


Verilog Tutorial

Abdul-Rahman Elshafei
COE-561



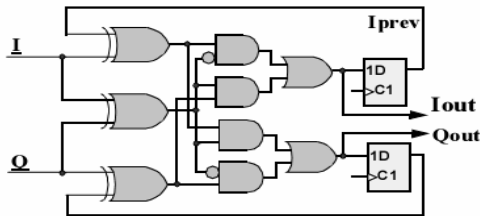
Introduction

Purpose of HDL:

1. Describe the circuit in algorithmic level (like c) and in gate-level (e.g. And gate)
2. Simulation
3. Synthesis
4. Words are better than pictures

Nov 16, 2006 Abdul-Rahman Elshafei 2

The best way to describe a circuit?



$$I_{out} = \overline{(I \oplus Q)}(I \oplus I_{prev}) + (I \oplus Q)(Q \oplus Q_{prev})$$

$$Q_{out} = \overline{(I \oplus Q)}(Q \oplus Q_{prev}) + (I \oplus Q)(I \oplus I_{prev})$$

(I,Q) prev		Iout, Qout			
		00	01	11	10
00	00	00	01	11	10
01	10	10	00	01	11
11	11	11	10	00	01
10	01	01	11	10	00

If both inputs are 1, change both outputs.
If one input is 1 change an output as follows:

If the previous outputs are equal change the output with input 0;
If the previous outputs are unequal change the output with input 1.

If both inputs are 0, change nothing.

Rahman Elshafei

3

Lexicography

■ Comments:

Two Types:

- *// Comment*
- */* These comments extend over multiple lines. Good for commenting out code */*

■ Character Set:

0123456789ABCD..YZabcd...yz_ \$

Cannot start with a number or \$

Nov 16, 2006

Abdul-Rahman Elshafei

4

Data Types

■ Data Values:

0,1,x,z

■ Wire

- Synthesizes into wires
- Used in structural code

■ Reg

- May synthesize into latches, flip-flops or wires
- Used in procedural code

■ Integer

32-bit integer used as indexes

■ Input, Output, inout

Defines ports of a module (wire by default)

```
module sample (a,b,c,d);
```

```
input a,b;  
output c,d;
```

```
wire [7:0] b;
```

```
reg c,d;
```

```
integer k;
```

Data Values

■ Numbers:

Numbers are defined by number
of bits

Value of 23:

5'b10111

5'd23

5'h17

■ Constants:

```
wire [3:0] t,d;
```

```
assign t = 23;
```

```
assign d= 4'b0111;
```

■ Parameters:

```
parameter n=4;
```

```
wire [n-1:0] t, d;
```

```
`define Reset_state = 0, state_B =1,  
Run_state =2, finish_state = 3;
```

```
if(state==`Run_state)
```

Operators

■ Arithmetic:

*,+,-,/,%

■ Relational

<,<=,>,>=,==,!=

■ Bit-wise Operators

- Not: ~
- XOR: ^
- And: & 5'b11001 & 5'b01101 ==> 5'b01001
- OR: |
- XNOR: ~^ or ^~

■ Logical Operators

Returns 1or 0, treats all nonzero as 1

- ! : Not
- && : AND 27 && -3 ==> 1
- || : OR

```
reg [3:0] a, b, c, d;
wire[7:0] x,y,z;
parameter n =4;
```

```
c = a + b;
d = a *n;
```

```
if(x==y) d = 1; else d =0;
```

```
d = a ~^ b;
```

```
if ((x>=y) && (z)) a=1;
else a = !x;
```

Nov 16, 2006

Abdul-Rahman Elshafei

7

Operators

■ Reduction Operators:

Unary operations returns single-bit values

- & : and
- | :or
- ~& : nand
- ~| : nor
- ^ : xor
- ~^ :xnor

■ Shift Operators

Shift Left: <<

Shift right: >>

■ Concatenation Operator

{ } (concatenation)

{ n{item} } (n fold replication of an item)

■ Conditional Operator

Implements if-then-else statement

(cond) ? (result if cond true) : (result if cond false)

```
module sample (a, b, c, d);
input [2:0] a, b;
output [2:0] c, d;
wire z,y;
```

```
assign z = ~| a;
c = a * b;
if(a==b) d = 1; else d =0;
```

```
d = a ~^ b;
```

```
if ((a>=b) && (z)) y=1;
else y = !x;
```

```
assign d << 2; //shift left twice
assign {carry, d} = a + b;
assign c = {2{carry},2{1'b0}};
// c = {carry,carry,0,0}
```

```
assign c= (inc==2)? a+1:a-1;
```

Nov 16, 2006

Abdul-Rahman Elshafei

8

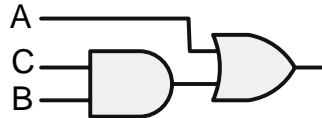
Verilog Structure

- All code are contained in modules
- Can't invoke other modules

- Modules cannot be contained in another module

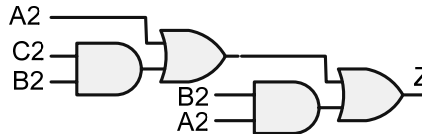
```

module gate(Z,A,B,C);
    output Z;
endmodule
    
```



```

module two_gates(Z2,A2,B2,C2);
    output Z2;
    gate gate_1(G2,A2,B2,C2);
    gate gate_2(Z2,G2,A2,B2);
endmodule
    
```



Structural Vs Procedural

Structural

- textual description of circuit
- order does not matter
- Starts with **assign** statements
- Harder to code
- Need to work out logic

```

wire c, d;
assign c = a & b;
assign d = c | b;
    
```

Procedural

- Think like C code
- Order of statements are important
- Starts with **initial** or **always** statement
- Easy to code
- Can use case, if, for

```

reg c, d;
always@ (a or b or c) begin
    assign c = a & b;
    assign d = c | b; end
    
```

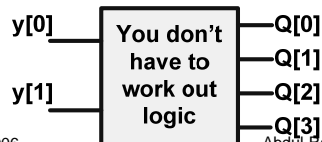
Structural Vs Procedural

Procedural

```

reg [3:0] Q;
wire [1:0] y;
always@(y)
begin
  Q=4'b0000;
  case(y) begin
    2'b00: Q[0]=1;
    2'b01: Q[1]=1;
    2'b10: Q[2]=1;
    2'b11: Q[3]=1;
  endcase
end

```

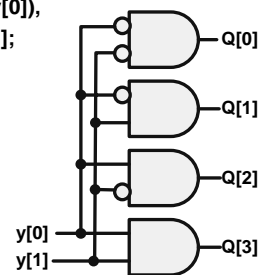


Structural

```

wire [3:0]Q;
wire [1:0]y;
assign
  Q[0]=(~y[1])&(~y[0]),
  Q[1]=(~y[1])&y[0],
  Q[2]=y[1]&(~y[0]),
  Q[3]=y[1]&y[0];

```



Blocking Vs Non-Blocking

Blocking

- <variable> = <statement>
- Similar to C code
- The next assignment waits until the present one is finished
- Used for combinational logic

Non-blocking

- <variable> <=> <statement>
- The inputs are stored once the procedure is triggered
- Statements are executed in parallel
- Used for flip-flops, latches and registers



Do not mix both assignments in one procedure



Blocking Vs Non-Blocking

Initial

begin

#1 e=2;

#1 b=1;

#1 b<=0;

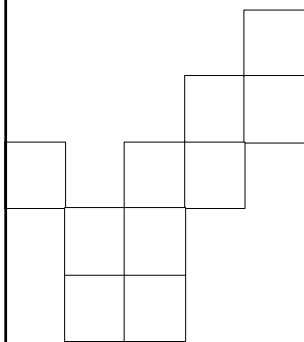
e<=b; // grabbed the old b

f=e; // used old e=2, did not wait e<=b

Nov 16, 2006

Abdul-Rahman Elshafei

13



Behavior Modeling

If Statements

Syntax

```
if (expression)
begin
...statements...
end
```

```
else if (expression)
begin
...statements...
end
...more else if blocks
```

```
else
begin
...statements...
end
```

```
if (alu_func == 2'b00)
aluout = a + b;
else if (alu_func == 2'b01)
aluout = a - b;
else if (alu_func == 2'b10)
aluout = a & b;
else // alu_func == 2'b11
aluout = a | b;
```

Case Statements

Syntax

```
case (expression)
case_choice1:
begin
...statements...
end
```

```
case_choice2:
begin
...statements...
end
```

...more case choices blocks...

```
default:
begin
...statements...
end
endcase
```

```
case (alu_ctr)
2'b00: aluout = a + b;
2'b01: aluout = a - b;
2'b10: aluout = a & b;
default: aluout = 1'bx; // Treated as don't cares for
endcase // minimum logic generation.
```


For loops

Syntax

```
for (count= value1;  
    count</<=>/>= value2;  
    count=count+/- step)  
begin  
    ...statements...  
end
```

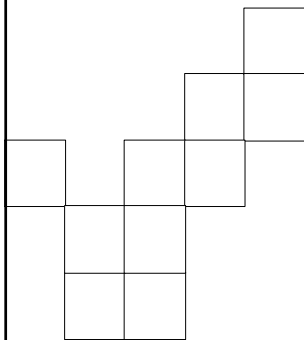
```
integer j;
```

```
for(j=0;j<=7;j=j+1)  
begin  
    c[j] = a[j] + b[j];  
end
```

Nov 16, 2006

Abdul-Rahman Elshafei

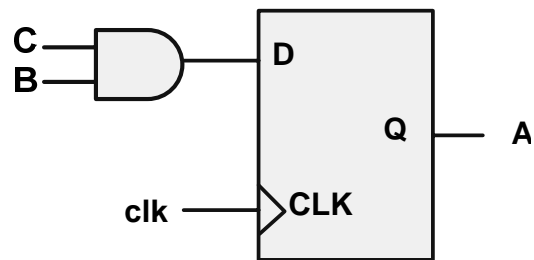
17



Component Inference

Flip-Flops

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



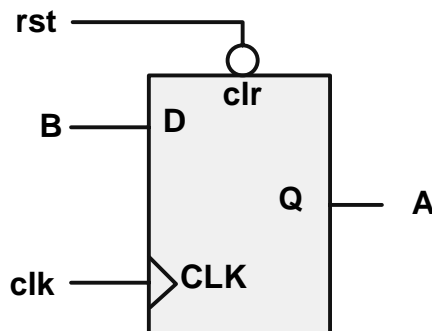
Nov 16, 2006

Abdul-Rahman Elshafei

19

D Flip-Flop with Asynchronous Reset

```
always@(posedge clk or
  negedge rst)
begin
  if (!rst) a<=0;
  else a<=b;
end
```



Nov 16, 2006

Abdul-Rahman Elshafei

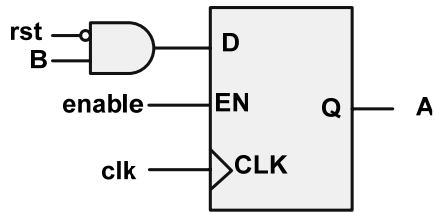
20

D Flip-flop with Synchronous reset and Enable

```

always@(posedge clk)
begin
  if (rst) a<=0;
  else if (enable) a<=b;
end

```



Nov 16, 2006

Abdul-Rahman Elshafei

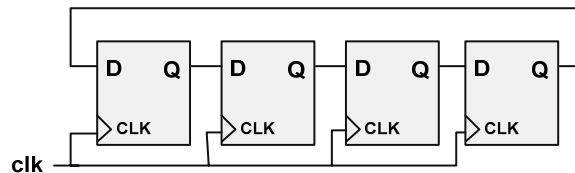
21

Shift Registers

```

reg[3:0] Q;
always@(posedge clk or
posedge rset )
begin
  if (rset) Q<=0;
  else begin
    Q <=Q << 1;
    Q[0]<=Q[3];
  end
end

```



Nov 16, 2006

Abdul-Rahman Elshafei

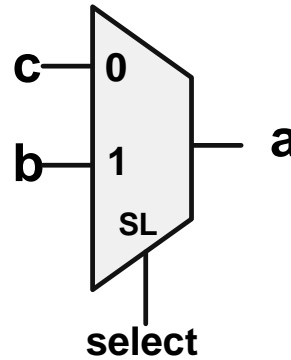
22

Multiplexers

Method 1
assign a = (select ? b : c);

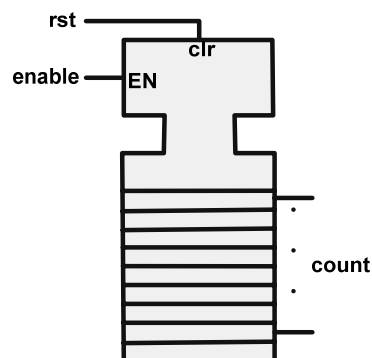
Method 2
always@(select or b or c) begin
 if(select) a=b;
 else a=c;
end

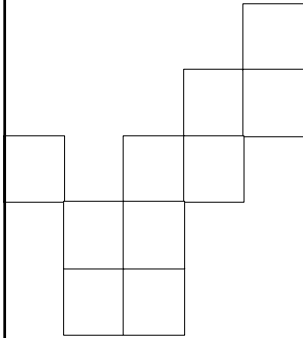
Method 2b
case(select)
 1'b1: a=b;
 1'b0: a=c;
endcase



Counters

```
reg [7:0] count;  
wire enable;  
always@(posedge clk or  
  negedge rst)  
begin  
  if (rst) count<=0;  
  else if (enable)  
    count<=count+1;  
end
```





Avoiding Unwanted Latches

Latches are BAD

Rule #1



**If the procedure has several paths,
every path must evaluate all outputs**

■ **Method1:**

Set all outputs to some value at the start of the procedure.
Later on different values can overwrite those values.

```
always @(...)
begin
  x=0;y=0;z=0;
  if (a) x=2; elseif (b) y=3; else z=4;
End
```

■ **Method2:**

Be sure every branch of every if and case generate every output

```
always @(...)
begin
  if (a) begin    x=2; y=0; z=0; end
  elseif (b) begin x=0; y=3; z=0; end
  else begin    x=0; y=0; z=4; end
end
```

Nov 16, 2006

Abdul-Rahman Elshafei

26

Rule #2



All inputs used in the procedure must appear in the trigger list

- **Right-hand side variables:**

Except variables both calculated and used in the procedure.

always @(a or b or c or ~~x or y~~)

begin

 x=a; y=b; z=c;

 w=x+y;

end

- **Branch controlling variables:**

Be sure every branch of every if and case generate every output

always @(a or b)

begin

 if (a) begin x=2; y=0; z=0; end

 elseif (b) begin x=0; y=3; z=0; end

 else begin x=0; y=0; z=4; end

end

Nov 16, 2006

Abdul-Rahman Elshafei

27

Rule #3



All possible inputs used control statements must be covered

- End all case statements with the default case whether you need it or not.

case(state)

...

default: next_state = reset;

endcase

- Do not forget the self loops in your state graph

if(a|b&c) next_state=S1;

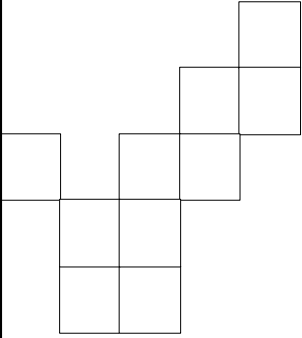
elseif(c&d) next_state=S2;

else next_state=reset;


Nov 16, 2006

Abdul-Rahman Elshafei

28



Finite State Machines



Standard Form for a Verilog FSM

```
// state flip-flops
reg [2:0] state, nxt_st;
// state definitions
parameter
  reset=0,S1=1,S2=2,S3=3,..

// REGISTER DEFINITION
always@(posedge clk)
begin
  state<=next_state;
end

// NEXT STATE CALCULATIONS
always@(state or inputs or ...)
begin
  ...
  next_state= ...
  ...
end

// OUTPUT CALCULATIONS
output= f(state, inputs)
```

Nov 16, 2006 Abdul-Rahman Elshafei 30

Example

```
module myFSM (clk, x, z)
input clk, x; output z;

// state flip-flops
reg [2:0] state, nxt_st;
// state definition
parameter S0=0,S1=1,S2=2,S3=3,S7=7

// REGISTER DEFINITION
always @(posedge clk)
begin
state<=nxt_st;
end

// OUTPUTCALCULATIONS
assign z = (state==S7);

// NEXT STATE CALCULATIONS
always @(state or x)
begin
case (state)
S0: if(x) nxt_st=S1;
else nxt_st=S0;
S1: if(x) nxt_st=S3;
else nxt_st=S2;
S2: if(x) nxt_st=S0;
else nxt_st=S7;
S3: if(x) nxt_st=S2;
else nxt_st=S7;
S7: nxt_st=S0;
default: nxt_st = S0;
endcase
end

endmodule
```

Nov 16, 2006

Abdul-Rahman Elshafei

31

Test Benches



System tasks

- Used to generate input and output during simulation. Start with **\$** sign.
- Display Selected Variables:
\$display ("format_string",par_1,par_2,...);
\$monitor("format_string",par_1,par_2,...);
Example: `$display("Output z: %b", z);`
- Writing to a File:
`$fopen`, `$fdisplay`, `$fmonitor` and `$fwrite`
- Random number generator: **\$random** (*seed*)
- Query current simulation time: `$time`

Test Benches

Overview

1. Invoke the verilog under design
2. Simulate input vectors
3. Implement the system tasks to view the results

Approach

1. Initialize all inputs
2. Set the clk signal
3. Send test vectors
4. Specify when to end the simulation.

Example

```
'timescale1 ns /100 ps
// timeunit =1ns; precision=1/10ns;
module my_fsm_tb;
reg clk, rst, x;
wire z;

/**** DESIGN TO SIMULATE (my_fsm)
INSTANTIATION *****/
myfsm dut1(clk, rst, x, z);

/****RESET AND CLOCK SECTION****/
Initial
begin
clk=0;
rst=0;
#1rst=1; /*The delay gives rst a posedge for
sure.*/
#200 rst=0; //Deactivate reset after two clock
cycles +1ns*/
end
always #50clk=~clk; /* 10MHz clock (50*1ns*2)
with 50% duty-cycle */

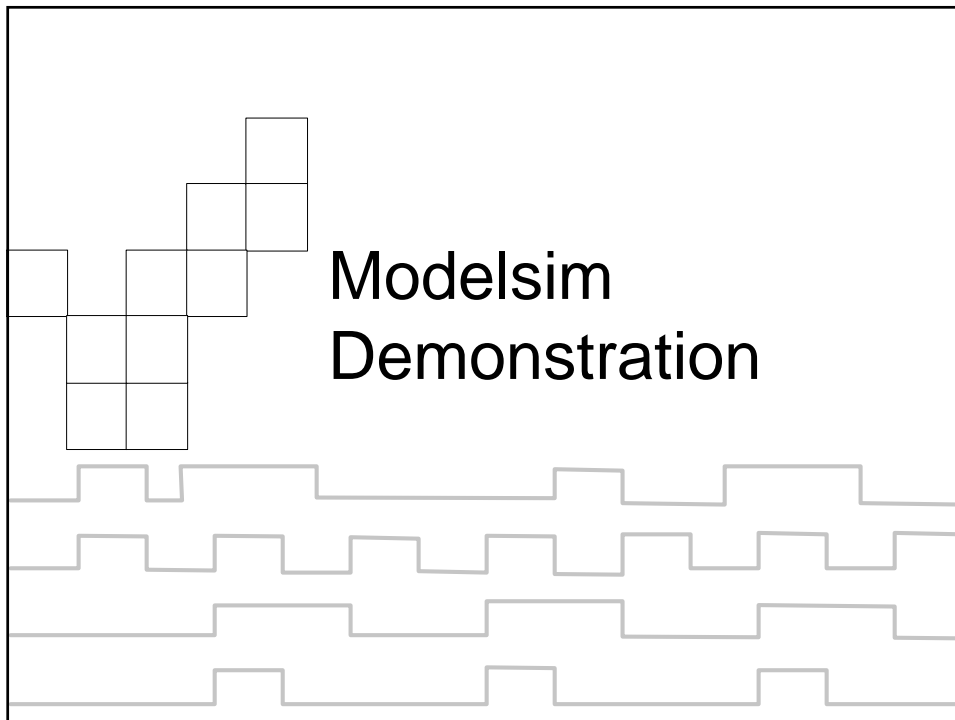
/****SPECIFY THE INPUT WAVEFORM x *****/
Initial begin
#1 x=0;
#400 x=1;
$display("Output z: %b", z);
#100 x=0;
@(posedge clk) x=1;

#1000 $finish; //stop simulation
//without this, it will not stop
end
endmodule
```

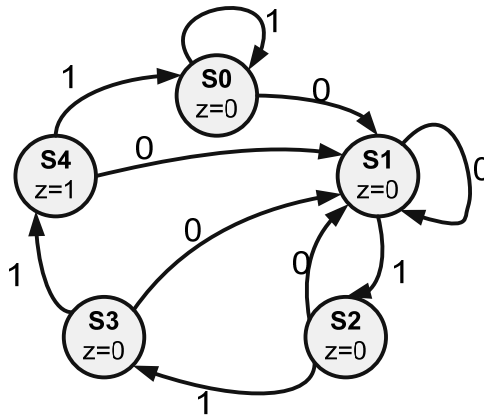
Nov 16, 2006

Abdul-Rahman Elshafei

35



0111 Sequence Detector



Nov 16, 2006

Abdul-Rahman Elshafei

37