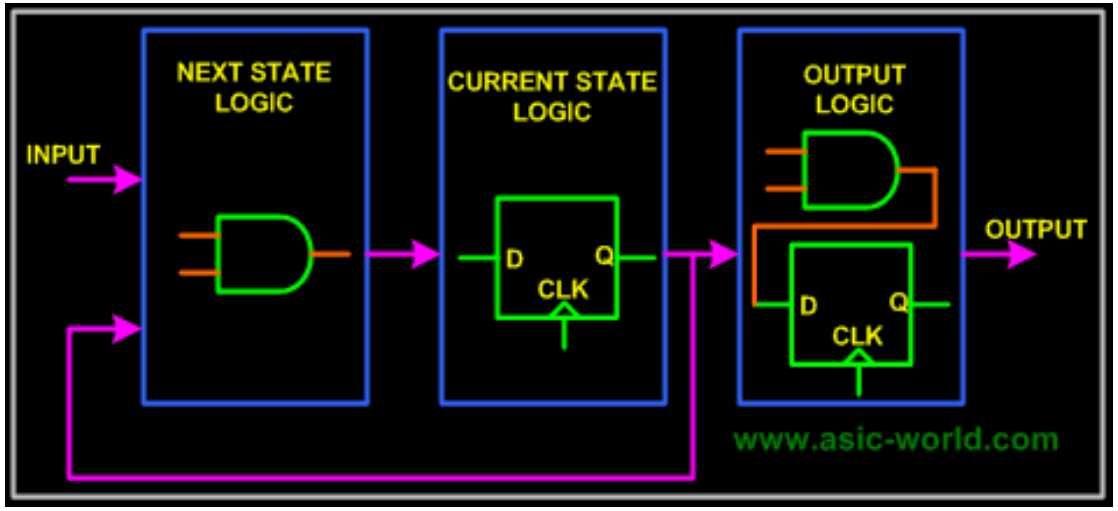
Logic Circuit Modeling: Sequential circuits

- Simple Flip-Flop
  - Example

```
module flif_flop (clk,reset, q, d);
   input clk, reset, d;
   output q;
   reg q;

   always @ (posedge clk )
   begin
      if (reset == 1) begin
         q <= 0;
      end else begin
         q <= d;
      end
   end

endmodule
```

FSMs in Verilog

- Basically a FSM consists of
  - Combinational logic
    - used to decide the next state of the FSM
  - Sequential logic
    - used to store the current state of the FSM
  - Output logic
    - a mixture of both combinational and sequential logic

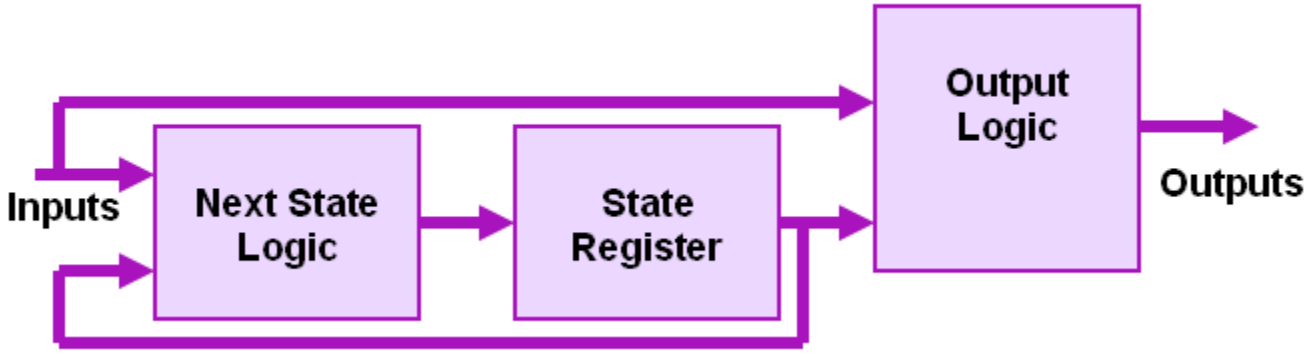


FSMs in Verilog

- There are two types of state machines:

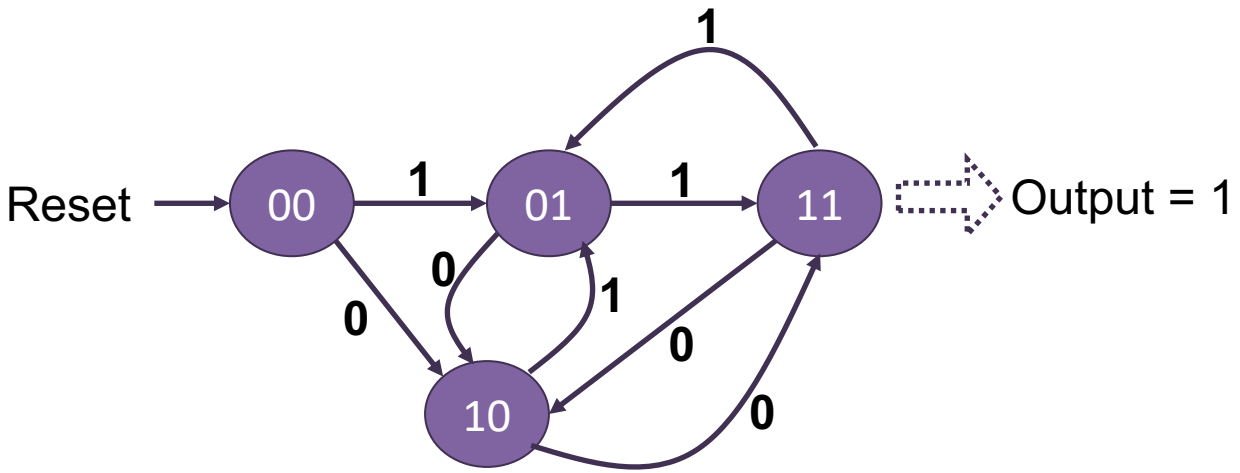
1. **Moore State Machine** : Its output depends on current state only

2. **Mealy State Machine** : Its output depends on current state and current inputs

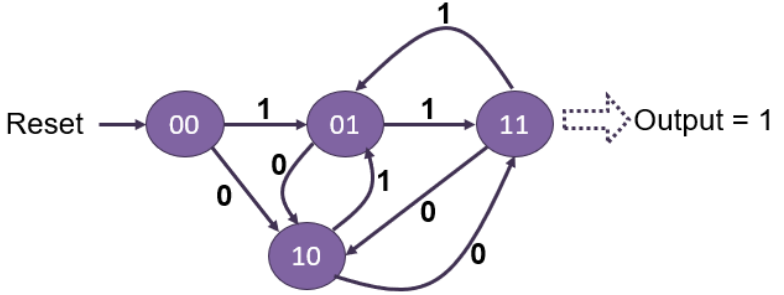


FSMs in Verilog

- Example of Moore State Machine:
  - Consider the case of a circuit to detect a pair of 1's or 0's in the single bit input. That is, input will be a series of one's and zero's
  - If two one's or two zero's comes one after another, output should go high. Otherwise output should be low



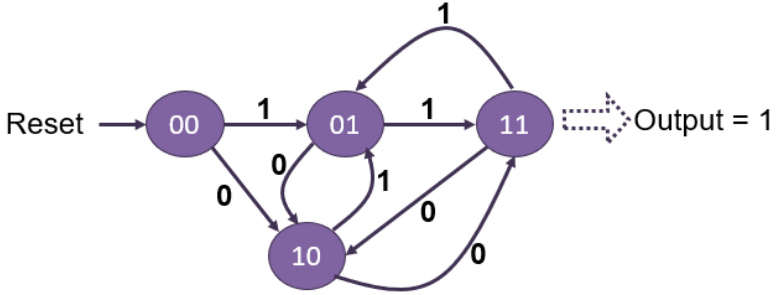
FSMs in Verilog



- Example of Moore State Machine:

```
module fsm( clk, rst, inp, outp);
   input clk, rst, inp;
   output outp;
   reg [1:0] state;
   reg outp;
   always @( posedge clk, posedge rst )

   begin
   if( rst )
      state <= 2'b00;
   else
   begin
      case( state )
      2'b00:
      begin
          if( inp ) state <= 2'b01;
          else state <= 2'b10;
      end
      2'b01:
      begin
         if( inp ) state <= 2'b11;
         else state <= 2'b10;
      end
      2'b10:
      begin
         if( inp ) state <= 2'b01;
         else state <= 2'b11;
      end
      2'b11:
      begin
         if( inp ) state <= 2'b01;
         else state <= 2'b10;
      end
      endcase
   end
end

always @(posedge clk, posedge rst)
begin
   if( rst )
      outp <= 0;
   else if( state == 2'b11 )
      outp <= 1;
   else outp <= 0;
end
endmodule
```

FSMs in Verilog

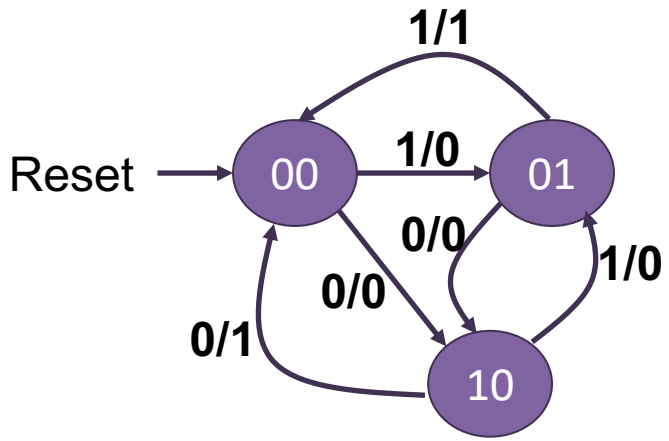


- Example of Moore State Machine:
  - Testbench

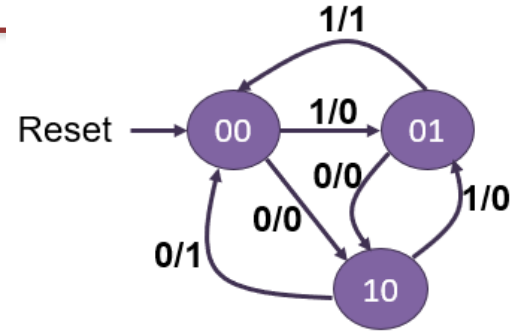
```
module fsm_test;

reg clk, rst, inp;
wire outp;
reg[15:0] sequence;
integer i;

fsm dut( clk, rst, inp, outp);

initial
begin

   clk = 0;
      rst = 1;
      sequence = 16'b0101_0111_0111_0010;
   #5 rst = 0;
   for( i = 0; i <= 15; i = i + 1)
   begin
      inp = sequence[i];
      #2 clk = 1;
      #2 clk = 0;
      $display("State = ", dut.state, " Input = ",
inp, ", Output = ", outp);

   end
      test2;
end

task test2;
   for( i = 0; i <= 15; i = i + 1)
   begin
      inp = $random % 2;
      #2 clk = 1;
      #2 clk = 0;
      $display("State = ", dut.state, " Input = ",
inp, ", Output = ", outp);

   end
endtask
endmodule
```

FSMs in Verilog

- Example of Mealy State Machine:
  - Consider the case of a circuit to detect a pair of 1's or 0's in the single bit input. That is, input will be a series of one's and zero's
  - When reset, state becomes idle, that is 00. Next, if 1 comes, state becomes 01 and if 0 comes state becomes 10 with output 0. If input bit repeats, output becomes 1 and state goes to 00



FSMs in Verilog



- Example of Mealy State Machine:

```
module mealy( clk, rst, inp, outp);
   input clk, rst, inp;
   output outp;
   reg [1:0] state;
   reg outp;

   always @( posedge clk, posedge rst ) begin
   if( rst ) begin
      state <= 2'b00;
      outp <= 0;
   end
   else begin
      case( state )
      2'b00: begin
         if( inp ) begin
            state <= 2'b01;
            outp <= 0;
         end
         else begin
            state <= 2'b10;
            outp <= 0;
         end
      end
```

```
      2'b01: begin
         if( inp ) begin
            state <= 2'b00;
            outp <= 1;
         end
         else begin
            state <= 2'b10;
            outp <= 0;
         end
      end

      2'b10: begin
         if( inp ) begin
            state <= 2'b01;
            outp <= 0;
         end
         else begin
            state <= 2'b00;
            outp <= 1;
         end
      end
```

```
      default: begin
         state <= 2'b00;
         outp <= 0;
      end
      endcase
   end
end
endmodule
```

FSMs in Verilog

- Difference between Moore and Mealy state machines
  - Moore state machine is easier to design than Mealy
    - First design the states depending on the previous state and input. Then design output only depending on state
    - Whereas in Mealy, the state and input must be considered while designing the output
  - Mealy state machine uses less states than the Moore
    - Since inputs influence the output in the immediate clock, memory needed to remember the input is less. So, it uses less flip flops and hence circuit is simpler
  - Mealy is faster than Moore
    - Mealy gives immediate response to input and Moore gives response in the next clock