



Wahba



2

Verilog Operators

- Ver

W and 3



Relational Operators

- $a < b$ a less than b
- $a > b$ a greater than b
- $a \leq b$ a less than or equal to b
- $a \geq b$ a greater than or equal to b
- The result is:
 - 0 if the relation is false
 - 1 if the relation is true
 - x if any of the operands has unknown x bits
- Note: If a value is x or z, then the result of that test is false

Java



Arithmetic Operators

- Binary: +, -, *, /, % (the modulus operator)
- Unary: +, -
- Integer division truncates any fractional part
- The result of a modulus operation takes the sign of the first operand.
- If any operand bit value is the unknown value x, then the entire result value is x
- Register data types are used as unsigned values
- Negative numbers are stored in two's complement form

Java



Equality Operators

- $a == b$ a equal to b, including x and z
- $a != b$ a not equal to b, including x and z
- $a == b$ a equal to b, result may be unknown
- $a != b$ a not equal to b, result may be unknown
- Operands are compared bit by bit, with zero filling if the two operands do not have the same length
- For the $= =$ and $!=$ operators: the result is x, if either operand contains an x or a z
- For the $== =$ and $!= =$ operators: bits with x and z are included in the comparison and must match for the result to be true. The result is always 0 or 1.

Java



Logical Operators

- ! 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 unknown x bits

And



Bit-Wise Operators

- \sim negation
- $\&$ and
- $|$ inclusive or
- \wedge exclusive or
- $\wedge \sim$ or $\sim \wedge$ exclusive nor (equivalence)

- Computations include unknown bits, in the following way:

$$\sim x = x$$

$$0 \& x = 0$$

$$1 \& x = x \& x = x$$

$$1 | x = 1$$

$$0 | x = x | x = x$$

$$0 \wedge x = 1 \wedge x = x \wedge x = x$$

$$0 \wedge \sim x = 1 \wedge \sim x = x \wedge \sim x = x$$

- When operands are of unequal bit length, the shorter operand is zero filled in the most significant bit positions

IN AND



Reduction Operators

• &	and
• $\sim\&$	nand
•	or
• $\sim $	nor
• ^	xor
• $^{\sim}$ or \sim^{\wedge}	xnor

- Reduction operators are unary.
- They perform a bit-wise operation on a single operand to produce a single bit result.
- Reduction unary NAND and NOR operators operate as AND and OR respectively, but with their outputs negated.
- Unknown bits are treated as described before.

Nand



Shift Operators

- << 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.

Nand



Concatenation Operators

- Concatenations are expressed using the brace characters { and }, with commas separating the expressions within.
- Examples
 - {a, b[3:0], c, 4'b1001} // if a and c are 8-bit numbers, the results has 24 bits.
Unsized constant numbers are not allowed in concatenations.
- Repetition multipliers that must be constants can be used:
{3{a}} // this is equivalent to {a, a, a}
- Nested concatenations are possible:
{b, {3{c, d}}} // this is equivalent to {b, c, d, c, d, c, d}

IN and



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 whether `cond_expr` evaluates to true or false

Example:

`out = (enable) ? data : 8'bz; // Tri state buffer`

IN AND



Operator Precedence

- Unary, Multiply, Divide, Modulus
- Add, Subtract, Shift.
- Relation, Equality
- Reduction
- Logic
- Conditional

+ - ! ~ * / %

, - , <<, >>

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

&, !&, ^, ^~, |, ~|

&&, ||

?:

Java



Verilog Assignments

- Ver

W and 3



Non-Blocking Assignment

```
c <= a&b;    <= non-blocking  
              always @(posedge  
              clk)  
              begin  
                test_1 <= 1'b1;  
                test_2 <= test_1;  
                test_3 <= test_2;  
              end
```

- Used only inside an **always** or **initial** block.
- All these statements are executed in parallel. So every LHS will take the value of the RHS at the moment of entering the block.
- Here 1 will be entered into test_3 at the 3rd edge of the clock.
- LHS should always be a **reg** or a vector of reg.
- RHS can be of type **reg** or **wire**.

Non-Blocking Assignment



Blocking Assignment

```
c = 1'b0;      = blocking assignment  
  
always @(posedge  
clk)  
begin  
    test_1 = 1'b1;  
    test_2 = test_1;  
    test_3 = test_2;  
end
```

- Used **only** inside an **always** or **initial** block.
- The statements are executed in sequence, one after the other, exactly like software.
- Here 1 will be entered into test_3 when at the 1st edge of the clock.
- **LHS** should always be a **reg** or a vector of reg.
- **RHS** can be of type **reg** or **wire**.

Java



Continuous Assignment

```
assign c = ~a;
```

= blocking assignment



- Signals of type **wire** require **continuous assignment** of a value.
- Used when you connect gates together.
- **LHS** should always be a **wire** or a vector of wires. It can never be **reg**.
- **RHS** can be of type **reg** or **wire**.
- The assignment is always active. Whenever any operand on the RHS changes in value, LHS will be updated with the new value.

IN & OUT



Data Types

- Ver

Value



Verilog Types

Verilog Language has two primary data types:

- **Nets:** represents structural connections between components.
- **Registers:** represent variables used to store data.

Every signal has a data type associated with it:

- Explicitly declared with a declaration in your Verilog code.
- Implicitly declared with no declaration but used to connect structural building blocks in your code.
- Implicit declaration is always a net of type wire and is one bit wide.

Wire:

Wire data type is used in the continuous assignments or ports list. It is treated as a wire So it cannot hold a value. It can be driven and read. Wires are used for connecting different modules.

- Wires are used for connecting different elements.
- They can be treated as physical wires.
- They can be read or assigned.
- No values get stored in them.
- They need to be driven by either continuous assign statement or from a port of a module.

Reg:

Reg is a date storage element in Verilog. It's not an actual hardware register but it can store values. Registers retain their value until next assignment statement.

- Contrary to their name, regs do not necessarily correspond to physical registers.
- They represent data storage elements in Verilog.
- They retain their value till next value is assigned to them.



Port Connection Rules

- **Inputs** : internally must always be type *net*, externally the inputs can be connected to variable *reg* or *net* type.
- **Outputs** : internally can be type *net* or *reg*, externally the outputs must be connected to a variable *net* type.
- **Inouts** : internally or externally must always be type *net*, can only be connected to a variable *net* type.

