Verilog Cheat Sheet by Santiago Jácome

Verilog is a **Hardware Description Language (HDL)** used to describe, simulate, and implement digital systems. Here's everything you need to get started and reference frequently!

Design Styles

Top-Down vs Bottom-Up

- **Bottom-Up**: Design starts at the gate level (small components). Difficult to maintain for complex designs.
- **Top-Down**: Begin with high-level abstraction (e.g., specifications, block diagrams). Easier testing and tech changes.
  ☆ **Most designs are a mix of both.**

Abstraction Levels in Verilog

1. **Behavioral Level**
   - Describes a system using **concurrent algorithms**.
   - **Example**: High-level functionality of a system.
2. **Register-Transfer Level (RTL)**
   - Describes data **transfer between registers** using operations.
3. **Gate Level**
   - Describes circuits using **logic gates** (AND, OR, NOT) and their timing.
   - **Example**: `assign y = a & b;`

Verilog Design Cycle

1. **Specifications**: Define restrictions and requirements.
2. **High-Level Design**: Create block diagrams.
3. **Low-Level Design**: Define details like state machines.
4. **RTL Coding**: Write Verilog code.
5. **Verification**: Test design with simulation.
6. **Synthesis**: Convert RTL to gate-level hardware.

Modules

Modules are the **building blocks** of Verilog. Think of them as **black boxes**.

Module Syntax

```
module arbiter (clock, reset, req_0, req_1, gnt_0, gnt_1);
    input clock, reset, req_0, req_1;
    output gnt_0, gnt_1;
    // Internal logic goes here
endmodule
```

- **Ports**: Defined as `input`, `output`, or `inout`.
- **Vector Signals**: Multi-bit signals like `[3:0]`. Endianness matters:
    - Little Endian: `[7:0]`
    - Big Endian: `[0:7]`

💾 Data Types

1. `reg`: Stores values (used in procedural blocks).
    - Example: `reg [7:0] address_bus;`
2. `wire`: Connects points (cannot store values).
    - Example: `wire gate_output;`

➕ Operators

Arithmetic

Operator	Meaning	Example
`+`	Addition	`a + b`
`-`	Subtraction	`a - b`
`*`	Multiplication	`a * b`
`/`	Division	`a / b`
`%`	Modulus	`a % b`

Logical

Operator	Meaning	Example
`&&`	Logical AND	`a && b`
`||`	Logical OR	`a || b`
`!`	Logical NOT	`!a`

Relational

Operator	Meaning	Example
`<`	Less than	`a < b`

Operator	Meaning	Example
`>`	Greater than	`a > b`
`<=`	Less/equal	`a <= b`
`>=`	Greater/equal	`a >= b`

Equality

Operator	Meaning	Example
`==`	Equal	`a == b`
`!=`	Not Equal	`a != b`
`===`	Case Equality	`a === b`
`!==`	Case Inequality	`a !== b`

Reduction

Operator	Meaning	Example
`&`	AND Reduction	`&a`
`|`	OR Reduction	`|a`
`^`	XOR Reduction	`^a`

⚡ Assignments: Blocking vs Non-Blocking

Blocking Assignment (`=`)

- Executes immediately in procedural blocks.
- Used for **combinational logic** or temporary variables.
- **Example**:

```
always @(*) begin
    y = a + b;  // Assigns immediately
end
```

Non-Blocking Assignment (`<=`)

- Queues the assignment to occur at the end of the time step or clock edge.
- Used for **sequential logic**.
- **Example**:

```
always @(posedge clk) begin
    y <= a + b;  // Queues for update
```

```
    end
```

Key Difference

- Use `=` for combinational logic (immediate execution).
- Use `<=` for sequential logic to model flip-flops or registers.

🔄 Control Statements

If-Else

```
if (condition)
    statement;
else
    statement;
```

Case Statement

```
case (selector)
    value1: statement1;
    value2: statement2;
    default: statement_default;
endcase
```

For Loop

```
for (init; condition; update)
    statement;
```

Repeat

```
repeat (n)
    statement;
```

🧠 Example: 2:1 Multiplexer

```
always @(a or b or sel) begin
    if (sel == 0)
        y = a;
```

```
    else
        y = b;
end
```

Testbenches

Testbenches validate designs by simulating inputs and observing outputs.

Example Testbench

```
module tb();
    reg clock, reset, enable;
    wire [3:0] counter_out;

    initial begin
        clock = 0;
        reset = 0;
        enable = 0;
        $monitor("%g: clock=%b, reset=%b, enable=%b, counter_out=%b",
                 $time, clock, reset, enable, counter_out);
    end

    always #5 clock = ~clock;

    first_counter DUT (clock, reset, enable, counter_out);
endmodule
```

Dumping Waveforms

To visualize waveforms:

1. Add the following to the testbench:

```
initial begin
    $dumpfile("simulation.vcd");  // VCD file
    $dumpvars(0, tb);             // Dump all signals in "tb"
end
```

2. Compile and simulate:

```
iverilog -o simulation.vvp tb.v first_counter.v
vvp simulation.vvp
```

3. View the waveform using GTKWave:

```
gtkwave simulation.vcd
```

Tasks and Functions

Tasks

- Can include delays, events, and multiple outputs.

```
task example_task(input a, output b);
    b = a + 1;
endtask
```

Functions

- Cannot include delays, used for single return values.

```
function [3:0] add_one(input [3:0] a);
    add_one = a + 1;
endfunction
```