Verilog Tutorial

VLSI DESIGN-MONSOON 2021

Agenda

Understanding the basic use of verilog

To understand the Design Representation

To primarily get familiar with the basic building block of verilog coding.

Getting enough conceptual understanding so that we can code some standard circuits in next tutorial.

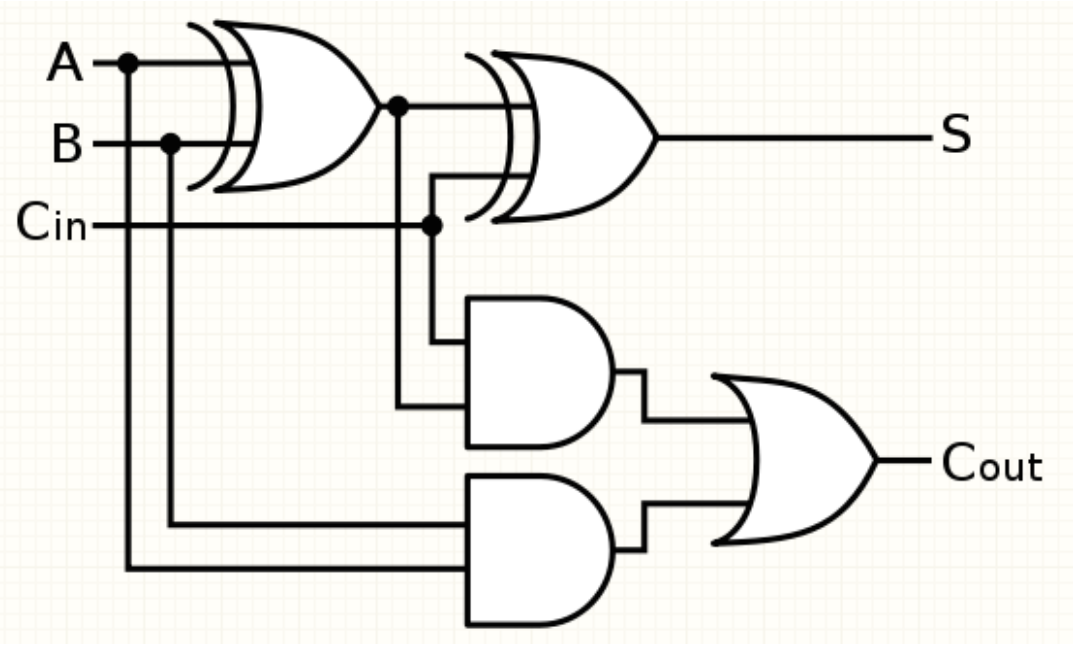
VLSI Design – Need For Verilog

Design complexity increasing rapidly
– Increased size and complexity
– Fabrication technology improving
– CAD tools are essential
– Conflicting requirements like area, speed, and energy consumption
• The present trend
– Standardize the design flow
– Emphasis on low-power design, and increased performance

Behavioral Representation

- 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.
- This can be done by various methods:-
  - Truth Table
  - Boolean expression (i.e logic expression)
  - Algorithm written in standard HDL language like C , System Verilog , VHDL or Verilog

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



1 Bit-full Adder

Let's see the code for "carry" term- Modelling using Truth table

```
primitive carry (Cy, A, B, C);
  input A, B, C;
  output Cy;
   table
      // A B C Cy
          1 1 ? : 1 ;
          1 ? 1 : 1 ;
          ? 1 1 : 1 ;
          0 0 ? : 0 ;
          0 ? 0 : 0 ;
          ? 0 0 : 0 ;
    endtable
endprimitive
```

Inputs			Outputs	
A	B	C_{in}	Sum	Carry
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

This method is not recommended for behavioral modelling

Using Boolean Expression

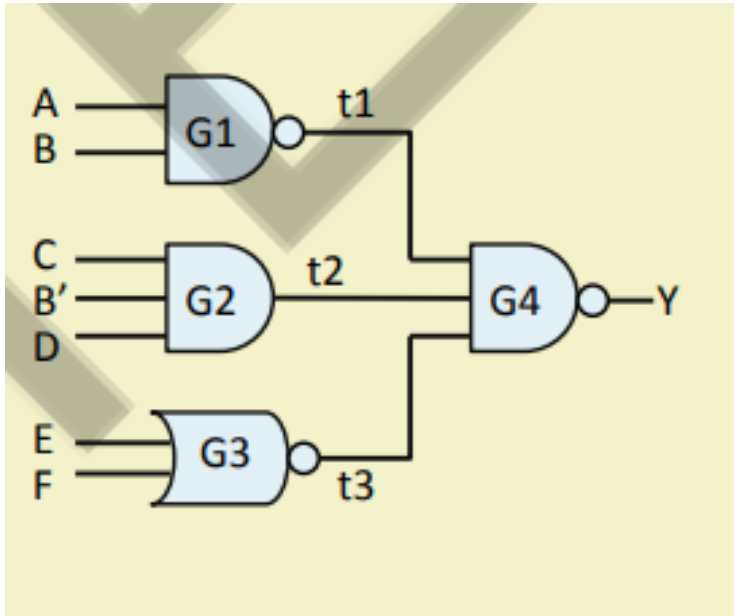
```
module carry(Sum, Cout, A, B, Cin);
    input A, B, Cin;
    output Sum , Cout;
    assign Sum= A ^ B ^ Cin;
    assign Cout = (A&B) | (B & Cin) | (A & Cin)
endmodule
```

Structural Modelling

- 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.

Let's take a Basic Combinational Circuit

```
module logic(Y,A,B,C,E,F);
  input A,B,C,D,E,F;
  output Y;
  wire t1,t2,t3;
  NAND G1 (t1,A,B);
  AND G2 (t2,C,~B,D);
  NOR G3 (t3,E,F);
  NAND G4 (Y,t1,t2,t3);
endmodule
```

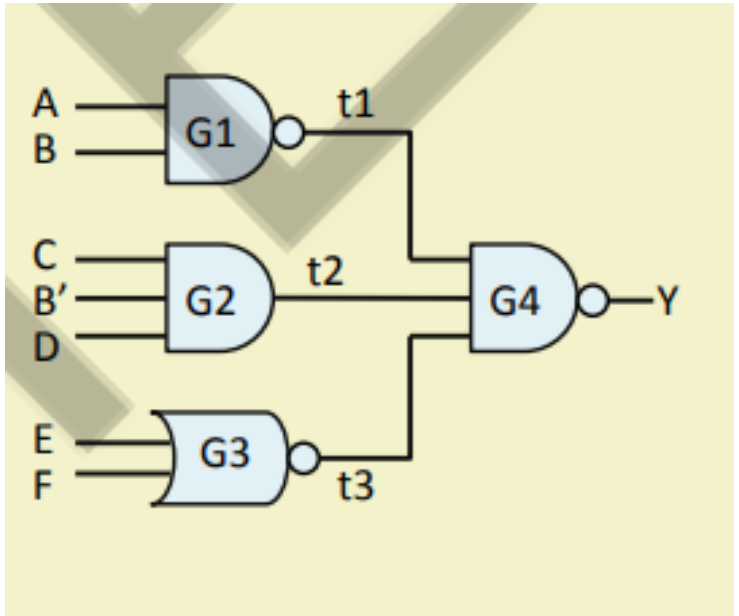


Behavioral Counterpart

Find the Boolean expression of Y

```
module logic (Y,A,B,C,D,E,F);
  input A,B,C,D,E,F;
  output Y;
  wire t1,t2,t3;
  assign t1= ~(A & B);
  assign t2= (C & (~B) & D);
  assign t3= ~(E | F);
  assign Y= ~(t1 & t2 & t3);

endmodule
```

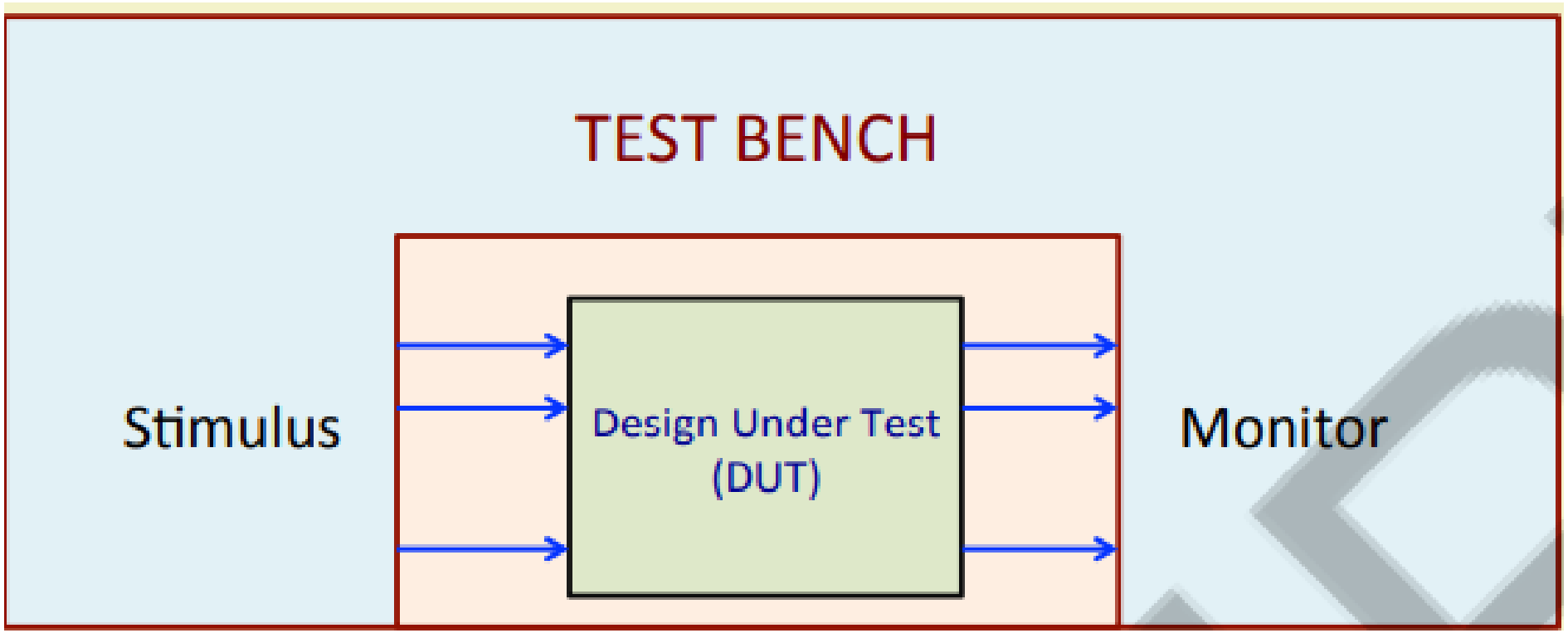


Simulating Verilog Modules

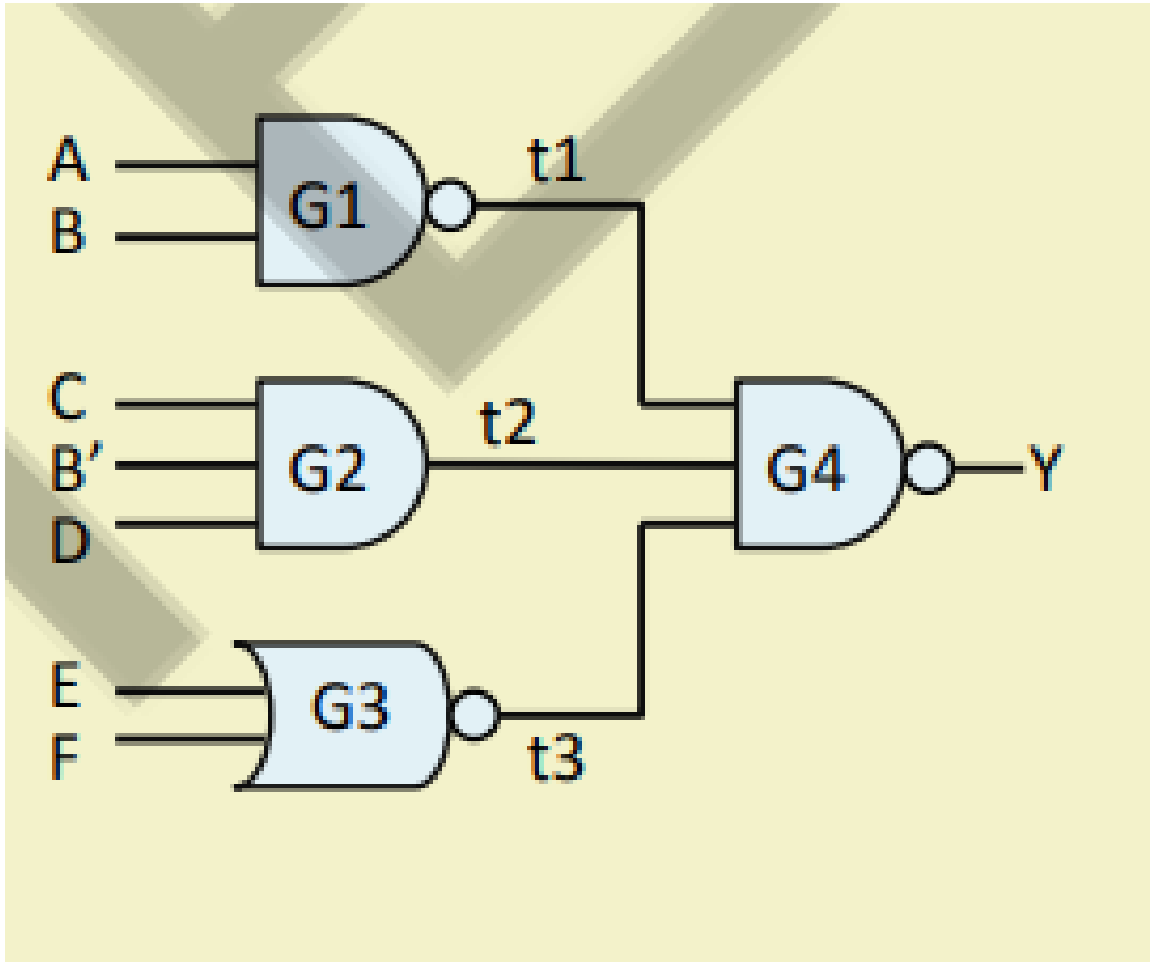
- Using a test bench to verify the functionality of a design coded in Verilog (called Design-under-Test or DUT), comprising of:
  - – A set of stimulus for the DUT.
  - – A monitor, which captures or analyzes the outputs of the DUT.
- Requirement:

– 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.



TESTING OF CIRCUIT



Previously taken circuit

```
module logic(Y,A,B,C,E,F);
    input A,B,C,E,F;
    output Y;
    wire t1,t2,t3;
    NAND G1 (t1,A,B);
    AND G2 (t2,C,~B,D);
    NOR G3 (t3,E,F);
    NAND G4 (Y,t1,t2,t3);
endmodule
```

Testbench

```
module testbench;
 reg A,B,C,D,E,F; wire Y;
 logic DUT(A,B,C,D,E,F,Y);
 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); // Intelligent Print Statement
   #5 A=1; B=0; C=0; D=1; E=0; F=0;
   #5 A=0; B=0; C=1; D=1; E=0; F=0;
   #5 A=1; C=0;
   #5 F=1;
   #5 $finish;
  end
endmodule
```

- Simulation results:

```
0 A=x, B=x, C=x, D=x, E=x, F=x, Y=x
5 A=1, B=0, C=0, D=1, E=0, F=0, Y=x
8 A=1, B=0, C=0, D=1, E=0, F=0, Y=1
10 A=0, B=0, C=1, D=1, E=0, F=0, Y=1
13 A=0, B=0, C=1, D=1, E=0, F=0, Y=0
15 A=1, B=0, C=0, D=1, E=0, F=0, Y=0
18 A=1, B=0, C=0, D=1, E=0, F=0, Y=1
20 A=1, B=0, C=0, D=1, E=0, F=1, Y=1
```

Running in iverilog

1.

a) iverilog -o (random name) example.v example-test.v

b) vvp random name

2.

a) iverilog example.v example-test.v

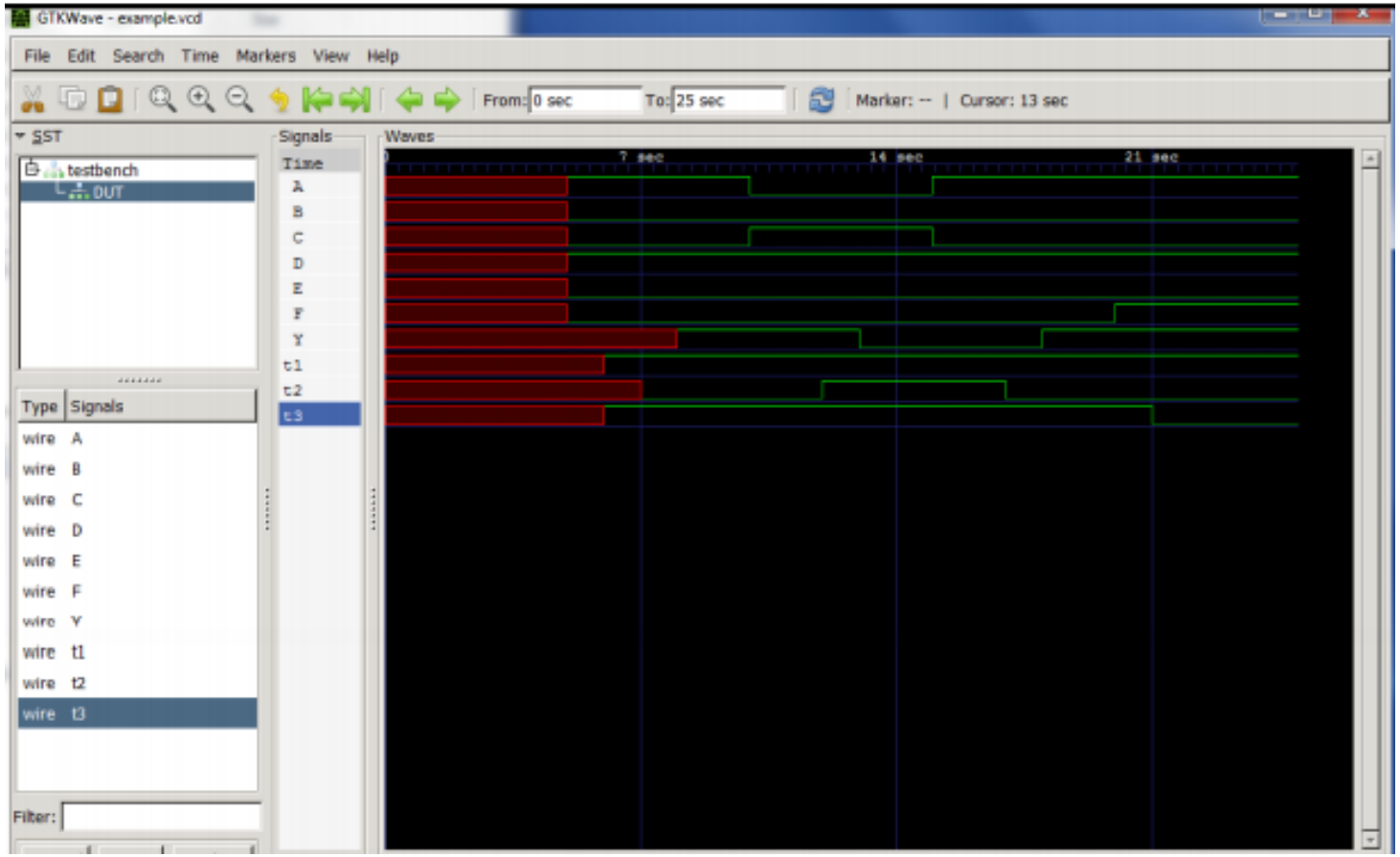
b) ./a.out

The image is just an example might not be the exact result for the previous circuit.

The Initial number in the simulation result is the time unit.

Dumpvariable

```
module testbench;
  reg A,B,C,D,E,F; wire Y;
  logic DUT(A,B,C,D,E,F,Y);
  initial
    begin
      $dumpfile (“logic.vcd”);
      $dumpvars (0,testbench);
      $monitor ($time,” A=%b, B=%b, C=%b, D=%b, E=%b, F=%b, Y=%b”, A,B,C,D,E,F,Y); // Intelligent Print Statement
      #5 A=1; B=0; C=0; D=1; E=0; F=0;
      #5 A=0; B=0; C=1; D=1; E=0; F=0;
      #5 A=1; C=0;
      #5 F=1;
      #5 $finish;
    end
endmodule
```



GTKWAVE

To display the waveforms

Run the command:

gtkwave example.vcd

Features of Verilog Language

1. Conventions in Verilog:

- The basic lexical conventions in Verilog HDL are similar to those in the C programming language.
- Verilog HDL is a case-sensitive language.
- All keywords in Verilog are in lowercase.

Comments

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
line comment*/
```

Multiple-line comments cannot be nested

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

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

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

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

Operators

- 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.

Numbers in Verilog

Numbers - <size>'<base><number>

2'b01 // 2-bit binary number

1'h0 // 1-bit hexadecimal number

12'h200// 12-bit hexadecimal number;

16'd255 // This is a 16-bit decimal number.

-14'h1212 // signed negative number

8'b1000_1000 //8-bit binary number

6'hA //stored as 001010

6'hAB //stored as 101011 (AB is 1010_1011)

If size is not specified, then default size is simulator and machine specific(>= 32-bit), If base is not specified then decimal is taken by default.

Unknown values – Denoted by **x**

2'bx // 2-bit binary unknown number; stored as xx

2'b0x // 2-bit binary unknown number with unknown LSB; stored as 0x

High Impedance values – Denoted by **z**

8'bz // 8-bit high impedance value; stored as zzzzzzzz

1.4 Strings -

Strings should be enclosed in double quotes

Strings cannot take multiple lines

```
"hello world" // example of string
```

1.5 Keywords –

All keywords in verilog are in lower case

1.6 Identifiers – Names given to an object such as function.

Identifiers can begin with alphabets or underscore character.

Identifiers may contain characters

```
input INPUT;// 'input' is keyword, 'INPUT' is identifier
wire _aBc;//'wire' is keyword, '_aBc' is identifier
reg hi!; //'reg' is keyword, 'hi!' Is identifier
```

Modules – Basic unit of HDL

In general, Multiple modules are made to create an exact circuit schematic with required functionality.

POINTS TO REMEMBER:-

– A module cannot contain definition of other modules.

– A module can, however, be instantiated within another module.

– Instantiation allows the creation of a hierarchy in Verilog description.

```
module verilog (output pins, input pins);
        Input port declaration;
        Output Port declaration;
        Local net;
        Assignment ;
        Program logic;
endmodule
```

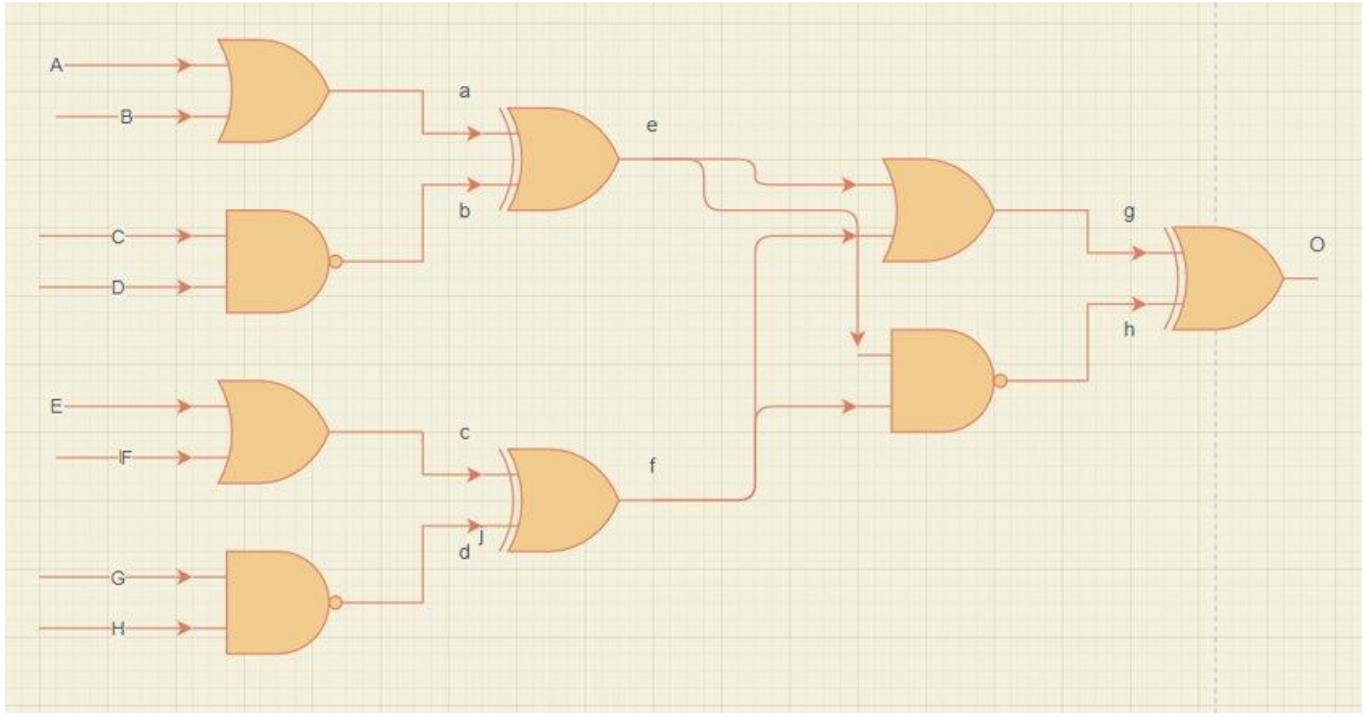
Recommended Practice

- For parameters always assign the output first and then input
- Systematic structure
- Before writing the code, select the representation.

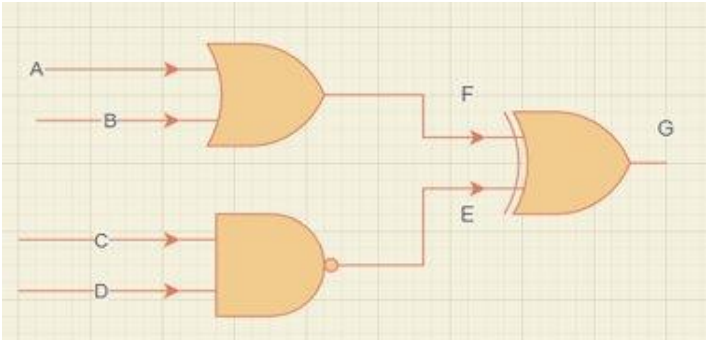
Ports - Ports allow communication between a module and its environment.

Explanation

Example – Let below circuit be the required design

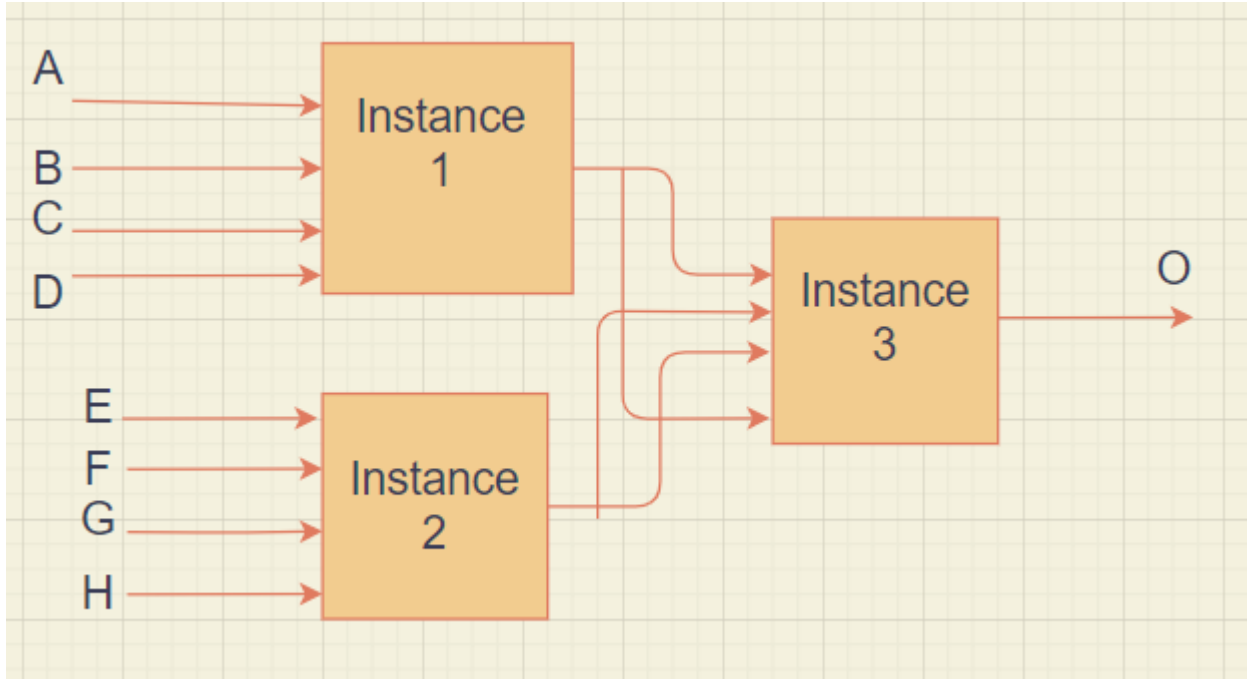


Building block for given design



A,B,C,D,E,F,G,H and O are the ports

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




```
module circuit(port1, port2, port3, port4, port0);//Module name and port list

input port1:

input port2:

input port3:

input port4:

output port0:


//Module internals//


endmodule // End of module
```

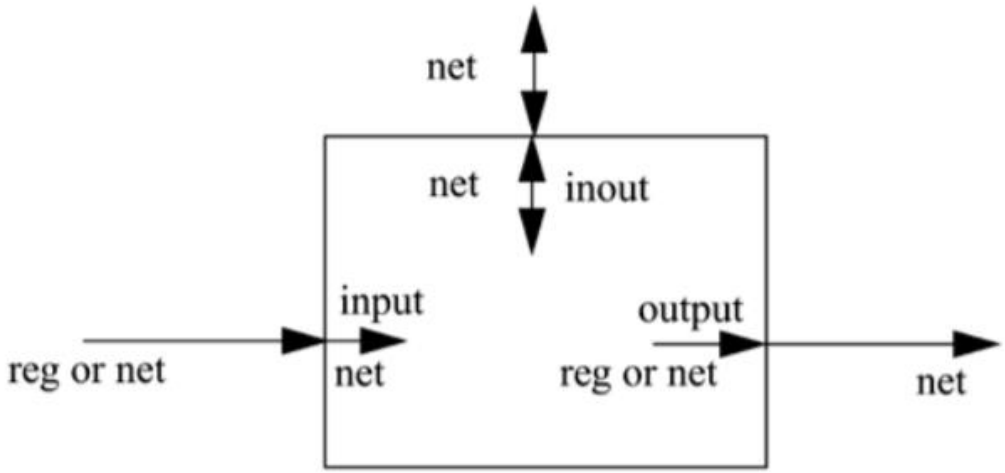
Instantiating a module

1. Port Connection rules

Ports can be connected by ordered list or by name

2. Instantiation of module

```
module_name instance_name(port_list);
```



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))
//instantiation by port name//
```

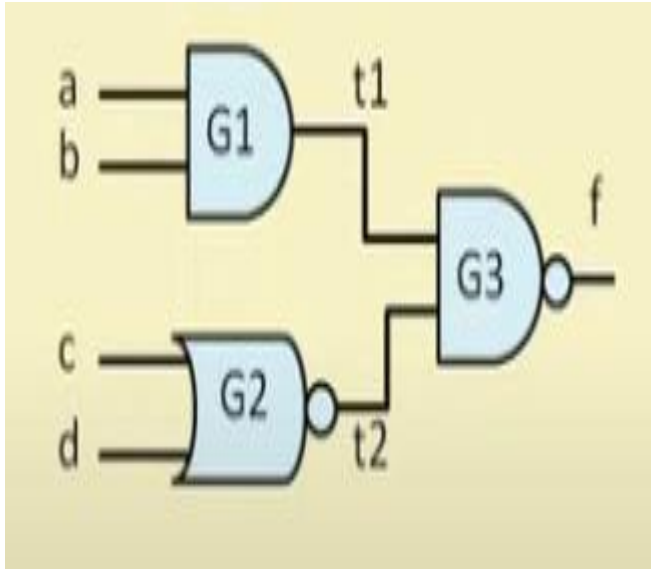
Getting familiar with coding

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

POINT TO REMEMBER: AND GATE is different than AND operator.

Revision

```
module logic (f,a,b,c,d);
  input a,b,c,d;
  output f;
  wire t1,t2;
  AND G1 (t1,a,b);
  EXOR G2 (t2,c,d);
  NAND G3 (f,t1,t2);
endmodule
```



Assign Statement

– 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.

Data Types in Verilog

- A variable in Verilog belongs to one of two data types:

a) Net

- Must be continuously driven.
- Cannot be used to store a value.
- Used to model connections between continuous assignments and instantiation.

b) Register

- 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”

"z"- This is known as the high Impedence state . Typical refered as open Circuit wire (more relevant in switch modelling).

- 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.

Example of wand and wor

```
module use_wire (f,A,B,C,D);
   input A, B, C, D;
   output f;
   wire f; // net f declared as ‘wire’
   assign f = A & B;
   assign f = C | D;
endmodule
```

```
module use_wand(f,A,B,C,D);
   input A,B,C,D;
   output f;
    wand f; //net declared as wand
   assign f= A & B ;
   assign f= C | D;
endmodule
```

Data Values and strength

• Verilog supports 4 value levels and 8 strength levels to model the functionality of real hardware.

– Strength levels are typically used to resolve conflicts between signal drivers of different strengths in real circuits.

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”.

(b) 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:

i. reg : Most widely used

ii. integer : Used for loop counting (typical use)

iii. real : Used to store floating-point numbers

iv. 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.

– Must be used when we model actual sequential hardware elements like counters, shift registers, etc

Understanding Always block

```
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
```

32-bit counter with synchronous reset.

- Count value increases at the positive edge of the clock.
- If "rst" is high, the counter is reset at the positive edge of the next clock.
- Key detail is we need to define count as register data.

Asynchronous Counter

```
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
```

32-bit counter with asynchronous reset.

- Here reset occurs whenever "rst" goes high.
- Does not synchronize with clock.

Initial block

- All statements inside an “initial” statement constitute an “initial block”.
  - Grouped inside a “begin … end” structure for multiple statements.
    - The statements starts at time 0 and execute only once.
    - If there are multiple “initial” blocks, all the blocks will start to execute concurrently at time 0.
- The “initial” block is typically used to write test benches for simulation:
  - Specifies the stimulus to be applied to the design-under-test (DUT).
  - Specifies how the DUT outputs are to be displayed / handled.
  - Specifies the file where the waveform information is to be dumped.

Initial Block

```
module testbench_example;
  reg a, b, cin, sum, cout;

  initial
    cin = 1'b0;

  initial
    begin
      #5 a = 1'b1; b=1'b1;
      #5 b = 1'b0;
    end

  initial
    #25 $finish;

endmodule
```

- The three “initial” blocks execute concurrently.
- The first block executes at time 0.
- The third block terminates simulation at time 25 units.

Always Block

- All behavioral statements inside an “always” statement constitute an “always block”.

– Multiple statements are grouped using “begin … end”.

- An “always” statement starts at >me 0 and executes the statements inside the block repeatedly, and never stops.

– Used to model a block of activity that is repeated indefinitely in a digital circuit.

– For example, a clock signal that is generated continuously.

– We can specify delays for simulation; however, for real circuits, the clock generator will be active as long as there is power supply.

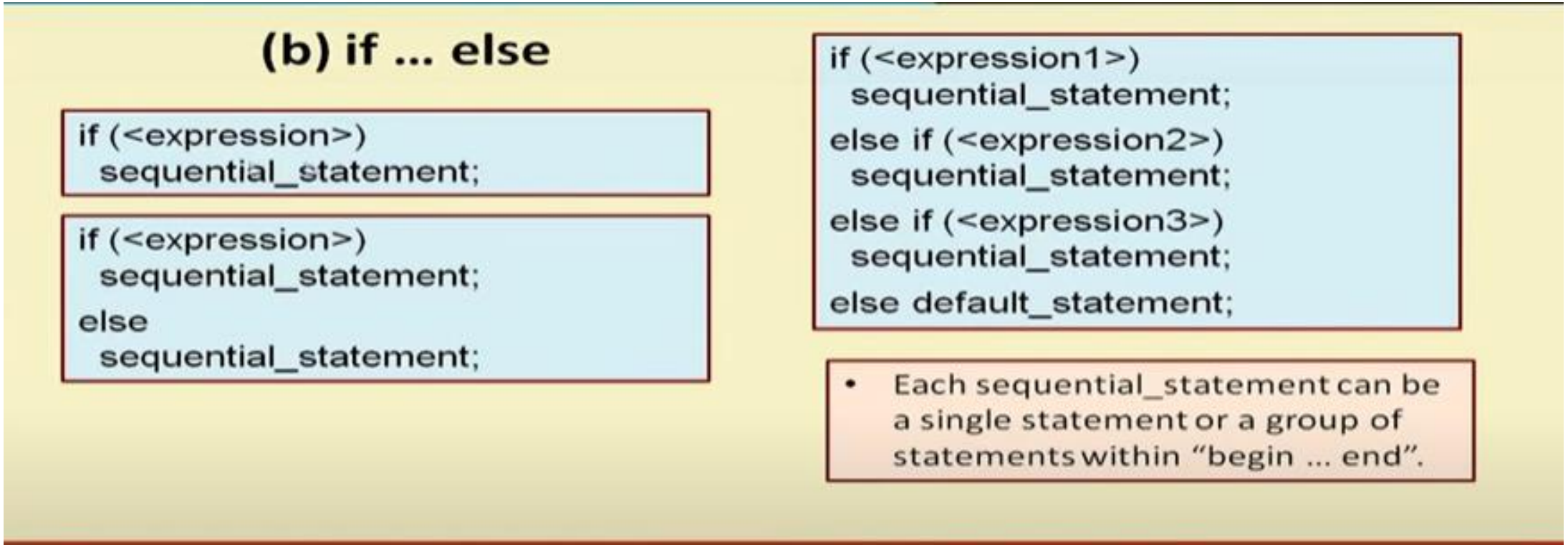
Basic syntax of "always" block:

```
always @(event_expression)
begin
   sequential_statement_1;
   sequential_statement_2;
   …
   sequential_statement_n;
end
```

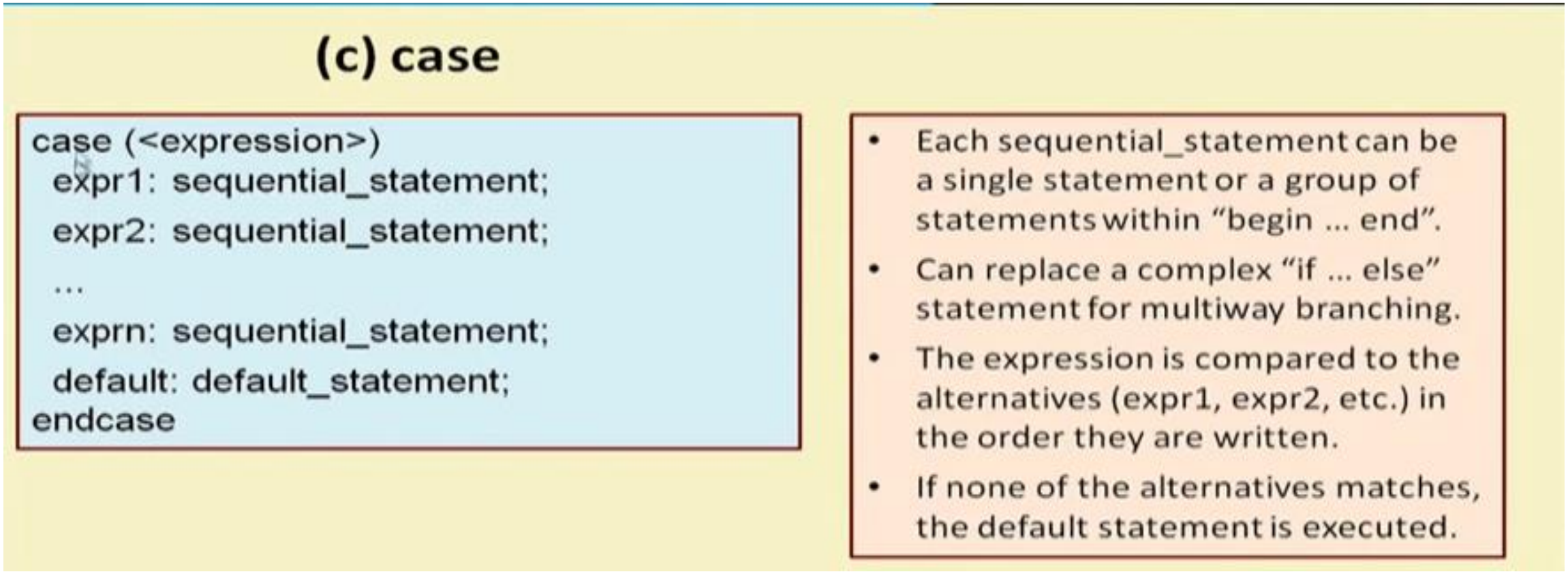
ALWAYS BLOCK

A module can contain any number of "always" blocks, all of which execute concurrently.

- The @(event_expression) part is required for both combinational and sequential circuit descriptions.


(b) if ... else
if (<expression>)
sequential_statement;
if (<expression>)
sequential_statement;
else
sequential_statement;
if (<expression1>)
sequential_statement;
else if (<expression2>)
sequential_statement;
else if (<expression3>)
sequential_statement;
else default_statement;
• Each sequential_statement can be a single statement or a group of statements within "begin ... end".

continued


(c) case
case (<expression>)
expr1: sequential_statement;
expr2: sequential_statement;
...
exprn: sequential_statement;
default: default_statement;
endcase
• Each sequential_statement can be a single statement or a group of statements within "begin ... end".
• Can replace a complex "if ... else" statement for multiway branching.
• The expression is compared to the alternatives (expr1, expr2, etc.) in the order they are written.
• If none of the alternatives matches, the default statement is executed.

Switch case:

(d) "while" loop

```
while (<expression>)
   sequential_statement;
```

- The "while" loop executes until the expression is *not true*.
- The sequential_statement can be a single statement or a group of statements within "begin ... end".

Example:

```
integer mycount;
initial
begin
  while (mycount <= 255)
    begin
      $display ("My count:%d", mycount);
      mycount = mycount + 1;
    end
end
```

While loop

For loop .

Syntax for For loop

for (expr1; expr2; expr3)

sequential statement;

Expr1:- Initial Condition

Expr2:- Condition to be satisfied

Expr3:- updating the incremental variable.

Primarily used to initialize array or memory element.

Example:

```
integer mycount;
reg [100:1] data;
integer i;
initial
  for (mycount=0; mycount<=255;
mycount=mycount+1)
      $display (“My count:%d”, mycount);
initial
  for (i=1; i<=100; i=i+1)
       data[i] = 1’b0;
```

End of tutorial -1

Topics for next tutorial

- Mainly focusing on blocking/non-blocking assignment
- More examples on Combinational circuit (multiplexer , encoder , decoder etc.)
- More example on sequential Circuit (Flip flops , counter, etc.)
- Further tutorial will have more regressive coding material so clear the basics properly before heading into that

REFERENCE :-

Verilog HDL: A Guide to Digital Design and Synthesis by Samir palnitkar

This slides for concise content

External Links -

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

2.http://www.asic-world.com/verilog/veritut.html