

## 1. Arithmetic Operators

These are like standard math operations.

- `+` : Addition (e.g., `a + b`) - Synthesizable.
- `-` : Subtraction (e.g., `a - b`) - Can also be used to negate a single number (e.g., `-5`). Synthesizable.
- `*` : Multiplication (e.g., `a * b`) - Synthesizable.
- `/` : Division (e.g., `a / b`) - Synthesizable for simple cases like dividing by powers of 2 (which become bit shifts). For complex division, it's often not directly synthesizable or requires specialized hardware.
- `%` : Modulus (remainder) (e.g., `a % b`) - Synthesizable for simple cases like modulus by powers of 2.
- `**` : Power (e.g., `a ** b`) - **Not synthesizable**. Used mainly for simulation or in testbenches.

## 2. Relational Operators

These compare two values and produce a single-bit result: 1 (true), 0 (false), or x (unknown if the comparison is uncertain due to x or z values).

- `>` : Greater than (e.g., `a > b`) - Synthesizable.
- `<` : Less than (e.g., `a < b`) - Synthesizable.
- `>=` : Greater than or equal to (e.g., `a >= b`) - Synthesizable.
- `<=` : Less than or equal to (e.g., `a <= b`) - Synthesizable.
- `==` : **Logical Equality** (e.g., `a == b`) - Synthesizable. **Important:** Returns x if any bit in a or b is x or z. This is usually what you want for hardware where x means "we don't know the value."
- `!=` : **Logical Inequality** (e.g., `a != b`) - Synthesizable. Returns x if any bit in a or b is x or z.
- `====` : **Case Equality** (e.g., `a ==== b`) - **Not synthesizable**. This performs a bit-for-bit comparison, including x and z. It will return 1 or 0, never x. Used almost exclusively in testbenches where you need to check exact bit patterns, including unknowns.
- `!==!` : **Case Inequality** (e.g., `a !== b`) - **Not synthesizable**. Compares all four states (0, 1, x, z). Used mainly in testbenches.

## 3. Logical Operators

These perform Boolean logic on entire expressions, treating any non-zero value as 1 (true) and zero as 0 (false). They always produce a single-bit result (1, 0, or x).

- `&&` : Logical AND (e.g., `cond1 && cond2`) - Synthesizable.
- `||` : Logical OR (e.g., `cond1 || cond2`) - Synthesizable.
- `!` : Logical NOT (e.g., `!condition`) - Synthesizable.

### Key Difference with Bitwise Operators:

- **Logical operators** (`&&`, `||`, `!`) work on the *truth value* of their operands. For example, `(4'b1010 && 4'b0000)` results in 0 because `4'b0000` is logically false.
- **Bitwise operators** (`&`, `|`, `~`, etc.) perform operations on *each individual bit* of their operands.

## 4. Bitwise Operators

These operate independently on each corresponding bit of the two operands. The result will have the same bit width as the widest operand.

- `~` : Bitwise NOT (e.g., `~my_data`) - Unary operator (operates on one value). Inverts each bit. Synthesizable.

- `&` : Bitwise AND (e.g., `a & b`) - Synthesizable.
- `|` : Bitwise OR (e.g., `a | b`) - Synthesizable.
- `^` : Bitwise XOR (e.g., `a ^ b`) - Synthesizable.
- `~^` or `~~` : Bitwise XNOR (e.g., `a ~^ b`) - Synthesizable.

## 5. Shift Operators

These move the bits of an operand left or right.

- `<<` : **Logical Left Shift** (e.g., `data << 2`) - Shifts bits to the left. Vacated bits on the right are filled with 0s. Synthesizable.
- `>>` : **Logical Right Shift** (e.g., `data >> 2`) - Shifts bits to the right. Vacated bits on the left are filled with 0s. Synthesizable.
- `<<<` : **Arithmetic Left Shift** (e.g., `signed_data <<< 2`) - For positive numbers, behaves like logical left shift. For negative numbers, can preserve sign in some contexts. Synthesizable.
- `>>>` : **Arithmetic Right Shift** (e.g., `signed_data >>> 2`) - Shifts bits to the right. Vacated bits on the left are filled with the **sign bit** (MSB) of the original number. **Crucial for signed numbers** to preserve their value. Synthesizable.

## 6. Concatenation and Replication Operators

These are powerful for manipulating bit vectors.

- **Concatenation (`{ , }`):** Combines multiple operands (wires, registers, or smaller numbers) into a single, wider vector. The order matters.
  - Example: `wire [7:0] combined_data = {high_nibble, low_nibble};` (If high\_nibble is 4 bits and low\_nibble is 4 bits, combined\_data becomes an 8-bit vector).
- **Replication (`{N{expression}}`):** Repeats an expression N times to create a wider vector. N must be a constant number.
  - Example: `wire [16:0] seventeen_ones = {17{1'b1}};` (Creates a 17-bit vector where all bits are 1).
  - Example: `wire [7:0] replicated_byte = {2{4'b1010}};` (Results in 8'b10101010).

These are extremely useful for tasks like bit extraction, padding, and implementing sign or zero extension.

## 7. Conditional Operator (Ternary Operator)

This is a three-operand operator that works like an "if-else" statement condensed into a single line. It's often used to describe multiplexers (MUXes).

- **Syntax:** `condition ? expression_if_true : expression_if_false;`
- **Example:** `assign out = (select_signal == 1'b1) ? input_one : input_zero;`
  - If select\_signal is 1, out will get the value of input\_one; otherwise, out will get the value of input\_zero.
- **Synthesizable:** Yes, it typically creates a multiplexer.