CS 207 Digital Logic - Spring 2019
Lab 3 - Operators and Primitives

Monday, Mar. 4, 2019

1 Experiment A

In Verilog, operators can be classified into multiple categories, i.e., arithmetic, relational, equality, logical, bit-wise, reduction, shift, concatenation, replication, and conditional operators. In this experiment, we investigate how they functions on input variables with examples.

When going through the following examples, first read the bullet points indicating the operators and properties of the specific operator category. Then implement the example and see if the result meets the theory. Note that the source codes in this experiment is available from the course website. It is recommended but not required to type some of the code for better understanding.

1.1 Arithmetic Operators

Operator	Description
+, -, *, /	binary modulus operator
+, -	unary operator used to specify the sign

- Integer division truncates any fractional part.
- If any operand bit value is the unknown value X, then the entire result value is X.

arithmetic_operators.v

```
1 module arithmetic_operators();
2
3 initial begin
4   $display (" 5  +  10 = %d", 5  + 10);
5   $display (" 5  -  10 = %d", 5  - 10);
6   $display (" 10 -  5  = %d", 10 - 5);
7   $display (" 10 *  5  = %d", 10 * 5);
```

```
8     $display (" 10 /  5  = %d", 10 / 5);
9     $display (" 10 /  -5 = %d", 10 / -5);
10    $display (" 10 %s 3  = %d", "%", 10 % 3);
11    $display (" +5       = %d", +5);
12    $display (" -5       = %d", -5);
13    #10 $finish;
14  end
15
16  endmodule
```

Assignment
Save the command line output in `arithmetic_operators.log`. Assignment 1 requires the output file.

1.2 Relational Operators

Operator	Description
a ¡ b	a less than b
a ¿ b	a greater than b
a ¡= b	a less than or equal to b
a ¿= b	a greater than or equal to b

- The result is a scalar value (example a ¡ b).
- 0 if the relation is false (a is bigger then b).
- 1 if the relation is true (a is smaller then b).
- x if any of the operands has unknown x bits (if a or b contains X).

relational_operators.v

```
1   module relational_operators();
2
3   initial begin
4     $display (" 5      <=  10 = %b", (5      <= 10));
5     $display (" 5      >=  10 = %b", (5      >= 10));
6     $display (" 1'bx   <=  10 = %b", (1'bx   <= 10));
7     $display (" 1'bz   <=  10 = %b", (1'bz   <= 10));
8     #10 $finish;
9   end
10
11  endmodule
```

Assignment
Save the command line output in `relational_operators.log`. Assignment 1 requires the output file.

1.3 Equality Operators

There are two types of Equality operators. Case Equality and Logical Equality.

Operator	Description
a === b	a equal to b, including X and Z (Case equality)
a !== b	a not equal to b, including X and Z (Case inequality)
a == b	a equal to b, result may be unknown (logical equality)
a != b	a not equal to b, result may be unknown (logical equality)

- Operands are compared bit by bit, with zero filling if the two operands do not have the same length.
- Result is 0 (false) or 1 (true).
- For the == and != operators, the result is X, if either operand contains an X or a Z (high impedance, or open circuit).
- For the === and !== operators, bits with X and Z are included in the comparison and must match for the result to be true.

equality_operators.v

```
1  module equality_operators();
2
3  initial begin
4    // Case Equality
5    $display (" 4'bx001 ===  4'bx001 = %b", (4'bx001 ===  4'
         ↪ bx001));
6    $display (" 4'bx0x1 ===  4'bx001 = %b", (4'bx0x1 ===  4'
         ↪ bx001));
7    $display (" 4'bz0x1 ===  4'bz0x1 = %b", (4'bz0x1 ===  4'
         ↪ bz0x1));
8    $display (" 4'bz0x1 ===  4'bz001 = %b", (4'bz0x1 ===  4'
         ↪ bz001));
9    // Case Inequality
10   $display (" 4'bx0x1 !==  4'bx001 = %b", (4'bx0x1 !==  4'
         ↪ bx001));
11   $display (" 4'bz0x1 !==  4'bz001 = %b", (4'bz0x1 !==  4'
         ↪ bz001));
12   // Logical Equality
13   $display (" 5       ==   10      = %b", (5       ==   10))
         ↪ ;
```

```
14    $display (" 5       ==    5       = %b", (5       ==    5));
15    // Logical Inequality
16    $display (" 5       !=    5       = %b", (5       !=    5));
17    $display (" 5       !=    6       = %b", (5       !=    6));
18    #10 $finish;
19  end
20
21  endmodule
```

Assignment
Save the command line output in `equality_operators.log`. Assignment 1 requires the output file.

1.4 Logical Operators

Operator	Description
!	logic negation
&&	logical and
||	logical or

- Expressions connected by `&&` and `||` are evaluated from left to right.
- Evaluation stops as soon as the result is known.
- The result is a scalar value:
  - 0 if the relation is false.
  - 1 if the relation is true.
  - X if any of the operands has X (unknown) bits.

logical_operators.v

```
1  module logical_operators();
2
3  initial begin
4    // Logical AND
5    $display ("1'b1 && 1'b1 = %b", (1'b1 && 1'b1));
6    $display ("1'b1 && 1'b0 = %b", (1'b1 && 1'b0));
7    $display ("1'b1 && 1'bx = %b", (1'b1 && 1'bx));
8    // Logical OR
9    $display ("1'b1 || 1'b0 = %b", (1'b1 || 1'b0));
10   $display ("1'b0 || 1'b0 = %b", (1'b0 || 1'b0));
11   $display ("1'b0 || 1'bx = %b", (1'b0 || 1'bx));
12   // Logical Negation
```

```
13    $display ("! 1'b1        = %b", (!   1'b1));
14    $display ("! 1'b0        = %b", (!   1'b0));
15    #10 $finish;
16  end
17
18  endmodule
```

Assignment
Save the command line output in `logical_operators.log`. Assignment 1 requires the output file.

1.5 Bit-wise Operators

Bitwise operators perform a bit wise operation on two operands. They take each bit in one operand and perform the operation with the corresponding bit in the other operand. If one operand is shorter than the other, it will be extended on the left side with zeroes to match the length of the longer operand.

Operator	Description
~	negation
&	and
|	inclusive or
^	exclusive or
^~ or ~^	exclusive nor (equivalence)

- Computations include unknown bits, in the following way:
  - ~X = X
  - 0&X = 0
  - 1&X = X&X = X
  - 1|X = 1
  - 0|X = X|X = X
  - 0^X = 1^X = X^X = X
  - 0^~X = 1^~X = X^~X = X
- When operands are of unequal bit length, the shorter operand is zero-filled in the most significant bit positions.

bitwise_operators.v

```
1  module bitwise_operators();
2
3  initial begin
```

```
4     // Bit Wise Negation
5     $display (" ~4'b0001           = %b", (~4'b0001));
6     $display (" ~4'bx001           = %b", (~4'bx001));
7     $display (" ~4'bz001           = %b", (~4'bz001));
8     // Bit Wise AND
9     $display (" 4'b0001 &  4'b1001 = %b", (4'b0001 &  4'b1001)
          ↪ );
10    $display (" 4'b1001 &  4'bx001 = %b", (4'b1001 &  4'bx001)
          ↪ );
11    $display (" 4'b1001 &  4'bz001 = %b", (4'b1001 &  4'bz001)
          ↪ );
12    // Bit Wise OR
13    $display (" 4'b0001 |  4'b1001 = %b", (4'b0001 |  4'b1001)
          ↪ );
14    $display (" 4'b0001 |  4'bx001 = %b", (4'b0001 |  4'bx001)
          ↪ );
15    $display (" 4'b0001 |  4'bz001 = %b", (4'b0001 |  4'bz001)
          ↪ );
16    // Bit Wise XOR
17    $display (" 4'b0001 ^  4'b1001 = %b", (4'b0001 ^  4'b1001)
          ↪ );
18    $display (" 4'b0001 ^  4'bx001 = %b", (4'b0001 ^  4'bx001)
          ↪ );
19    $display (" 4'b0001 ^  4'bz001 = %b", (4'b0001 ^  4'bz001)
          ↪ );
20    // Bit Wise XNOR
21    $display (" 4'b0001 ~^ 4'b1001 = %b", (4'b0001 ~^ 4'b1001)
          ↪ );
22    $display (" 4'b0001 ~^ 4'bx001 = %b", (4'b0001 ~^ 4'bx001)
          ↪ );
23    $display (" 4'b0001 ~^ 4'bz001 = %b", (4'b0001 ~^ 4'bz001)
          ↪ );
24    #10 $finish;
25  end
26
27  endmodule
```

Assignment
Save the command line output in `bitwise_operators.log`. Assignment 1 requires the output file.

1.6 Reduction Operators

- Reduction operators are unary.
- They perform a bit-wise operation on a single operand to produce a single bit result.

Operator	Description
&	and
~&	nand
|	or
~|	nor
^	xor
^~ or ~^	xnor

- Reduction unary NAND and NOR operators operate as AND and OR respectively, but with their outputs negated.
- Unknown bits are treated as described before.

reduction_operators.v

```
1  module reduction_operators();
2
3  initial begin
4    // Bit Wise AND reduction
5    $display (" &  4'b1001 = %b", (&  4'b1001));
6    $display (" &  4'bx111 = %b", (&  4'bx111));
7    $display (" &  4'bz111 = %b", (&  4'bz111));
8    // Bit Wise NAND reduction
9    $display (" ~& 4'b1001 = %b", (~& 4'b1001));
10   $display (" ~& 4'bx001 = %b", (~& 4'bx001));
11   $display (" ~& 4'bz001 = %b", (~& 4'bz001));
12   // Bit Wise OR reduction
13   $display (" |  4'b1001 = %b", (|  4'b1001));
14   $display (" |  4'bx000 = %b", (|  4'bx000));
15   $display (" |  4'bz000 = %b", (|  4'bz000));
16   // Bit Wise OR reduction
17   $display (" ~| 4'b1001 = %b", (~| 4'b1001));
18   $display (" ~| 4'bx001 = %b", (~| 4'bx001));
19   $display (" ~| 4'bz001 = %b", (~| 4'bz001));
20   // Bit Wise XOR reduction
21   $display (" ^  4'b1001 = %b", (^  4'b1001));
22   $display (" ^  4'bx001 = %b", (^  4'bx001));
23   $display (" ^  4'bz001 = %b", (^  4'bz001));
24   // Bit Wise XNOR
25   $display (" ~^ 4'b1001 = %b", (~^ 4'b1001));
26   $display (" ~^ 4'bx001 = %b", (~^ 4'bx001));
27   $display (" ~^ 4'bz001 = %b", (~^ 4'bz001));
28   #10 $finish;
29 end
30
31 endmodule
```

Assignment
Save the command line output in `reduction_operators.log`. Assignment 1 requires the output file.

1.7 Shift Operators

Operator	Description
<<	left shift
>>	right shift

- The left operand is shifted by the number of bit positions given by the right operand.
- The vacated bit positions are filled with zeroes.

shift_operators.v

```
1  module shift_operators();
2
3  initial begin
4    // Left Shift
5    $display (" 4'b1001 << 1 = %b", (4'b1001 << 1));
6    $display (" 4'b10x1 << 1 = %b", (4'b10x1 << 1));
7    $display (" 4'b10z1 << 1 = %b", (4'b10z1 << 1));
8    // Right Shift
9    $display (" 4'b1001 >> 1 = %b", (4'b1001 >> 1));
10   $display (" 4'b10x1 >> 1 = %b", (4'b10x1 >> 1));
11   $display (" 4'b10z1 >> 1 = %b", (4'b10z1 >> 1));
12   #10 $finish;
13 end
14
15 endmodule
```

Assignment
Save the command line output in `shift_operators.log`. Assignment 1 requires the output file.

1.8 Concatenation Operators

- Concatenations are expressed using the brace characters `{` and `}`, with commas separating the expressions within.

  - Example: `+ {a, b[3:0], c, 4'b1001}` // if a and c are 8-bit numbers, the results has 24 bits.

- Un-sized constant numbers are not allowed in concatenations.

concatenation_operator.v

```
1  module concatenation_operator();
2
3  initial begin
4    // concatenation
5    $display (" {4'b1001,4'b10x1}  = %b", {4'b1001,4'b10x1});
6    #10 $finish;
7  end
8
9  endmodule
```

Assignment
Save the command line output in `concatenation_operators.log`. Assignment 1 requires the output file.

1.9 Replication Operators

Replication operator is used to replicate a group of bits n times. Say you have a 4 bit variable and you want to replicate it 4 times to get a 16 bit variable: then we can use the replication operator.

Operator	Description
`{n{m}}`	Replicate value m, n times

- Repetition multipliers (must be constants) can be used:
  - `{3{a}}` // this is equivalent to `{a, a, a}`
- Nested concatenations and replication operator are possible:
  - `{b, {3{c, d}}}` // this is equivalent to `{b, c, d, c, d, c, d}`

replication_operator.v

```
1  module replication_operator();
2
3  initial begin
4    // replication
5    $display (" {4{4'b1001}}      = %b", {4{4'b1001}});
6    // replication and concatenation
```

```
7     $display (" {4{4'b1001,1'bz}} = %b", {4{4'b1001,1'bz}});
8     #10 $finish;
9   end
10
11  endmodule
```

Assignment
Save the command line output in `replication_operators.log`. Assignment 1 requires the output file.

1.10 Conditional Operators

The conditional operator has the following C-like format: `cond_expr ? true_expr : false_expr`. The `true_expr` or the `false_expr` is evaluated and used as a result depending on what `cond_expr` evaluates to (true or false).

conditional_operator.v

```
1   module conditional_operator();
2
3   wire out;
4   reg enable,data;
5   // Tri state buffer
6   assign out = (enable) ? data : 1'bz;
7
8   initial begin
9     $display ("time\t enable data out");
10    $monitor ("%g\t %b       %b     %b",$time,enable,data,out);
11    enable = 0;
12    data = 0;
13    #1 data = 1;
14    #1 data = 0;
15    #1 enable = 1;
16    #1 data = 1;
17    #1 data = 0;
18    #1 enable = 0;
19    #10 $finish;
20  end
21
22  endmodule
```

Assignment
Save the command line output in `conditional_operators.log`. Assignment 1 requires the output file.

1.11 Operator Precedence

The operator precedence relationship, from top priority to least: Unary, Multiply, Divide, Modulus > Add, Subtract, Shift > Relation, Equality > Reduction > Logic > Conditional.

2 Experiment B

Verilog has built-in primitives like gates, transmission gates, and switches. This is a rather small number of primitives; if we need more complex primitives, then Verilog provides User Defined Primitives (UDPs).

2.1 User Defined Primitive

UDP begins with reserve word `primitive` and ends with `endprimitive`. Ports/terminals of primitive should follow. This is similar to what we do for module definition. UDPs should be defined outside `module` and `endmodule`:

```
// This code shows how input/output ports
// and primitives are declared
primitive udp_syntax (
a, // Port a
b, // Port b
c, // Port c
d  // Port d
);
output a;
input b,c,d;

// UDP function code here

endprimitive
```

In the above code, `udp_syntax` is the primitive name, it contains ports a, b, c, d.

Functionality of primitives is described inside a table, and it ends with reserved word `endtable` as shown in the code below. For sequential UDP, we can use initial to assign an initial value to output.

```
// This code shows how UDP body looks like
primitive udp_body (
a, // Port a
b, // Port b
c  // Port c
);
output a;
input b,c;

// UDP function code here
```

```
// A = B | C;
table
 // B  C    : A
    ?  1    : 1;
    1  ?    : 1;
    0  0    : 0;
endtable

endprimitive
```

Note that An UDP cannot use `z` for high impedance in the input table. We may use the following to test the primitive:

```
module udp_body_tb();

reg b,c;
wire a;

udp_body udp (a,b,c);

initial begin
  $monitor(" B = %b C = %b  A = %b",b,c,a);
  b = 0;
  c = 0;
   #1  b = 1;
   #1  b = 0;
   #1  c = 1;
   #1  b = 1'bx;
   #1  c = 0;
   #1  b = 1;
   #1  c = 1'bx;
   #1  b = 0;
   #1  $finish;
end

endmodule
```

Finally, `initial` statement is used for initialization of sequential UDPs. The statement that follows must be an assignment statement that assigns a single bit literal value to the output terminal `reg`. A partial example:

```
primitive udp_initial (a,b,c);
output a;
input b,c;
reg a;
// a has value of 1 at start of sim
initial a = 1'b1;

table
// udp_initial behaviour
endtable
```

```
11
12 endprimitive
```

2.2 Have a Try

Use data-flow model with Verilog operators to implement the following Boolean expressions in the last lab session:

- $(b + d)(a' + b' + c)$
- $a'b + a'c' + abc$

Then create a UDP for each of them, and write a testbench to verify their equivalency.

Assignment
Save the source code and testbench in `udp.v`. Save the command line output in `udp.log`. Assignment 1 requires the source and output files.