SystemVerilog Cheat Sheet V1.0

Data Types

SystemVerilog provides various data types to define the signals and variables used in the design. These include logic types and bit vectors. **Logic Type:** Used for general-purpose variables that can hold 0, 1, undefined X or high-impedance Z.

```
logic a, b; // Two logic variables
```

Bit Vectors: Arrays of bits used to represent multi-bit data.

```
logic [3:0] bus; // 4-bit vector
```

Packed Array: Contiguous bits stored in memory.

```
// Packed array with 4 elements
logic [7 : 0][3 : 0] packed_array ;
```

Packed arrays in SystemVerilog are useful when you need to manipulate contiguous bits as a single entity. For example, `packed_array[0]` represents an 8-bit value, `packed_array[1]` the next 8-bit value, and so on.

Unpacked Array: Independent elements stored in memory.

```
// Unpacked array with 8 elements
logic [3 : 0] unpacked_array [0 : 7];
```

Unpacked arrays allow you to access individual elements independently. Each element, such as `unpacked_array[0]`, `unpacked_array[1]`, etc., is a 4-bit variable.

Module Definition

A module is the basic building block in SystemVerilog. It defines a hardware block with inputs, outputs, and internal logic. Modules can be instantiated within other modules to create complex designs.

```
module and_gate (
  input logic a,
  input logic b,
  output logic y
);
  // Continuous assignment for combinational logic
  assign y = a & b; // AND operation
endmodule
```

Always Block

The always block is used to describe both sequential and combinational logic.
Sequential Logic: Triggered by clock edges or asynchronous reset signals.

```
always_ff @(posedge clk or posedge reset) begin
  if (reset) begin
    q <= 0; // Asynchronous reset
  end else begin
    q <= d; // Register assignment on clock edge
    end
end
```

Combinational Logic: Evaluated whenever any input changes.

```
always_comb begin
  if (sel) begin
    // Select a if sel is true
    y = a;
  end else begin
    // Select b if sel is false
    y = b;
  end
end
```

Continuous Assignment

Continuous assignments are used for simple combinational logic outside of always blocks. They are always active and drive the assigned values continuously.

```
// AND operation
assign y = a & b;
```

Conditional Operator

The conditional operator (?:) is a compact way to express if-else logic. It can be used in continuous assignments and always blocks for decision-making.

```
// If sel is true, y = a; otherwise y = b
assign y = (sel) ? a : b;
```

Case Statement

The case statement is used for multi-way branching based on the value of an expression. It is particularly useful for implementing finite state machines (FSMs) and decoders. Case statements with multiple lines must use `begin ··· end`.

```
always_comb begin
  case (opcode)
    2'b00: result = a + b; // Addition
    2'b01: result = a - b; // Subtraction
    2'b10: result = a & b; // AND operation
    2'b11: result = a | b; // OR operation
    default: result = 0;   // Default case
  endcase
end
```

For Loop

The for loop iterates over a block of code a fixed number of times. It is useful for generating repetitive logic, such as summing elements of an array or initializing registers.

```
always_comb begin
  sum = 0; // Initialize sum
  for (int i = 0; i < 4; i++) begin
    // Sum elements of data array
    sum = sum + data[i];
  end
end
```

Parameters

Parameters in SystemVerilog allow modules to be **configured or customized** during instantiation. They're useful for creating **reusable and flexible** designs (e.g., parameterized bus widths, address sizes, etc.).

Declaring Parameters

Use the `parameter` keyword inside the module definition.

```
module adder #(parameter WIDTH = 8) (
    input  logic [WIDTH-1:0] a, b,
    output logic [WIDTH-1:0] sum
);
    assign sum = a + b;
endmodule
```

Multiple Parameters

You can declare multiple parameters with default values:

```
module memory #(
    parameter ADDR_WIDTH = 8,
    parameter DATA_WIDTH = 16
) (
    input  logic [ADDR_WIDTH-1:0] addr,
    input  logic [DATA_WIDTH-1:0] wdata,
    output logic [DATA_WIDTH-1:0] rdata
);
// Module body
endmodule
```

Instantiating Parameterized Modules

Override parameters using the `#()` syntax at instantiation:

```
adder #(
  .WIDTH(16)
) my_adder (
    .a(data1),
    .b(data2),
    .sum(result)
);
```

Parameters are constant expressions evaluated at compile time. Parameters can also be defined using `localparam` for internal constants.

Generate Block

The generate block is used to create multiple instances of a block of code. It is particularly useful for creating parameterized hardware designs that need multiple instances of a module or logic.

```
generate // <-- you don't have to mark the beginning of
↪  a generate block by writing this (in SystemVerilog)
  for (genvar i = 0; i < 4; i = i + 1) begin : gen_block
    // Assign each bit of bus
    assign bus[i] = data[i];

    // Create instances
    module_def i_module ( /* port map */ );
  end

endgenerate // <-- you don't have to mark the beginning
↪  of a generate block by writing this (in
↪  SystemVerilog)

// Other generate construct is a Generate-if
if (A == 2) begin
  // do something
end
```

Finite State Machine (FSM)

FSMs are used to manage state transitions in a design. They consist of states, transitions, and actions associated with the states.

```
// Enum definition with explicit datatype
typedef enum logic [1:0] {
// Or Enum definition without explicit type
typedef enum {
  /* You don't need assign values, if so SystemVerilog
  ↪  creates a default value assignment for you. */
  IDLE = 2'b00,
  READ = 2'b01,
  WRITE = 2'b10
} state_t;

state_t current_state, next_state;

always_ff @(posedge clk or posedge reset) begin
  if (reset) begin
    // Asynchronous reset to IDLE
    current_state <= IDLE;
  end else begin
    // State transition on clock edge
    current_state <= next_state;
  end
end

always_comb begin
  case (current_state)
    IDLE: begin
      if (start) begin
        next_state = READ;
      end
    end
    READ: begin
      next_state = WRITE;
    end
    WRITE: begin
      next_state = IDLE;
    end
    default: begin
      next_state = IDLE;
    end
  endcase
end
```

Operators

Arithmetic Operators

Division, modulus and multiplication are very expensive and sometimes you cannot synthesize division and modulus. Assume that `a = 5`, `b = 10`, `c = 2'b01`.

Character	Operation performed	Example
+	Add	`b + c = 11`
-	Subtract	`b - c = 9, -b = - 10`
/	Divide	`b / c = 2`
*	Multiply	`a * b = 50`
%	Modulus	`b % a = 0`

Bitwise Operators

Each bit is operated, result is the size of the largest operand and the smaller operand is left extended with zeroes to the size of the bigger operand. Assume `a = 3'b101`, `b = 3'b110` and `c = 3'b01x`.

Character	Operation performed	Example
~	Invert each bit	`~a = 3'b010`
&	And each bit	`b & c = 3'b100`
|	Or each bit	`a | b = 3'b111`
^	Xor each bit	`a ^ b = 3'b011`
^~ or ~^	Xnor each bit	`a ^~ b = 3'b100`

Reduction Operators

These operators reduces the vectors to only one bit. If there are the characters z and x the result can be a unknown value. Assume `a = 5'b10101`, `b = 4'b0011`, `c = 3'bz00` and `d = 3'bx011`.

Character	Operation performed	Example
&	And all bit	`&a = 1'b0, &d = 1'b0`
~ &	Nand all bit	`~&a = 1'b1`
|	Or all bit	`|a = 1'b1, |c = 1'bx`
~ |	Nor all bit	`~|a= 1'b0`
^	Xor all bit	`^a = 1'b1`
^~ or ~^	Xnor all bit	`~^a = 1'b0`

Relation Operators

These operators compare operands and results a 1 bit scalar boolean value. The case equality and inequality operators can be used for unknown or high impedance values (z or x) and if the two operands are unknown the result is a 1. However, using z or x is not synthesizable. Assume `a = 3'b010`, `b = 3'b100`, `c = 3'b111`, `d = 3'b01z` and `e = 3'b01x`.

Character	Operation performed	Example
>	Greater than	`a > b = 1'b1`
<	Smaller than	`a < b = 1'b0`
>=	Greater than or equal	`a >= d = 1'bx`
<=	Smaller than or equal	`a <= d = 1'bx`
==	Equality	`a == b = 1'b0`
!=	Inequality	`a != b = 1'b1`

Logical Operators

These operators compare operands and results a 1 bit scalar boolean value. Assume `a = 3'b010` and `b = 3'b000`.

Character	Operation performed	Example
!	Not	`!(a && b) = 1'b1`
&&	Logical And	`a && b = 1'b0`
||	Logical Or	`a || b = 1'b1`

Shift Operators

These operators shift operands to the right or left, the size is kept constant, shifted bits are lost and the vector is filled with zeroes. Assume `a = 4'b1010` and `b = 4'b10x0`.

Character	Operation performed	Example
>>	Shift right	`b >> 1 = 4'b010x`
<<	Shift left	`a << 2 = 4'b1000`

Other Operators

These are operators used for condition testing and to create vectors. Assume `a = 4'b1010` and `b = 4'b10x0`.

Character	Operation performed	Example
?:	Condition testing	`cond. ? true stmt. : false stmt.`
{}	Concatenate	`c = {a,b} = 8'b101010x0`
{{}}	Replicate	`{3{2'b10}}= 6'b101010`

Operator Precedence

+, -, !, ~ (Unary) → +,- (Binary) → <<, >> → <, >, <= , >= → ==, != → & → ^, ^~ or ~^ → | → && → || → ?:

System Tasks

System tasks provide mechanisms for displaying information, stopping simulation, and file operations. They are primarily used in testbenches and debugging.

$display: Prints a message to the console.

```
$display("Time: %0t, a: %b, b: %b", $time, a, b);
```

$finish: Ends the simulation.

```
$finish;
```

$write: Similar to $display but does not add a newline at the end.

```
$write("Data: %h", data);
```

$fopen: Opens a file for writing.

```
integer file;
file = $fopen("output.txt", "w");
```

$fwrite: Writes to an open file.

```
$fwrite(file, "Time: %0t, Data: %h\n", $time, data);
```

$signed: Converts an unsigned number to a signed number of the specified width. It is used to interpret bit patterns as signed integers for arithmetic operations.

```
logic [7:0] unsigned_data = 8'b10000001;
logic signed [7:0] signed_data;

initial begin
  // Convert unsigned to signed
  signed_data = $signed(unsigned_data);
  // Output: Signed data: -127
  $display("Signed data: %d", signed_data);
end
```

Testbench (Simulation only)

Initial Block

The initial block is used for simulation purposes to initialize values. It is not synthesizable and should be used only in testbenches.

```
initial begin
  a = 0;
  b = 1;
  #10 a = 1; // Wait for 10 time units and change a
end
```

Testbench Example

A basic testbench to simulate and verify the functionality of a 4-bit adder. Testbenches are used to apply stimulus to the design and observe the outputs.

```
module tb_adder4;
  logic [3:0] a, b;
  logic [3:0] sum;
  logic carry;

  // Instantiate a 4-bit adder
  adder4 uut (
    .a(a),
    .b(b),
    .sum(sum),
    .carry(carry)
  );

  initial begin
    // Test case
    a = 4'b0001; b = 4'b0010;
    #10;
    // Test case 2
    a = 4'b0101; b = 4'b0101;
    #10;
    // Test case 3
    a = 4'b1111; b = 4'b0001;
    #10;
  end
endmodule
```

Additional Information

The following resources offer additional assistance and in-depth information. Please note that not all functionalities presented in these resources are required for our lecture.

HDLBits

HDLBits is an educational platform focused on teaching Verilog through interactive exercises. It offers a variety of problems ranging from basic to advanced levels, allowing users to practice and improve their digital logic design skills. The platform provides immediate feedback on submitted solutions, enhancing the learning experience.

`https://hdlbits.01xz.net/wiki/Main_Page`

ChipVerify

ChipVerify is an educational platform designed to help users learn and enhance their skills in chip verification using SystemVerilog and UVM (Universal Verification Methodology). It offers comprehensive tutorials, practical examples, and a wealth of resources on verification techniques and methodologies. The site aims to provide both beginners and experienced professionals with the knowledge needed to effectively verify complex digital designs.

`https://www.chipverify.com/`