Verilog - Operators

- Verilog operators operate on several data types to produce an output
- Not all Verilog operators are synthesible (can produce gates)
- Some operators are similar to those in the C language
- Remember, you are making gates, not an algorithm (in most cases)
Verilog - Operators

Arithmetic Operators

- There are two types of operators: binary and unary
- Binary operators:
  - add(+) -, subtract(-), multiply(*), divide(/), power(**), modulus(%)

//suppose that: a = 4’b0011;
//            b = 4’b0100;
//            d = 6; e = 4; f = 2;
//then,
a + b //add a and b; evaluates to 4’b0111
b - a //subtract a from b; evaluates to 4’b0001
a * b //multiply a and b; evaluates to 4’b1100
d / e //divide d by e, evaluates to 4’b0001. Truncates fractional part
e ** f //raises e to the power f, evaluates to 4’b1111
    //power operator is most likely not synthesizable

If any operand bit has a value "x", the result of the expression is all "x".
If an operand is not fully known the result cannot be either.
Arithmetic Operators (cont.)
Modulus operator yields the remainder from division of two numbers
It works like the modulus operator in C
May or may not be synthesizable

3 % 2; //evaluates to 1
16 % 4; //evaluates to 0
-7 % 2; //evaluates to -1, takes sign of first operand
7 % -2; //evaluates to 1, takes sign of first operand
Arithmetic Operators (cont.)

- Unary operators
  - Operators "+" and "-" can act as unary operators
  - They indicate the sign of an operand

i.e., -4 // negative four
+5 // positive five

!!! Negative numbers are represented as 2’s compliment numbers !!!
!!! Use negative numbers only as type integer or real !!!
!!! Avoid the use of <sss>’<base><number > in expressions !!!
!!! These are converted to unsigned 2’s compliment numbers !!!
!!! This yields unexpected results in simulation and synthesis !!!
Verilog - Operators

Arithmetic Operators (cont.)

- The logic gate realization depends on several variables
  - coding style
  - synthesis tool used
  - synthesis constraints (more later on this)

- So, when we say “+”, is it a...
  - ripple-carry adder
  - look-ahead-carry adder (how many bits of lookahead to be used?)
  - carry-save adder

When writing RTL code, keep in mind what will eventually be needed
Continually thinking about structure, timing, size, power
16-bit adder with loose constraints:
set_max_delay 2 [get_ports sum*]
max delay = 0.8ns, area = 472 = 85 gates
16-bit adder with tighter constraints:
set_max_delay 0.5 [get_ports sum*]
max delay = 0.5ns, area = 2038 = 368gates
Verilog - Operators

Logical Operators

- Verilog Logical Operators
  - logical-and(&&) //binary operator
  - logical-or(||) //binary operator
  - logical-not(!) //unary operator

//suppose that: a = 3 and b = 0, then...
(a && b) //evaluates to zero
(b || a) //evaluates to one
(!a) //evaluates to 0
(!b) //evaluates to 1

//with unknowns: a = 2'b0x; b = 2'b10;
(a && b) // evaluates to x

//with expressions...
(a == 2) && (b == 3) //evaluates to 1 only if both comparisons are true
Verilog - Operators

Logical Operators (.cont)

- Logical operators evaluate to a 1 bit value
  - 0 (false), 1 (true), or x (ambiguous)
- Operands not equal to zero are equivalent to one
- Logical operators take variables or expressions as operators
Verilog - Operators

Relational Operators (.cont)

- greater-than (>)
- less-than (<)
- greater-than-or-equal-to (>=)
- less-than-or-equal-to (<=)

Relational operators return logical 1 if expression is true, 0 if false

```verilog
//let a = 4, b = 3, and...
//x = 4'b1010, y = 4'b1101, z = 4'b1xxx
a <= b  //evaluates to logical zero
a > b   //evaluates to logical one
y >= x  //evaluates to logical 1
y < z   //evaluates to x
```

!!! Note: These are expensive and slow operators at gate level !!!
Verilog - Operators

Equality Operators - "LT" is big and slow

// 8-bit less than detector
// if a is less than b, output is logic one
module less8(
    input [7:0] a, b,
    output z
);
    assign z = (a < b) ? 1'b1 : 1'b0;
endmodule

Results from synthesis:
Equality Operators

- logical equality (==)
- logical inequality (!=)
- logical case equality (===)
- logical case inequality (!==)

Equality operators return logical 1 if expression is true, else 0

Operands are compared bit by bit

Zero filling is done if operands are of unequal length (Warning!)

Logical case inequality allows for checking of x and z values

Checking for X and Z is most definitely non-synthesible!
Equality Operators (cont.)

//let a = 4, b = 3, and...
//x = 4'b1010, y = 4'b1101,
//z = 4'b1xxz, m = 4'b1xxz, n = 4'b1xxx

a == b  //evaluates to logical 0
x != y  //evaluates to logical 1
x == z  //evaluates to x
z === m  //evaluates to logical 1
z === n  //evaluates to logical 0
m !== n  //evaluates to logical 1
Verilog - Operators

Bitwise Operators

- negation (∼), and(&), or(|), xor(^), xnor(¬,¬)
- Perform bit-by-bit operation on two operands (except ∼)
- Mismatched length operands are zero extended
- x and z treated the same

<table>
<thead>
<tr>
<th>bitwise AND</th>
<th>bitwise OR</th>
<th>bitwise XOR</th>
<th>bitwise XNOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 x</td>
<td>0 1 x</td>
<td>0 1 x</td>
<td>0 1 x</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>0 0 1 x</td>
<td>0 0 1 x</td>
<td>0 1 0 x</td>
</tr>
<tr>
<td>1 0 1 x</td>
<td>1 1 1 1</td>
<td>1 1 0 x</td>
<td>1 0 1 x</td>
</tr>
<tr>
<td>x 0 x x</td>
<td>x x 1 x</td>
<td>x x x x</td>
<td>x x x x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bitwise negation</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
Bitwise Operators (cont.)

- Logical operators result in logical 1, 0 or x
- Bitwise operators result in a bit-by-bit value

```verbatim
//let x = 4'b1010, y = 4'b0000
x | y  //bitwise OR, result is 4'b1010
x || y  //logical OR, result is 1
```
Bitwise operators give bit-by-bit results

//8-bit wide AND
module and8(
    input [7:0] a, b,
    output [7:0] z
);
    assign z = a & b;
endmodule
Verilog - Operators

Reduction Operators

- and(&), nand(∼&), or(||), nor(∼||) xor(^), xnor(∼^,∼^)
- Operates on only one operand
- Performs a bitwise operation on all bits of the operand
- Returns a 1-bit result
- Works from right to left, bit by bit

```verbatim
//let x = 4'b1010
&x  //equivalent to 1 & 0 & 1 & 0. Results in 1'b0
|x  //equivalent to 1 | 0 | 1 | 0. Results in 1'b1
^x  //equivalent to 1 ^ 0 ^ 1 ^ 0. Results in 1'b0
```

A good example of the XOR operator is generation of parity
//8-bit parity generator
//output is one if odd # of ones
module parity8(
    input [7:0] d_in,
    output parity_out
);
    assign parity_out = ^d_in;
endmodule
Verilog - Operators

Shift Operators

- right shift (>>)
- left shift (<<)
- arithmetic right shift (>>>)
- arithmetic left shift (<<<)

Shift operator shifts a vector operand left or right by a specified number of bits, filling vacant bit positions with zeros.

- Shifts do not wrap around.
- Arithmetic shift uses context to determine the fill bits.

```
// let x = 4'b1100
y = x >> 1; // y is 4'b0110
y = x << 1; // y is 4'b1000
y = x << 2; // y is 4'b0000
```
Arithmetic Shift Operators

- arithmetic right shift (\(>>>\))
  - Shift right specified number of bits, fill with value of sign bit if expression is signed, otherwise fill with zero.

- arithmetic left shift (\(<<<\))
  - Shift left specified number of bits, filling with zero.
Verilog - Operators

Concatenation Operator 

- Provides a way to append busses or wires to make busses
- The operands must be sized
- Expressed as operands in braces separated by commas

```verbatim
// let a = 1'b1, b = 2'b00, c = 2'b10, d = 3'b110
y = {b, c} // y is then 4'b0010
y = {a, b, c, d, 3'b001} // y is then 11'b10010110001
y = {a, b[0], c[1]} // y is then 3'b101
```
Replication Operator \{ \{ \} \}

- Repetitive concatenation of the same number
- Operands are number of repetitions, and the bus or wire

```verbatim
//let a = 1'b1, b = 2'b00, c = 2'b10, d = 3'b110
y = \{ 4\{a\} \}          // y = 4'b1111
y = \{ 4\{a\}, 2\{b\} \}    // y = 8'b11110000
y = \{ 4\{a\}, 2\{b\}, c \} // y = 8'b1111000010
```
Verilog - Operators

Conditional Operator ?:
▶ Operates like the C statement
  - `conditional_expression ? true_expression : false_expression ;`
▶ The conditional_expression is first evaluated
  - If the result is true, `true_expression` is evaluated
  - If the result is false, `false_expression` is evaluated
  - If the result is x:
    - both true and false expressions are evaluated,...
    - their results compared bit by bit,...
    - returns a value of x if bits differ, OR...
    - the value of the bits if they are the same.

This is an ideal way to model a multiplexer or tri-state buffer.
Verilog - Operators

Conditional Operator (cont.)

//8-bit wide, 2:1 mux
module mux2_1_8wide(
    input sel,
    input [7:0] d_in1, d_in0,
    output [7:0] d_out
);
    assign d_out = sel ? d_in1 : d_in0;
endmodule
Verilog - Operators

Conditional Operator (cont.)

//8-bit wide,
//active-low enabled tri-state buffer
module ts_buff8(
    input [7:0] d_in,
    input en_n,
    output [7:0] d_out
);
    assign d_out = ~en_n ? d_in : 8'bz;
endmodule
Verilog - Operators

More Lexical Conventions

► The "assign" statement places a value (a binding) on a wire
► Also known as a continuous assign
► A simple way to build combinatorial logic
► Confusing for complex functions
► Must be used outside a procedural statement (always)

```verilog
//two input mux, output is z, inputs in1, in2, sel
assign z = (a | b);
assign a = in1 & sel;
assign b = in2 & ~sel;
```
Some More Lexical Conventions

- The order of execution of the assign statements is unknown
- We must fake parallel execution... gates operate in parallel
- The assign statements ”fire” when the RHS variables change
- RHS = a, b, in1, in2, sel
- The values of a, b, and z are updated at the end of the timestep
- In the next time step if variables changed the next result is posted
- This repeats until no further changes occur
- Then...... time advances

```verilog
//two input mux, output is z, inputs in1, in2, sel
assign z = (a | b);
assign a = in1 & sel;
assign b = in2 & ~sel;
```