VHDL

HDL (Hardware Description Language)

- **Purpose**: Used to describe and model the behavior and structure of digital circuits.
- **Types**: Verilog, VHDL, and SystemVerilog are popular examples.
- **Usage**: Allows for simulation (testing digital circuit behavior) and synthesis (converting designs to hardware).
- **Applications**: Used in designing FPGAs (Field-Programmable Gate Arrays), ASICs (Application-Specific Integrated Circuits), microprocessors, etc.

Verilog

- **Purpose**: A popular HDL for modeling and simulating digital systems.
- **Syntax**: Similar to C, making it easier for software engineers to learn.
- **Key Features**:
  - Loosely typed (less strict than VHDL).
  - Supports both behavioral and structural modeling of hardware.
  - Commonly used in the U.S. and commercial applications.
- **Example**:

```
module AND_GATE (input A, input B, output Y);
    assign Y = A & B;
endmodule
```

C vs Verilog

Aspect	C (Software Language)	Verilog (Hardware Language)
Execution	Sequential execution.	Concurrent execution (multiple events happening in parallel).
Purpose	General-purpose programming.	Describing hardware behavior (digital circuits).
Target	CPU or software-based systems.	FPGAs, ASICs, or digital logic circuits.
Data Types	Standard data types (int, float, etc.).	Hardware-specific types (e.g., wires, registers).
Concurrency	Not inherently concurrent.	Supports concurrent processes.
Design Level	High-level, abstract programming.	Low-level, closer to hardware design.
Compilation	Compiled into machine code (software).	Synthesized into hardware logic (gates, flip-flops, etc.).

VHDL (VHSIC Hardware Description Language)

- **Full Form**: VHSIC (Very High-Speed Integrated Circuit) Hardware Description Language.
- **Purpose**: Used for modeling, simulation, and synthesis of digital circuits, similar to Verilog but with a different syntax and type system.
- **Key Features**:
  - Strongly typed (strict type-checking, more verbose than Verilog).
  - Popular in defense and European industries.

    - Supports modularity with packages and libraries.
- **Concurrency**: Models concurrent processes and has sequential execution blocks within processes.

Levels of Abstraction

In VHDL (VHSIC Hardware Description Language), there are different levels of abstraction that allow designers to model and describe hardware systems at varying degrees of detail. These levels help transition from high-level abstract design to low-level physical implementation. The four main levels of abstraction in VHDL are:

1. Behavioral Level (Algorithmic Level)

- **Purpose**: Describes *what* the system does without focusing on how it will be implemented in hardware.
- **Details**:
    - Focuses on algorithms and functionalities.
    - It uses high-level constructs such as `if-else` statements, loops, and processes to define the system behavior.
    - Does not concern itself with timing details or hardware components.
- **Example**: Describing an AND gate by its behavior (`Y <= A and B;`).
- **Use Case**: Early stages of design where the logic is more important than the physical structure.
- 2×1 MUX

```
always@(i0,i1,sel)
begin
if(sel)
    out=i1;
else
    out=i0;
end
```

2. Dataflow Level (RTL - Register Transfer Level)

- **Purpose**: Describes how data flows between different components of the system.
- **Details**:
    - Focuses on the transfer of data between registers and how operations are performed on data.
    - Uses signals, registers, and combinational logic to describe the circuit behavior.
    - Often used for synthesis into hardware, where the timing and flow of data are crucial.
- **Example**: Defining the data path between components and how operations take place on signals.
- **Use Case**: Intermediate-level description of how the data flows and how components interact in the system.
- e.g. - assign z=x&y

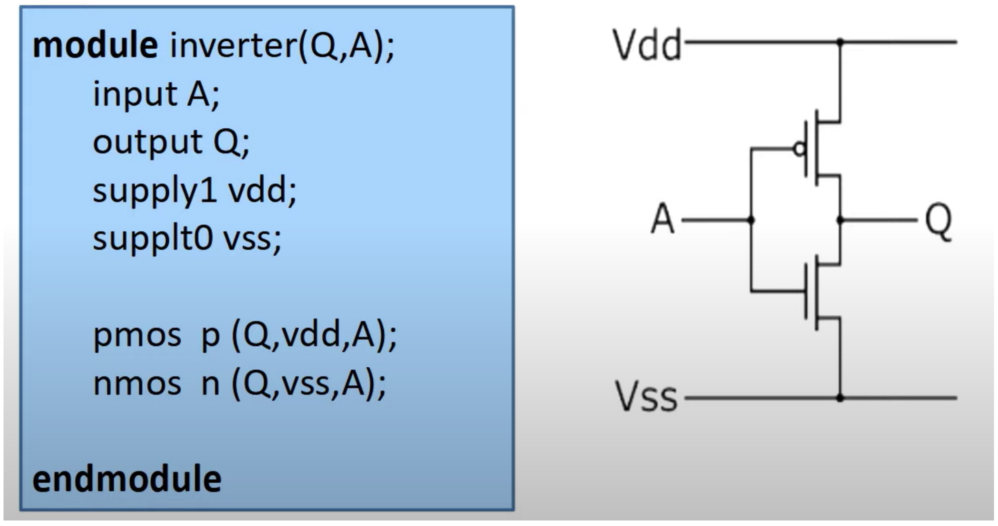
3. Structural Level (Gate Level)

- **Purpose**: Describes *how* the system is built using interconnected components (like gates and flip-flops).
- **Details**:
    - Focuses on the physical structure of the circuit, where individual components (AND, OR, NAND gates, etc.) and connections are explicitly defined.
    - Hierarchical in nature: larger designs are built from smaller components.
    - Used to describe the actual hardware that will be synthesized.
- **Example**: Interconnecting gates and modules to form a complete circuit.

- **Use Case**: Detailed implementation that is closer to the actual hardware, useful in the later stages of design or synthesis.
- **Syntax -** primitive_name instance_name (output, inputs)

4. Physical Level (Switch/Transistor Level)

- **Purpose**: Describes the system in terms of physical components such as transistors and switches.
- **Details**:
    - This level of abstraction gets down to the actual physical components, like MOSFETs and transistors.
    - Describes how the circuit will be laid out on a chip or board.
    - Rarely used in VHDL, as it's more related to physical design tools (used for layout, fabrication, etc.).
- **Use Case**: Generally handled by tools during the process of translating higher-level abstractions into physical circuits.
- **Syntax -** mos_name instance_name (output,data,control)



```
module inverter(Q,A);
    input A;
    output Q;
    supply1 vdd;
    supplt0 vss;

    pmos  p (Q,vdd,A);
    nmos  n (Q,vss,A);

endmodule
```

Modules

Modules are used to create reusable components, similar to how functions or classes work in software programming. A **module** in VHDL refers to the logical block of hardware described using an entity and architecture pair.

```
module module_name(x,y,z);
    input x,y;
    output z;
    statements;
    ...........;
    ...........;
endmodule
```

Module Instatiation

In VHDL, **module instantiation** refers to the process of using a previously defined module (entity + architecture) inside another module. This is similar to calling a function in programming, allowing for the reuse of existing components in larger designs. Each instance of a module is a separate, unique block of logic in the system.

💡

Simulation

Simulation is the process of testing the functionality of the hardware design in a virtual environment before physically implementing it. It helps to verify that the design behaves as expected under different conditions.

Synthesis

Synthesis is the process of translating high-level hardware descriptions (in VHDL or Verilog) into a low-level hardware implementation, typically in terms of gates or flip-flops. This step converts your design into something that can be physically implemented on a chip or programmable logic device (like an FPGA).

Data Type

Reg Data Type

The `reg` data type in Verilog is used to represent variables that store a value until explicitly assigned a new one. Despite the name, `reg` does not always correspond to a hardware register; it is a modeling construct used primarily in **procedural blocks** (`always` , `initial` , etc.).

Key Features:

- Holds a value until it is explicitly updated.
- Used in **procedural assignments** inside `always` or `initial` blocks.
- Can represent both combinational and sequential logic depending on usage.
- Default value: `x` **(unknown)**.

Syntax

```
reg [msb:lsb] variable_name;
```

Net Data Type

The `net` data type in Verilog is used to model connections between components, like wires in hardware. Nets **continuously reflect** the value being driven onto them, meaning they don't store values like `reg` .

Key Features:

- Represents physical connections (e.g., wires).
- Cannot hold a value; must always be driven by some source.
- Default value: `z` **(high impedance)** if undriven.
- Common types: `wire` , `tri` , `wand` , `wor` .

```
wire [msb:lsb] net_name;
```

Numbers

The general syntax for numbers in Verilog is:

```
[size]'[base][value]
```

- `size` : Optional. Specifies the number of bits for the literal.
    - Example: `4'b1010` (4 bits: 1010)

- `base`: Indicates the base of the number.
  - `'b` or `'B`: Binary (base 2)
  - `'o` or `'O`: Octal (base 8)
  - `'d` or `'D`: Decimal (base 10)
  - `'h` or `'H`: Hexadecimal (base 16)
- `value`: The actual number, written in the specified base.

Examples

Literal	Interpretation	Value (Decimal)
`4'b1010`	4-bit binary: `1010`	10
`8'd255`	8-bit decimal: `255`	255
`4'o12`	4-bit octal: `12`	10
`8'hFF`	8-bit hexadecimal: `FF`	255
`5'bz`	5-bit high-impedance	ZZZZZ
`4'bx`	4-bit unknown	XXXX

💡

1. Zero Extension

- **Zero extension** occurs when an **unsigned number** or a smaller variable is assigned to a larger-sized variable.
- Extra bits are filled with `0`.

2. `x` (Unknown) Extension

- `x` represents an unknown value (indeterminate or uninitialized state).
- When a smaller `x` is extended to a larger size, all extra bits are filled with `x`.

3. `z` (High-Impedance) Extension

- `z` represents a high-impedance state (e.g., an unconnected wire).
- When a smaller `z` value is extended to a larger size, all extra bits are filled with `z`.

Different Reg types

1. `integer`

The `integer` type is used for general-purpose **signed integers**. Unlike `reg`, it is not restricted to binary values and can represent **positive and negative** values.

Key Characteristics:

- Typically 32 bits wide (simulator-dependent).
- Signed by default, so it can represent negative values.
- Used for **loop counters**, **indices**, and **calculations** where signed numbers are required.

```
integer variable_name;
```

---

2. `real`

The `real` data type is used for **floating-point numbers** in Verilog. It is essential for simulations involving non-integer calculations, although it is **not synthesizable** (cannot be used in hardware implementation).

Key Characteristics:

- Represents **double-precision floating-point values** (64 bits).
- Useful for simulations requiring non-integer precision, such as **timing** and **analog behavior**.
- Cannot be used in hardware design or synthesis; purely for testbenches.

```
real variable_name;
```

---

3. `time`

The `time` data type is specifically designed to hold **simulation time values**. It is an unsigned integer, typically **64 bits wide**, that can store large values to represent time in simulations.

Key Characteristics:

- **Unsigned integer** (64 bits), storing simulation time units.
- Useful for tracking or measuring simulation time intervals.
- Typically used with `$time`, `$stime`, or `$realtime` system functions.
- Not synthesizable.

```
time variable_name;
```

Vectors, Arrays & Memories

1. Vectors

A **vector** is a single-dimensional collection of bits that is declared with a bit range. It is commonly used to represent **multi-bit signals**, such as buses, registers, or memory words.

Characteristics:

- Represents a **single signal** with multiple bits.
- Can be indexed to access individual bits (bit-select) or subsets of bits (part-select).
- Declared using square brackets `[]` to specify the range.

Syntax:

```
type [msb:lsb] variable_name;
```

- `type`: Typically `reg`, `wire`, or `logic`.
- `msb` **and** `lsb`: Most significant bit and least significant bit.

Examples:

```
reg [7:0] byte_data;    // 8-bit vector
wire [15:0] address;    // 16-bit vector
logic [3:0] nibble;     // 4-bit vector
```

Usage:

- **Bit-select**: Access a single bit.

- **Part-select**: Access a subset of bits.

2. Arrays

An **array** is a collection of **multiple vectors** or **scalar values**, indexed by integers. Arrays allow for storing and accessing multiple elements with the same data type.

Characteristics:

- Represents **multiple instances** of a signal (vector or scalar).
- Supports **multi-dimensional declarations**.
- Used for **memory modeling** or **lookup tables**.

Syntax:

```
type [msb:lsb] array_name [size];
```

- `type`: Typically `reg`, `wire`, or `logic`. Also allow **integer**, **time**, **real**.
- `[msb:lsb]`: Defines the bit width of each element (optional).
- `[size]`: Defines the number of elements in the array.

Examples:

```
reg [7:0] memory [0:15]; // 16x8 array (16 elements, each 8 bits wide)
reg data [0:31];         // 32x1 array (32 elements, each 1 bit wide)
```

Usage:

- Access an element using an **index.**
- Arrays can be **multi-dimensional.**

Feature	**Vector**	**Array**
Purpose	Represents a multi-bit signal.	Represents a collection of signals.
Dimension	Single-dimensional.	Single or multi-dimensional.
Access	Bit-select or part-select.	Accessed via indices.
Usage	For buses, registers, and signals.	For memory, lookup tables, or data sets.

3. Memories

A memory in Verilog is declared as a
1-dimensional array of vectors. Each element in the memory array represents a **word**, and its width determines the **word size**.

Syntax:

```
type [msb:lsb] memory_name [depth];
```

- `type`: Usually `reg` (used for storage).
- `[msb:lsb]`: Defines the size (number of bits) of each word.
- `[depth]`: Defines the number of words in the memory.

Memory Access

You can write to a memory location by assigning a value using its index:

```
memory[0] = 8'hFF; // Write 8'hFF to the first memory word
```

Reading from Memory:

You can read a value from a memory location using its index:

```
data = memory[0]; // Read the value from the first memory word
```

Parameters & Strings

1. Parameters

Parameters are constants that are defined at compile time and are typically used to make your code more flexible and reusable. They are most commonly used for defining constants such as bit widths, memory sizes, or other configuration values that may need to be adjusted without changing the rest of the code.

Characteristics of Parameters:

- **Compile-time constants**.
- Cannot be changed during simulation or synthesis.
- Can have default values or can be overridden during module instantiation.

Syntax:

```
parameter parameter_name = value;
```

Example

```
module adder #(parameter WIDTH = 8);  // Parameter with default value of 8
  input [WIDTH-1:0] a, b;   // Input signals with WIDTH size
  output [WIDTH-1:0] sum;   // Output signal with WIDTH size

  assign sum = a + b;  // Adder logic
endmodule
```

---

2. Strings

In Verilog, strings are used primarily for simulation and debugging purposes. They are used to store and manipulate sequences of characters.

Characteristics of Strings:

- Verilog strings are **arrays of characters**.
- Typically used with system tasks like `$display`, `$monitor`, and `$write` for debugging.
- Verilog strings are **not synthesizable**; they are used for simulation purposes only.

Declaring Strings:

In Verilog, strings are declared as arrays of type `reg` with the character data type `8` (for ASCII characters).

Syntax:

```
reg [7:0] string_name [string_length:0];
```

Examples:

1. **Basic String Declaration**:

```
reg [7:0] my_string [0:15]; // Declare a string with 16 characters (16 ASCII charac
ters)
```

2. **Assigning Values to Strings**:
   Strings can be assigned character-by-character or using system tasks.

```
my_string = "Hello, Verilog!";
```

3. **Using `$display` with Strings**:
   Verilog allows you to print strings to the console using the
   `$display` task.

```
initial begin
  $display("String value is: %s", my_string);  // Print the string
end
```

Operators

1. Arithmetic Operators

Arithmetic operators perform mathematical operations on numeric values.

Operators:

- `+` : Addition
- : Subtraction
- : Multiplication
- `/` : Division
- `%` : Modulus (remainder after division)

---

2. Logical Operators

Logical operators are used to perform logical operations, typically on **single bits**.

Operators:

- `&&` : Logical AND
- `||` : Logical OR
- `!` : Logical NOT

---

3. Bitwise Operators

Bitwise operators work on each bit of the operands individually.

Operators:

- `&` : Bitwise AND
- `|` : Bitwise OR
- `^` : Bitwise XOR
- `~` : Bitwise NOT (inverts all bits)
- `~&` : NAND (NOT AND)

- `~|` : NOR (NOT OR)
- `^~` : XNOR (NOT XOR)

4. Equality Operators

Equality operators are used to compare two values for equality or inequality.

Operators:

- `==` : Equal to
- `!=` : Not equal to

5. Relational Operators

Relational operators compare two values in terms of their relative size.

Operators:

- `>` : Greater than
- `<` : Less than
- `>=` : Greater than or equal to
- `<=` : Less than or equal to

6. Reduction Operators

Reduction operators perform bitwise operations across all bits of a vector.

Operators:

- `&` : AND reduction (AND all bits)
- `|` : OR reduction (OR all bits)
- `^` : XOR reduction (XOR all bits)
- `~&` : NAND reduction
- `~|` : NOR reduction
- `^~` : XNOR reduction

7. Shift Operators

Shift operators shift bits of a value to the left or right.

Operators:

- `<<` : Logical left shift (shifts bits to the left, filling with 0s)
- `>>` : Logical right shift (shifts bits to the right, filling with 0s)
- `<<<` : Arithmetic left shift (preserves sign bit in signed numbers)
- `>>>` : Arithmetic right shift (preserves sign bit in signed numbers)

8. Concatenation Operator

Concatenation combines multiple bits or vectors into a single vector.

Operator:

- `{}` : Concatenation

Example:

```
reg [3:0] a, b;
reg [7:0] result;
initial begin
  a = 4'b1010; // 10
  b = 4'b1100; // 12
  result = {a, b}; // Concatenate a and b to form 8-bit result (result = 8'b10101100)
end
```

You can also concatenate multiple values together:

```
result = {a, b, 4'b0001}; // Concatenate a, b, and 4'b0001
```

9. Conditional Operator (Ternary Operator)

The conditional operator is used to make decisions based on a condition.

Syntax:

```
condition ? expression1 : expression2;
```

If the condition is true, `expression1` is selected, otherwise `expression2` is selected.

Example:

```
reg [7:0] a, b, result;
initial begin
  a = 8'b00001111; // 15
  b = 8'b11110000; // 240
  result = (a > b) ? a : b; // If a > b, result = a, else result = b
end
```

Gate Primitives

Gate primitives are the built-in logical gates provided by Verilog. These represent simple combinational circuits and are directly supported in Verilog. Gate primitives are used for basic logic operations, such as AND, OR, NOT, and more complex functionalities like tri-state buffers and enable/disable behavior.

1. Basic Gate Primitives

Verilog supports basic logic gates like **AND**, **OR**, **NOT**, **BUF**, etc. These gates are modeled as **built-in primitives** and have predefined functionality.

Syntax for Basic Gates:

```
gate_type [instance_name] (output, input1, input2, ...);
```

Supported Gates:

Gate Type	Description	Inputs
`and`	Logical AND	2+
`or`	Logical OR	2+
`nand`	Logical NAND	2+
`nor`	Logical NOR	2+

`xor`	Logical XOR	2+
`xnor`	Logical XNOR	2+
`not`	Logical NOT (Inverter)	1
`buf`	Buffer	1

Example: Basic Gates

```
// AND Gate
 and u1 (out_and, a, b); // out_and = a & b

 // OR Gate
 or u2 (out_or, a, b); // out_or = a | b

 // NOT Gate
 not u3 (out_not, a); // out_not = ~a
```

2. Buffer (`buf`) and NOT (`not`) Gates

Buffers and inverters are also gate primitives, primarily used to pass or invert signals.

Syntax:

- **Buffer** (`buf`): `buf [instance_name] (output, input);`
- **NOT Gate** (`not`): `not [instance_name] (output, input);`

Example: Buffer and NOT Gate

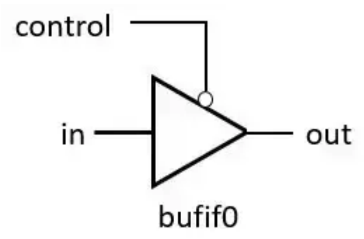
```
// Buffer: Passes input to output
  buf u1 (out_buf, a); // out_buf = a

  // NOT Gate: Inverts input
  not u2 (out_not, a); // out_not = ~a
```

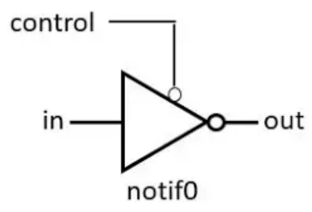
3. Tri-State Gates (`bufif` , `notif`)

Tri-state gates allow for conditional driving of outputs. They are useful for modeling shared buses and enable/disable functionality.

Types:

- `bufif1` : Passes input to output when control signal is **1**.
- `bufif0` : Passes input to output when control signal is **0**.
- `notif1` : Inverts input to output when control signal is **1**.
- `notif0` : Inverts input to output when control signal is **0**.





Syntax:

```
bufif1 [instance_name] (output, input, control);
bufif0 [instance_name] (output, input, control);
notif1 [instance_name] (output, input, control);
notif0 [instance_name] (output, input, control);
```

Example: Tri-State Gates

```
  // Buffer if Control is 1
  bufif1 u1 (out_bufif1, a, enable);

  // Buffer if Control is 0
  bufif0 u2 (out_bufif0, a, enable);

  // NOT if Control is 1
  notif1 u3 (out_notif1, a, enable);

  // NOT if Control is 0
  notif0 u4 (out_notif0, a, enable);
```

Explanation:

- When `enable` is **1**:
  - `bufif1` passes `a` to the output.
  - `notif1` inverts `a` and passes it to the output.
- When `enable` is **0**:
  - `bufif0` passes `a` to the output.
  - `notif0` inverts `a` and passes it to the output.

Gate Delay

Delays in Verilog are used to model the timing characteristics of gates, such as the time it takes for a signal to propagate through a gate or change its state. Verilog supports three types of delays:

1. **Rise Delay**: Time taken for a signal to transition from `0` to `1`.
2. **Fall Delay**: Time taken for a signal to transition from `1` to `0`.
3. **Turn-off Delay**: Time taken for a signal to transition to a high-impedance (`z`) state.

Syntax for Delays

```
gate_type #(rise_delay, fall_delay, turn_off_delay) [instance_name] (o/p,i/p1, i/p2);
```

- **`rise_delay`**: Time for the gate to output a `1`.
- **`fall_delay`**: Time for the gate to output a `0`.
- **`turn_off_delay`**: Time for the gate to output a `z` (tri-state or undefined state).

Continuous Assignment

A **continuous assignment** in Verilog is used to drive values to `wire` data types. This is typically done using the `assign` statement, which defines a continuous relationship between a signal and an expression.

Whenever any variable in the expression changes, the output is automatically updated to reflect the new value.

Syntax

```
assign <LHS> = <expression>;
```

LHS is of net type, whereas RHS is of reg or net type.

```
assign c = a & b; // c is continuously updated as the AND of a and b
```

Delays

Delays in Verilog are used to model the time taken for a signal to propagate or for a transition to occur. They can be specified in different contexts, such as assignments, continuous assignments, or during net declaration. Here's an overview of the types of delays:

---

1. Regular Assignment Delays

Delays in procedural assignments (inside `initial` or `always` blocks) specify the time after which the assignment takes effect.

Syntax

```
#<delay> <assignment>;
```

Example

```
#5 b = a;  // Assigns the value of `a` to `b` after 5 time units
```

Explanation

- The value of `a` is assigned to `b` after a delay of **5 time units**.

---

2. Implicit Continuous Assignment Delays

Continuous assignments (`assign` statements) can include delays, which specify the time taken for a net to update after any change in the driving expression.

Syntax

```
assign #<delay> <net> = <expression>;
```

Example

```
assign #10 c = a & b; // `c` updates 10 time units after `a` or `b` changes
```

Explanation

- Any change in `a` or `b` causes `c` to be updated after a delay of **10 time units**.

---

3. Net Declaration Delays

Delays can be specified during net declaration, which apply to all assignments driving that net. This models the intrinsic delay of the net itself.

Syntax

```
<net_type> #<delay> <net_name>;
```

Example

```
  wire #5 a, b, c;

  assign c = a & b; // Output `c` will be updated 5 time units after any change
```

Explanation

- The **delay of 5 time units** applies to the net itself, regardless of the driving logic.

Key Differences

Type	When Specified	Scope of Delay	Use Case
Regular Assignment Delays	Inside procedural blocks	Affects only that particular assignment	Sequential or procedural logic
Continuous Assignment Delays	With `assign` keyword	Affects the specific continuous assignment	Combinational logic
Net Declaration Delays	During net declaration	Affects all drivers of that net	Models intrinsic net delay

Structural and Procedural Constructs

Verilog supports **structural** and **procedural** constructs for modeling hardware. Here, we focus on **initial statements**, **always blocks**, **procedural assignments**, and **non-blocking assignments**.

---

1. Structural Procedure: Initial Statement

The `initial` block is a procedural construct that executes **only once** at the start of the simulation. It is typically used for initializing values or defining testbench behavior.

Syntax

```
initial begin
  // Code block
end
```

Key Features

- Executes only **once** during simulation.
- Executes sequentially inside the block.
- Commonly used in **testbenches** to apply stimuli.

Example

```
module initial_example;
  reg a, b;

  initial begin
    a = 0;
    b = 1;
    #5 a = 1; // Changes value after 5 time units
    #10 b = 0; // Changes value after 10 time units
  e
endmodule
```

2. Structural Procedure: Always Statement

The `always` block executes **repeatedly** based on a sensitivity list or event trigger. It is used for modeling **combinational** or **sequential logic**.

Syntax

```
always @(<sensitivity list>) begin
  // Code block
end
```

Key Features

- Executes whenever the **sensitivity list** is triggered.
- Can model combinational logic (use all inputs in the sensitivity list) or sequential logic (use `posedge` or `negedge`).

Example: Combinational Logic

```
module always_comb_example;
  reg a, b;
  wire c;

  always @(a or b) begin
    c = a & b; // Executes whenever a or b changes
  end
endmodule
```

Example: Sequential Logic

```
module always_seq_example;
  reg clk, d;
  reg q;

  always @(posedge clk) begin
    q <= d; // Executes at every positive clock edge
  end
endmodule
```

3. Procedural Assignment

Procedural assignments are used inside `initial` or `always` blocks to assign values to variables (`reg`, `integer`, `real`, etc.). There are two types of procedural assignments:

- **Blocking Assignment** (`=`): Executes **immediately**, blocking subsequent statements until completion.
- **Non-Blocking Assignment** (`<=`): Executes **concurrently**, allowing subsequent statements to execute immediately.

Key Differences

Feature	Blocking Assignment (`=`)	Non-Blocking Assignment (`<=`)
Execution Type	Sequential	Concurrent

Usage	Combinational logic, assignments within the same clock cycle	Sequential logic (flip-flops)
Blocking Behavior	Blocks subsequent code until completed	Does not block subsequent code

4. Blocking Assignment

Syntax

```
<variable> = <expression>;
```

Example

```
module blocking_example;
  reg a, b, c;

  always @(*) begin
    a = 1;
    b = a; // This executes only after `a` is updated
    c = b; // This executes only after `b` is updated
  end
endmodule
```

Simulation Timeline

- `a` is updated first, then `b`, and finally `c`. Each step **blocks** the next.

5. Non-Blocking Assignment

Syntax

```
<variable> <= <expression>;
```

Example

```
module nonblocking_example;
  reg a, b, c;

  always @(posedge clk) begin
    a <= 1; // Updates in the next clock cycle
    b <= a; // Uses the old value of `a` (before the clock edge)
    c <= b; // Uses the old value of `b` (before the clock edge)
  end
endmodule
```

Simulation Timeline

- All assignments (`a`, `b`, `c`) are evaluated concurrently at the same clock edge, but updates happen in the **next clock cycle**.

6. Use Cases

Construct	**Use Case**
Initial Statement	Testbenches, one-time initialization
Always Statement	Combinational or sequential logic modeling

Blocking Assignment	Combinational logic
Non-Blocking Assignment	Sequential logic, especially in flip-flops

Delays in Procedural Assignments

Procedural delays in Verilog control the timing of updates in procedural blocks (inside `initial` or `always`). They are classified as:

1. **Regular Delay**
2. **Intra-Assignment Delay**

---

1. Regular Delay

A **regular delay** introduces a delay before the execution of the assignment. It applies to the **entire assignment statement**.

Syntax

```
#<delay> <variable> = <expression>;
```

Key Features

- The delay occurs **before** the assignment is performed.
- The assignment itself is instantaneous after the delay.

Example

```
module regular_delay_example;
  reg a, b;

  initial begin
    a = 1;       // Immediate assignment
    #5 b = a;    // Delay of 5 time units before `b` is assigned the value of `a`
  end
endmodule
```

Simulation Timeline

- `a` is assigned `1` immediately.
- After **5 time units**, `b` is assigned the value of `a`.

---

2. Intra-Assignment Delay

An **intra-assignment delay** introduces a delay **during the execution** of the assignment. The expression on the right-hand side is evaluated immediately, but the update to the left-hand side occurs after the specified delay.

Syntax

```
<variable> = #<delay> <expression>;
```

Key Features

- The **right-hand side** of the assignment is evaluated **immediately**.
- The **left-hand side** update occurs **after the delay**.

Example

```
module intra_assignment_delay_example;
  reg a, b;

  initial begin
    a = 1;         // Immediate assignment
    b = #5 a;      // Delay of 5 time units before assigning the evaluated value of `a
` to `b`
  end
endmodule
```

Simulation Timeline

- `a` is assigned `1` immediately.
- After **5 time units**, `b` is updated to the evaluated value of `a`.

Key Differences Between Regular and Intra-Assignment Delays

Aspect	**Regular Delay**	**Intra-Assignment Delay**
When the Delay Occurs	Before the entire assignment	During the assignment
Evaluation Timing	Both sides are delayed	Right-hand side is evaluated immediately
Update Timing	Left-hand side is updated after delay	Left-hand side is updated after delay
Example	`#5 b = a;`	`b = #5 a;`

Block Statement

1. Sequence Block

A **sequence block** groups multiple statements that will be executed **sequentially** (one after another), just like the execution in an `initial` or `always` block. It's commonly used when modeling sequential logic or scenarios where specific timing or order of operations matters.

Syntax

```
begin
  // statement 1;
  // statement 2;
  // statement 3;
end
```

Key Features

- Statements inside a **sequence block** are executed **sequentially**.
- The execution is similar to how procedural code behaves inside `initial` or `always` blocks.
- Used for modeling simple sequential logic or setting up conditions that must execute in order.

---

2. Parallel Block

A **parallel block** allows multiple statements to execute **concurrently**. This is particularly useful in Verilog when modeling hardware behavior that can operate in parallel, such as describing combinational logic or parallel operations.

Syntax

```
fork
  // statement 1;
  // statement 2;
  // statement 3;
join
```

Key Features

- The statements inside a **parallel block** (created using `fork...join`) are executed **concurrently**, meaning they start executing at the same time.
- The `join` statement at the end ensures that the block finishes after all the parallel statements are done.

Example

```
module parallel_block_example;
  reg a, b, c;

  initial begin
    a = 0;
    b = 1;
    c = 0;

    // Parallel block - statements run concurrently
    fork
      a = 1;  // executed concurrently with other statements
      b = 0;
      c = 1;
    join
  end
endmodule
```

Explanation

- The values of `a`, `b`, and `c` are set to `0`, `1`, and `0`, respectively, initially.
- Inside the `fork...join` block, the assignments to `a`, `b`, and `c` occur **concurrently**. All assignments are performed at the same simulation time.

Key Differences Between Sequence and Parallel Blocks

Aspect	**Sequence Block (`begin...end`)**	**Parallel Block (`fork...join`)**
Execution	Executes statements **sequentially**	Executes statements **concurrently**
Use Case	Used when order of execution matters	Used for parallel processes or events
Control Flow	Follows normal sequential flow	All statements start at the same time
Simulation Timing	Each statement completes before the next starts	All statements execute simultaneously

Conditional Statement

1. `if` Only

The `if` statement evaluates a condition, and if the condition is **true**, it executes a block of code. If the condition is **false**, the block of code is skipped.

Syntax

```
if (condition) begin
  // Statements to execute if condition is true
end
```

2. `if-else`

The `if-else` statement provides an **alternative block** of code that executes if the condition is **false**. It’s useful when you want to specify one action for a **true** condition and a different action for a **false** condition.

Syntax

```
if (condition) begin
  // Statements to execute if condition is true
end else begin
  // Statements to execute if condition is false
end
```

3. `if-else if`

The `if-else if` ladder allows you to check multiple conditions sequentially. It is used when you have more than two possible outcomes. Each `else if` block provides an additional condition to check.

Syntax

```
if (condition1) begin
  // Statements to execute if condition1 is true
end else if (condition2) begin
  // Statements to execute if condition2 is true
end else begin
  // Statements to execute if all conditions are false
end
```

Multiway Branching

`case` Statement

The `case` statement is more suited for **multiway branching** when you have several **discrete, non-overlapping values** to check. It's more efficient and easier to read when dealing with multiple conditions based on a **single expression**.

Syntax

```
case (expression)
  value1: begin
    // Statements to execute if expression == value1
  end
  value2: begin
    // Statements to execute if expression == value2
  end
```

Looping Constructs

In Verilog, loops allow you to execute a block of code repeatedly. There are four primary types of looping constructs in Verilog:

1. `for` **Loop**
2. `while` **Loop**
3. `repeat` **Loop**
4. `forever` **Loop**

Each type of loop has specific use cases, and the choice depends on how many iterations you need and the condition under which the loop should terminate.

---

1. `for` Loop

The `for` **loop** is used when you know the number of iterations in advance. It is typically used for **count-controlled loops**, where you specify the **initialization**, **condition**, and **iteration** all in one line.

Syntax

```
for (initialization; condition; iteration) begin
  // Statements to execute
end
```

- **Initialization**: This part is executed only once, before the loop starts.
- **Condition**: The loop continues as long as this condition is true.
- **Iteration**: This part is executed at the end of each iteration.

Example

```
module for_example;
  reg [3:0] i;
  reg [7:0] result;

  initial begin
    result = 0;
    for (i = 0; i < 8; i = i + 1) begin
      result = result + i;  // Add the value of i to result
    end
  end
endmodule
```

Explanation

- This loop starts with `i = 0` and runs until `i < 8`.
- The `result` accumulates the sum of the values of `i` (from 0 to 7).

---

2. `while` Loop

The `while` **loop** continues as long as the **condition** is true. It is typically used when you don't know how many iterations are needed in advance but want to keep looping until a specific condition is met.

Syntax

```
while (condition) begin
  // Statements to execute
end
```

- **Condition**: The loop continues as long as the condition evaluates to true.

Example

```
module while_example;
  reg [3:0] count;
  reg [7:0] result;

  initial begin
    count = 0;
    result = 0;
    while (count < 8) begin
      result = result + count;  // Add the value of count to result
      count = count + 1;        // Increment count
    end
  end
endmodule
```

Explanation

- The loop starts with `count = 0` and runs as long as `count < 8`.
- The value of `result` is the sum of `count` from 0 to 7.

Key Points

- The `while` loop is useful when the number of iterations is **not known** in advance but depends on a condition being satisfied.

---

3. `repeat` Loop

The `repeat` **loop** is similar to the `for` loop, but the number of iterations is specified as an **expression** that is evaluated only once, at the start of the loop. This makes the `repeat` loop ideal for fixed iterations that are determined at compile time.

Syntax

```
repeat (number_of_iterations) begin
  // Statements to execute
end
```

- `number_of_iterations`: The number of times the loop should execute.

Example

```
module repeat_example;
  reg [3:0] i;
  reg [7:0] result;

  initial begin
    result = 0;
    repeat (8) begin
      result = result + i;  // Add the value of i to result
      i = i + 1;            // Increment i
    end
  end
endmodule
```

Explanation

- The loop will repeat 8 times (since `repeat(8)` is specified).
- In each iteration, `i` is added to `result`, and `i` is incremented.

Key Points

- The number of iterations in the `repeat` loop is **fixed** and evaluated before the loop starts, making it suitable for known iteration counts.

---

4. `forever` Loop

The `forever` **loop** is an infinite loop. It runs endlessly unless there is a break condition or the process is explicitly terminated (like using `disable` or `$stop` in a testbench). This loop is often used in **simulation environments** or for creating **clocking** or **event-driven behavior**.

Syntax

```
forever begin
  // Statements to execute
end
```

Example

```
module forever_example;
  reg clock;

  initial begin
    clock = 0;
    forever begin
      #5 clock = ~clock;  // Toggle the clock every 5 time units
    end
  end
endmodule
```

Explanation

- This is a **clock generation** example. The `clock` signal is toggled every 5 time units.
- The `forever` loop runs infinitely, constantly toggling the `clock` signal.

Key Points

- The `forever` loop is ideal for generating continuous signals or running **endless tasks** like **clock generation** in simulations.
- Use `forever` with caution, as it will **never terminate** unless explicitly stopped.

Comparison of Looping Constructs

Aspect	`for` Loop	`while` Loop	`repeat` Loop	`forever` Loop
Use Case	When the number of iterations is known in advance.	When the number of iterations is not known, but a condition is checked.	When you need a fixed number of iterations (known at the start).	For infinite loops (e.g., continuous events).
Termination Condition	Checked before each iteration (fixed number of iterations).	Checked before each iteration (based on a condition).	Fixed iteration count (checked once).	Runs forever unless explicitly stopped.
Typical Use	Iterating over a fixed range.	Iterating while a condition is true.	Repeating a block a specific number of times.	Clock generation or simulation-driven

				events.
Example Use Case	Iterating through an array.	Waiting for a specific event or condition.	Repeating a task a fixed number of times.	Continuously toggling a signal.

System Tasks in Verilog

Verilog provides several **system tasks** that assist in simulation control, time monitoring, and debugging. These tasks are built into the Verilog language and are widely used during simulation and debugging. System tasks can be categorized into the following groups:

1. **Internal Variable Monitoring System Tasks**
2. **Simulation Control Tasks**
3. **Simulation Time-Related Tasks**

---

i. Internal Variable Monitoring System Tasks

These system tasks are primarily used for monitoring and displaying the values of internal variables and signals during simulation. These tasks are essential for debugging and tracking the behavior of a design.

1. `$display`

The `$display` system task is used to print text or the values of variables to the simulation output (typically the console or terminal). It can print formatted output.

Syntax

```
$display("text", expression1, expression2, ...);
```

- `text`: A string of text to be printed.
- `expression1, expression2, ...`: Variables or expressions whose values are printed.

Example

```
$display("The value of signal is: %b", signal);
```

This will print the value of `signal` in binary format.

2. `$monitor`

The `$monitor` system task is similar to `$display`, but it continuously monitors and prints the values of specified signals whenever any of them change.

Syntax

```
$monitor("text", expression1, expression2, ...);
```

- This task continuously outputs the current values of the variables whenever they change.

Example

```
$monitor("time = %0t, signal1 = %b, signal2 = %b", $time, signal1, signal2);
```

This will display the time and values of `signal1` and `signal2` whenever any of these signals change.

3. `$strobe`

The `$strobe` task is similar to `$monitor`, but it only prints the value of the signals at the **end of the current simulation time step** (not when the signal changes).

Syntax

```
$strobe("text", expression1, expression2, ...);
```

- This is useful for capturing the values of signals after all assignments in a simulation time step have been completed.

Example

```
$strobe("At time %0t, signal1 = %b, signal2 = %b", $time, signal1, signal2);
```

---

ii. Simulation Control Tasks

These system tasks control the simulation flow, allowing you to pause, stop, or restart simulations, among other tasks.

1. `$finish`

The `$finish` system task ends the simulation. It halts the simulation after completing the current time step.

Syntax

```
$finish
```

- This is typically used when the simulation has reached a desired state, or when you want to force the simulation to stop.

Example

```
if (done) begin
  $finish;  // Ends the simulation when 'done' is true
end
```

2. `$stop`

The `$stop` system task suspends the simulation, allowing you to interact with the simulation environment, such as inspecting variables and stepping through the simulation. It's often used for debugging.

Syntax

```
$stop;
```

- This pauses the simulation at the current point, and the simulator typically waits for user input to resume.

3. `$restart`

The `$restart` system task restarts the simulation from the beginning. This is useful when you want to restart the simulation after a particular state has been reached or after a test case has been executed.

- It resets the simulation and starts from the initial conditions.

Example

```
if (failure_detected) begin
  $restart;  // Restart the simulation if a failure occurs
end
```

4. `$display` vs. `$monitor` vs. `$strobe`

- `$display`: Prints messages once at the point of execution.
- `$monitor`: Continuously monitors and prints when a signal changes.
- `$strobe`: Prints messages at the end of the time step, after all assignments.

---

iii. Simulation Time-Related Tasks

These tasks provide access to the simulation time and allow manipulation of time for debugging and control purposes.

1. `$time`

The `$time` system task returns the **current simulation time**. This is a constant system task that outputs the simulation time in the format of the time resolution being used.

- It returns the current time in **simulation time units**.

Example

```
$display("Current simulation time: %0t", $time);
```

2. `$stime`

The `$stime` system task returns the **current simulation time** as an integer (in time units defined by the simulator).

- This provides the simulation time in **integer format**.

Example

```
$display("Simulation time in integer: %0d", $stime);
```

3. `$now`

The `$now` system task is another time-related task that provides the **current simulation time**, but it returns the time in a **more precise format**.

- It returns the simulation time as a floating-point value, which is useful for high-precision applications.

Example

```
$display("Current time is: %0f", $now);
```

4. `$delay`

The `$delay` system task inserts a delay in the simulation time, allowing you to simulate time lags between events.

- `delay_time`: Time to delay the current execution.

Example

```
$delay(10);  // Delay simulation for 10 time units
```

5. `$realtime`

The `$realtime` system task returns the current simulation time as a **real number** (floating-point value), which is useful for real-time applications.

Syntax

- This returns the simulation time as a **real number**.

Example

```
$display("Real-time simulation time: %0f", $realtime);
```

Compiler Directive

1. `define` (Macro Definitions)

The `define` directive is used to define a macro. A macro can be a constant or an expression that can be replaced in the code during compilation. This is similar to defining constants or global variables.

Syntax:

```
`define MACRO_NAME value
```

- `MACRO_NAME`: The name of the macro.
- `value`: The value or expression that the macro will be replaced with.

Example:

```
`define WIDTH 8  // Defines a constant macro for width

module my_module;
  reg [`WIDTH-1:0] data;  // Use of the macro 'WIDTH'
endmodul
```

---

2. `include`

The `include` directive is used to include the contents of another Verilog file at the point where the directive is used. This is useful for reusing code in multiple modules.

Syntax:

```
`include "filename.v"
```

- `filename.v`: The name of the Verilog file to include.

Example:

```
`include "constants.v"  // Include constants from another file

module my_module;
  // The constants defined in "constants.v" will be available here
endmodule
```

In this example, the contents of `constants.v` will be included in the current file, making any defined macros, parameters, or other code available.

---

3. `timescale`

The `timescale` directive is used to specify the time unit and time precision for the simulation. It sets the time scale for the simulation, affecting how time values are interpreted in the Verilog code.

Syntax:

```
`timescale time_unit / time_precision
```

- `time_unit`: The time unit (e.g., `1ns`, `1ps`).
- `time_precision`: The precision of the time (e.g., `1ps`, `1fs`).

Example:

```
`timescale 1ns / 1ps  // Set time unit to 1ns and precision to 1ps

module my_module;
  reg clk;
  initial begin
    clk = 0;
    #5 clk = 1;
  end
endmodule
```

This directive specifies that time in the module will be interpreted in nanoseconds with a precision of picoseconds.

Tasks & Functions

In Verilog, **tasks** and **functions** are used to organize and modularize code, allowing repetitive operations to be executed more efficiently. Both provide a way to encapsulate logic into reusable blocks, but they have some key differences in their syntax and behavior.

---

1. Tasks

A **task** is a block of Verilog code that can perform a set of operations. Tasks can contain any type of procedural statements, including delays, procedural assignments, and control statements (like loops and conditionals). A task may also accept input and output arguments.

Key Features of Tasks:

- A task can have zero or more input, output, or inout arguments.
- Tasks **can** contain delays (`#` or `@`).
- Tasks can perform multiple operations including procedural assignments, loops, or delays.
- A task can be called multiple times from different parts of the code.

Syntax:

```
task task_name(input_type input_name, output_type output_name);
  // Task body
  // Code or procedural statements
endtask
```

Example:

```
module example;
  reg [3:0] a, b;
  wire [3:0] sum;

  // Task to add two numbers
  task add_numbers(input [3:0] num1, input [3:0] num2, output [3:0] result);
```

```
    begin
      result = num1 + num2;
    end
  endtask

  initial begin
    a = 4'b1100;
    b = 4'b0011;
    add_numbers(a, b, sum);  // Calling the task
    $display("Sum: %b", sum);
  end
endmodule
```

- **Calling a Task**: In the `initial` block, the task `add_numbers` is called with `a`, `b`, and `sum` as arguments.

Important Notes:

- Tasks may use delays, which means they can pause execution and continue after some time.
- A task may have `input`, `output`, or `inout` arguments.

---

2. Functions in Verilog

A **function** is similar to a task but is more restrictive in its behavior. Functions are used to perform a small operation that returns a value. Unlike tasks, functions cannot contain delays and must complete in zero simulation time.

Key Features of Functions:

- A function **must** return a value (using the `return` keyword).
- Functions **cannot** have delays (`#` or `@`).
- Functions **cannot** have procedural assignments.
- Functions can be used as expressions in other parts of the Verilog code.
- A function can only have input arguments (no output or inout).

Syntax:

```
function return_type function_name(input_type input_name);
  // Function body
  // Code to return a result
  return return_value;
endfunction
```

Example:

```
module example;
  reg [3:0] a, b;
  wire [3:0] sum;

  // Function to add two numbers and return the result
  function [3:0] add_numbers(input [3:0] num1, input [3:0] num2);
    begin
      add_numbers = num1 + num2;  // Return the sum
    end
  endfunction
  initial begin
    a = 4'b1100;
    b = 4'b0011;
    sum = add_numbers(a, b);  // Calling the function
    $display("Sum: %b", sum);
  end
endmodule
```

- **Calling a Function**: The function `add_numbers` is used in the `initial` block, where the result is directly assigned to the `sum` wire.

Important Notes:

- Functions return a value and must be used as part of an expression.
- Functions cannot contain `#` or `@` delays, meaning they execute instantly and return a value in zero simulation time.

Differences Between Tasks and Functions

Feature	Task	Function
Return Value	No return value (can use `output` arguments).	Must return a value (using `return`).
Delays	Can include delays (`#`, `@`), i.e., takes simulation time.	Cannot include delays.
Procedural Assignment	Can have procedural assignments.	Cannot have procedural assignments.
Usage in Expressions	Cannot be used in expressions directly.	Can be used in expressions (e.g., `sum = func(a, b)`).
Argument Types	Can have `input`, `output`, or `inout` arguments.	Can only have `input` arguments.
Call Behavior	Called as a separate statement.	Called as part of an expression.

Key Differences Between Automatic and Static Tasks

Feature	Automatic Task	Static Task
Memory Allocation	Variables are dynamically allocated (each call gets a new set of variables).	Variables retain their value between calls (shared memory).
Scope of Variables	Local to each call (i.e., each invocation gets a fresh instance of variables).	Shared across all invocations (retains values between calls).
Thread Safety	Thread-safe, since each call gets its own local variables.	Not thread-safe, since all invocations share the same variables.
Execution Speed	Generally slower due to memory allocation/deallocation for each call.	Generally faster but can lead to unexpected behavior due to shared state.
Use Cases	Ideal for tasks that are called concurrently and require independent local variables.	Ideal for tasks where shared state needs to be retained across calls.
Memory Overhead	Higher memory overhead per call (due to dynamic allocation).	Lower memory overhead (shared state across calls).

Test Bench Writing

1. Combinational Circuit Testbench

For **combinational circuits**, the testbench applies input values and checks the outputs.

Example: 4-bit Adder

Design:

```
module four_bit_adder (input [3:0] A, B, input Cin, output [3:0] Sum, output Cout);
  assign {Cout, Sum} = A + B + Cin;
endmodule
```

Testbench:

```
module tb_four_bit_adder;
  reg [3:0] A, B;
  reg Cin;
  wire [3:0] Sum;
  wire Cout;

  // Instantiate DUT
  four_bit_adder DUT (A, B, Cin, Sum, Cout);

  initial begin
    A = 4'b0001; B = 4'b0010; Cin = 0; #10;
    $display("Sum = %b, Cout = %b", Sum, Cout);

    A = 4'b1111; B = 4'b0001; Cin = 1; #10;
    $display("Sum = %b, Cout = %b", Sum, Cout);

    $finish;
  end
endmodule
```

2. Sequential Circuit Testbench

For **sequential circuits**, you simulate the clock and reset signals to test the design.

Example: D Flip-Flop

Design:

```
module d_flip_flop (input D, clk, reset, output reg Q);
  always @(posedge clk or posedge reset) begin
    if (reset) Q <= 0;
    else Q <= D;
  end
endmodule
```

Testbench:

```
module tb_d_flip_flop;
  reg D, clk, reset;
  wire Q;

  // Instantiate DUT
  d_flip_flop DUT (D, clk, reset, Q);

  // Clock generation
  always #5 clk = ~clk;

  initial begin
    clk = 0; reset = 0; D = 0;
    reset = 1; #10; $display("Q = %b", Q);

    reset = 0; D = 1; #10; $display("Q = %b", Q);

    $finish;
  end
endmodule
```