Commuter Aided Design
CAD

LECTURE 5

Modeling Concurrent Functionality in Verilog

Learning Outcomes—After completing this chapter, you will be able to:

- Describe the various built-in operators within Verilog.
- Design a Verilog model for a combinational logic circuit using continuous assignment and logical operators.
- Design a Verilog model for a combinational logic circuit using continuous assignment and conditional operators.
- Design a Verilog model for a combinational logic circuit using continuous assignment with delay.

1- Verilog Operators

1.1 Assignment Operator

Verilog uses the equal sign (=) to denote an assignment.

Example:

```
F1 = A;        // F1 is assigned the signal A
F2 = 8'hAA;    // F2 is an 8-bit vector and is assigned the value 10101010₂
```

1.2 Continuous Assignment

Verilog uses the keyword **assign** to denote a continuous signal assignment.

The left-hand side (LHS) of the assignment is the target signal and must be a net type.

Example:

```
assign F1 = A;       // F1 is updated anytime A changes, where A is a signal
assign F2 = 1'b0;    // F2 is assigned the value 0
```

Each individual assignment will be executed concurrently and synthesized as separate logic circuits.

Example:

```
assign X = A;
assign Y = B;
assign Z = C;
```

1- Verilog Operators

1.3 List of Verilog operators:

Category	Symbol
Bit-wise	~ & | ^ ~^
Reduction	& ~& | ~| ^ ~^ ^~
Logical	! && ||
Arithmetic	** * / % + -
Shift	<< >> <<< >>>
Relational	< > <= >=
Equality	== != === !==
Conditional	?:
Concatenation	{}
Replication	{{}}

1- Verilog Operators

1.3 Bitwise Logical Operators

Syntax	Operation
~	Negation
&	AND
|	OR
^	XOR
~^ or ^~	XNOR
<<	Logical shift left (fill empty LSB location with zero)
>>	Logical shift right (fill empty MSB location with zero)

Example:

```
~X          // invert each bit in X
X & Y       // AND each bit of X with each bit of Y
X | Y       // OR each bit of X with each bit of Y
X ^ Y       // XOR each bit of X with each bit of Y
X ~^ Y      // XNOR each bit of X with each bit of Y
X << 3      // Shift X left 3 times and fill with zeros
Y >> 2      // Shift Y right 2 times and fill with zeros
```

Hands-on time

Results:

```
# MON x=000000, y=000000, result=000000
# MON x=000101, y=110001, result=000001
# MON x=000101, y=110001, result=111110
# MON x=100101, y=011011, result=111111
# MON x=100101, y=011011, result=000000
# MON x=010110, y=011011, result=001101
# MON x=010110, y=011011, result=110010
# MON x=011011, y=011011, result=111111
```

```
module bitwise_operators(
    // no inputs here
    );

    reg [5:0] x = 0;        // 6bit variable
    reg [5:0] y = 0;        // 6bit variable
    reg [5:0] result = 0;   // 6bit variable

    // Procedure used to continuously monitor 'x', 'y', and 'result'
    initial begin
      $monitor("MON x=%b, y=%b, result=%b", x, y, result);
    end

    // Procedure used to generate stimulus
    initial begin
        #1;  // wait some time between examples
        x = 6'b00_0101;
        y = 6'b11_0001;
        result = x & y; // AND

        #1;  // Use the same values for x and y from above (reg stores the value)
        result = ~(x & y); // NAND Try: x ~& y to see what happens

        #1;
        x = 6'b10_0101;
        y = 6'b01_1011;
        result = x | y; // OR

        #1;
        result = ~(x | y); // NOR Try: x ~| y to see what happens

        #1;
        x = 6'b01_0110;
        y = 6'b01_1011;
        result = x ^ y; // XOR

        #1; // NXOR is used to check if x = y
        result = x ~^ y; // NXOR

        #1
        x = y; // This should make all bits 1
        result = ~(x ^ y); // NXOR
    end

endmodule
```

1- Verilog Operators

1.4 Reduction Logic Operators

Syntax	Operation
&	AND all bits in the vector together (1-bit result)
~&	NAND all bits in the vector together (1-bit result)
|	OR all bits in the vector together (1-bit result)
~|	NOR all bits in the vector together (1-bit result)
^	XOR all bits in the vector together (1-bit result)
~^ or ^~	XNOR all bits in the vector together (1-bit result)

Example:

```
&X      // AND all bits in vector X together
~&X     // NAND all bits in vector X together
|X      // OR all bits in vector X together
~|X     // NOR all bits in vector X together
^X      // XOR all bits in vector X together
~^X     // XNOR all bits in vector X together
```

Hands-on time

Results:

```
MON my_val1=11111, my_val2=101011110, result=1
MON my_val1=11111, my_val2=101011110, result=0
MON my_val1=11111, my_val2=101011110, result=1
MON my_val1=11111, my_val2=101011110, result=0
MON my_val1=11111, my_val2=101011110, result=1
MON my_val1=11111, my_val2=101011110, result=0
MON my_val1=11111, my_val2=101011110, result=1
MON my_val1=11111, my_val2=101011110, result=0
```

```
module reduction_operators();

  reg [4:0] my_val1 = 5'b1_1111; // 5bit variable
  reg [8:0] my_val2 = 9'b1_0101_1110;
  reg result;

    // Procedure used to continuously monitor 'my_val1', 'my_val2', and 'result
    initial begin
      $monitor("MON my_val1=%b, my_val2=%b, result=%b", my_val1, my_val2, resul
    end

    // Procedure used to generate stimulus
    initial begin
        result = &my_val1;   // AND reduction
        #1;                  // wait some time between examples
        result = &my_val2;

        #1;
        result = ~&my_val2;  // NAND reduction
        #1;
        result = ~&my_val1;

        #1;
        result = |my_val2;   // OR reduction

        #1;
        result = ~|my_val2;  // NOR reduction

        #1;
        result = ^my_val1;   // XOR reduction

        #1;
        result = ~^my_val1;  // XNOR reduction

        // Change the values of my_val1/2 and perform some bit reduction operat
        // Ex: my_val1 = 5'b1_0010
        //     result = ~& my_val1
    end

endmodule
```

1- Verilog Operators

1.5 Boolean Logic Operators

Syntax	Operation
!	Negation
&&	AND
||	OR

Example:

```
!X          // TRUE if all values in X are 0, FALSE otherwise
X && Y      // TRUE if the bitwise AND of X and Y results in all ones, FALSE otherwise
X || Y      // TRUE if the bitwise OR of X and Y results in all ones, FALSE otherwise
```

Hands-on time

Results:

```
# MON my_val1=111, my_val2=0000, result=0
# MON my_val1=111, my_val2=0000, result=1
# MON my_val1=111, my_val2=0000, result=0
# MON my_val1=111, my_val2=0000, result=1
# MON my_val1=z0x, my_val2=0000, result=x
# MON my_val1=z0x, my_val2=0000, result=0
```

```
module logical_operators();

    reg [2:0] my_val1 = 3'b111;   // 3bit variable
    reg [3:0] my_val2 = 4'b0000;  // 4bit variable
    reg result;                   // 1bit variable

    // Procedure used to continuously monitor 'my_val1', 'my_val2', and 'result'
    initial begin
      $monitor("MON my_val1=%b, my_val2=%b, result=%b", my_val1, my_val2, result);
    end

    // Procedure used to generate stimulus
    initial begin
        result = !my_val1;   // Logical NOT
        #1;                  // wait some time between examples
        result = !my_val2;   // Logical NOT

        #1;
        result = my_val1 && my_val2; // Logical AND

        #1;
        result = my_val1 || my_val2;

        #1;
        my_val1 = 3'bz0X; // Add some unknown bits
        result  = !my_val1;

        #1;
        result = my_val1 || my_val2;

        #1;
        result = my_val1 && my_val2;

        // Change the values of my_val1/2 and perform some logical operations
        // Ex: my_val2 = 4'b0101
        //     result = my_val1 && my_val2
        //     $display("MON my_val1=%b, my_val2=%b, result=%b", my_val1, my_val2, re
    end

endmodule
```

1- Verilog Operators

1.6 Relational Operators

Syntax	Description
==	Equality
!=	Inequality
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

Example:

```
X == Y     // TRUE if X is equal to Y, FALSE otherwise
X != Y     // TRUE if X is not equal to Y, FALSE otherwise
X < Y      // TRUE if X is less than Y, FALSE otherwise
X > Y      // TRUE if X is greater than Y, FALSE otherwise
X <= Y     // TRUE if X is less than or equal to Y, FALSE otherwise
X >= Y     // TRUE if X is greater than or equal to Y, FALSE otherwise
```

Hands-on time

Results:

```
# MON result = x
# MON result = 0
# MON result = 1
# MON result = 0
# MON result = 1
# MON result = x
# MON result = 1
```

```
module relational_operators();

  reg result;


    initial begin
      $monitor("MON result = %1b", result);
    end


    initial begin
        #1; result = 3 < 0;
        #1; result = 3 < 6'b00_1111; // 3 < 15?
        #1; result = 6 > 6;
        #1; result = 4'b1001 <= 4'b1010; // 9 <= 10?
        #1; result = 4'b100X > 4'b1010;
        #1; result = 99 >= 98;
    end

endmodule
```

1- Verilog Operators

1.7 Conditional Operators

The keyword for the conditional operator is ? with the following syntax:

```
<target_net> = <Boolean_condition> ? <true_assignment> : <false_assignment>;
```

Example:

```
F = (A == 1'b0) ? 1'b1 : 1'b0;                    // If A is a zero, F=1, otherwise F=0.
                                                  //     This models an inverter.

F = (sel == 1'b0) ? A : B;                        // If sel is a zero, F=A, otherwise F=B.
                                                  //     This models a selectable switch.

F = ((A == 1'b0) && (B == 1'b0)) ? 1'b'0 :        // Nested conditional statements.
    ((A == 1'b0) && (B == 1'b1)) ? 1'b'1 :        //     This models an XOR gate.
    ((A == 1'b1) && (B == 1'b0)) ? 1'b'1 :
    ((A == 1'b1) && (B == 1'b1)) ? 1'b'0;
```

1- Verilog Operators

1.8 Concatenation Operator

In Verilog, the curly brackets (i.e., {}) are used to concatenate multiple signals. The target of this operation must be the same size of the sum of the sizes of the input arguments.

Example:

```
Bus1[7:0] = {Bus2[7:4], Bus3[3:0]};  // Assuming Bus1, Bus2, and Bus3 are all 8-bit
                                     // vectors, this operation takes the upper
                                        4-bits of
                                     // Bus2, concatenates them with the lower
                                        4-bits of
                                     // Bus3, and assigns the 8-bit combination
                                        to Bus1.

BusC = {BusA, BusB};                 // If BusA and BusB are 4-bits, then BusC
                                     //  must be 8-bits.

BusC[7:0] = {4'b0000, BusA};         // This pads the 4-bit vector BusA with
                                        4x leading
                                     // zeros and assigns to the 8-bit vector BusC.
```

1- Verilog Operators

1.9 Replication Operator

Verilog provides the ability to concatenate a vector with itself through the replication operator. This operator uses double curly brackets (i.e., **{{}}**) and an integer indicating the number of replications to be performed. The replication syntax is as follows:

```
{<number_of_replications>{<vector_name_to_be_replicated>}}
```

Example:

```
BusX = {4{Bus1}};          // This is equivalent to: BusX = {Bus1, Bus1, Bus1, Bus1};
BusY = {2{A,B}};           // This is equivalent to: BusY = {A, B, A, B};
BusZ = {Bus1, {2{Bus2}}}; // This is equivalent to: BusZ = {Bus1, Bus2, Bus2};
```

Hands-on time

Results:

```
# a = 10101010
# a = 1x0z1x0z
# a = 10101111
# b = 01100110011001100110011001100110
# b = 01110001011100001xz01xz01xz01xz01
# b = 10101010101010101010101010101010
```

```
module replication_operator();

    reg [7:0] a;
    reg [31:0] b;

    // Procedure used to generate stimulus
    initial begin
        // Concatenation of {2'b10, 2'b10, 2'b10, 2'b10}
        #1; a = {4{2'b10}};
        $display("a = %b", a);

        // Concatenation of {4'b1X0Z, 4'b1X0Z}
        #1; a = {2{4'b1X0Z}};
        $display("a = %b", a);

        // Concatenation of {4'b1010, 1'b1, 1'b1, 1'b1, 1'b1}
        #1; a = {4'b1010, {4{1'b1}}};
        $display("a = %b", a);

        #1; b = {8{4'b0110}};
        $display("b = %b", b);

        #1; b = {{2{8'b0111_0001}}, {4{4'bXZ01}}};
        $display("b = %b", b);

        #1; b = {{16{2'b10}}};
        $display("b = %b", b);

        // Do by yourself some replication examples
    end

endmodule
```

1- Verilog Operators

1.10 Numerical Operators

Syntax	Operation
+	Addition
–	Subtraction (when placed between arguments)
–	2's complement negation (when placed in front of an argument)
*	Multiplication
/	Division
%	Modulus
**	Raise to the power
<<<	Shift to the left, fill with zeros
<<<	Shift to the right, fill with sign bit

Example:

```
X + Y       // Add X to Y
X - Y       // Subtract Y from X
-X          // Take the two's complement negation of X
X * Y       // Multiply X by Y
X / Y       // Divide X by Y
X % Y       // Modulus X/Y
X ** Y      // Raise X to the power of Y
X <<< 3     // Shift X left 3 times, fill with zeros
X >>> 2     // Shift X right 2 times, fill with sign bit
```

1- Verilog Operators

1.11 Operator Precedence

Operators	Precedence	Notes
! ~ + –	Highest	Bitwise/Unary
{} {{}}		Concatenation/Replication
()	↓	No operation, just parenthesis
**		Power
* / %		Binary Multiply/Divide/Modulo
+ –	↓	Binary Addition/Subtraction
<< >> <<< >>>		Shift Operators
< <= > >=		Greater/Less than Comparisons
== !=	↓	Equality/Inequality Comparisons
& ~&		AND/NAND Operators
^ ~^		XOR/XNOR Operators
| ~|	↓	OR/NOR Operators
&&		Boolean AND
||		Boolean OR
?:	Lowest	Conditional Operator

Hands-on time

Results:

```
# a = 0001
# a = 0001
# a = 0001
# b = 40
# b = 1
# b = 2
```

```
module operators_precedence();

  reg [3:0] a;
  int b;

  initial begin
    #1; a = ~4'b1110 & |4'b1000; // unary executed before bit-wise
    // ~4'b1110 = 4'b0001, |4'b1000 = 1'b1, 4'b0001 & 1'b1 = 4'b0001
    $display("a = %b", a);

    #1; a = ~4'b1100 & |4'b1000; // unary executed before bit-wise
    // ~4'b1100 = 4'b0011, |4'b1000 = 1'b1, 4'b0011 & 1'b1 = 4'b0001
    $display("a = %b", a);

    #1; a = |4'b0100 & ~&4'b1011; // unary executed before bit-wise
    // |4'b0100 = 1'b1, ~&4'b1011 = 1'b1, 1'b1 & 1'b1 = 4'b0001
    $display("a = %b", a);
    // Best practice: (|4'b0100) & (~&4'b1011)

    #1; b = 2 * 5 << 2; // power execute before shift
    // b = 10 << 2 = 40; * executes before <<
    $display("b = %0d", b);
    // Always use paranthesis b = (2 * 5) << 2 to make clear you intent
    // for you and for others

    #1; b = 2 < 4 && -33 > -34; // relational executed before logical
    // b = (2 < 4) && (-33 > -34) = 1 && 1 = 1
    $display("b = %0d", b);

    #1; b = 2 << 3 - 3; // arithmetic before shift
    // b = 2 << (3 - 3) = 2 << 0 = 2;
    $display("b = %0d", b);

    // Do some other examples to play with operators precedence

  end

endmodule
```