Verilog HDL

A Brief Introduction

Fall 2019

https://web.cs.hacettepe.edu.tr/~bbm231/
https://piazza.com/hacettepe.edu.tr/fall2019/bbm231233
Outline

- Introduction
- Hardware Description Languages
- Different Levels of Abstraction
- Getting Started With Verilog
- Verilog Language Features
- Test benches and Simulation
Introduction

As digital and electronic circuit designs grew in size and complexity, capturing a large design at the gate level of abstraction with schematic-based design became

- Too complex,
- Prone to error,
- Extremely time-consuming.

Complex digital circuit designs require a lot more time for development, synthesis, simulation and debugging.

Solution? Computer Aided Design (CAD) tools

- Based on Hardware Description Languages

Nowadays, billions of transistors per chip!

Moore’s Law?
Hardware Description Language (HDL)

HDLs are specialized computer languages used to program electronic and digital logic circuits.

- High level languages with which we can specify our HW to analyze its design before actual fabrication.

**Two most popular HDLs:**

- Verilog
- VHDL

**Other popular HDLs:**

- SystemC
- SystemVerilog
- ...
Some other steps in design:

- **Simulation** to verify the design (at different levels)
- **Formal verification**
- etc.
Different Levels of Abstraction

Behavioral vs. Structural Design

- **Behavioral**: the highest level of abstraction - specifying the functionality in terms of its *behavior* (e.g. Boolean equations, truth tables, algorithms, code, etc.). *WHAT, not HOW.*

- **Structural**: a netlist specification of components and their interconnections (e.g. gates, transistors, even functional modules).

We will use both.
Different Levels of Abstraction

- **Behavioral modeling:**
  - Truth table:
    | A | B | Cin | Sum | Cout |
    |---|---|-----|-----|-----|
    | 0 | 0 | 0   | 0   | 0   |
    | 0 | 0 | 1   | 1   | 0   |
    | 0 | 1 | 0   | 1   | 0   |
    | 0 | 1 | 1   | 0   | 1   |
    | 1 | 0 | 0   | 1   | 0   |
    | 1 | 0 | 1   | 0   | 1   |
    | 1 | 1 | 0   | 0   | 1   |
    | 1 | 1 | 1   | 1   | 1   |

- **Structural modeling:**
  - Circuit diagram:
  - Verilog module with structural design:

```verilog
module full_adder(x, y, cin, s, cout);
  input x, y, cin;
  output s, cout;
  wire s1, c1, c2, c3;
  xor(s1, x, y);
  xor(s, cin, s1);
  and(c1, x, y);
  and(c2, y, cin);
  and(c3, x, cin);
  or(cout, c1, c2, c3);
endmodule
```

Example: Full Adder

- **Verilog module:**
- **Circuit diagram:**
- **Table:**
- **Verilog module with structural design:**

In terms of Boolean expressions:

\[
\begin{align*}
    S &= A' B' C_{in} + A' B C_{in} + A B C_{in}' + A B (C_{in} + 1) \\
    &= A' (B' C_{in} + 1) + A (B C_{in} + 1) \\
    &= A' Z + A' Z' + A Z + A C_{in} \\
    &= A \text{xor} Z = A \text{xor} (B \text{xor} C_{in})
\end{align*}
\]
Getting Started With Verilog

Describing a digital system as a set of modules

- Modules can have interfaces to other modules (*instantiation* = creating a copy).
- Modules are connected using *nets*. 

```
module ...

module 1
module 2
module 3
module 4

endmodule
```
What can we do?

- **Simulation** to verify the system (test benches)
- **Synthesis** to map to hardware (low-level primitives, ASIC, FPGA)

We’ll be doing this.
Development Process

**SIMULATION**
- Verilog Module(s)
- Testbench

Open source or commercial software available

Evaluate Result

**SYNTHESIS**
- Verilog Module(s)
- FPGA
- ASIC

Specific software (e.g., Xilinx ISE or Vivado)

Commercial CAD tools
Verilog Language Features

Operators

Arithmetic Operators:

- unary (sign) plus
- unary (sign) minus
+ binary plus (add)
- binary minus (subtract)
* multiply
/ divide
% modulus
** exponentiation

Examples:

- \(-(b + c)\)
- \((a - b) + (c \times d)\)
- \((a + b) / (a - b)\)
- \(a \% b\)
- \(a ** 3\)
Verilog Language Features

Operators

Logical Operators:
- `!` logical negation
- `&&` logical AND
- `||` logical OR

Examples:
- (done && ack)
- (a || b)
- ! (a && b)
- ((a > b) || (c == 0))
- ((a > b) && ! (b > c))

Evaluates to True or False
Verilog Language Features

Operators

Relational Operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!=</td>
<td>not equal</td>
</tr>
<tr>
<td>==</td>
<td>equal</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater or equal</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater</td>
</tr>
<tr>
<td>&lt;</td>
<td>less</td>
</tr>
</tbody>
</table>

Examples:

(a != b)
((a + b) == (c - d))
((a > b) && (c < d))
(count <= 0)

Operate on numbers, return True or False
Verilog Language Features

Operators

**Bitwise Operators:**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>bitwise NOT</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>bitwise exclusive-OR</td>
</tr>
<tr>
<td>^^</td>
<td>bitwise exclusive-NOR</td>
</tr>
</tbody>
</table>

**Examples:**

```plaintext
wire a, b, c, d, f1, f2, f3, f4;
assign f1 = ~a | b;
assign f2 = (a & b) | (b & c) | (c & a);
assign f3 = a ^ b ^ c;
assign f4 = (a & ~b) | (b & c & ~d);
```
Verilog Language Features

Operators

Shift Operators:

- `>>` shift right
- `<<` shift left
- `>>>` arithmetic shift right

Examples:

```verilog
wire [15:0] data, target;
assign target = data >> 3;
assign target = data >>> 2;
```

Reduction, conditional, concatenation, and replication operators also available.
Verilog Language Features

Verilog supports 4 value levels:

<table>
<thead>
<tr>
<th>Value Level</th>
<th>Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Logic 0 state</td>
</tr>
<tr>
<td>1</td>
<td>Logic 1 state</td>
</tr>
<tr>
<td>x</td>
<td>Unknown logic state</td>
</tr>
<tr>
<td>z</td>
<td>High impedance state</td>
</tr>
</tbody>
</table>

- All unconnected nets are set to ‘z’.
- All register variables are set to ‘x’.
Verilog Language Features

Module - the basic unit of hardware in Verilog

- Cannot contain definitions of other modules,
- Can be *instantiated* within another module - hierarchy of modules.

Different than calling a function in programming lang. Every instantiation adds to area!

```
module module_name (list_of_ports);
  input/output declarations
  Local net declarations  Temporary connections (wires)
  Parallel statements
endmodule
```

Why parallel?
Verilog Language Features

Module example: A simple AND function

```verilog
// A simple AND function
module simple_AND(t1, a, b);
  input a, b;
  output t1;
  assign t1 = a & b;
endmodule
```

Assign statement:

```
assign var = expression;
```

- used typically for combinational circuits.
- continuous assignment
- LHS must be “net” type var (usually “wire”)
- RHS can be both “register” or “net” type

Is this a structural or behavioral description?

```
assign t1 = a & b;
```

Not synced with clock!
Module example 2: A 2-level combinational circuit

```verilog
// A 2-level combinational circuit
module two_level(a, b, c, d, f);
    input a, b, c, d;
    output f;
    wire t1, t2; // intermediate lines
    assign t1 = a & b;
    assign t2 = ~(c | d);
    assign f = ~(t1 & t2);
endmodule
```

This is also a behavioral description.
Verilog Language Features

A variable can be:

A. **Net**  *wire, wor, wand, tri, supply0, supply1, etc.*
   - Must be continuously driven,
   - Cannot be used to store a value,
   - Models connections between continuous assignments and instantiations,
   - 1-bit values by default, unless declared as vectors explicitly.
   - Default value of a *net* is “Z” - high impedance state.

B. **Register**  *reg, integer, real, time*
   - Retains the last value assigned to it,
   - Usually used to represent storage elements (sometimes in combinational circuits),
   - May or may not map to a HW register during synthesis.
   - Default value of a *reg* data type is “X”.

Data Types

```
wire sbar;
assign sbar = ~S;
```
Verilog Language Features

Data Types

net example:

```verilog
module use_wire(a, b, c, d, f);
    input a, b, c, d;
    output f;
    wire f; // net f declared as wire
    assign f = a & b;
    assign f = c | d;
endmodule
```
Verilog Language Features  Data Types

net example:

```
module use_wire(a, b, c, d, f);
    input a, b, c, d;
    output f;
    wire f; // net f declared as wire
    assign f = a & b;
    assign f = c | d;
endmodule
```

Wrong design!

For these inputs, f will be indeterminate!
Verilog Language Features

Data Types

net example - correct design:

```
// A 2-level combinational circuit
module using_wand(a, b, c, d, f);
    input a, b, c, d;
    output f;
    wand f; // net f declared as wand
    assign f = a & b;
    assign f = c | d;
endmodule
```

Here, function realized will be

\[ f = (A \& B) \& (C \mid D) \]

\[ f = t_1 \& t_2 \]
Verilog Language Features

Data Types

Net example 2:

```verilog
module using_supply_wire(a, b, c, f);
  input a, b, c;
  output f;
  wire t1, t2;
  supply0 gnd;
  supply1 vcc;
  nand G1 (t1, vcc, a, b);
  xor G2 (t2, c, gnd);
  and G3 (f, t1, t2);
endmodule
```

Is this a structural or behavioral description?
Verilog Language Features

reg example:

- Declaration explicitly specifies the size (default is 1-bit):
  - `reg x, y;` // 1-bit register variables
  - `reg [7:0] bus;` // An 8-bit bus
- Treated as an unsigned number in arithmetic expressions.
- **MUST** be used when modeling actual **sequential** HW, e.g. counters, shift registers, etc.
- Two types of assignments possible:
  - `A = B + C;`
  - `A <= B + C;`
Verilog Language Features

Data Types

**reg example - 32-bit counter with synchronous reset:**

```verilog
module simple_counter(clk, rst, count);
    input clk, rst;
    output [31:0] count;
    reg [31:0] count;

    always @(posedge clk)
    begin
        if(rst)
            count = 32'b0;
        else
            count = count + 1;
    end
endmodule
```

**Because we must have a reg type var at LHS**
Otherwise: compiler error

**How to fix this?**

If rst is high, reset occurs at the positive edge of the next clock.

Any variable assigned within the *always* block must be of type *reg*. 
Verilog Language Features

**reg example - solution: 32-bit counter with asynchronous reset:**

```verilog
module simple_counter(clk, rst, count);
  input clk, rst;
  output [31:0] count;
  reg [31:0] count;

  always @(posedge clk or posedge rst)
  begin
    if(rst)
      count = 32'b0;
    else
      count = count + 1;
  end
endmodule
```

Reset occurs whenever `rst` goes high.
Verilog Language Features

Data Types

Integer example:

- General purpose register data type,
- 2’s complement signed integer in arithmetic expressions,
- Default size is 32 bits.

```
wire [15:0] X, Y;
integer C;
C = X + Y;
```

Synthesis tool deduces that C is 17 bits (16 bits + a carry)

Other register data types: `real`, `time`. 
Verilog Language Features

Vectors

- Both `Net` or `reg` type variables can be declared as vectors: **multiple bit widths**
- Specifying width with: `[MSB:LSB]`

```verilog
wire x, y, z;  // single bit variables
wire[7:0] sum; // MSB is sum[7], LSB is sum[0]
reg [1:10] data; // MSB is data[1], LSB is data[10]
reg clock;
```
Verilog Language Features

Vectors

- Parts of a vector can be addressed and used in an expression:

```verilog
reg [31:0] IR;
reg [5:0] opcode;
reg [4:0] reg1, reg2, reg3;
reg [10:0] offset;

opcode = IR[31:26];
reg1 = IR[25:21];
reg2 = IR[20:16];
reg3 = IR[15:11];
offset = IR[10:0];
```

\[ \text{SUM} = \text{IR}[25:21] + \text{IR}[20:16] \]
Verilog Language Features

Constant Values

- **Sized** or **unsized** form,
- Syntax:
  - `<size>'<base><number>
- Examples:

```
4'b0101           // 4-bit binary number 0101
1'b0              // Logic 0 (1-bit)
12'hB3C           // 12-bit number 1011 0011 1100
12'h8xF           // 12-bit number 1000 xxxx 1111
25                // signed number, in 32 bits (size not specified)
```
Verilog Language Features

Parameters

- Constants with a given name,
- Size deduced from the constant value itself:

```verilog
module counter(clk, rst, count);
    parameter N = 31;
    input clk, rst;
    output [0:N] count;
    reg [0:N] count;

    always @(negedge clk)
    begin
        if (rst)
            count = 0;
        else
            count = count + 1;
    end
endmodule
```

```verilog
parameter HI = 5, LO = 0;
parameter up = 2'b00, down = 2'b01, steady = 2'b10;
```
Verilog Language Features

Predefined Logic Gates

- Can be instantiated within a module to create a structural design.

<table>
<thead>
<tr>
<th>2-input AND</th>
<th>2-input OR</th>
<th>2-input EXOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 &amp; 0 = 0</td>
<td>0</td>
<td>0 = 0</td>
</tr>
<tr>
<td>0 &amp; 1 = 0</td>
<td>0</td>
<td>1 = 1</td>
</tr>
<tr>
<td>1 &amp; 1 = 1</td>
<td>1</td>
<td>1 = 1</td>
</tr>
<tr>
<td>1 &amp; x = x</td>
<td>1</td>
<td>x = 1</td>
</tr>
<tr>
<td>0 &amp; x = 0</td>
<td>0</td>
<td>x = x</td>
</tr>
<tr>
<td>1 &amp; z = x</td>
<td>1</td>
<td>z = x</td>
</tr>
<tr>
<td>z &amp; x = x</td>
<td>z</td>
<td>x = x</td>
</tr>
<tr>
<td>0 ^ 0 = 0</td>
<td>0</td>
<td>1 = 1</td>
</tr>
<tr>
<td>0 ^ 1 = 1</td>
<td>1</td>
<td>1 = 0</td>
</tr>
<tr>
<td>1 ^ x = x</td>
<td>1</td>
<td>x = x</td>
</tr>
<tr>
<td>0 ^ x = x</td>
<td>0</td>
<td>x = x</td>
</tr>
<tr>
<td>1 ^ z = x</td>
<td>1</td>
<td>z = x</td>
</tr>
<tr>
<td>z ^ x = x</td>
<td>z</td>
<td>x = x</td>
</tr>
</tbody>
</table>

Remember that Verilog supports 4 value levels.
Verilog Language Features

List of Some Primitive Gates

```verilog
and Gate1 (out, in1, in2);
nand Gate2 (out, in1, in2);
or Gate3 (out, in1, in2);
nor Gate4 (out, in1, in2);
xor Gate5 (out, in1, in2);
xnor Gate6 (out, in1, in2);
not Gate7 (out, in1);
```

- Number of inputs can be arbitrary.
- Output ports must be connected to a `net`.
- Input ports may be either `net` or `reg` type vars.

A 5-input AND gate
Verilog Language Features

Two Ways to Specify Connectivity During Module Instantiation

Connectivity of the signal lines between two modules can be specified in 2 ways:

- **Positional Association (Implicit)**
  - Parameters listed *in the same order* as in the original module description.

- **Explicit Association**
  - Parameters *explicitly* listed *in arbitrary order*. 

```verilog
`include "module2_name.v"
```
Verilog Language Features

- **Positional Association Example** - Full Adder Using Half Adder Module

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Sum</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Half Adder**

**Full Adder**

```
A   B   S   Carry
0   0   0   0
0   1   1   0
1   0   1   0
1   1   0   1
```
Verilog Language Features

- **Positional Association Example** - Full Adder Using Half Adder Module

```
module half_adder (Sum, Carry, A, B);
    input A, B;
    output Carry, Sum;
    //structural description
    xor G1(Sum, A, B);
    and G2(Carry, A, B);
endmodule
```
Verilog Language Features

- **Positional Association Example** - Full Adder Using Half Adder Module

```
module half_adder (Sum, Carry, A, B);
  input A, B;
  output Carry, Sum;
  //structural description
  xor G1(Sum, A, B);
  and G2(Carry, A, B);
endmodule
```

```
//behavioral description
assign Sum = A ^ B;
assign Carry = A & B;
```
Verilog Language Features

- Positional Association Example - Full Adder Using Half Adder Module

```
module half_adder (Sum, Carry, A, B);
    input A, B, Cin;
    output Cout, Sum;
    wire w1, w2, w3;
    half_adder HA1 (w1, w2, A, B);
    half_adder HA2 (Sum, w3, Cin, w1);
    or (Cout, w2, w3);
endmodule
```

Note the port order.
Verilog Language Features

- Explicit Association Example - Full Adder Using Half Adder Module

```verilog
module full_adder (Sum, Cout, A, B, Cin);
    input A, B, Cin;
    output Cout, Sum;
    wire w1, w2, w3;
    half_adder HA1 (.A(A), .B(B), .Sum(w1), .Carry(w2));
    half_adder HA2 (.Sum(Sum), .Carry(w3), .B(Cin), .A(w1));
    or (Cout, w2, w3);
endmodule
```

Ports are explicitly specified - order is not important

Less chance for errors
Verilog Language Features

Combinational vs. Sequential Circuits

**Combinational**: The output only depends on the present input.

**Sequential**: The output depends on both the present input and the previous output(s) (the state of the circuit).
Verilog Language Features

Combinational vs. Sequential Circuits

Sequential logic: Blocks that have memory elements: Flip-Flops, Latches, Finite State Machines.

- Triggered by a ‘clock’ event.
  - Latches are sensitive to level of the signal.
  - Flip-flops are sensitive to the transitioning of clock

Combinational constructs are not sufficient. We need new constructs:

- always
- initial
Verilog Language Features

Sequential Circuits

```verilog
always @(sensitivity list)
    statement;
```

Whenever the event in the sensitivity list occurs, the statement is executed.

```verilog
module simple_counter(clk, rst, count);
    input clk, rst;
    output [31:0] count;
    reg [31:0] count;

    always @(posedge clk)
    begin
        if(rst)
            count = 32'b0;
        else
            count = count + 1;
    end
endmodule
```
Verilog Language Features

Sequential Circuits

- Sequential statements are within an ‘always’ block,
- The sequential block is triggered with a change in the sensitivity list,
- Signals assigned within an always block must be declared as reg,
  - The values are preserved (memorized) when no change in the sensitivity list.
- We do not use ‘assign’ within the always block.

**Always blocks allow powerful statements**

- if .. then .. else
- case
Non-blocking and Blocking Statements

**Non-blocking**

```verilog
counter #2; // all assignments are made here // b is not (yet) 2'b01
end
```

- Values are assigned at the end of the block.
- All assignments are made in parallel, process flow is not-blocked.

**Blocking**

```verilog
counter #2; // a is 2'b01
begin
  a = 2'b01;
  b = a;
  // b is now 2'b01 as well
end
```

- Value is assigned immediately.
- Process waits until the first assignment is complete, it blocks progress.

Blocking statements allow sequential descriptions.

---

**Verilog Language Features**

**Sequential Circuits**

Blocking statements allow sequential descriptions.
How to Simulate Verilog Module(s)

Testbench: provides stimulus to Unit-Under-Test (UUT) to verify its functionality, captures and analyzes the outputs.

Requirements:
Inputs and outputs need to be connected to the test bench
How to Simulate Verilog Module(s)  Example

Suppose we want to design and simulate this circuit.

We can choose either behavioral or structural design.
How to Simulate Verilog Module(s)  Example

Now we need to provide stimulus and monitor the outputs - TESTBENCH

Let’s choose structural design:

```verilog
module function_Y (A, B, C, D, E, F, Y);
  input A, B, C, D, E, F;
  output Y;
  wire t1, t2, t3, Y;
  //structural description
  nand G1(t1, A, B);
  and G2(t2, C, ~B, D);
  nor G3(t3, E, F);
  nand G4(Y, t1, t2, t3);
endmodule
```
How to Simulate Verilog Module(s)  

Example

**Unit Under Test**

```verilog
module function_Y (A, B, C, D, E, F, Y);
    input A, B, C, D, E, F;
    output Y;
    wire t1, t2, t3, Y;
    //structural description
    nand G1(t1, A, B);
    and G2(t2, C, ~B, D);
    nor G3(t3, E, F);
    nand G4(Y, t1, t2, t3);
endmodule
```

**TESTBENCH**

```verilog
module function_Y_testbench;
    reg A, B, C, D, E, F, Y;
    wire Y;

    function_Y UUT(A, B, C, D, E, F, Y);

    initial
    begin
        #10 A = 0; B = 0; C = 0; D = 0; E = 0; F = 0;
        #10 A = 1; B = 0; C = 1; D = 1; E = 0; F = 0;
        #10 A = 0; B = 1;
        #10 F = 1;
        #10 $finish;
    end

endmodule
```

Saved as function_Y.v

Saved as function_Y_testbench.v

Vars MUST be declared as reg

Stimulus
How to Simulate Verilog Module(s)  Example

Results can be viewed as waveforms:
How to Simulate Verilog Module(s)  Example

We can also monitor the changes and print them to the console using `$monitor`:

```verilog
initial
begin
  $monitor ($time, "A = %b, B = %b, C = %b, D = %b, E = %b, F = %b, Y = %b", A, B, C, D, E, F, Y);
  #10 A = 0; B = 0; C = 0; D = 0;
  #10 A = 1; B = 0; C = 1; D = 1;
  #10 A = 0; B = 1;
  #10 F = 1;
  #10 $finish;
end
```

We can also use `$dumpfile` to dump variable changes to a file.
Questions?
Lab Example

Implement a **4-Bit Ripple Carry Adder** in Verilog in the following steps:

1. Implement a **1-Bit Full Adder** using **behavioral** design approach. Fill in the truth table, find the corresponding functions for *Sum* and *Carry_out*, write a Verilog module, test it by writing a testbench for all possible cases.

2. Implement a **4-Bit Ripple Carry Adder** by instantiating your 1-Bit Full Adder module as many times as necessary. Use **structural** design approach and **explicit association**. Test it by writing an appropriate testbench.
Lab Example

Solution:

\[
\text{sum} = (A^B)^C \\
\text{Cout} = AB + BC + AC
\]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>S</th>
<th>Cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Full_Adder.v**

```verilog
module Full_Adder(
    input A,
    input B,
    input Cin,
    output S,
    output Cout
);

assign S = (A&B)^Cin;
assign Cout = (A&B)||(B&Cin)||(Cin&A);
endmodule
```

Note the behavioral description
module Full_Adder_Testbench;
  //inputs
  reg A, B, Cin;
  //outputs
  wire S, Cout;
  // Instantiate the Unit Under Test (UUT)
  Full_Adder UUT(.A(A), .B(B), .Cin(Cin), .S(S), .Cout(Cout));
  //Provide stimulus
initial begin
  // Initialize Inputs
  A = 0; B = 0; Cin = 0;
  #10 A = 0; B = 0; Cin = 1;
  #10 A = 0; B = 1; Cin = 0;
  #10 A = 0; B = 1; Cin = 1;
  #10 A = 1; B = 0; Cin = 0;
  #10 A = 1; B = 0; Cin = 1;
  #10 A = 1; B = 1; Cin = 0;
  #10 A = 1; B = 1; Cin = 1;
  #10 $finish;
end
Four_Bit_RCA.v module

```
#include "Full_Adder.v"
module Four_Bit_RCA(
  input [3:0] A,
  input [3:0] B,
  input Cin,
  output [3:0] S,
  output Cout
);
wire [2:0] Carries;
Full_Adder A1(.A(A[0]),.B(B[0]),.Cin(Cin),.S(S[0]),.Cout(Carries[0]));
Full_Adder A2(.A(A[1]),.B(B[1]),.Cin(Carries[0]),.S(S[1]),.Cout(Carries[1]));
Full_Adder A3(.A(A[2]),.B(B[2]),.Cin(Carries[1]),.S(S[2]),.Cout(Carries[2]));
Full_Adder A4(.A(A[3]),.B(B[3]),.Cin(Carries[2]),.S(S[3]),.Cout(Cout));
endmodule
```
module Four.Bit_RCA_Testbench;

//inputs
reg [3:0] A, B;
reg Cin;

//outputs
wire [3:0] S;
wire Cout;

// Instantiate the Unit Under Test (UUT)
Four.Bit_RCA UUT(.A(A), .B(B), .Cin(Cin), .S(S), .Cout(Cout));

//Provide stimulus
initial begin

// Initialize Inputs
A = 4'b0000; B = 4'b0000; Cin = 0;
#10 A = 4'b0000; B = 4'b0000; Cin = 1;
#10 A = 4'b1100; B = 4'b0011; Cin = 0;
#10 A = 4'b1100; B = 4'b0011; Cin = 1;
#10 A = 4'b1110; B = 4'b0001; Cin = 1;
#10 A = 4'b1100; B = 4'b0011; Cin = 1;
#10 A = 4'b1111; B = 4'b0001; Cin = 0;
#10 A = 4'b1111; B = 4'b1111; Cin = 1;
end

Only some test cases. Why not all?
References

- Slides mostly based on: NPTEL Online Certification Course on Hardware Modeling Using Verilog, by Prof. Indranil Sengupta, Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur - Available online at: 
  https://www.youtube.com/playlist?list=PLUtfVcb-iqu-EkuBs3arreilxa2UKIChl


- Verilog for Sequential Circuits, Design of Digital Circuits 2014, Srdjan Capkun, Frank K. Gürkaynak. Available online at: 
  https://syssec.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/system-security-group-dam/education/Digitaltechnik_14/09_Verilog_Sequential.pdf