Verilog Synthesis Perspective

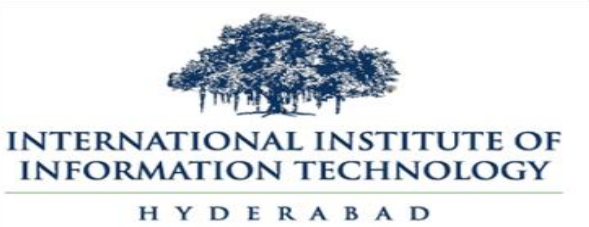
Student: Sri Rama Rathan Reddy Koluguri

Advisor: Abhishek Srivastava

Research Lab

Integrated **C**ircuits inspired by **Wi**reless
and **B**iomedical **E**lectronic **S**ystems

4 June, 2024



Center for VLSI and Embedded System Technologies (CVEST)
International Institute of Information Technology (IIIT) Hyderabad, India

Project Summary

Project Title	**Verilog Synthesis Perspective**
PI	Dr. Abhishek Srivastava
Research lab	***IC WiBES*** **I**ntegrated **C**ircuits inspired by **Wi**reless and **B**iomedical **E**lectronic **S**ystems
Research center	Center for VLSI and Embedded System Technologies (CVEST)
Institute	International Institute of Information Technology Hyderabad

Installation Procedure

You are required to download two softwares:

1. Icarus Verilog
2. GTK Wave.

The installation procedures for both are provided below. If you encounter difficulties running Verilog codes on your local system, you can alternatively use the online EDA platform called EDA Playground, which is very convenient.

Installation Procedure

1. **For Debian based operating systems like Ubuntu**:
   - To install iverilog: sudo apt-get install iverilog
   - To install GTK Wave: sudo apt-get install gtkwave
   - Any Issues Refer to the Link: Verilog Install Guide - Ubuntu
2. **For mac users**:
   - To install iverilog: brew install icarus-verilog
   - To install GTKWave: brew install gtkwave

Installation Procedure

For Windows Users:

1. Install Icarus Verilog from the given link:- Verilog Install for Windows
2. Download the version v11 or above.
3. Once downloaded , Run the file as administrator
4. Accept the license agreement.
5. Select both the Components given (MinGw and GTKwave)
6. Remember the destination of the installed file (cause you will run the programs from there)
7. In the last step ,select the check box which says “Add executable folder(s) to the user PATH.

Installation Procedure

The above Setup comes with GTKwave so you won't be required to download it separately.

Once Installed,you need to include the path in the environment Variable

1. Open the iverilog folder.
2. Copythe path of the bin folder.
3. Search for “Systems” in the windows search bar. Open the Systems.
4. Select Advanced System Setting. Click on Environment variables.
5. Click on New. Give any name to the Variable name (example: Iverilog) and paste the path of the bin folder path(which you have copied in earlier steps mentioned) in variable value

Any Issues Refer to the Link: [Verliog Installation Guide - Windows]

EDA Play Ground

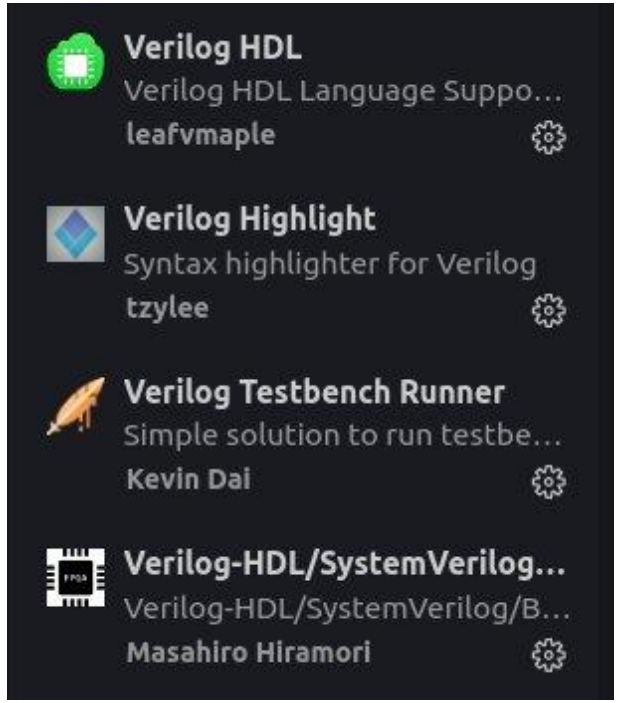
- You need to have internet access to run your verilog codes. This environment provides two places to write codes, one for modules and another for writing testbench.
- To run the files ,select these below given option in the left side toolbar available
  - Testbench+design :- System Verilog/Verilog
  - Tools and Simulator :- Icarus Verilog 12
  - Compile options :- give all your filenames with.v extension which you want to run(you can give multiple files here) after the flags given

Execution of Code

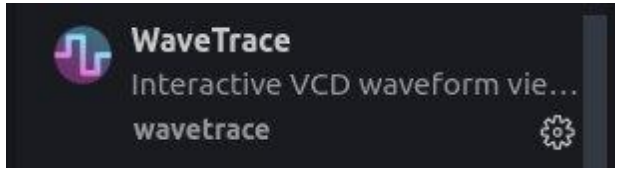
1. File Formats:
   - Modules and testbenches must have the extension (.v)
   - Dumpfiles should have the extension (.vcd)
2. Simulating the code files:
   - To compile the written code, just type:
     iverilog <FileName> <Included Modules>
   - To execute the compiled code, as usual run: ./a.out
   - To checkthe waveforms on GTKWave, use the command:
     gtkwave <dumpFileName>
3. To observe waveforms on GTKWave,select the module name in the left side section, then drag whichever variable you want to observe.
4. The above Procedures need to be executed in the terminal (make sure you enter the terminal when you are in the specific folder).

VS Code Extensions

1. Better to use a Linux system than Windows
2. Better to use installed version than on EDA Playground



For Running Verilog Codes



For GTKWave Plot tabs

External References

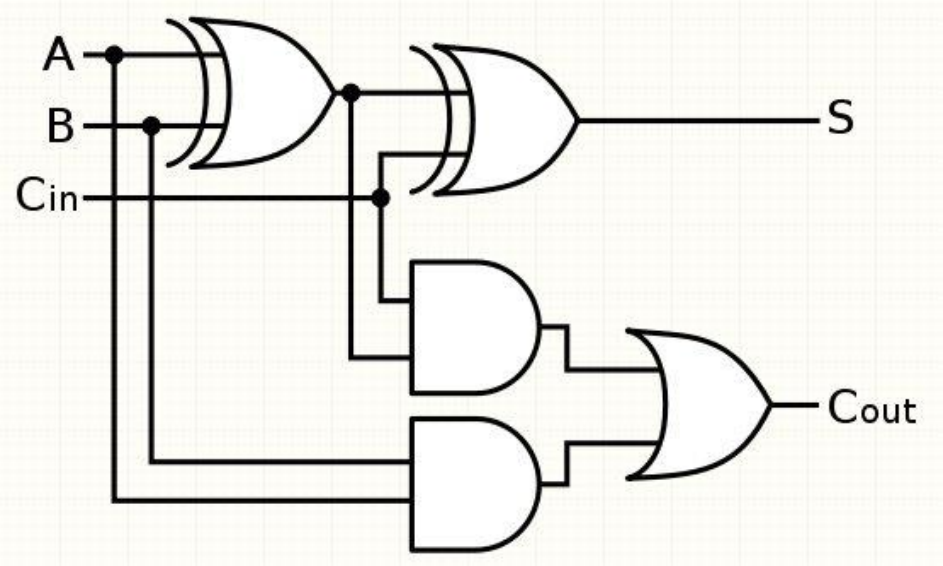
- Chip Verify - Overall reference
- SOC - Good examples for Structural vs Behavioral
- HDL Bits - Concept and Practice
- ASIC World
- **Textbooks:**
  - Verilog HDL: A Guide to Digital Design and Synthesis by Samir Palnitkar
  - Verilog HDL synthesis – A practical Primer by J.Bhasker
  - Fundamentals of Digital Logic with Verilog Design by Stephan Brown and Zvonko vranesik

Behavioral Representation

- All the Electronic Components work based on some Scientific Principles involved
- Most of the simulations we do are Based on Mathematical Models of the Devices which are derived from their behaviour
- In Verilog, We can code its Behavior
- Modeling only the functionality of the design Irrespective of the exact circuit schematic. Basically, specifying how the device should responds to a set of input.
- Various Methods:
  - Truth Table
  - Boolean expression (i.e logic expression)
  - Written Algorithm

Example Circuit

1 – bit Full Adder



Input			Output	
A	B	C_in	Sum	C_out
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

Behavioral Representation

FA_truthtable.v

```
module FA_truthtable(input A, input B, input Cin, output reg Cout, output reg Sum);
    always @(A,B,Cin) begin
        if (A == 1'b0 && B == 1'b0 && Cin == 1'b0) begin
            Cout = 1'b0;
            Sum = 1'b0;
        end
        if (A == 1'b0 && B == 1'b0 && Cin == 1'b1) begin
            Cout = 1'b0;
            Sum = 1'b1;
        end
        if (A == 1'b0 && B == 1'b1 && Cin == 1'b0) begin
            Cout = 1'b0;
            Sum = 1'b1;
        end
        if (A == 1'b0 && B == 1'b1 && Cin == 1'b1) begin
            Cout = 1'b1;
            Sum = 1'b0;
        end
        if (A == 1'b1 && B == 1'b0 && Cin == 1'b0) begin
            Cout = 1'b0;
            Sum = 1'b1;
        end
        if (A == 1'b1 && B == 1'b0 && Cin == 1'b1) begin
            Cout = 1'b1;
            Sum = 1'b0;
        end
        if (A == 1'b1 && B == 1'b1 && Cin == 1'b0) begin
            Cout = 1'b1;
            Sum = 1'b0;
        end
        if (A == 1'b1 && B == 1'b1 && Cin == 1'b1) begin
            Cout = 1'b1;
            Sum = 1'b1;
        end
    end
endmodule
```

FA_expression.v

```
module FA_expression(input A, input B, input Cin, output Cout, output Sum);
  assign Cout = ((A ^ B)&Cin) | (A & B);
  assign Sum = A ^ B ^ Cin;
endmodule
```

tb_FA.v

```
`include "FA_truthtable.v"
`include "FA_expression.v"

module tb_FA;
    reg A, B, Cin;
    wire Cout, Sum;
    // FA_truthtable dut(A, B, Cin, Cout, Sum);
    FA_expression dut(A, B, Cin, Cout, Sum);

    initial begin
        // $dumpfile("tb_FA.vcd");
        // $dumpvars(0, tb_FA);
        $monitor("at time %3t: A = %0b, B = %0b, Cin = %0b, Cout = %0b, Sum = %0b", $time, A, B, Cin, Cout, Sum);

        #0 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
endmodule
```

Behavioral Representation


```
[Running] tb_FA.v
at time   0: A = 0, B = 0, Cin = 0, Cout = 0, Sum = 0
at time  10: A = 0, B = 0, Cin = 1, Cout = 0, Sum = 1
at time  20: A = 0, B = 1, Cin = 0, Cout = 0, Sum = 1
at time  30: A = 0, B = 1, Cin = 1, Cout = 1, Sum = 0
at time  40: A = 1, B = 0, Cin = 0, Cout = 0, Sum = 1
at time  50: A = 1, B = 0, Cin = 1, Cout = 1, Sum = 0
at time  60: A = 1, B = 1, Cin = 0, Cout = 1, Sum = 0
at time  70: A = 1, B = 1, Cin = 1, Cout = 1, Sum = 1
[Done] exit with code=0 in 0.025 seconds
```

Output for code with expression

```
[Running] tb_FA.v
at time   0: A = 0, B = 0, Cin = 0, Cout = 0, Sum = 0
at time  10: A = 0, B = 0, Cin = 1, Cout = 0, Sum = 1
at time  20: A = 0, B = 1, Cin = 0, Cout = 0, Sum = 1
at time  30: A = 0, B = 1, Cin = 1, Cout = 1, Sum = 0
at time  40: A = 1, B = 0, Cin = 0, Cout = 0, Sum = 1
at time  50: A = 1, B = 0, Cin = 1, Cout = 1, Sum = 0
at time  60: A = 1, B = 1, Cin = 0, Cout = 1, Sum = 0
at time  70: A = 1, B = 1, Cin = 1, Cout = 1, Sum = 1
[Done] exit with code=0 in 0.027 seconds
```

Output for code with truthtable

Structural Representation

- Here we try to code to get the exact specific circuit that we are intending to design.
- This modelling gives the description which components are used and how they are interconnected between them
- This is also called as netlist
- Various level of description can be given to get more specific design needed.

Structral Representation

FA_structral.v

```v
module FA_structral(input A, input B, input Cin, output Cout, output Sum);
    wire S1C1,C2;
    xor(S1, A, B);
    xor(Sum, S1,Cin);
    and(C1, S1, Cin);
    and(C2, A, B);
    or(Cout, C1, C2);
endmodule
```

```
[Running] tb_FA.v
at time   0: A = 0, B = 0, Cin = 0, Cout = 0, Sum = 0
at time  10: A = 0, B = 0, Cin = 1, Cout = 0, Sum = 1
at time  20: A = 0, B = 1, Cin = 0, Cout = 0, Sum = 1
at time  30: A = 0, B = 1, Cin = 1, Cout = 1, Sum = 0
at time  40: A = 1, B = 0, Cin = 0, Cout = 0, Sum = 1
at time  50: A = 1, B = 0, Cin = 1, Cout = 1, Sum = 0
at time  60: A = 1, B = 1, Cin = 0, Cout = 1, Sum = 0
at time  70: A = 1, B = 1, Cin = 1, Cout = 1, Sum = 1
[Done] exit with code=0 in 0.03 seconds
```

tb_FA.v

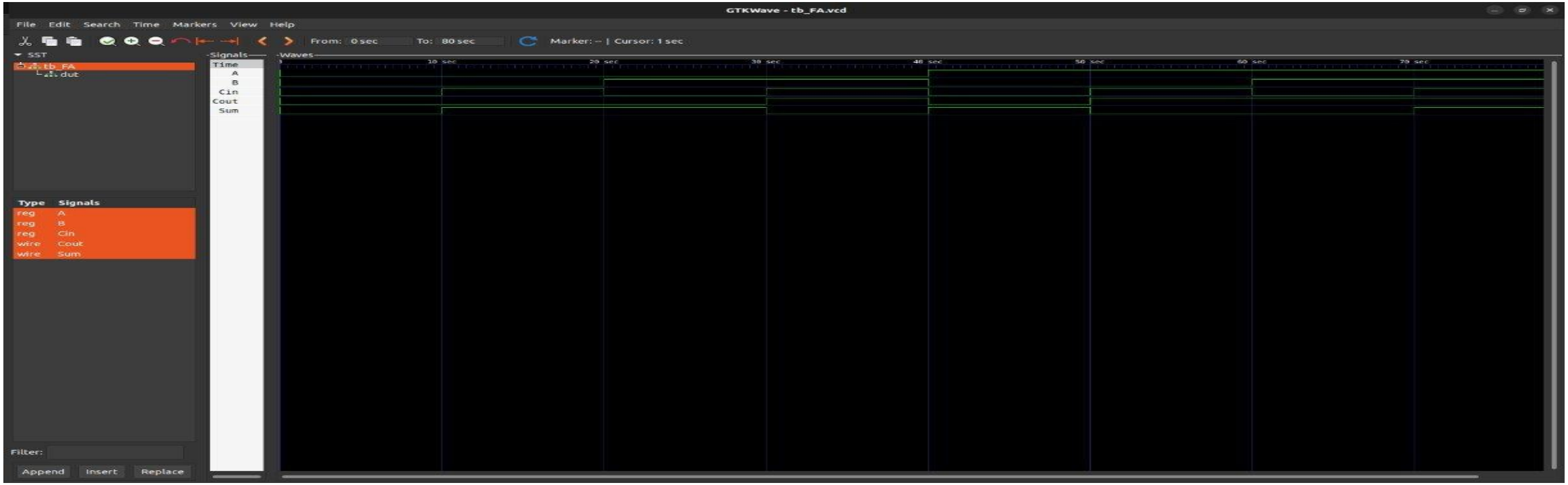
```v
// `include "FA_truthtable.v"
// `include "FA_expression.v"
`include "FA_structral.v"

module tb_FA;
    reg A, B, Cin;
    wire Cout, Sum;
    // FA_truthtable dut(A, B, Cin, Cout, Sum);
    // FA_expression dut(A, B, Cin, Cout, Sum);
    FA_structral dut(A, B, Cin, Cout, Sum);

    initial begin
        // $dumpfile("tb_FA.vcd");
        // $dumpvars(0, tb_FA);
        $monitor("at time %3t: A = %0b, B = %0b, Cin = %0b, Cout = %0b, Sum = %0b", $time, A, B, Cin, Cout, Sum);

        #0 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
endmodule
```

Dumpvariables

- To see the waveforms in GTKwave, you need to export them into it.
- `$dumpfile("<Dump_File_Name>.vcd");`
- `$dumpvars(0, <Testbench_Name>);`
- The above two statements dump the selected outputs into a vcd file for plotting in GTK Wave
- In terminal, Run the command
-

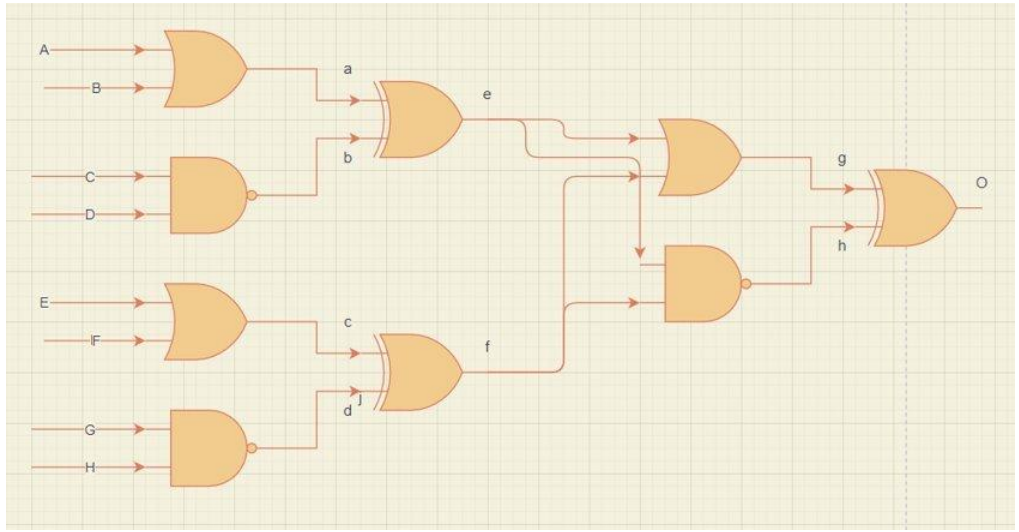
GTK Plots


tb_FA.vcd
BIN
10 s
20 s
30 s
40 s
50 s
60 s
70 s
tb FA.Sum 1
tb FA.Cout 1
tb FA.dut.A 1
tb FA.dut.B 1
tb FA.dut.Cin 1

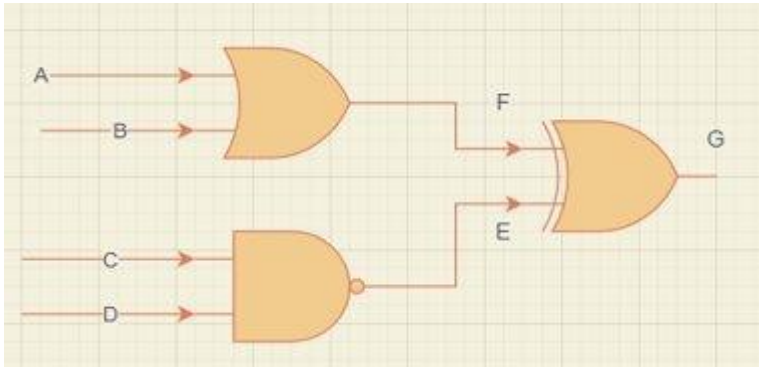

GTKWave - tb_FA.vcd
File Edit Search Time Markers View Help
From: 0 sec
To: 80 sec
Marker: -- | Cursor: 1 sec
SST
tb_FA
dut
Signals
Waves
Time
A
B
Cin
Cout
Sum
Type Signals
reg A
reg B
reg Cin
wire Cout
wire Sum
Filter:
Append
Insert
Replace

Explanation

- Example – Let below circuit be the required design

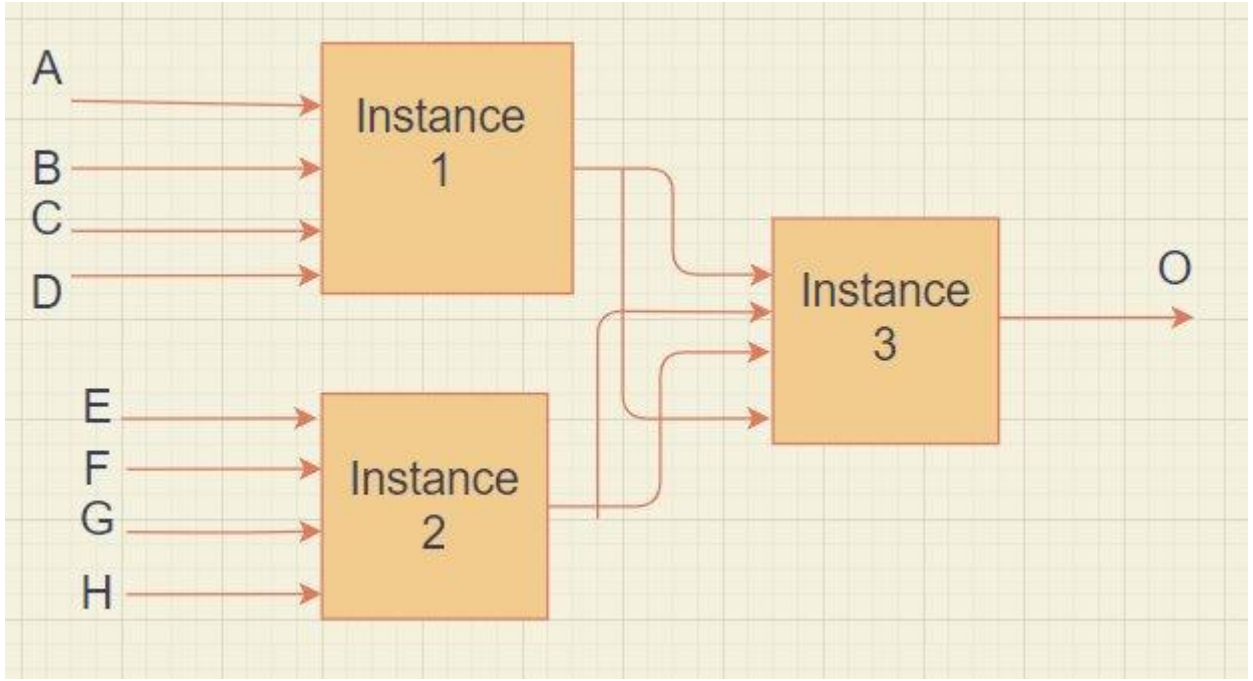


Building block for given design



Instantiating a module

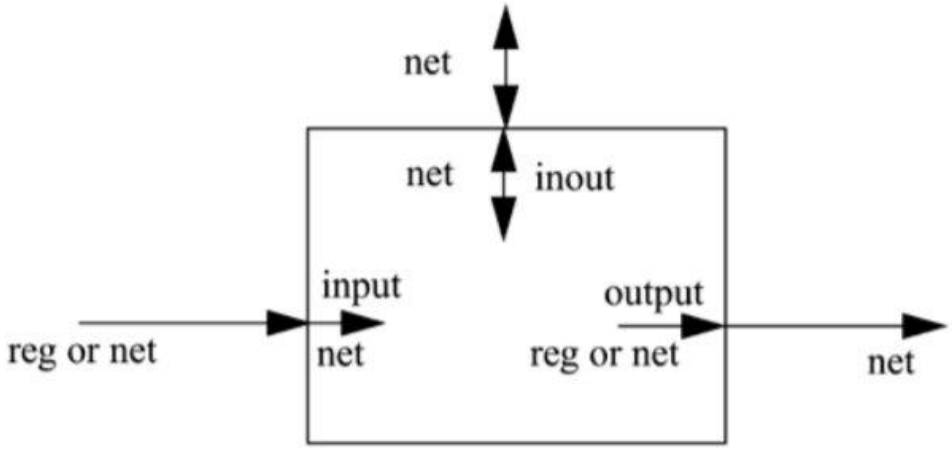
- Building block can be defined as a module and then be instantiated in the main circuit



Instantiating a module

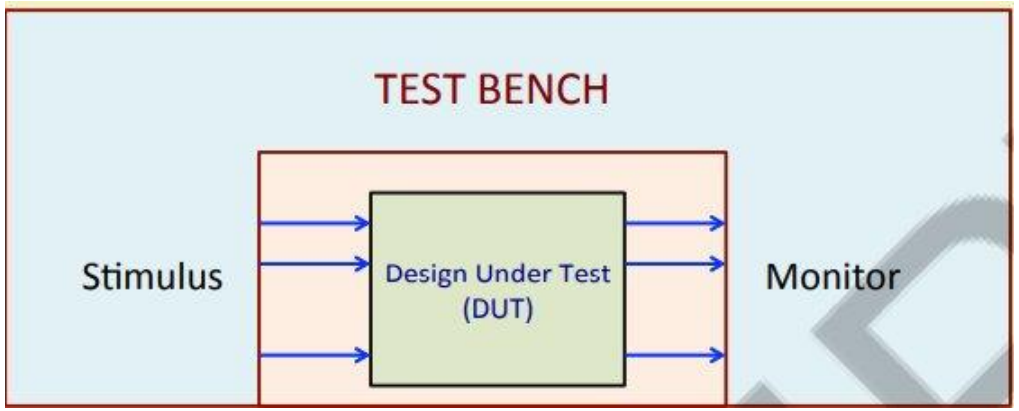
1. **Port Connection rules**
   - Ports can be connected by ordered list or by name
2. **Instantiation of module**
   - module_name instance_name(port_list);
3. **Example**
   - circuit c1(A,B,C,D,O);//Port connection by ordered list //

   Or
   - Circuit c1(.port1(A),.port2(B),.port3(C),.port4(D), .port0(O))



Simulating Verilog Modules

- Using a test bench to verify the functionality of a design coded in Verilog (called Design-under-Test or DUT), comprising of:
  - A set of stimulus for the DUT.
  - A monitor, which captures or analyzes the outputs of the DUT.
- Requirement:
  - The inputs of the DUT need to be connected to the test bench.
  - The outputs of the DUT needs also to be connected to the test bench.



Operators in Verilog

- They take one/two/three values and operate on them to yield a result
- Based on number of operands they can be called unary, binary, and ternary.
- Based on functionality they can be called logical, relational, arithmetic, bitwise etc.

Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
!	logical negation	logical
~	negation	bit-wise
&	reduction AND	reduction
|	reduction OR	reduction
~&	reduction NAND	reduction
~|	reduction NOR	reduction
^	reduction XOR	reduction
~^ or ^~	reduction XNOR	reduction
+	unary (sign) plus	arithmetic
-	unary (sign) minus	arithmetic
{ }	concatenation	concatenation
{{ }}	replication	replication
*	multiply	arithmetic
/	divide	arithmetic
%	modulus	arithmetic
+	binary plus	arithmetic
-	binary minus	arithmetic
<<	shift left	shift
>>	shift right	shift
>	greater than	relational
>=	greater than or equal to	relational
<	less than	relational
<=	less than or equal to	relational
==	case equality	equality
!=	case inequality	equality
&	bit-wise AND	bit-wise
^	bit-wise XOR	bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional

Comments in Verilog:

- There are two ways to write comments
- A one-line comment starts with "//". Verilog skips from that point to the end of line

```
y = !b; // one line comment
```

- A multiple-line comment starts with "/*" and ends with "*/".

```
y = !b; /* multiple
```

- Multiple-line comments cannot be nested

```
/* illegal /* comment*/ */
```

- one-line comments can be embedded in multiple-line comments.

```
/* legal//comment*/
```

Data Types in Verilog

- A variable in Verilog belongs to one of two data types:
  - Net
  - Register
- Must be continuously driven.
- Cannot be used to store a value.
- Used to model connections between continuous assignments and instantiation.
- Retains the last value assigned to it.
- Often used to represent storage elements, but sometimes it can translate to combinational circuits also.

Net Data Type

- Nets represents connection between hardware elements.
- Nets are continuously driven by the outputs of the devices they are connected to.
- Nets are 1-bit values by default unless they are declared explicitly as vectors.
- Default value of a net is “z”
  - This is known as the high Impedence state . Typical refered as open Circuit wire
- Various “Net” data types are supported for synthesis in Verilog: – wire, wor, wand, tri, supply0, supply1, etc.
- “wor” and “wand” inserts an OR and AND gate respectively at the connection.
- “supply0” and “supply1” model power supply connections.
- The Net data type “wire” is most common.

Data Values

Verilog supports 4 value levels to model the functionality of real hardware.

Value Level	Represents
0	Logic 0 state
1	Logic 1 state
x	Unknown logic state
z	High impedance state

Initialization:

- All unconnected nets are set to “z”.
- All register variables set to “x”.

Register Data Type

- In Verilog, a "register" is a variable that can hold a value.
  - Unlike a "net" that is continuously driven and cannot hold any value.
  - Does not necessarily mean that it will map to a hardware register during synthesis.
  - Combinational circuit specifications can also use register type variables.
- Register data types supported by Verilog:
  - reg : Most widely used
  - integer : Used for loop counting (typical use)
  - real : Used to store floating-point numbers
  - time : Keeps track of simulation time (not used in synthesis)

Register Data Type

- "reg" data type:
  - Default value of a "reg" data type is "x".
  - It can be assigned a value in synchronism with a clock or even otherwise.
  - The declaration explicitly specifies the size (default is 1-bit):
- reg x, y; // Single-bit register variables
- reg [15:0] bus; // A 16-bit bus
- Treated as an unsigned number in arithmetic expressions unless mentioned.
- reg signed[15:0] bus_with_Sign // 2's complement representation of the number
- Must be used when we model sequential hardware elements like counters, shift registers, etc...

Assign Statement in Verilog

- The "assign" statement represents continuous assignment, whereby the variable on the LHS gets updated whenever the expression on the RHS changes.

```
assign variable = expression;
```

- The LHS must be a "net" type variable, typically a "wire".
- The RHS can contain both "register" and "net" type variables.
- A Verilog module can contain any number of "assign" statements; they are typically placed in the beginning after the port declarations.
- The "assign" statement models behavioral design style and is typically used to model combinational circuits.

Blocking and Non-Blocking Assignments

- **Blocking Assignments:**
  - Blocks the assignments of other statements until the current statement is done
  - `Variable = expression;`
  - Generally used with Combinational Circuit Design
- **Non-Blocking Assignments:**
  - All assignments are done at once
  - `Variable <= expression;`
  - Generally used with Sequential Circuit Design
  - Mostly in Always blocks as they are building blocks of Sequential circuits

Always Block


```
always @ (event)
    [statement]

always @ (event) begin
    [multiple statements]
end
```

- The block gets executed whenever at least one of the sensitive list (change in the event) gets executed
- Most commonly used in sequential block
- D flip is the basic building block of any sequential circuit
  - Any circuit can be made of D flip flops + combinational logic
- always@(*): sensitive list is every variable that is being assigned inside the block

- Difference between Asynchronous reset and synchronous reset ?
- Difference between Active low and active high reset ? - Which would be safe to use when you want initialized states
- What type of D flip flop is shown here

Active high

All the always blocks that are triggered at once are simultaneously executed. For this purpose, you will need to use non-blocking assignments

```
module D_ff(D,clk,reset,Q);
    parameter N = 3;
    // parameter clk2QDel = 0;
    input [N-1:0] D;
    input clk,reset;
    output reg [N-1:0] Q;

    always @(posedge clk or posedge reset) Q = reset? {N{1'b0}}: D;
endmodule
```

Initial Block


```
initial
    [single statement]

initial begin
    [multiple statements]
end
```

- Not synthesizable
- Will be executed only once for variable initialization
- Starts execution at the very beginning at time = 0 units, finishes once all the statements are executed
- Any number of initial blocks is permitted
- Used mostly ion test benches
- Never use in design modules

```
module tb_D_ff;
parameter N = 3;
    reg [N-1:0] D;
    reg clk, reset;
    wire [N-1:0] Q;

    reg [2:0] delay;
    D_ff DUT(.D(D),.clk(clk),.reset(reset),.Q(Q));
    always #5 clk = ~clk;
    initial begin
        $dumpfile("D_ff.vcd");
        $dumpvars(0);
        $monitor("At time %3t: D = %3b (%d), clk = %1b, reset = %1b, Q = %3b (%d)",$time,D,D,clk,reset,Q,Q);
        clk <= 1'b0; D <= 3'b000; reset <= 1'b1;
        #8 D <= 3'b001; // 8
        #5 reset <= 0; // 13
        #10 D <= 3'b010; // 23
        #3 D <= 3'b011; // 26
        #2 D <= 3'b100; // 28
        #11 D <= 3'b101; // 39
        #13 D <= 3'b110; // 52
        #17 D <= 3'b111; // 69
        #6 $finish; // 75
    end
endmodule
```

For loop


```
for (<initial_condition>; <condition>; <step_assignment>) begin
    // Statements
end
```

Three parts:

1. Initial condition to specify initial values of signals
2. A check to evaluate if the given condition is true
3. Update control variable for the next iteration

- Primarily used to replicate hardware logic in Verilog
- Iterate a set of statements given within the loop as long as the given condition is true.

While loop and do-while loop

- Looping constructs that execute the given set of statements as long as the given condition is true.

1. A while loop first checks if the condition is true and then executes the statements if it is true. If the condition turns out to be false, the loop ends right there.
2. A do while loop first executes the statements once, and then checks for the condition to be true. If the condition is true, the set of statements are executed until the condition turns out to be false. If the condition is false, the loop ends right there.

```
while (<condition>) begin
    // Multiple statements
end

do begin
    // Multiple statements
end while (<condition>);
```

Switch Case statements


```
// Here 'expression' should match one of the items (item 1,2,3 or 4)
case (<expression>)
    case_item1 :    <single statement>
    case_item2,
    case_item3 :    <single statement>
    case_item4 :    begin
                        <multiple statements>
                        end
    default      : <statement>
endcase
```

The case statement checks if the given expression matches one of the other expressions in the list and branches accordingly. It is typically used to implement a multiplexer. The if-else construct may not be suitable if there are many conditions to be checked and would synthesize into a priority encoder instead of a multiplexer.

Force and release; Generate variable; $finish

Generate variable

- Parameter based
- Repeats the hardware units that many number of times
- Synthesizability dependent on usage – Yes in most cases
- No ports must be used as genvar – Not synthesizable
- Can be used with for loop, while loop, if else and switch case statements
- Synthesizability depends on the loop or statements used inside it.

Force and release

- Force statement assigns the given value to the variable and the boolean state of the variable is unchanged irrespective of any number and types of assignments until released.

```
1  // Design for a half-adder
2  module ha ( input   a, b,
3              output  sum, cout);
4
5    assign sum  = a ^ b;
6    assign cout = a & b;
7  endmodule
8
9  // A top level design that contains N instances of half adder
10 module my_design
11     #(parameter N=4)
12         (   input [N-1:0] a, b,
13             output [N-1:0] sum, cout);
14
15     // Declare a temporary loop variable to be used during
16     // generation and won't be available during simulation
17     genvar i;
18
19     // Generate for loop to instantiate N times
20     generate
21         for (i = 0; i < N; i = i + 1) begin
22           ha u0 (a[i], b[i], sum[i], cout[i]);
23         end
24     endgenerate
25 endmodule
```

Thank you