UMASS AMHERST

ECE 697CE
Foundations of Computer Engineering

Lesson 8
Verilog Continued

Rationale

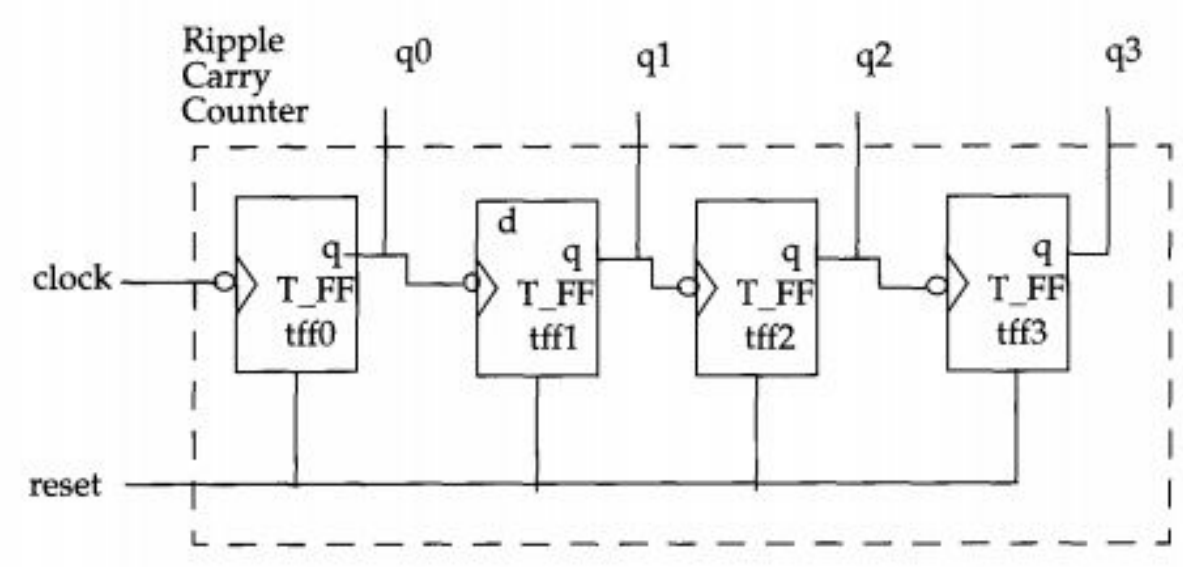
- Coding in Verilog creates fuller understanding of data flow
- Verilog offers tools for both simulation and debugging
- Hardware Description languages (HDL) allows one to describe the functionality of the circuit in a high level language rather than to design the digital logic needed to implement the circuit.

Objectives

- Apply the basics of coding in Verilog
- Analyze continuous data flow and procedural assignments
- Demonstrate the use of blocking and nonblocking assignments

Poll: Prior Knowledge

Given the following circuit implemented in Verilog, what is used to compute the output of the circuit?



1) Design Module
2) Top-level Block
3) 4 Bit Ripple Carry Counter
4) Simulation

Prior Knowledge

- Let's review from Lesson 4: Verilog
  - Introduction to HDL
  - Datatypes and Operators
  - Modules
  - Simulation Tools

Orchestrated Discussion (Hand Raise): Critical Thinking Exercise

- Discuss 2 questions about Lesson 4

Basics: Coding in Verilog

- Constants
  - Binary ‘b
  - Octal ‘o
  - Decimal ‘d
  - Hexadecimal ‘x
- Example:
  - V = 8’b1011, V = 00001011

- Verilog predefined logic values
  - 0 - represents number zero, logic zero, logical false
  - 1 - represents number one, logic one, logical true
  - x - represents an unknown logic value ; driven unknown
  - z - represents high impedance logic value ; undriven

Relational and Arithmetic Operators in Verilog

Operator	Application
<	a < b // is a less than b? // return 1-bit true/false
>	a > b // is a greater than b?
>=	a >= b // is a greater than or // equal to b
<=	a <= b // is a less than or // equal to b

Operator	Application
*	c = a * b ; // multiply a with b
/	c = a / b ; // int divide a by b
+	sum = a + b ; // add a and b

-	diff = a - b ; // subtract b
	// from a

%	amodb = a % b ; // a mod(b)

Logical, Equality, and Identity Operators in Verilog

Operator	Application

&&	a && b ; // is a and b true?
	// returns 1-bit true/false

||	a || b ; // is a or b true?
	// returns 1-bit true/false

!	if (!a) ; // if a is not true
	c = b ; // assign b to c

Operator	Application
=	c = a ; // assign a to c
==	c == a ; /* is c equal to a returns 1-bit true/false applies for 1 or 0, logic equality, using X or Z oper- ands returns always false 'hx == 'h5 returns 0 */

!=	c != a ; // is c not equal to
	// a, retruns 1-bit true/
	// false logic equality

===	a === b ; // is a identical to
	// b (includes 0, 1, x, z) /
	// 'hx === 'h5 returns 0

!==	a !== b ; // is a not
	// identical to b returns 1-
	// bit true/false

Unary Bitwise and Reduction Operators in Verilog

Operator	Application
+	Unary plus & arithmetic(binary) addition
-	Unary negation & arithmetic (binary) sub-traction
&	`b = &a ; // AND all bits of a`
|	`b = |a ; // OR all bits`
^	`b = ^a ; // Exclusive or all bits of a`
~&, ~|, ~^	NAND, NOR, EX-NOR all bits to-gether `c = ~& b ; d = ~| a; e = ^c ;`

~,&, |, ^	bit-wise NOT, AND, OR, EX-OR
	`b = ~a ; // invert a`
	`c = b & a ; // bitwise AND a,b`
	`e = b | a ; // bitwise OR`
	`f = b ^ a ; // bitwise EX-OR`

~&, ~|, ~^	bit-wise NAND, NOR, EX-NOR
	`c = a ~& b ; d = a ~| b ;`
	`e = a ~^ b ;`

Shift Operators and Others in Verilog

Operator	Application
<<	a << 1 ; // shift left a by 1-bit
>>	a >> 1 ; // shift right a by 1
?:	c = sel ? a : b ; /* if sel is true c = a, else c = b , ?: ternary operator */
{}	{co, sum } = a + b + ci ; /* add a, b, ci assign the overflow to co and the result to sum: operator is called concatenation */
{{}}	b = {3{a}} /* replicate a 3 times, equivalent to {a, a, a} */

Operator Precedence in Verilog

Operator	Precedence
+, -, !, ~ (unary)	Highest
*, / %	
+, - (binary)	
<<. >>	
<, < =, >, >=	
=, ==. !=	
===, !==	
&, ~&	
^, ^~	
|, ~|	
&&	
||	
?:	Lowest

Poll: Operators in Verilog

Which of the following define exactly the same value as 8'hF0?

1. **8'd240**
2. **{{4{1'b1}}, {4{1'b0}}}**
3. **{4'b1, 4b'0}**
4. **All of the above**
5. **None of the above**

Example of a Verilog Module

```
module adder( input a, b, out, outbar, sel);
                          //Declare and name a module
                          //Specify a list of ports
input  a, b, sel; //Specify each port as input, output or inout

output out, outbar;//
assign out= sel?0:1;//Executes in parallel
assign outbar=~out; //Executes in parallel
endmodule
```

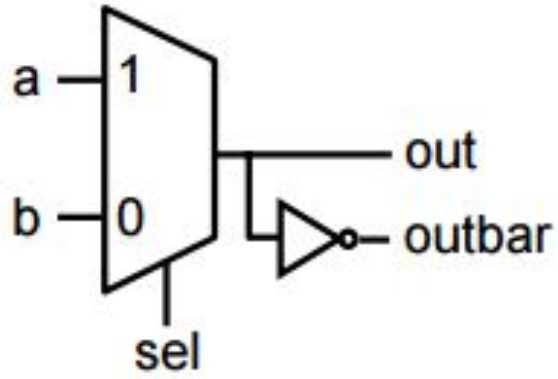


Fig: Two input multiplexer with inverted output

Blocks in Verilog Modules

- always
  - Commonly used, synthesizable
  - Evaluated whenever a signal in sensitivity list changes in simulator
  - Evaluated regardless of sensitivity list in actual hardware

- initial
  - Commonly used, non-synthesizable
  - Useful for testbench creation
  - Setting initial conditions else triggered by external events

- forever
  - Commonly used for generating clocks

    forever clk = #5 ~clk;

Continuous (Dataflow) Assignments

- Use the keyword assign
- Simple way to represent combinational logic
- Conceptually, the right-hand expression is continuously evaluated as a function of arbitrarily-changing inputs…just like dataflow
- Order doesn't matter
- Target: net driven by combinational logic

- Left side of the assignment
  - scalar or vector net
  - concatenation of scalar and vector nets
  - Can't be a scalar or vector register (discussed later)
- Right side can be register or nets
- Dataflow operators are fairly low-level;

Gate Level Description of a Circuit

```
module muxgate (a, b, out, outbar, sel);
   input a, b, sel;
   output out, outbar;
   wire out1, out2, selb;
   and a1 (out1, a, sel);
   not i1 (selb, sel);
   and a2 (out2, b , selb);
   or o1 (out, out1, out2);
   assign outbar = ~out;
endmodule
```

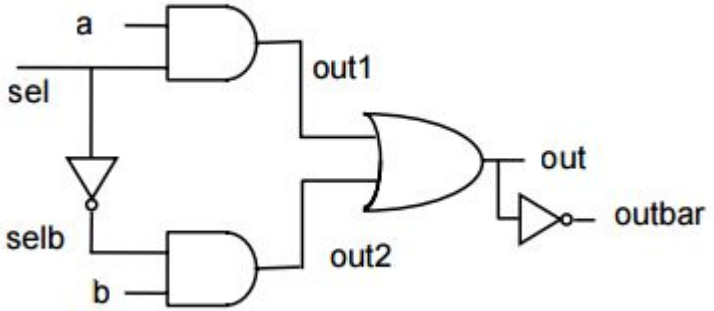


Fig: Two input multiplexer with inverted output

- Verilog supports basic logic gates as primitives

and, nand, or, nor, xor, xnor, not, buf

- Net represent connection between hardware elements
- Nets are declared with the keyword wire

Procedural Assignment of a Module Block

- An alternative higher-level, behavioral description of combinational logic
- Two structured procedure statements: initial and always
- Supports richer, C-like control structures such as

  if, for, while, case

- Anything assigned in an always block must also be declared as type reg
- Conceptually, the always block runs once whenever a signal in the sensitivity list changes value
- Statements within the always block are executed sequentially.
- Order matters!

- Surround multiple statements in a single always block with begin/end.

```
module mux_2_to_1(a, b, out, outbar, sel);
input a, b, sel;
output out, outbar;
reg out, outbar;
always @ (a or b or sel)
begin
   if (sel) out = a;
   else out = b;
   outbar = ~out;
end
endmodule
```

Mix and Match Assignments in Modules

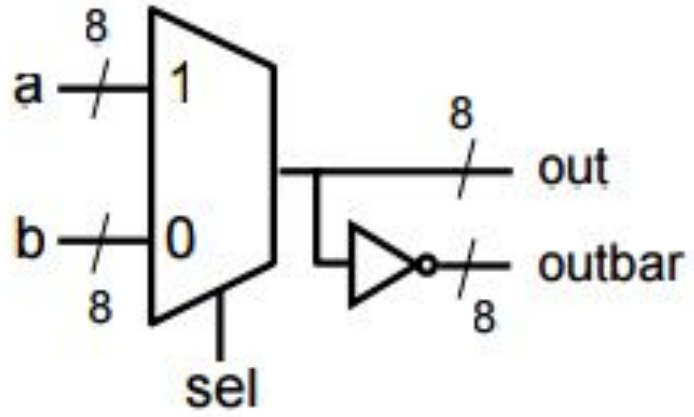
- Procedural and continuous assignments often co-exist within a module
- A variable updated by procedural assignment will remain unchanged till another procedural assignment updates the variable.
- This is the main difference with continuous assignments in which the right hand expression is constantly placed on the left-side

```
module mux_2_to_1(a, b, out, outbar, sel);
input a, b, sel;
output out, outbar;
reg out;
always @ (a or b or sel) //Procedural assignment
begin
    if (sel) out = a;
    else out = b;
end
assign outbar = ~out;//Continuous assignment
endmodule
```

N Bit Signal Examples

- E.g.: 2:1 multiplexer with 8 bit operands shown below:

```
module mux_2_to_1(a, b, out, outbar, sel);
input[7:0] a, b;
input sel;
output[7:0] out, outbar;
reg[7:0] out;
always @ (a or b or sel)
begin
 if (sel)
 out = a;
 else out = b;
 end
assign outbar = ~out;
endmodule
```



- E.g.: reg[31:0] mema[0:4095]; is a typical two dimensional memory, 4096 words of 32 bits

Group Discussion and Report Back (Short Answer): N Bit Signals

Which statement(s) is/are true of the following example: wire[31:0] a?

A. **declares a to be 32 wires.**

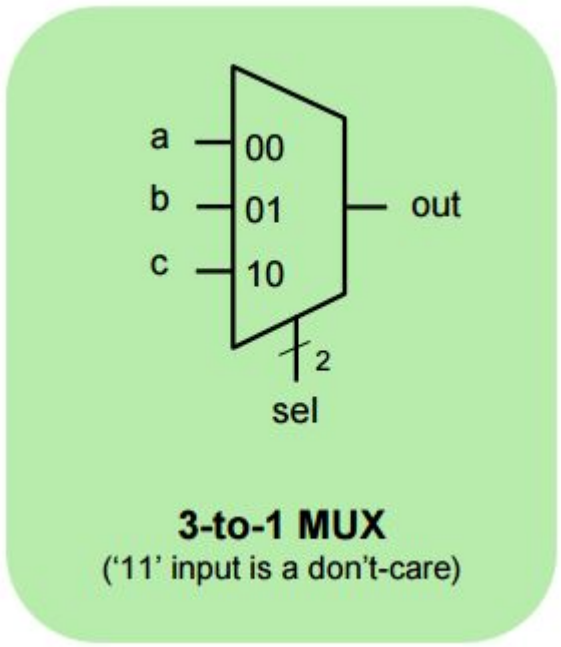
B. **just 'a' is all 32 wires,**

C. **a[31] is the most significant bit a[0] is the least significant bit**

Incomplete Specification in Verilog

- Note that if sel = '11' is a don't care condition

GOAL




```
module maybe_mux_3to1(a, b, c, sel, out);
input [1:0] sel;
input a, b, c;
output out;
reg out;
always @(a or b or c or sel)
begin
    case (sel)
    2'b00: out = a;
    2'b01: out = b;
    2'b10: out = c;
    endcase
end
endmodule
```

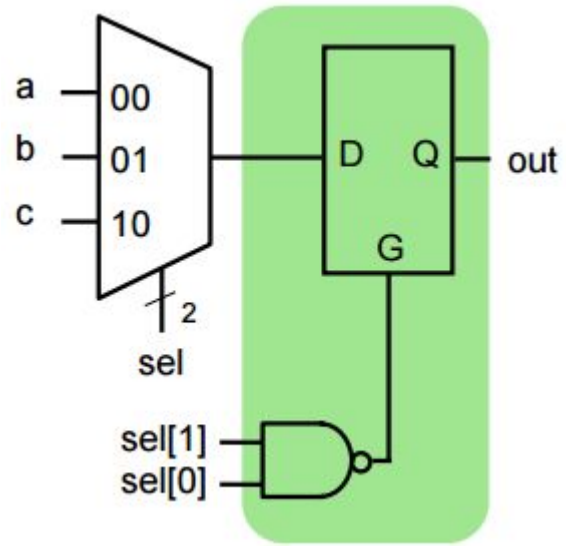
Incomplete Specification in Verilog: Latch Problem

- If out is not assigned during any pass through the always block, then the previous value is retained!
- The latch was not intended

```
module maybe_mux_3to1(a, b, c, sel, out);
input [1:0] sel;
input a, b, c;
output out;
reg out;
always @(a or b or c or sel)
begin
    case (sel)
    2'b00: out = a;
    2'b01: out = b;
    2'b10: out = c;
    endcase
end
endmodule
```

Whiteboard: Latch Problem Demonstration

SYNTHESIZED RESULT



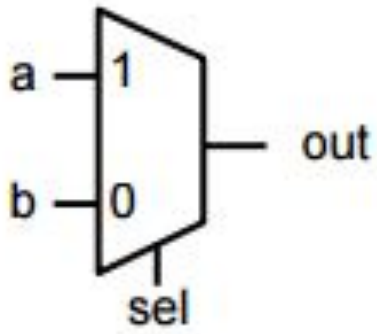

```
module maybe_mux_3to1(a, b, c, sel, out);
input [1:0] sel;
input a, b, c;
output out;
reg out;
always @(a or b or c or sel)
begin
    case (sel)
    2'b00: out = a;
    2'b01: out = b;
    2'b10: out = c;
    endcase
end
endmodule
```

The Sequential ALWAYS Block

- Edge triggered circuits are described using a sequential always block

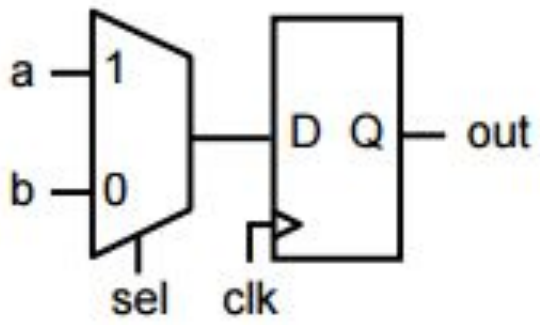
Combinational

```
module comb(a,b,sel,out);
input a,b;
input sel;
output out;
reg out;
always @(a or b or sel)
begin
    if (sel) out=a;
    else out=b;
end
endmodule
```



Sequential

```
module seq(a,b,sel,clk, out);
input a,b;
input sel,clk;
output out;
reg out;
always @(posedge clk)
begin
    if (sel) out<=a;
    else out<=b;
end
endmodule
```



Blocking Vs. Nonblocking Assignments

- Verilog supports two types of assignments within always blocks, with subtly different behaviors.
- Blocking assignment: evaluation and assignment are immediate
- Nonblocking assignment: all assignments deferred until all right-hand sides have been evaluated (end of simulation time step)

```
module block_nonblock();
reg a, b, c, d , e, f ;
initial // Blocking assignments
begin
   a = #10 1'b1 ; assigns 1 to a at time 10
   b = #20 1'b0 ; assigns 0 to b at time 30
   c = #40 1'b1 ; assigns 1 to c at time 70
end
endmodule
initial //Non blocking assignment
begin
    a <= #10 1'b1 ;assigns 1 to a at time
   10
    b <= #20 1'b0 ;assigns 0 to b at time
   20
    c <= #40 1'b1 ;assigns 1 to c at time
   40
 end
```

Orchestrated Discussion (Hand Raise): Blocking Vs. Nonblocking Assignments

Initial value of a=1 and b=2

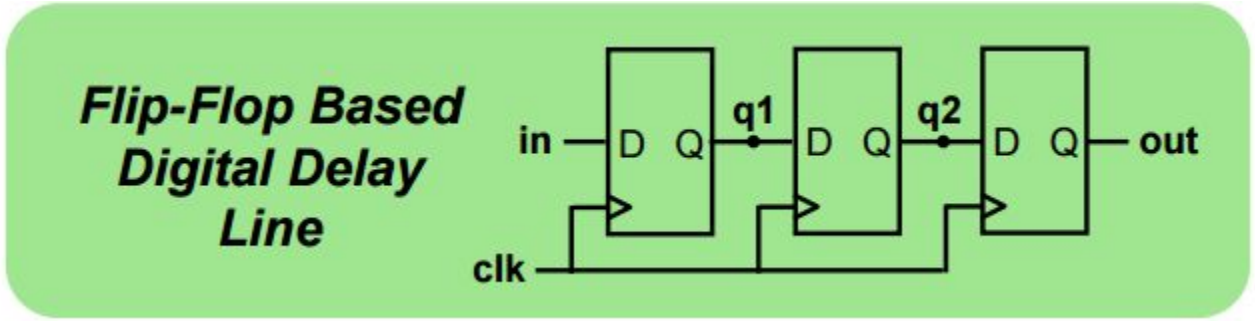
Part 1: Blocking

```
always @ (posedge clock)
  begin
     a=b;
     b=a;
  end
```

- Final value of a and b when using blocking assignments?

Part 2: Nonblocking

Sequential Logic: Assignment Style




```
//Nonblocking
always @ (posedge clk)
begin
      q1 <= in;
      q2 <= q1;
      out <= q2;
end
```

```
//Blocking
always @ (posedge clk)
begin
      q1 = in;
      q2 = q1;
      out = q2;
end
```

- Will both blocking and nonblocking assignments produce the desired result?

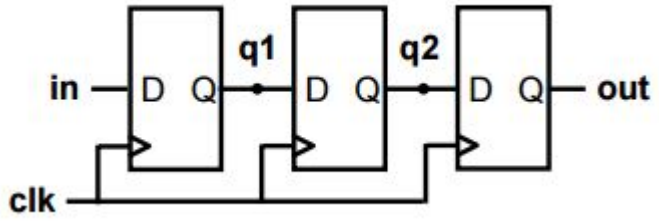
Use Nonblocking for Sequential Logic

- Use non blocking assignments for sequential always blocks

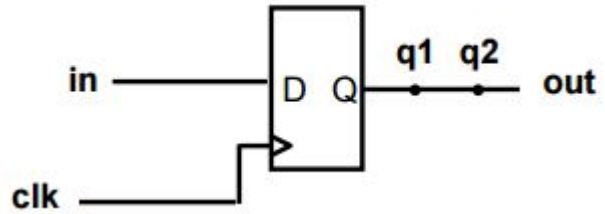
```
//Non blocking
always @ (posedge clk)
begin
      q1 <= in;
      q2 <= q1;
      out <= q2;
end
```

```
//Blocking
always @ (posedge clk)
begin
      q1 = in;
      q2 = q1;
      out = q2;
end
```

SYNTHESIZED RESULT



SYNTHESIZED RESULT



Recommendations for Style of Coding

- Many considerations like the quality of expected/resulting synthesis but also ease of debugging
- A good convention is to separate combinational and sequential blocks entirely
  - No combinational code in the sequential block!
  - Sequential block has mainly assignments to latch signals at clock edge or reset!
  - E.g.,
    - state <= state_nxt
    - signal <= signal_nxt
  - This keeps your code easy to read and debug and avoids subtle flaws
- Use non blocking assignments for sequential logic
- Use blocking assignments for combinational logic

Summary of this Lesson

- Answer questions related to Project 1: Finite State Machines in Verilog

Post-work for Lesson 5

Lesson Reflection

After the Live Lecture, you will reflect on what you learned. Then, you will answer questions and share your observations. Go to the online classroom to view the questions and submit your responses.

To Prepare for the Next Lesson

- Complete and submit the Post-work for Lesson 5.
- Read the Required Readings for Lesson 6.
- Complete the Pre-work for Lesson 6.
- Complete and submit the Project.

Go to the online classroom for details.