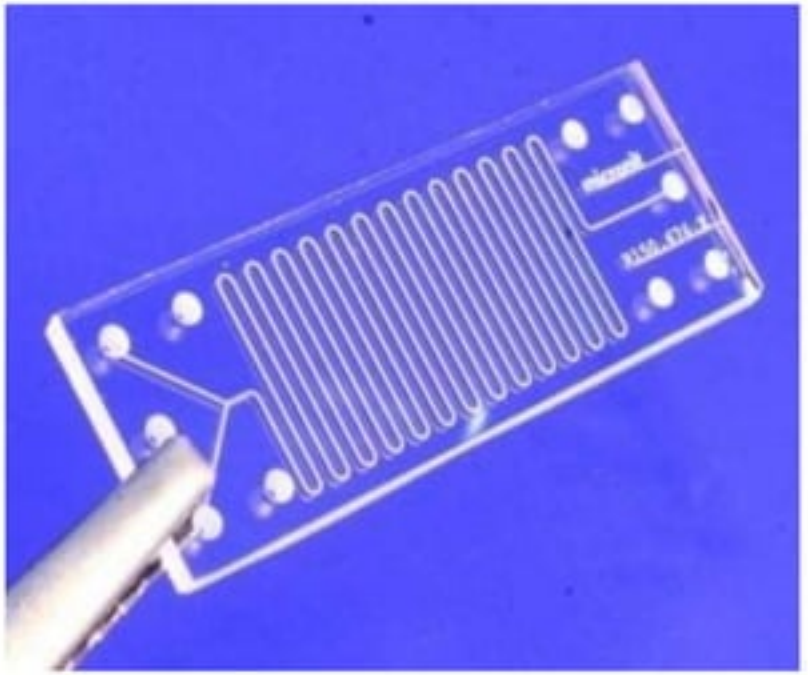



VERILOG HDL

Mantra

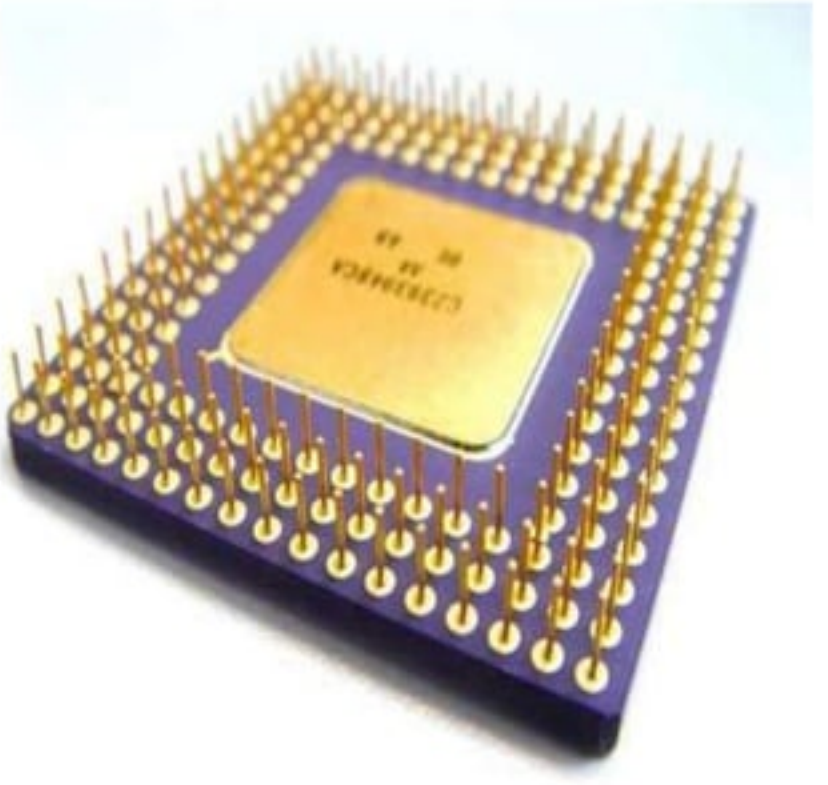
VLSI


Table of Contents



Welcome to verilog

1. Introduction to VLSI
2. Introduction to Verilog HDL
3. History
4. Design Methodology
5. Verilog Keywords
6. Verilog data types
7. Abstraction
8. Gate level Abstraction
9. Data flow level Abstraction
10. Behavioral level Abstraction
11. Switch level Abstraction
12. Advance verilog Keywords



Introduction to VLSI

- Small scale integration (SSI) 1 – 10 gates
- Medium Scale Integration (MSI) 10 – 100 gates
- Large Scale Integration (LSI) 100 – 1000 gates
- Very Large Scale Integration (VLSI) 1000 – 100000 gates
- Ultra High Scale Integration (ULSI) > 100000 gates

Design Flow in VLSI

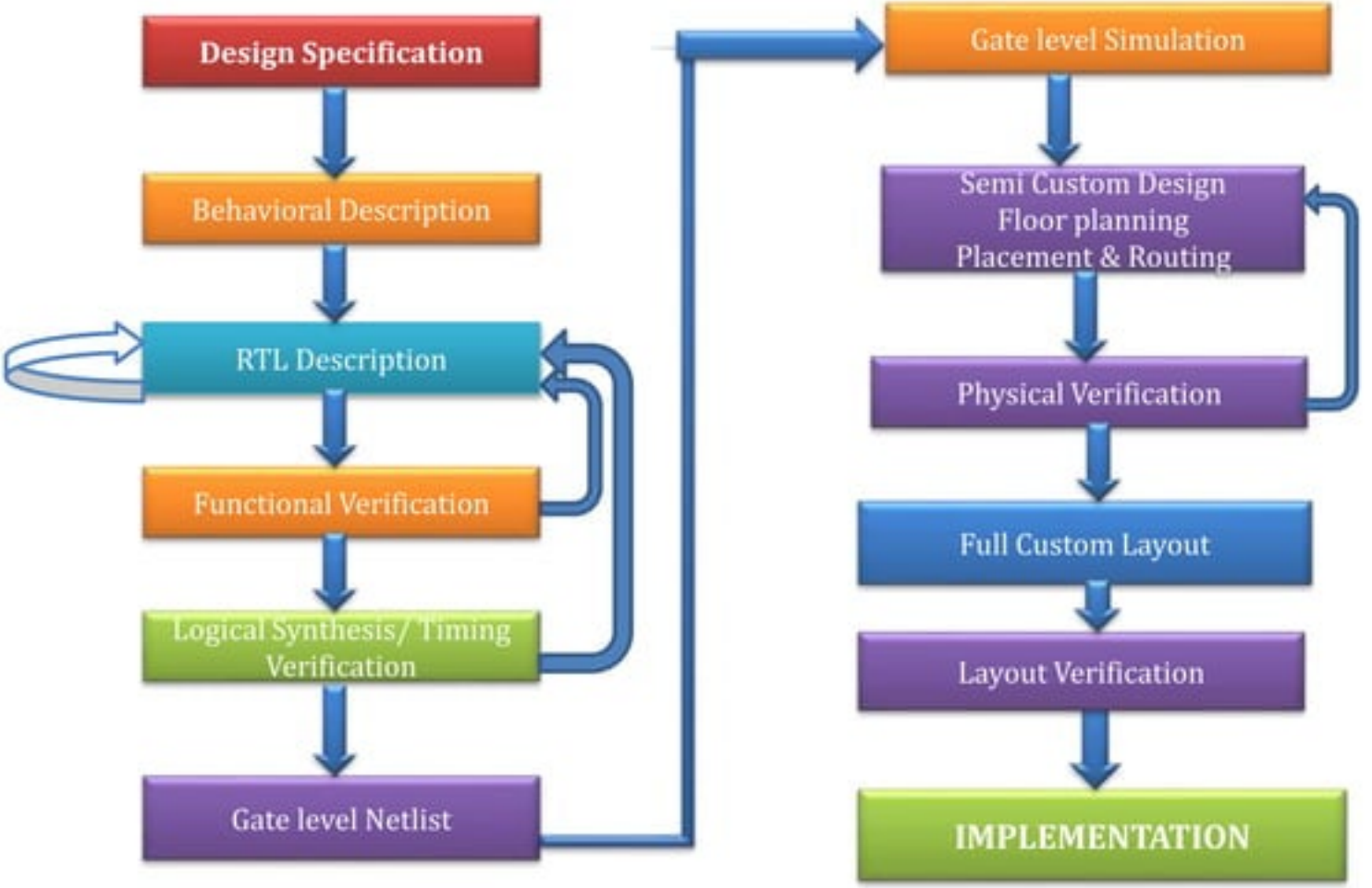
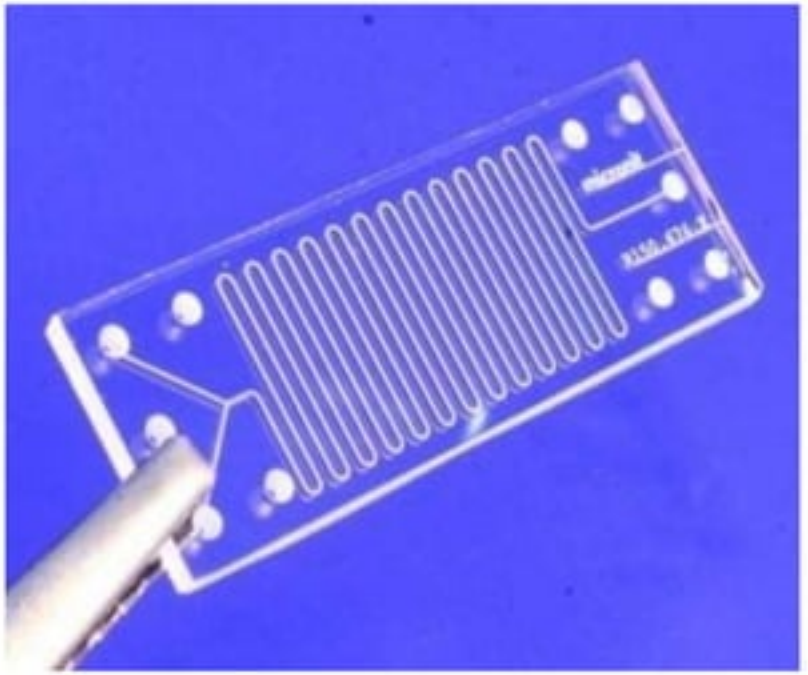


Table of Contents



Introduction to Verilog HDL

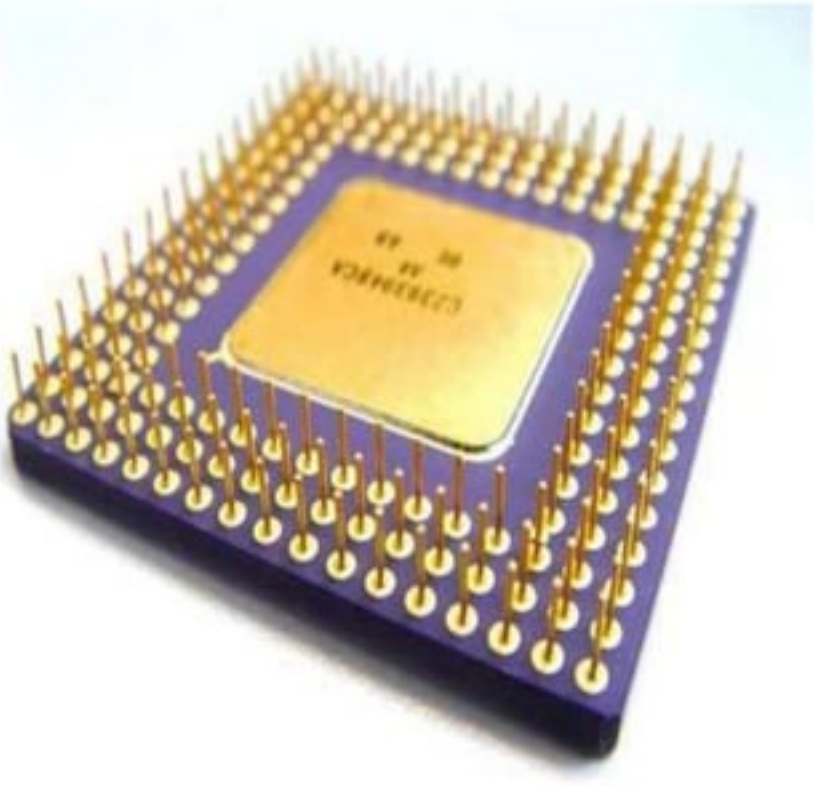
- Programming language.
- Hardware Description Language.
- Understand the Behavior of Hardware.
- Its syntax are similar to C language.
- Easy to Learn & Use.
- The Verilog HDL is both a behavioral and structural language.
- Verilog is case sensitive language.

Models in Verilog

- Models in the Verilog HDL can describe both the function of a design and the components .
- It can also define connections of the components in the design.

Welcome to verilog

1. Introduction to VLSI
2. Introduction to Verilog HDL
3. History
4. Design Methodology
5. Verilog Keywords
6. Verilog data types
7. Abstraction
8. Gate level Abstraction
9. Data flow level Abstraction
10. Behavioral level Abstraction
11. Switch level Abstraction
12. Advance verilog Keywords



History

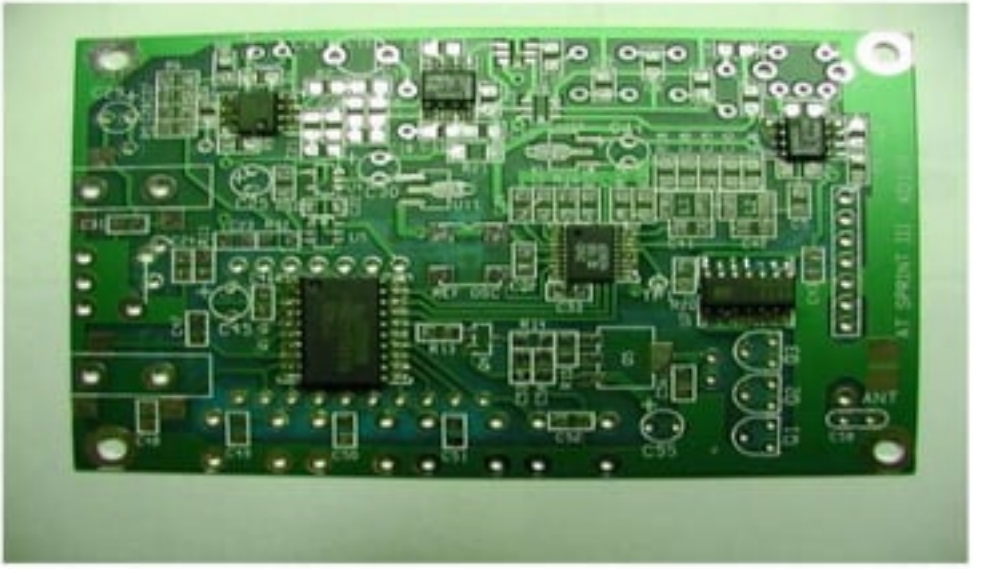
- Verilog was invented by Phil Moorby and Prabhu Goel in 1983/1984 at Gateway Design Automation
- GDA was purchased by Cadence Design Systems in 1990.
- Originally, Verilog was intended to describe and allow simulation.
- Later support for synthesis added.

History (cont...)

- Cadence transferred Verilog into the public domain under the Open Verilog International (OVI).
- Now it is known as Accellera organization.
- Verilog was later submitted to IEEE and became IEEE Standard 1364-1995, commonly referred to as Verilog-95.

History (cont...)

- **Verilog 2001** Extensions to Verilog-95 were submitted back to IEEE to cover few limitation of Verilog -95.
- This extensions become IEEE Standard 1364-2001 known as **Verilog-2001**.



History (cont...)

- Verilog-2001 is the dominant flavor of Verilog supported by the majority of commercial EDA software packages.
- Today all EDA developer companies are using Verilog - 2001

History (cont...)

- **Verilog 2005** Don't be confused with System Verilog, *Verilog 2005* (IEEE Standard 1364-2005) consists of minor corrections.
- A separate part of the Verilog standard, Verilog-AMS, attempts to integrate analog and mixed signal modeling with traditional Verilog.

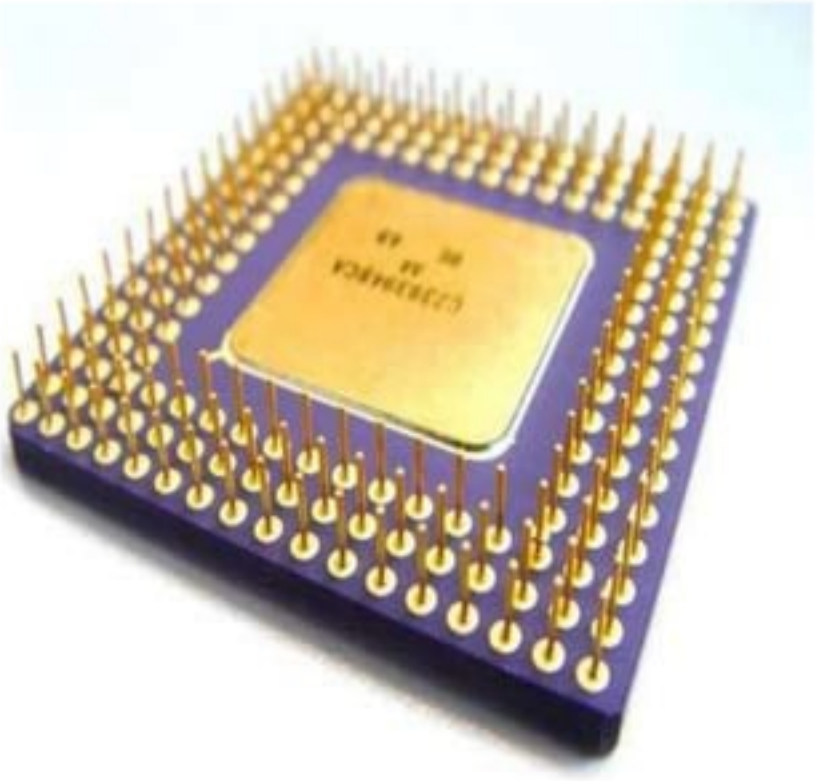
- **System Verilog** is a superset of Verilog-2005, with new features and capabilities to aid design-verification and design-modeling.
- As of 2009, the System Verilog and Verilog language standards were merged into System Verilog 2009 (IEEE Standard 1800-2009).

History (cont...)

- The advent of hardware verification languages such as Open Vera, System C encouraged the development of Superlog by Co-Design Automation Inc.
- Co-Design Automation Inc was later purchased by Synopsys.
- The foundations of Superlog and Vera were donated to Accellera, which later became the IEEE standard 1800-2005: System Verilog.

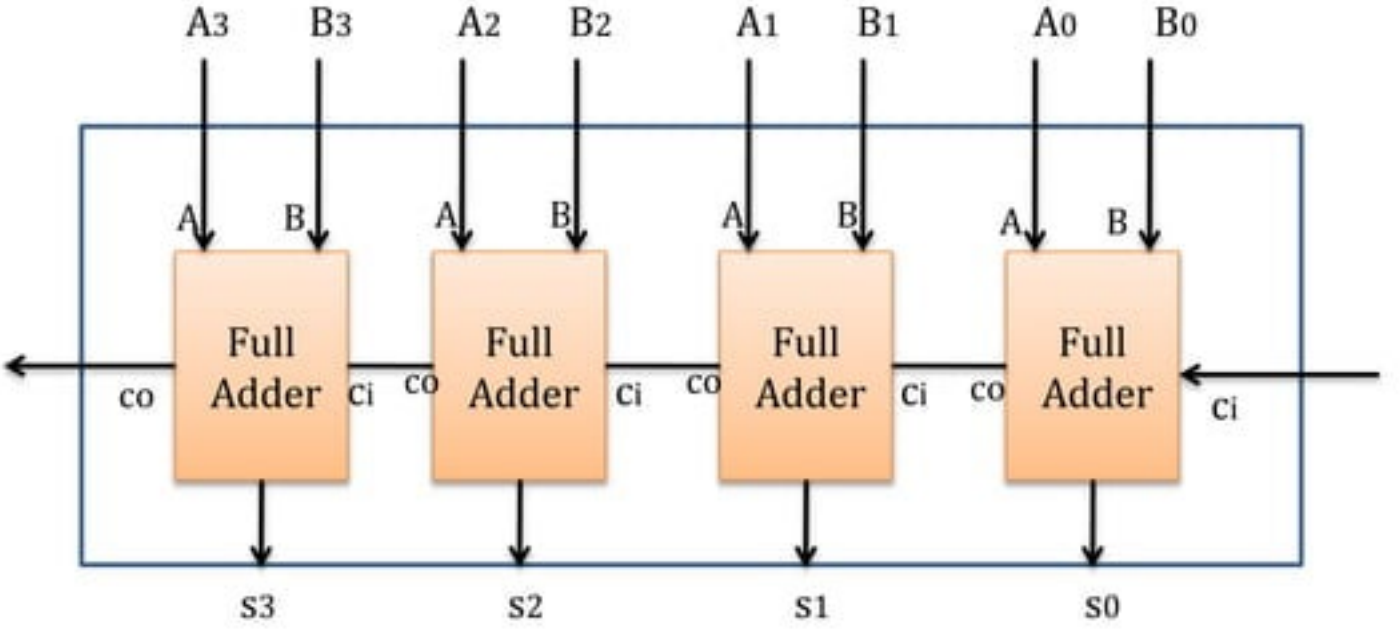
Continues to verilog

1. Introduction to VLSI
2. Introduction to Verilog HDL
3. History
4. Design Methodology
5. Verilog Keywords
6. Verilog data types
7. Abstraction
8. Gate level Abstraction
9. Data flow level Abstraction
10. Behavioral level Abstraction
11. Switch level Abstraction
12. Advance verilog Keywords



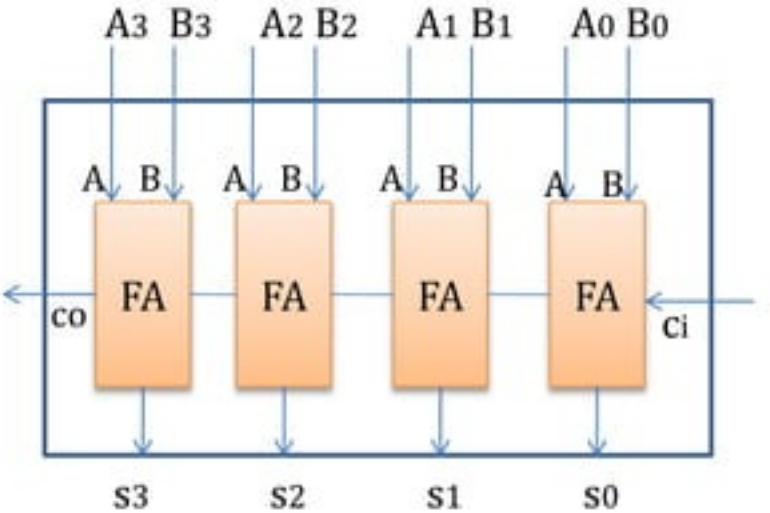
Design Methodology

- Based on Design Hierarchy
- Based on Abstraction

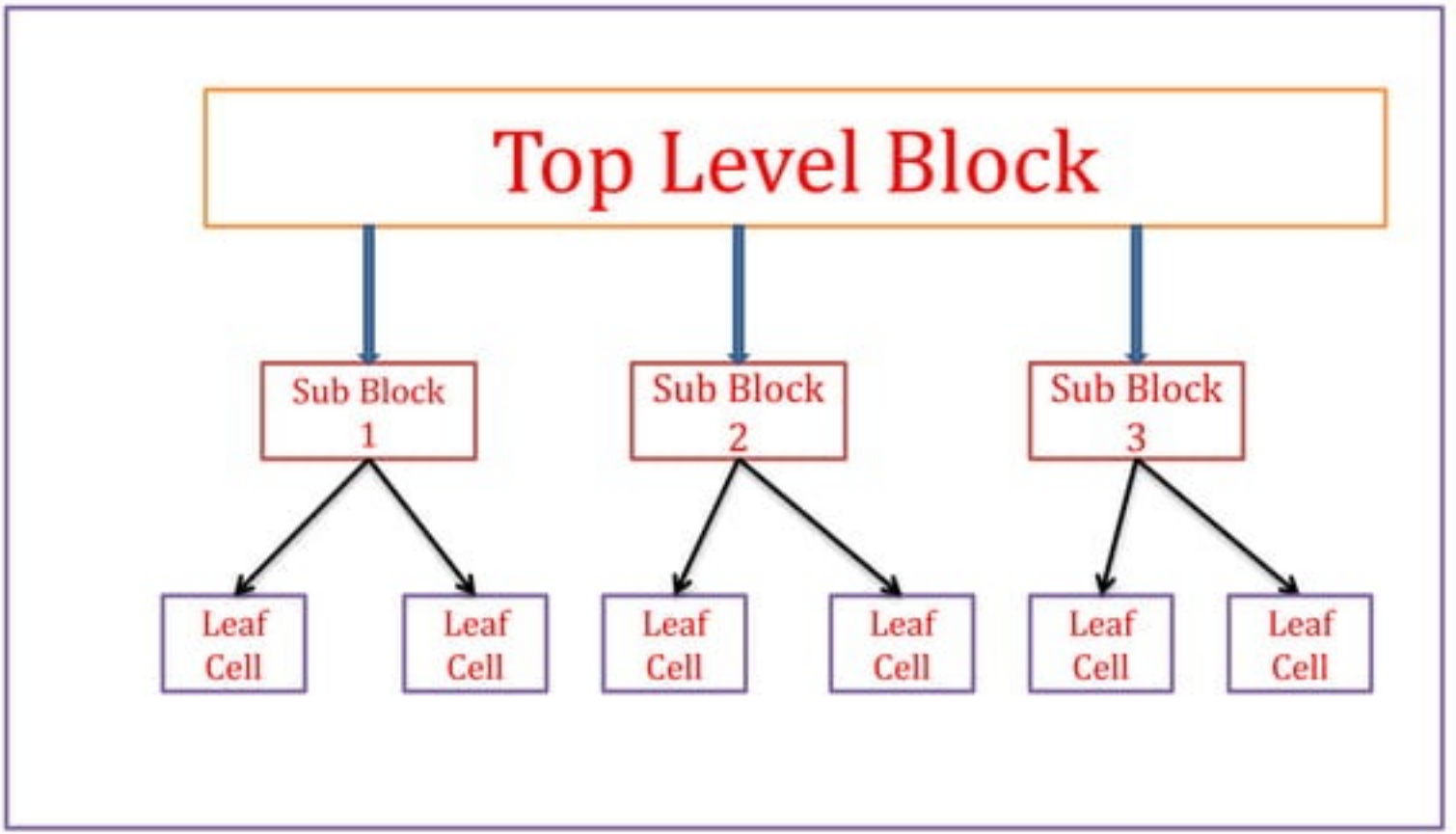


Based on Design Hierarchy

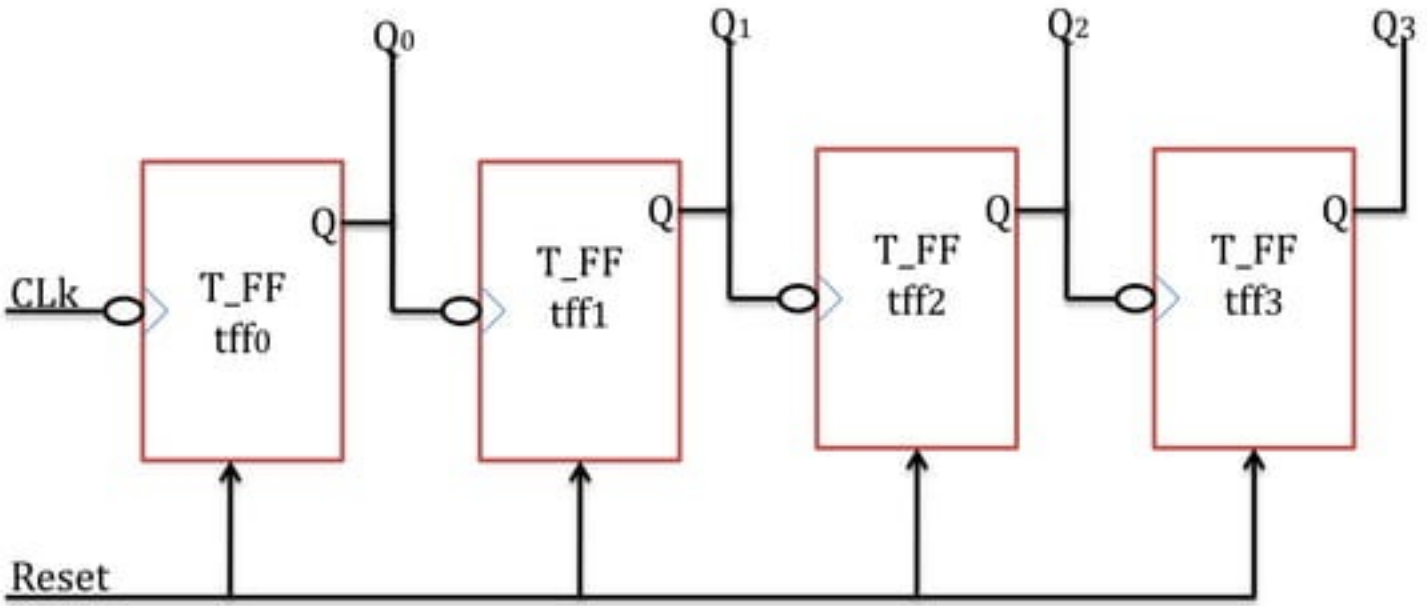
- Top Down Methodology.
- Bottom Up Design Methodology.



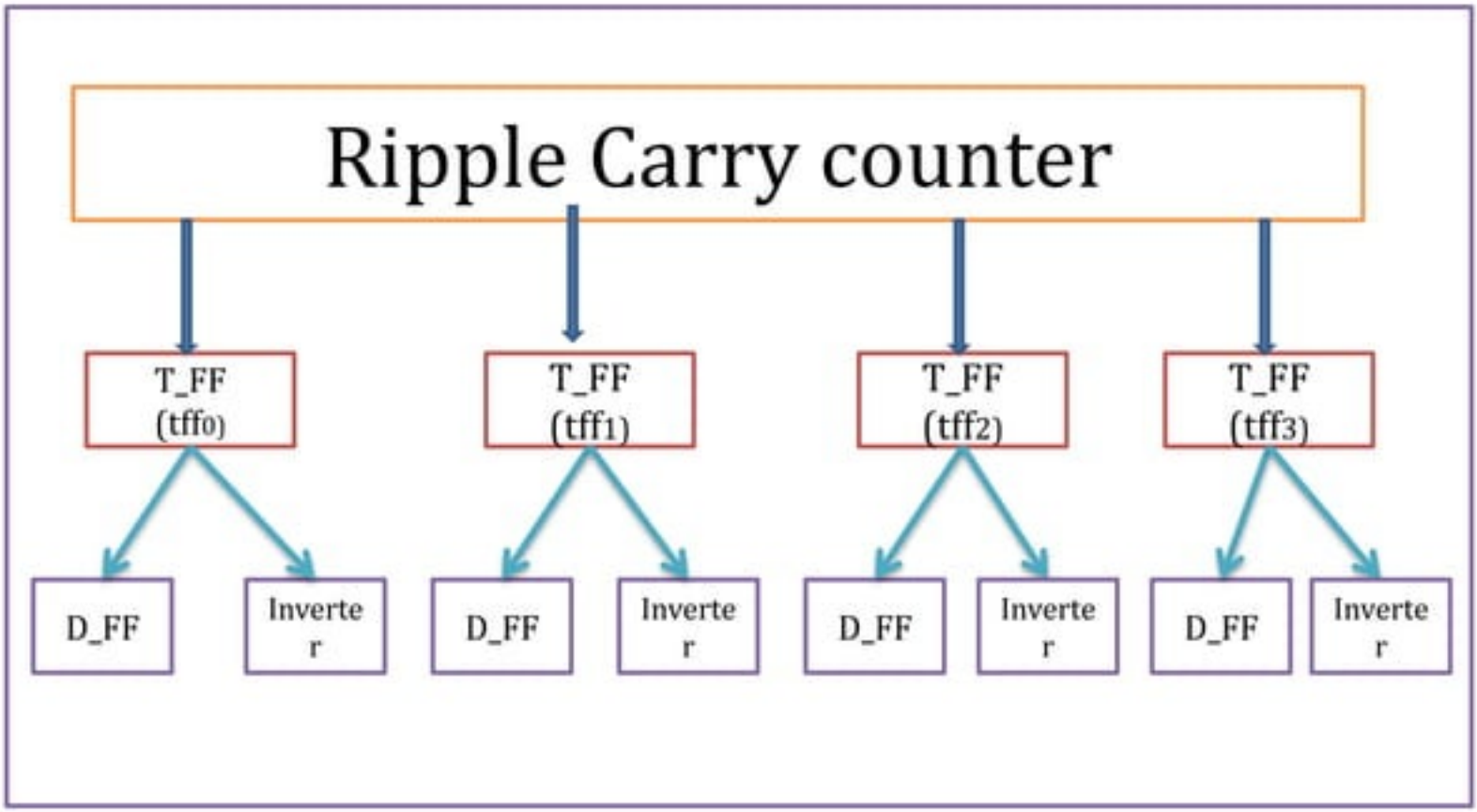
Top Down Design Methodology



Top Level for Four Bit Ripple carry Counter



Top Down Method



Verilog Code For Ripple Carry Counter

```
module ripple_carry_counter(clk,reset,q);          // starting of the module

  input clk;                                       // Input signal
  input reset;                                     // Input signal
  output [3:0] q;                                  // counter output
                                                   // to the external world

  T_FF tff0 (q[0],clk,reset);                      // instantiation of T_FF
  T_FF tff1 (q[1],q0,reset);                       // instantiation of T_FF
  T_FF tff2 (q[2],q1,reset);                       // instantiation of T_FF
  T_FF tff3 (q[3],q2,reset);                       // instantiation of T_FF

endmodule                                          // end of module
```

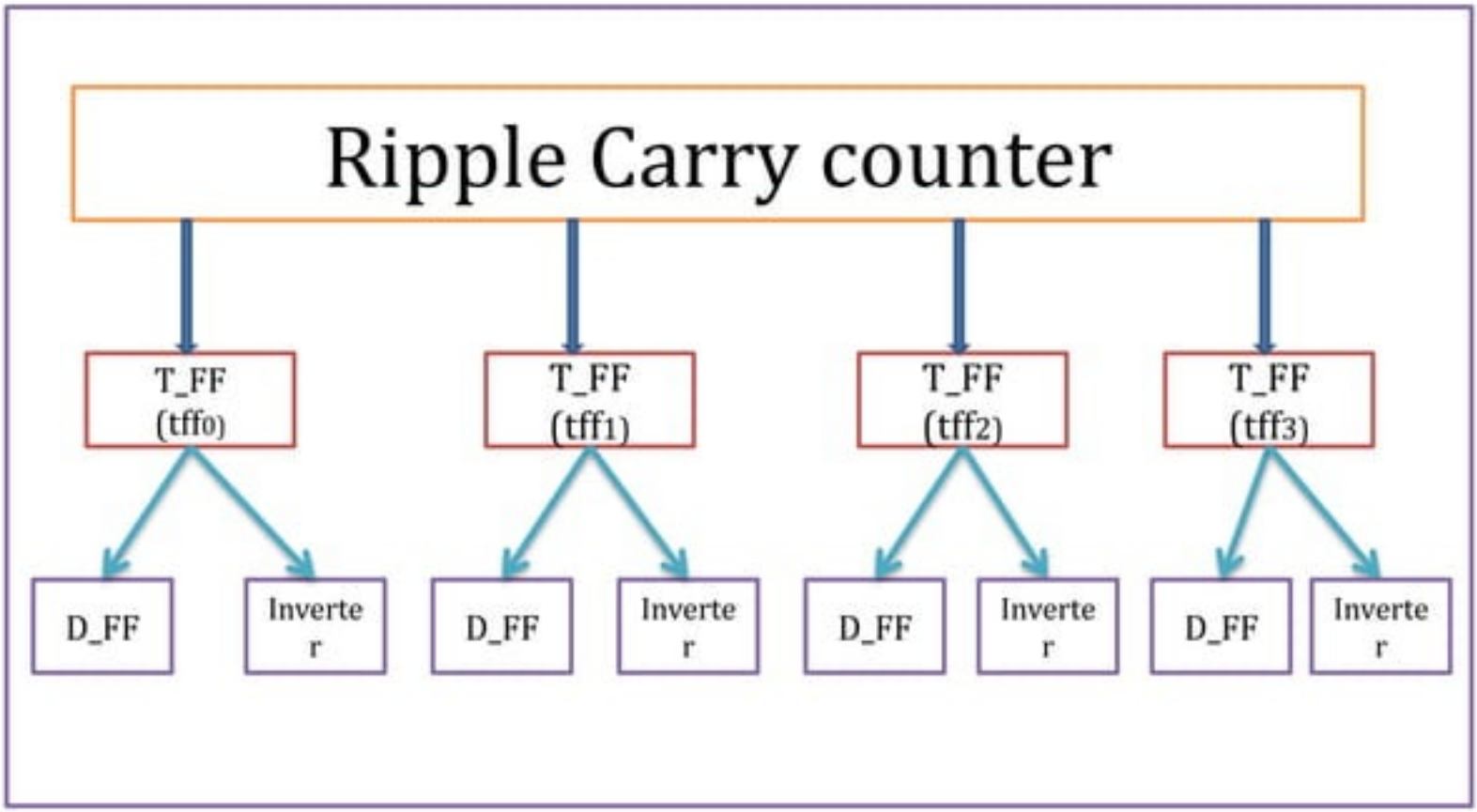
Lower level module

```
module T_FF(clk,reset,q);                          // starting of the module

   input clk;                                       // T_FF Input signal
   input reset;                                     // T_FF Input signal
   output  q;                                       // T_FF output
                                                    // to the external world

   wire d;                                          // internal signal

   D_FF dff0 (q,clk,reset,d);                       // instantiation of D_FF
   not n1 (d , q) ;                                 // instantiation of Not Gate

endmodule                                          // end of module
```

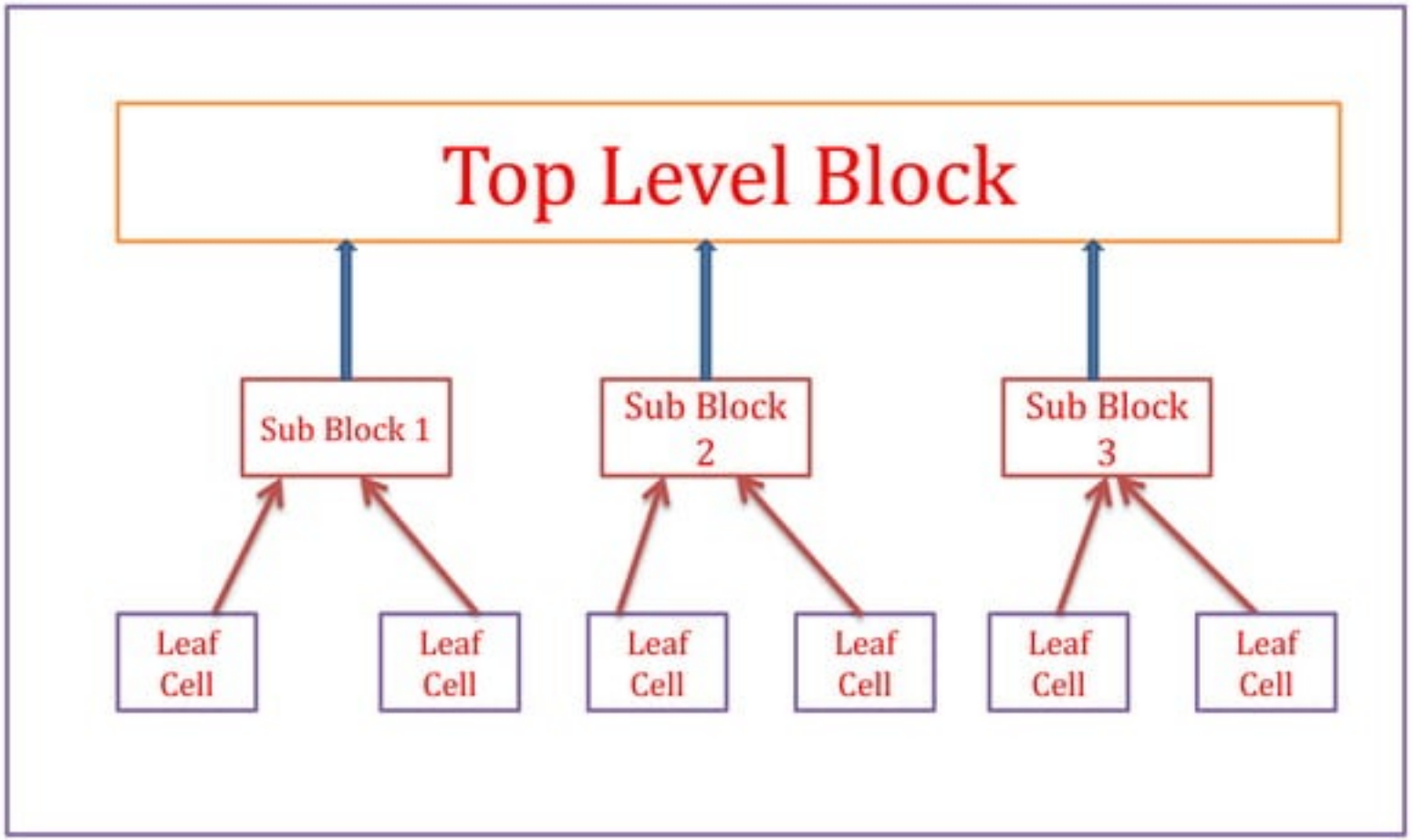
Lowest level for Counter

```
Module D_FF(clk,reset,q,d);                    // starting of the module
  input clk, d;                                 // Input signal
  input reset;                                 // Input signal
  output  q;                                  // D_FF output
                                             // to the external world

 always @(posedge clk , negedge reset)
     if (!reset)                             // main body of the D_FF
       q <=  1'b0;                           // Behavioral  level  code for D_FF
     else
       q <=  d;

endmodule                                   // end of module
```

Top Down Method

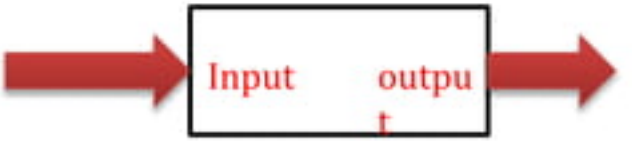


Bottom Up design Methodology

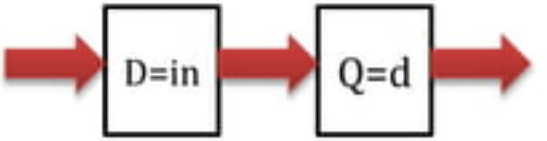


Based on Abstraction

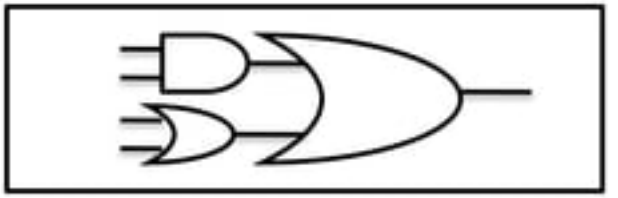
- Behavioral Level



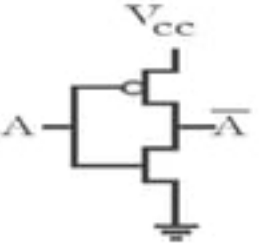
- Data flow Level



- Gate level



- Switch level



Based on Abstraction

Behavioral Level → `count<= count + 1;`

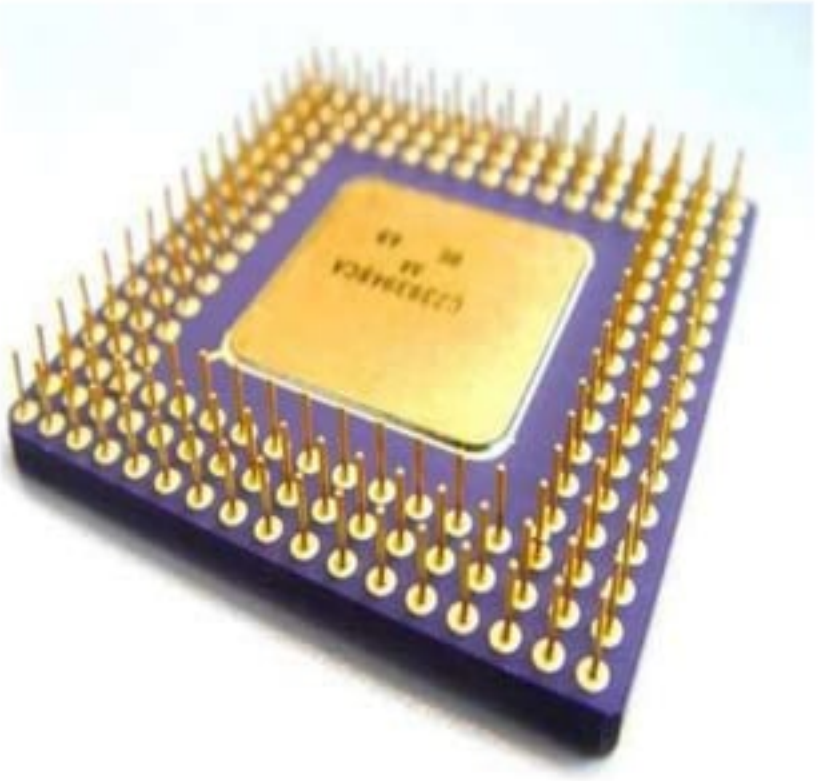
Data flow Level → `assign a = b&c`

Gate level →
```
and a1(y1,a,b);
or o1(y2,c,d);
or o2(y,y1,y2);
```

Switch level →
```
pmos p1 (out,pwr,in);
nmos n1 (out,gnd,in);
```

Continues to verilog

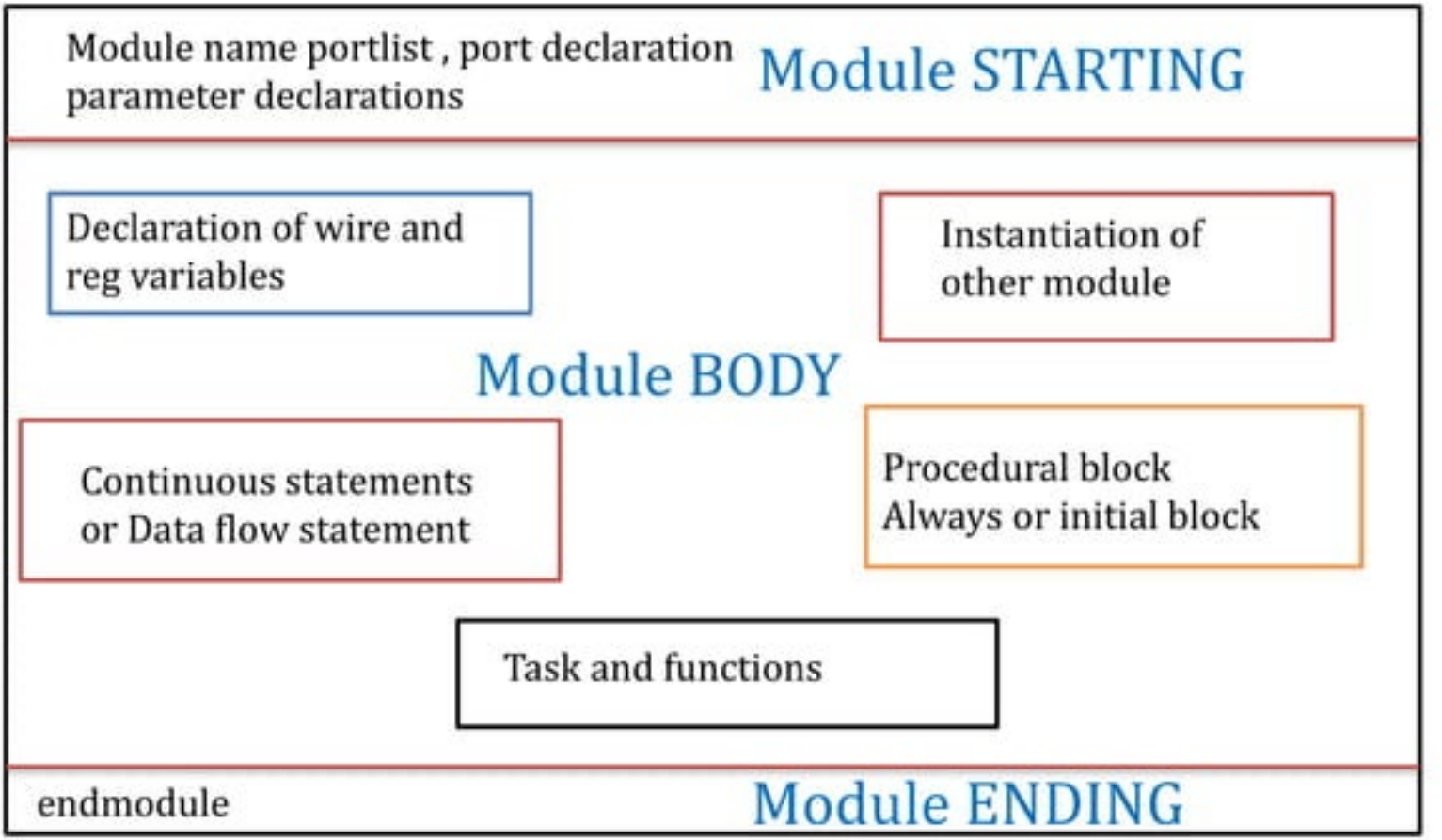
1. Introduction to VLSI
2. Introduction to Verilog HDL
3. History
4. Design Methodology
5. Verilog Keywords
6. Verilog data types
7. Abstraction
8. Gate level Abstraction
9. Data flow level Abstraction
10. Behavioral level Abstraction
11. Switch level Abstraction
12. Advance verilog Keywords



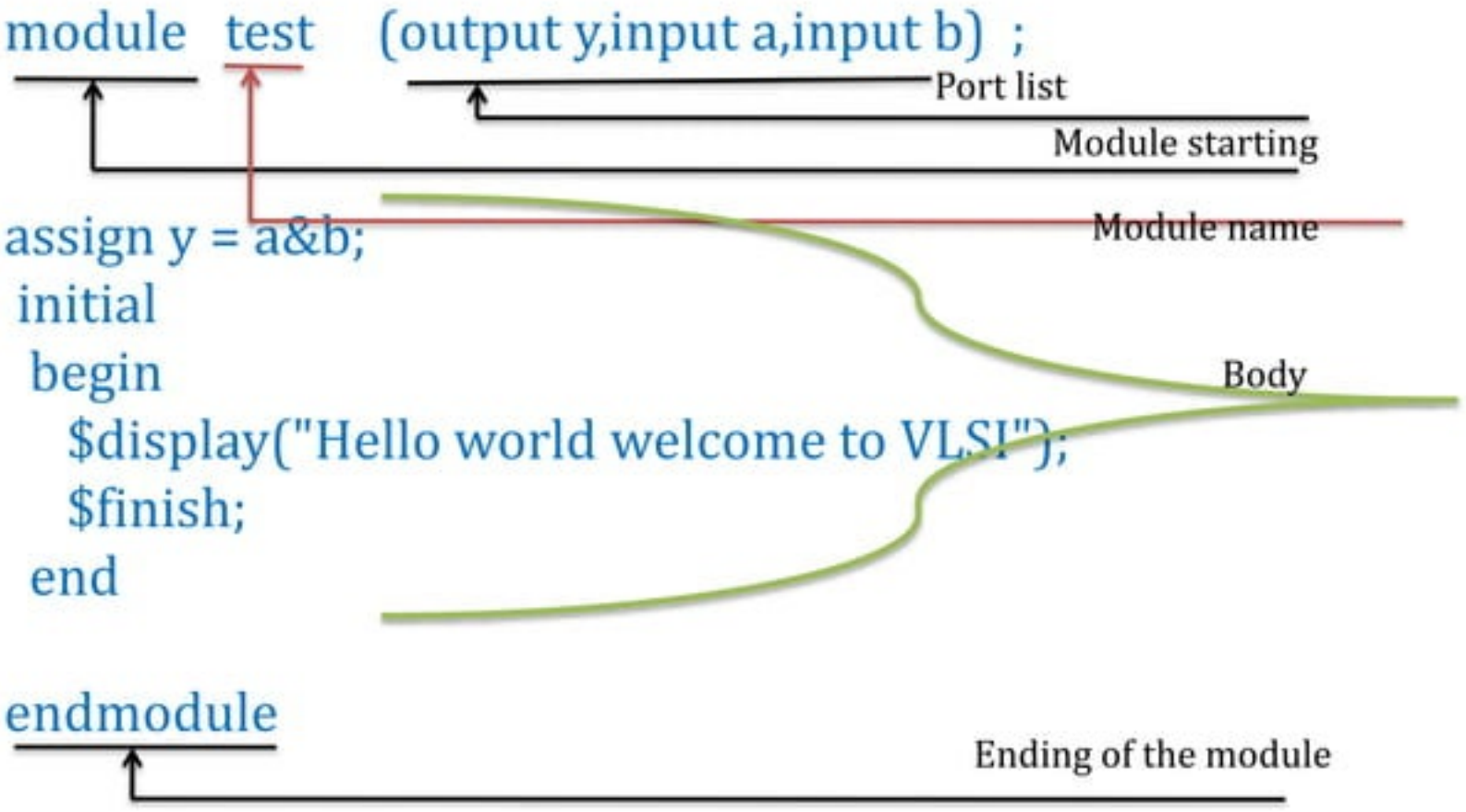
Module

- This is the basic building block of Verilog.
- Followed by the name of the Design.
- The Body of the Design is Enclosed inside the module.
- Module is ended by the keywords endmodule

Module Structure

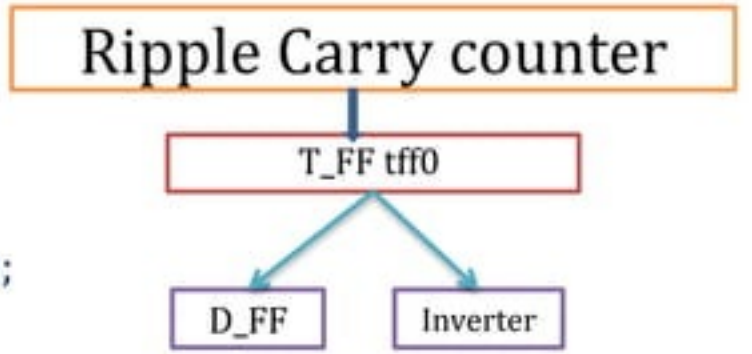


Example of module



Instantiation

- The Word Instantiation means to transfer.
- Transfer > transfer the property of Design
- Instantiation is a process to create an object for the Design which will replicate its all property where it is instantiated.




```
module ripple_carry_counter(clk,reset,q);
  input clk,reset;
   output [3:0] q;
  T_FF  tff0  (q[0],clk,reset);        // instance module name must be same as
                                       // declared in s main module
endmodule
```

instantiation

instance name

Port list

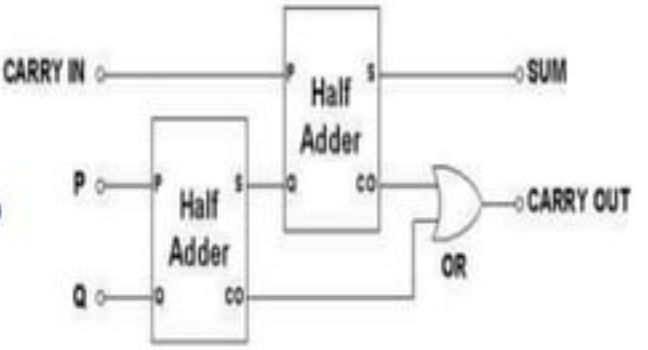
Instantiation methods

- Named Based
- Ordered Based

```
// this is half adder verilog code
module half_adder (S, C , A, B )   // module name & port declaration

   output S;                       // output port declarations
   output C;                       // output port declarations

    input A;                       // input port declarations
    input B;                       // input port declarations

   xor  x1 ( S, A, B);             // instantiation of xor gate
   and  a1 (C, A, B);              // instantiation of and gate

endmodule
```

Instantiation

- Named Based

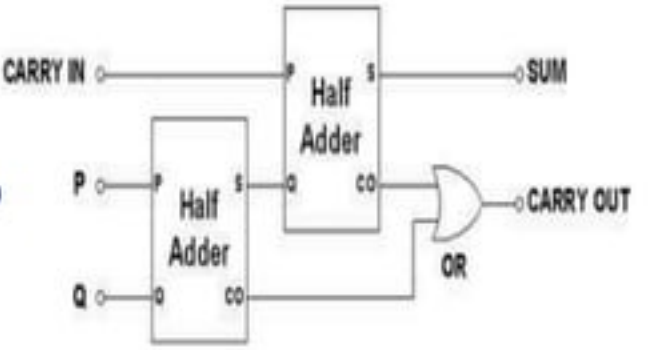



```
// This is Full adder verilog code
module  Full_adder (SUM, Cout, P,Q,Cin)
   output SUM , Cout;                    //output p
   input P,Q, Cin                        // input ports

   wire  S1, Co1 ,Co2;

    half_adder  H1 (.S1(S), .Co1(C) , .P(A),.Q(B))        // ports are
   connected by

    half_adder  H2(.SUM(S),.Co2(C),.S1(A) ,.Cin(B)) ;   // name to each
   signal

     or o1 ( Cout,Co1,Co2);

Endmodule   //module half_adder ( S, C , A, B )
```

Instantiation

- Ordered Based

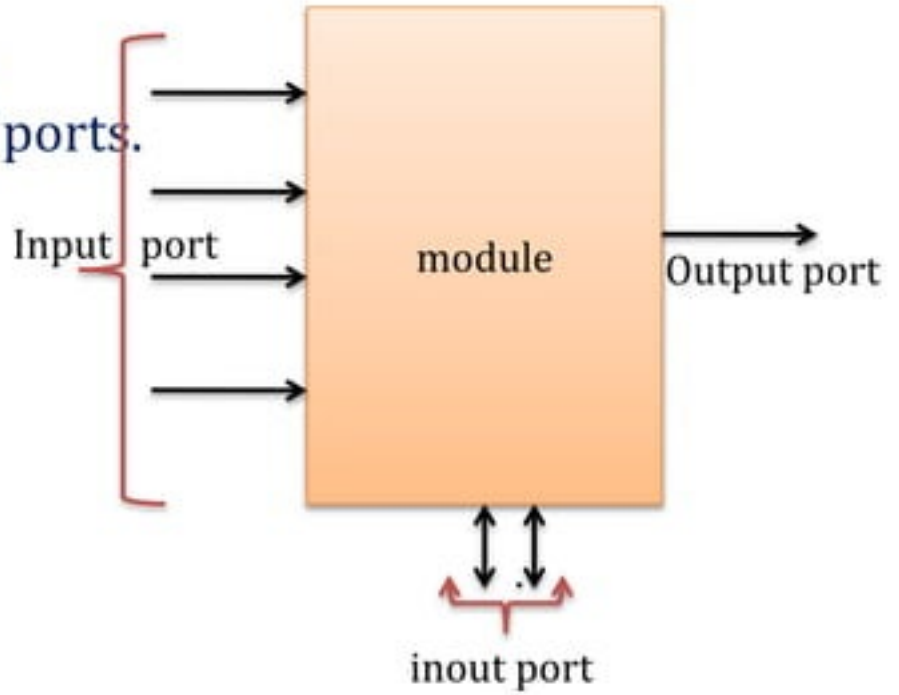
```
// This is Full adder verilog code
module  Full_adder (SUM, Cout, P,Q,Cin)
   output SUM , Cout;                    //output p
    input P,Q, Cin                       // input ports

    wire  S1, Co1 ,Co2;
     half_adder  H1 (S1, Co1 , P,Q) ;          // ports are connected by the
    same
     half_adder  H2(SUM, Co2,S1 ,Cin) ;    // Order as these are defined in
    main module

      or o1 ( Cout,Co1,Co2);
endmodule
```



// module half_adder (S, C , A, B)

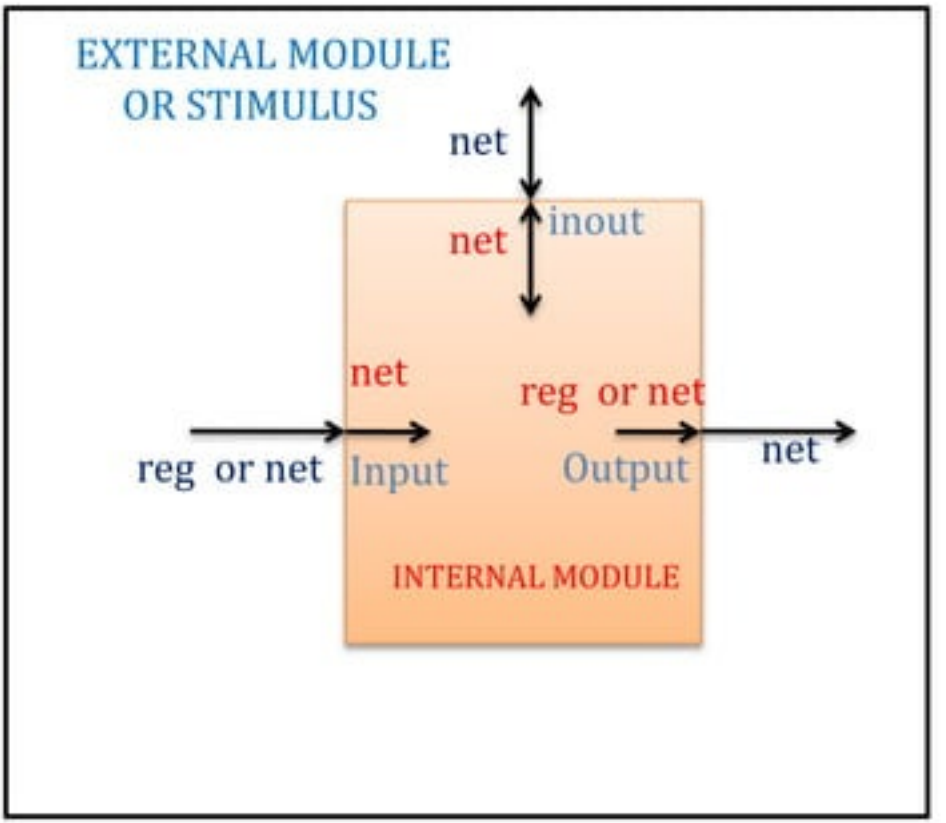
Input Output Ports

- I/O ports are the signal to interface with the external World.
- In verilog any module support three types of ports.
  - Input
  - Output
  - Inout



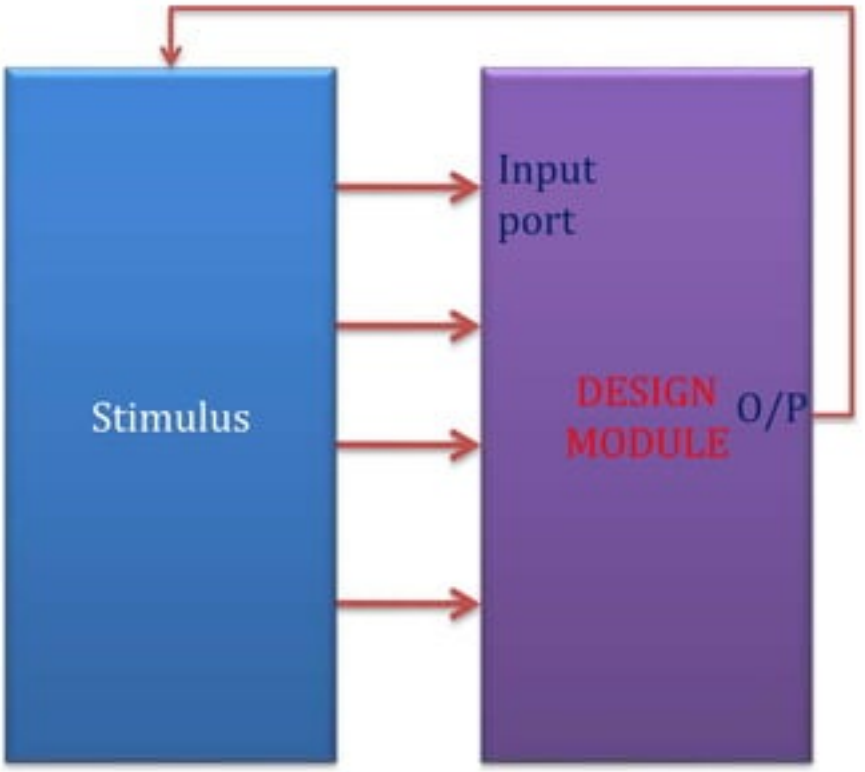
Port Connection Rules

- Input ports should be always wire & can be connected to net as well reg to the External World
- Inout ports are always net type in both the module Internal as well as External .
- Output ports may be reg as well wire or net internally but should be connected to net only
- Width Matching: suppose you defined any port with multi bits then at the time of interconnection both the data width must be same.
- If it is single bit then no need to declare bit size.



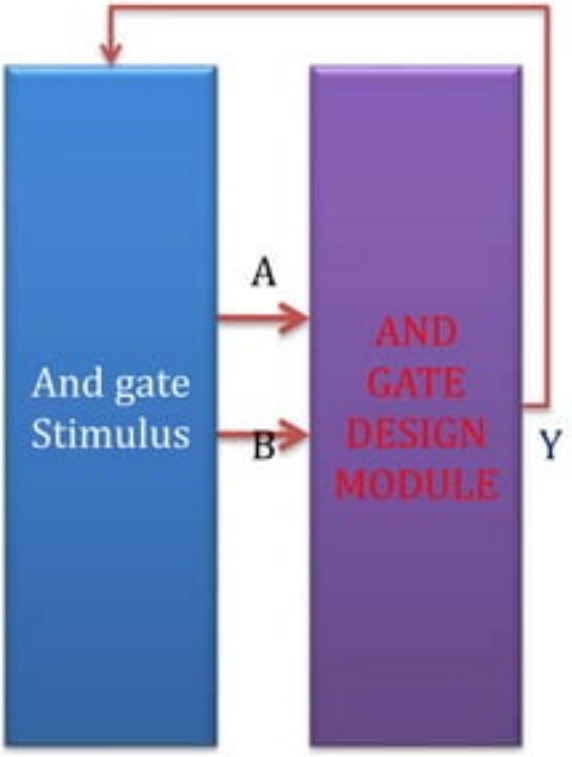
Stimulus

1. To check the functionality of the main design we require certain value of the input signal.
2. To model these input value we will use the Stimulus block.
3. This is same as test bench.
4. In this module we provide the input to the design module and check its response back to the stimulus.
5. The stimulus coding is written in verilog code.



Stimulus

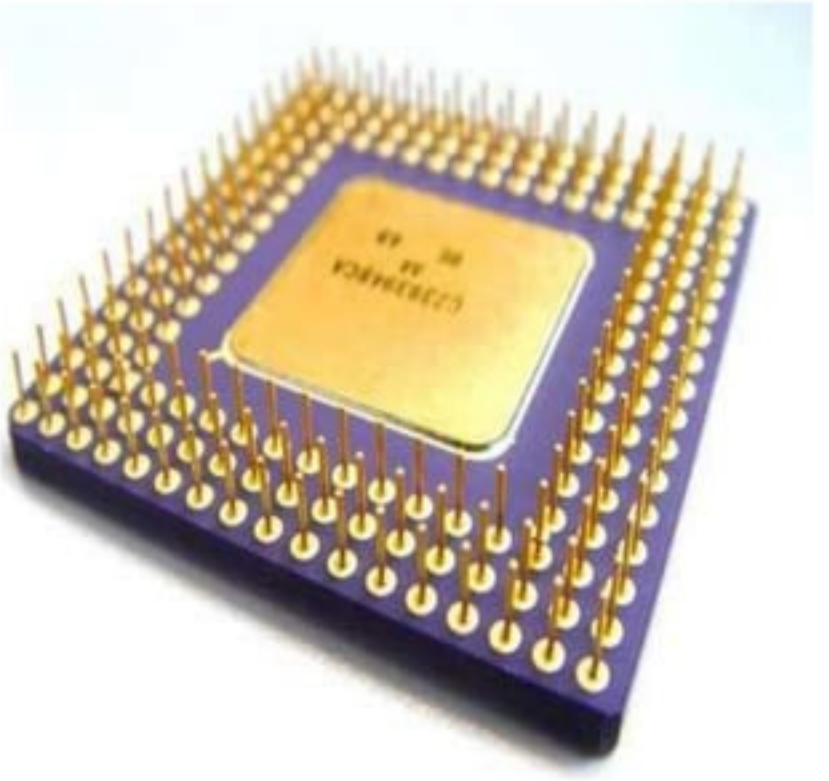
```
module stimulus_and_gate;
 reg A,B;;
 wire Y;
and_gate  and1 (Y,A,B); //
      //instantiation of
      design
 initial // stimulus
      generator
 begin
   $monitor  (" %b", Y);
   A=1'b0; B=1'b0;
   #10 A =1'b0; B=1'b1;
   #10 A=1'b1; B=1'b0;
   #10 A=1'b1; B = 1'b1;
  end
endmodule
```




```
module and_gate(Y,A,B);
 output Y;
 input A,B;
  and a1 ( Y , A , B);
endmodule
```

Continues to verilog

1. Introduction to VLSI
2. Introduction to Verilog HDL
3. History
4. Design Methodology
5. Verilog Keywords
6. Lexical Conventions
7. Verilog data types
8. Abstraction
9. Gate level Abstraction
10. Data flow level Abstraction
11. Behavioral level Abstraction
12. Switch level Abstraction
13. Advance verilog Keywords



Lexical Conventions

- To use lexical convention following option can be used.
  - \b Blank Space
  - \t Tab Space
  - \n New line
- Comments
  - // this is comment line //single line Coment
  - /* this is multi line comment // Multi line Comment /*
  - This is multi line comment */ // Multi line Comment */

- Escape character : escape character begin with the character to escape the property of any wildcard character.
  - ** // escape the property of *

Lexical Conventions

- Operator
  - Three types of operator is supported in verilog
  - Unary, binary & ternary.
  - A = ~ B ; // unary operator
  - A = B & C; // binary operator
  - A = B ? C : D; //ternary operator

Lexical Conventions

- Number Specification: **Sized Number**
  - <size > ‘ < base > < number >;
  - Size is the size of the identifier.
  - Base is the base like binary or decimal.
  - Number is the value of the identifier.

  - 8 ’ b0110011; // this is 8 bit binary value
  - 128 ‘ d 567; ; // this is 128 bit decimal value
  - 32 ‘ h 783; // this is 32 bit hexadecimal value
  - 64 ‘ o 3432; // this is 64 bit octal value

Lexical Conventions

- Number Specification : Unsized Number :
- without declaring the size of the identifier size is Macine dependence or 32/64 bit by default.
  - 'h 892; // Machine dependent / 32 bit
    // hexadecimal data

- Similarly the default base format is decimal.
  - Number = 23; //decimal number 32 bit width

Lexical Conventions

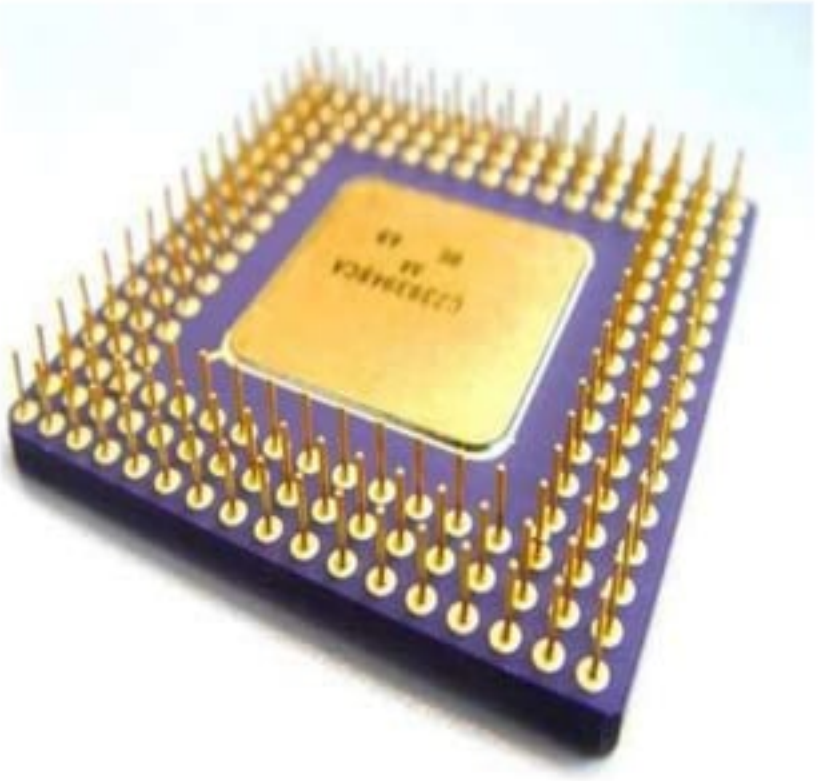
- Underscore character:
  - Underscore (_) can be place in between the digit of the number.
  - Dat a = 16 ‘b 1001_1010_1110_0001;
  - Note: never put the starting digit as underscore.
- Strings :
  - Verilog support string type data assignment and treated as ASCII character.
  - But there is no data type for string.
  - String must be enclosed inside double inverted commas “ “.
  - VAR = “ Hello world this is my first String”; // this is string

IDENTIFIER

- An identifier is any sequence of letters, digits, dollar signs ($), and underscore (_) symbol, except that:
  - the first must be a letter or the underscore.
  - the first character may not be a digit or $.
  - Upper and lower case letters are considered to be different.
  - Identifiers may be up to 1024 characters long.
  - Some Verilog-based tools do not recognize identifier characters beyond the 1024th as a significant part of the identifier.
  - Escaped identifiers start with the backslash character (\) and may include any printable ASCII character.
  - An escaped identifier ends with white space.
  - The leading backslash character is not considered to be part of the identifier.

Continues to verilog

1. Introduction to VLSI
2. Introduction to Verilog HDL
3. History
4. Design Methodology
5. Verilog Keywords
6. Lexical Conventions
7. Verilog data types
8. Abstraction
9. Gate level Abstraction
10. Data flow level Abstraction
11. Behavioral level Abstraction
12. Switch level Abstraction
13. Advance verilog Keywords



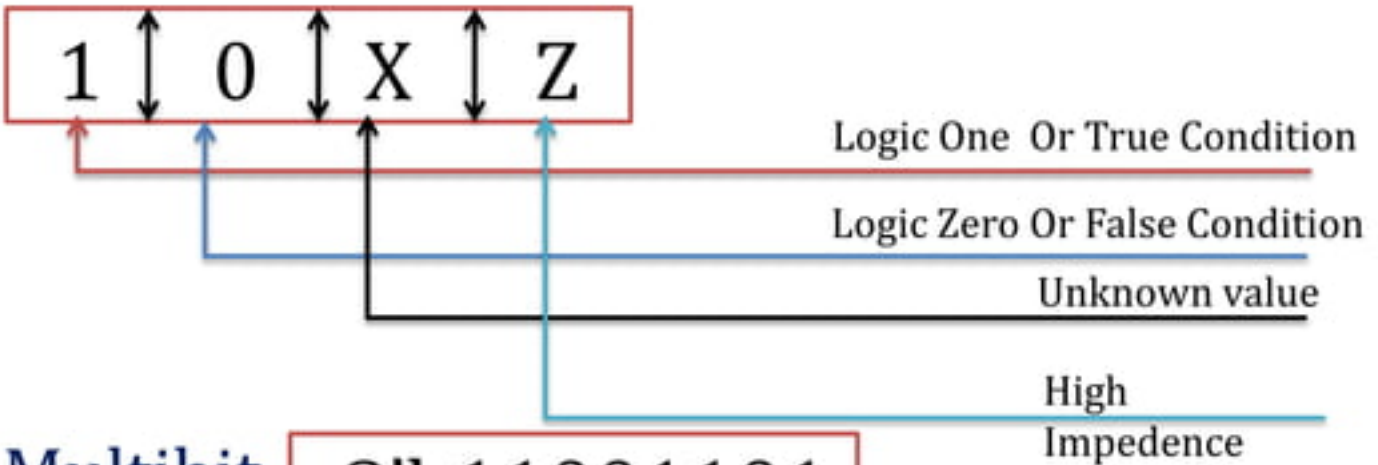
Verilog data types

- What is data?
- What is strength?
- What is type?

- In verilog there are some codes that perform some data calculation or manipulation.
- To store this data we require some variable .
- These data storage variable are known as Identifier.
- On the basis of the data storage, these variable have some data types.

What is data?

- Single bit four valued data.

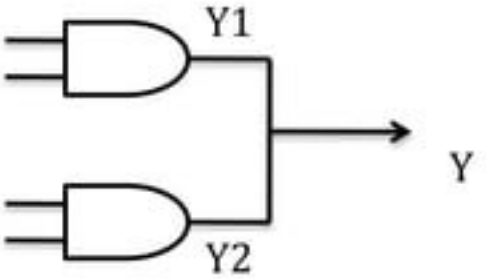


- Multibit 8'b11001101

Integer 231

Real 231.43

What is strength?

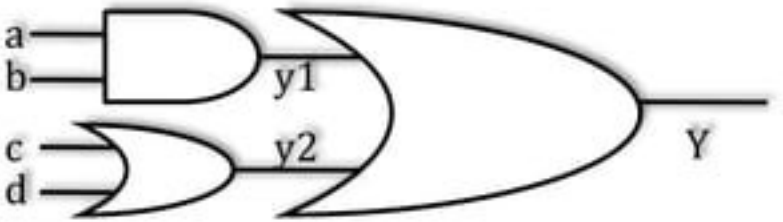


- The driving strength of a continuous assignment can be specified by the user.
- If two signal are driving the same net then the driving strength can be applied to get the require output.

Strength level	Degree	Strength Type
supply 1		driving
strong 1		driving
pull 1		driving
large 1		driving
weak 1		driving
medium 1		driving
small 1	Weakest 1	driving
highz 1	Weakest 0	high impedance
highz1		high impedance
small0		strong
medium0		strong
weak0		strong
large0		strong
pull0		strong
Strong0		strong
Supply0		strong

What is type?

- **Nets :**
- A net is declared as wire.
- Wire means a wire connection between two gates or hardware.
- The output of one logic goes to input of another logic.
- In verilog the default declaration of any variable is of net type.
- A net does not store a value (except for the trireg net.
- it must be driven by a driver, such as a gate or a continuous assignment




```
module
logic_wire(Y,a,b,c,d)
    output Y ;
    input a , b , c , d;

    wire y1 , y2 ;

     and a1 ( y1 , a , b );
     or o1 (y2 , c , d);
     assign Y = y1 | y2;

endmodule
```

Verilog data types : Nets

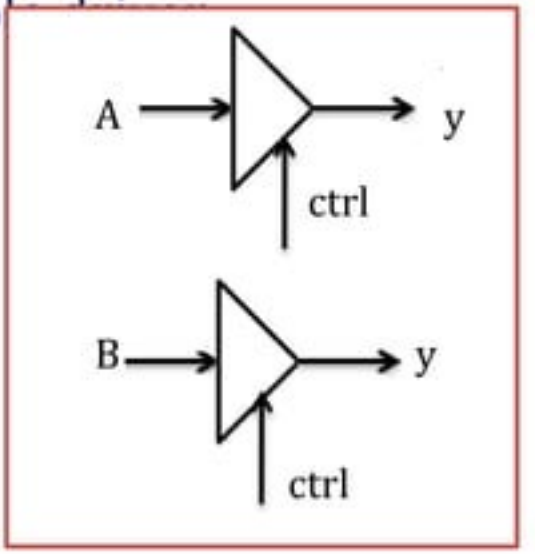
- it must be driven by a driver, such as a gate or a continuous assignment.
- If no driver is connected to a net, its value will be high-impedance (z).

- Multi bits wire
- wire [31:0] data_bus32;

Advanced Net Types

- **tri:** it is similar to wire as syntax wise as well as functionally.
- The only difference between wire and tri is wire denote single driver while tri means multiple drivers

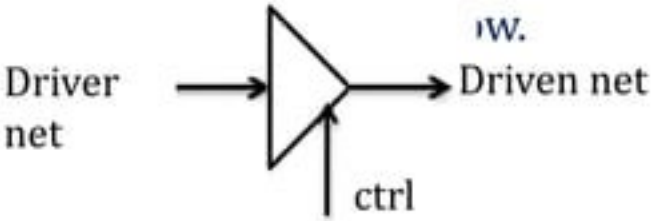
```
module buffif(y,A,B,ctrl);
 output y;
 input A,B,ctrl;

 tri y;
 wire A,B,ctrl;

  buffif0  bf0 ( y , A , ctrl);
  buffif1 bf1 ( y , B  , ctrl);
endmodule
```



Advanced Net Types

- **trireg :** trireg is same as wire except that when the net having capacitance effect.
- Capacitance effect means it will store the previous value.
- Therefore trireg works on two state.
- Driven state: when the driver net having a value 1 ,0 ,X then the driven net will follow the driver net.
- Capacitance state: when the driver net is unconnected or having a value of Z or high impedance then the driven net will hold the last value.

- Ex : buffif1 b1 (y , A, ctrl); // net y get value whenever the value of A // when ctrl is high. Store the last value when



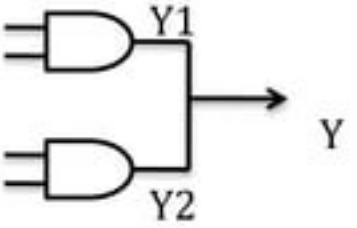
Advanced Net Types

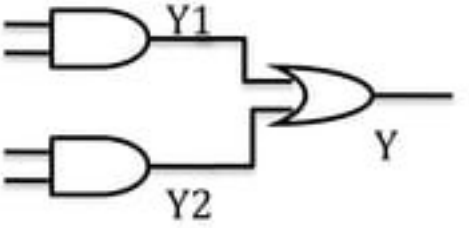
- **tri0 & tri1:** tri0 & tri1 are resistive Pulldown and pullup devices.
- When the value of the driving net is high then driven net will get a value of the input .
- When the value of the driving net is low the driven net get a value of pulldown or pullup.

- Ex : tri0 y;
- buff bf0 (y ,A,ctrl); // when ctrl is high y = A;
- // when ctrl is low y = 0; instead of high impedence.
- **supply0 & supply1:** these are used to model the power and Ground.
- supply1 VDD;
- supply 0 GND;

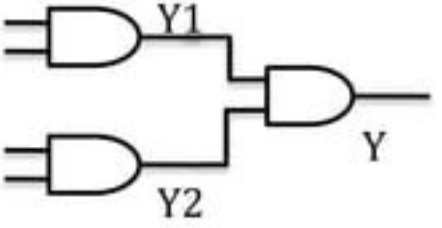
Advanced Net Types

- **wor, wand,trior & triand:**
- When one single net is driven by two net having same signal strength then it is difficult to determine the output on the driven net.
- In this case we can use the output oring or anding on both the driving nets.





- wor Y ; // or gate for Y1 & Y2
- wand Y; // and of Y1& Y2
- wor : or gate on the output driving nets.
- wand : and gates on the output driving nets.



Verilog data types

- Registers
- A register is an abstraction of a data storage element.
- A reg variable can be assign by another value only inside the procedural block.
- It is not always true that reg variable will infer a FF. Sometime it infer Combo logic.
- Registers can be assigned negative values.
- When a register is an operand in an expression, its value is treated as an unsigned (positive) value.
- reg is the keyword for the register data type.

Verilog Register data type

Declaration

reg <signed> <range> <list_of_register_variables> ;

```
reg data;                          // single bit variable.
reg signed [7:0] data_bus.         // multiple bits signed variable
reg [7:0] data_bus;                // multi bits unsigned variable
```

- **Integer:** This is a general reg type variable. Used to manipulate mathematical calculation.
- This is integer 32 bit long signed number.

```
integer data;        // 32 bit signed value
```

Default value

Verilog Register data type

- real: This is real register data types. Real data type variable has no range.
- Default value for real type variable is zero.
- real data = 3.34;
- real data= 2e10; // 3 * 10 ^6
- NOTE: Real value can not be passed from one module to another in verilog.

- time: the simulation is done with respect to time.
- To store the time we can use the time type variable.
- $time is used to see the system time. It is 64 bits integer type.
- $stime is use to display the simulation time & it is 32 bit integer type.
- $realtime is real value and it is 64 bits long.

Verilog data types

- **Vectors:** all the data types variable can be declared as multi bits.
- If the bit width is not specified then the bit width is one bit.
- reg data; // single bit variable
- reg [7 : 0] data_bus; // multi bits data
- wire data; // data is single
- wire [7 : 0] data_bus; // data_bits multi bits

Verilog Vector data type

- Vector bit select: reg [31 : 0] data;

  bit_select = data [7];
- Vector part select: reg [31 : 0] data;

  part_select = data [7:0];

  reg [0 : 31] data_bus;

  part_select = data_bus[0:7];
- Variables Vector part select:

  variabe_name [< starting_bit> + : width]

  byte_select= data_bus [16+ : 8]; // starting from 16 to 23

  variabe_name [< starting_bit> -: width]

  byte_select= data_bus [16- : 8]; // starting from 9 to 16

Verilog data type

- Arrays:
- Array is a collection of similar data types.
- Array can be multi dimensions.
- Declaration of array is similar to C language.
- reg array_data [7 : 0] [15 : 0]; // two dimension array.
- reg [7:0] mult_bits_array [7 :0]; // Single dimension array with each // element having 8 bits of data.
- integer data_bus[31:0] // 32 variable data_bus

- NOTE: Array can not be passed from one module to another in verilog.

Verilog data type

- Memories:
- **Memories:** Memory can be model as defining them multi bits array.
- reg mem_data [0: N-1];

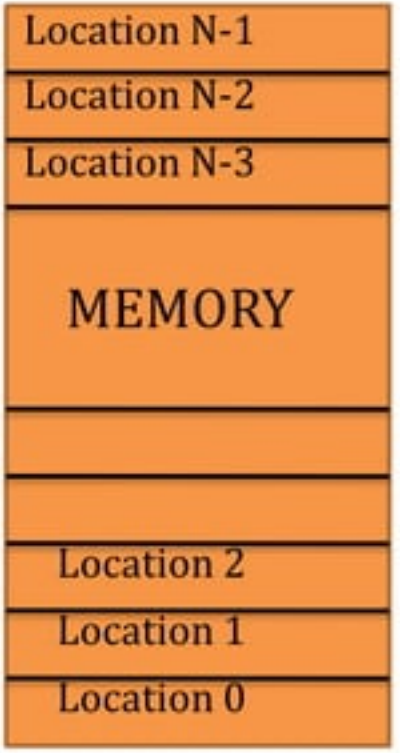
- Parameter

Parameter: To declare constant values in verilog we can used data type of parameter.

parameter Data_width=8; // this type of parameter global in nature and can be override at the simulation or from other module.

localparam: This is same as parameter expect that other module can not overwrite the constant value from outside of the module

localparam DW= 32;



Verilog system task

- Verilog provide standard system task to perform some routine operation.
- These system task start with a character $(keyword).
- These routine are used to display output values to the terminal or to display simulation time ect.
- ex: $display
- $monitor
- $strobe
- $write
- $time
- $finish
- $recordfile
- $dumpfile

Verilog system task

- Display Information:
- In verilog there are four types of representation
- These value can be displayed in binary, hexadecimal, decimal or octal.

Format	Display Action
%d or %D	display in decimal
% b or %B	display in binary
%s or %S	display string
%o or %O	display in octal
%h or %H	display in hexadecimal
%c or %C	display ASCII character
%m %M	display hierarchy name
%v or %V	display strength
%t or %T	display current time format
%e or %E	display real number in scientific
%f or %F	display real number in decimal
%g or %G	display real number in scientific

Verilog system task

- Displaying task : $display
- $display("hello world welcome to VLSI % b" , data);
- The above system task will display the string as well the value of the variable in binary format.
- Whenever this system task is used in the program it will display according to the variable and the string values & then it bring the **Cursor** to next line automatically.
- $write : it is same like $display except the cursor will remain in the same line at end of the system task.
- $write ("%d",a); // display the value of a in decimal and remain in the same line.

Verilog system task

- Displaying task : **$monitor**
- This system task is used as same as $display but it will keep monitoring the value whenever the value of the variable will change.
- **$monitor** (a,b,c); // it will display in decimal whenever the value of the any of the variable will change.
- **$monitoron** is used to enable the system task.
- **$monitoroff** is used to disable the system task.

- Displaying task : **$strobe**
- **$strobe** is same as display except that it will display the value at the end of the simulation time.
- $strobe("%h", a); // will display the value of a in hex at the end of simulation time.

Verilog system task

- Simulation control task **$stop:**
- It will stop the simulation at the given simulation time and provide control to the user to an interactive mode.
- #200 $stop // stop the simulation at the 200 time unit .

- Simulation control task **$finish:**
- It will terminate the simulation at the given simulation time.
- # 500 $finish; // terminate the simulation at the 500 time unit.

Verilog Compiler directive

- Compiler directive: compiler directive are defined in verilog as macro.
- These macro has certain property and the effect of these macro is global.
- These macro are defined by ` <keyword> or back quote < keywords>

Ex ‘ timescale 1ns/1ns

` define DATA_WIDTH 8

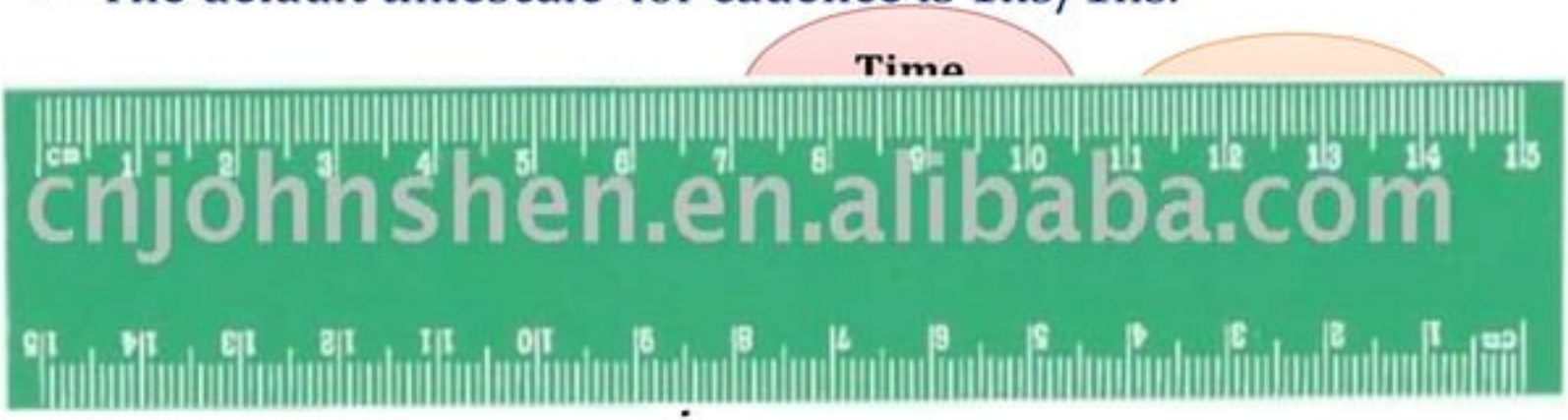
`include

`ifdefined

- These can be defined inside module as well outside module except `timescale which is always defined outside the main module

Timescale

- To define the simulation time unit we need a reference time unit.
- `timescale is used for specifying the reference time unit for the simulator.
- This time unit we can define at the top of the module or outside module.
- `timescale <reference_time_unit>/<time_precision>
- The default timescale depends on vender to vender.
- The default timescale for cadence is 1ns/1ns.



Timescale

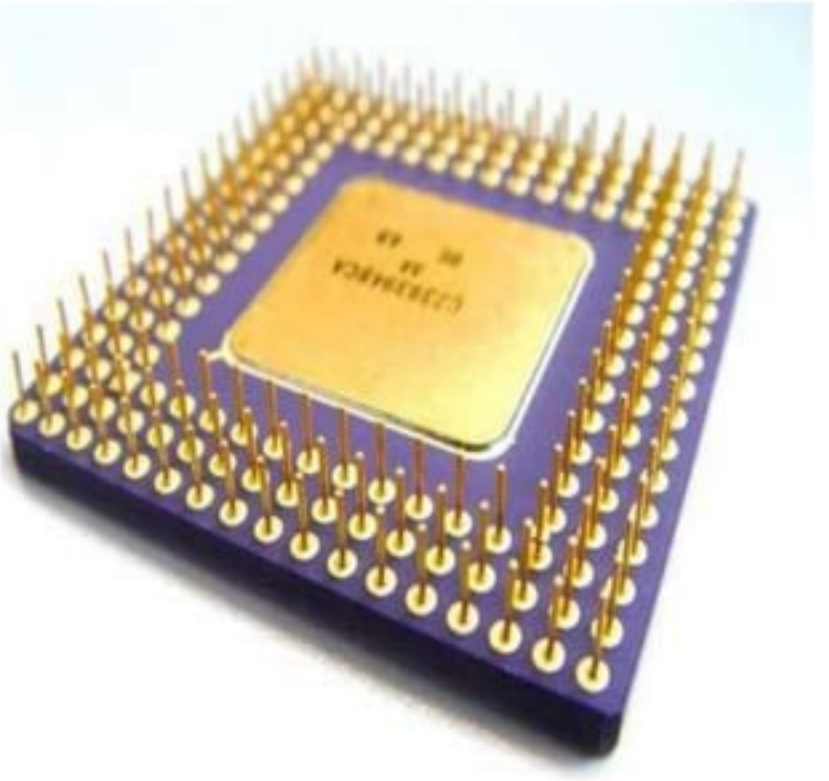
- `timescale time_unit / precision
- **Time Unit:** this is the basic simulation time unit.
- This time unit is responsible for all the signal to pass the data at the given time multiplied by the time unit.
- **Precision:** this is the minimum value up to which the precision can be measured.
- Precisions represents the minimum delay which needs to be considered during simulation.
- It decides that how many decimal point would be used with the time unit.
- **Range of Timescale**
- The range for time unit can be from seconds to
- ms(mili-second), us(micro-second), ns(nano-second), ps(pico-second) and fs(femto-second).

Timescale

- **`timescale 1ns/1ns**
- 1ns = 1ns
- #1; // = 1ns delay
- **`timescale 1ns/1ps**
- #1.003; // = will be considered as a valid delay
- #1.0009; // = will be taken as 1 ns only since it is out of the precision value.

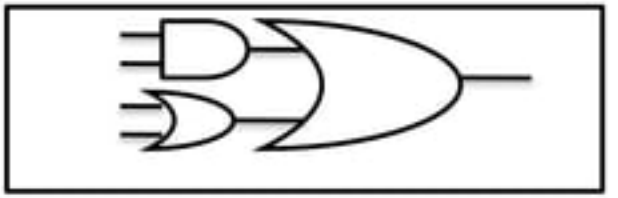
Continues to verilog

1. Introduction to VLSI
2. Introduction to Verilog HDL
3. History
4. Design Methodology
5. Verilog Keywords
6. Lexical Conventions
7. Verilog data types
8. Abstraction
9. Gate level Abstraction
10. Data flow level Abstraction
11. Behavioral level Abstraction
12. Switch level Abstraction
13. Advance verilog Keywords

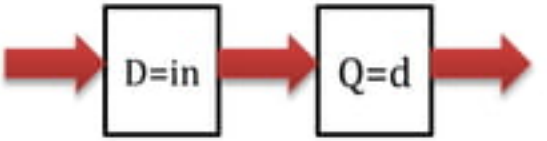


Abstraction

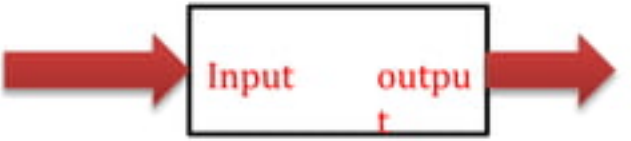
- Gate level



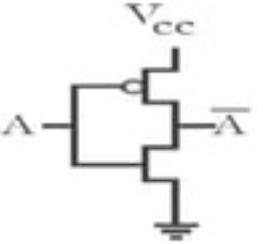
- Data flow Level



- Behavioral Level

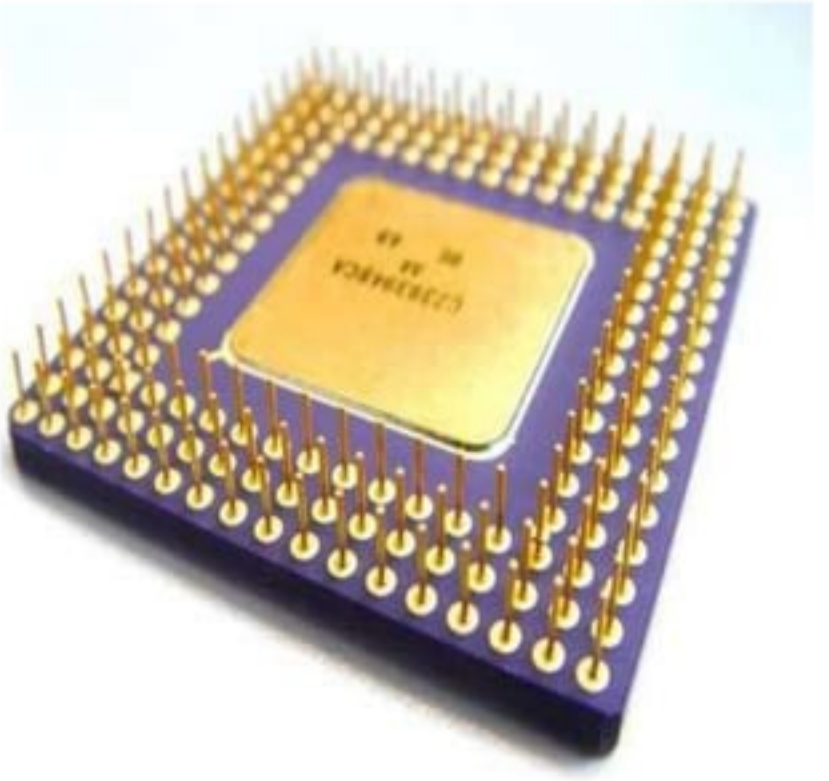


- Switch level



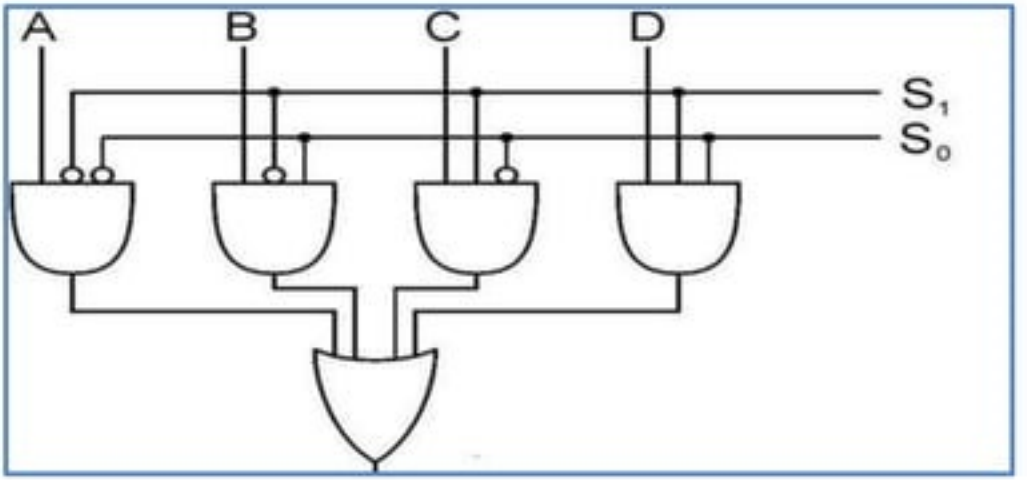
Continues to verilog

1. Introduction to VLSI
2. Introduction to Verilog HDL
3. History
4. Design Methodology
5. Verilog Keywords
6. Lexical Conventions
7. Verilog data types
8. Abstraction
9. Gate level Abstraction
10. Data flow level Abstraction
11. Behavioral level Abstraction
12. Switch level Abstraction
13. Advance verilog Keywords

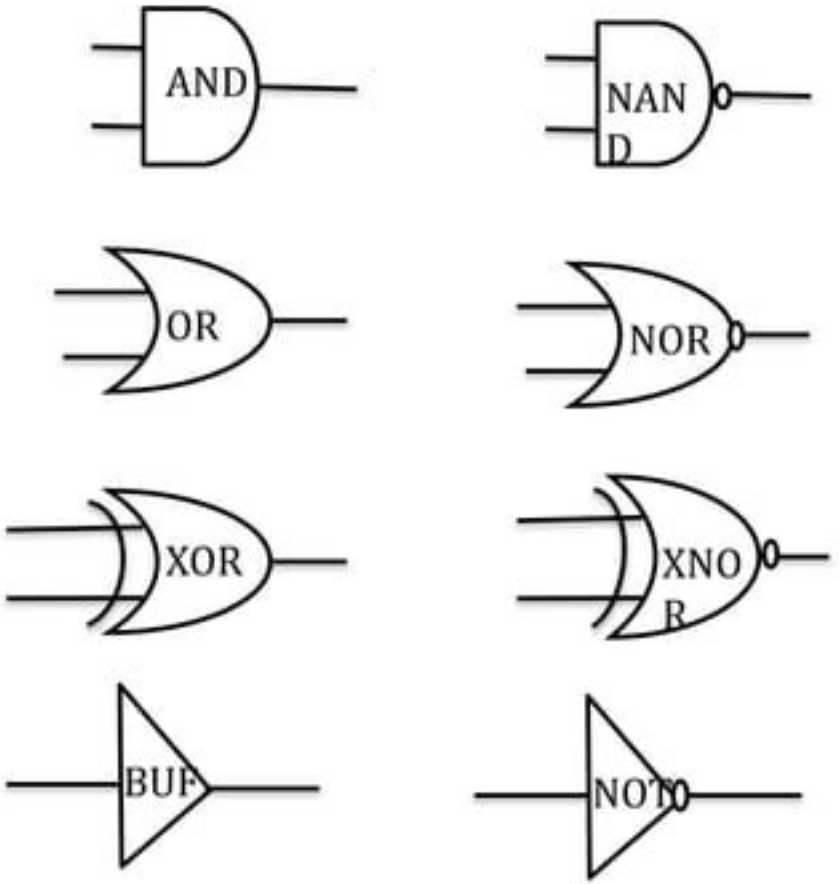


Gate level Abstraction

- This is also known as structural level Abstraction.
- In this level the Design is described in terms of gates.
- It is very easy to design any circuit in verilog if we have the structure.
- For large circuit its difficult to implement in gate level.



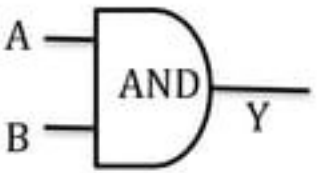
Basic gate primitive in verilog



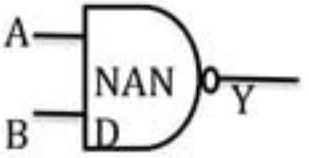
Instantiation of gates

```
input wire  a, b ;
output wire y;
and  a1 ( y , a, b);
nand n1 ( y , a, b);
or o1 ( y , a, b);
nor no1  ( y , a, b);
xor x1 ( y , a, b);
xnor xn1 ( y , a, b);
buf b1 ( out,in);
not n1 (out,in);
```

Basic gate primitive in verilog

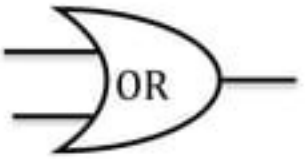


AND	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

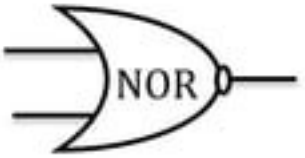


NAND	0	1	X	Z
0	1	1	1	1
1	1	0	X	X
X	1	X	X	X
Z	1	X	X	X

Basic gate primitive in verilog



OR	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X



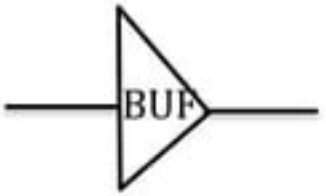
NOR	0	1	X	Z
0	1	0	X	X
1	0	0	0	0
X	X	0	X	X
Z	X	0	X	X

Basic gate primitive in verilog

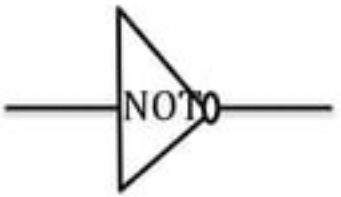
XOR	0	1	X	Z
0	0	1	X	X
1	1	0	X	X
X	X	X	X	X
Z	X	X	X	X

XNOR	0	1	X	Z
0	1	0	X	X
1	0	1	X	X
X	X	X	X	X
Z	X	X	X	X

Basic gate primitive in verilog

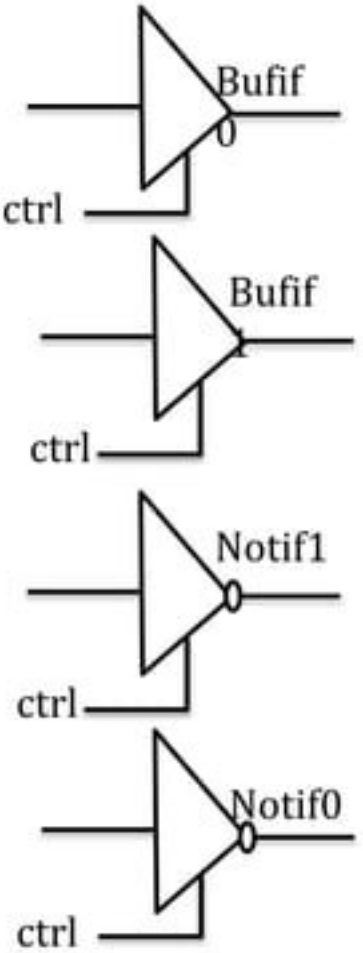


Buf	in	out
	0	0
	1	1
	X	X
	Z	X



Buf	in	out
	0	1
	1	0
	X	X
	Z	X

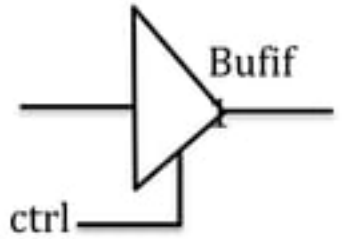
Basic gate primitive in verilog



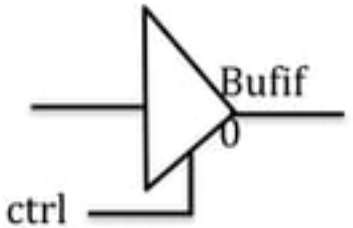
- Instantiation of bufif1 & bufif0
  - ❖ bufif1 (out, in, ctrl);
  - ❖ bufif0 (out, in, ctrl);

- Instantiation of notif1 & notif0
  - ❖ notif1 (out, in, ctrl);
  - ❖ notif0 (out, in, ctrl);

Basic gate primitive in verilog

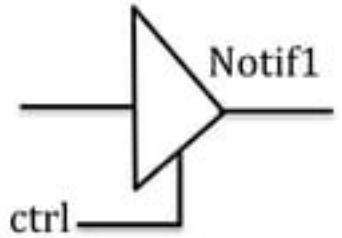


Bufif 1	in \ ctrl	0	1	X	Z
	0	Z	0	L	L
	1	Z	1	H	H
	X	Z	X	X	X
	Z	Z	X	X	X

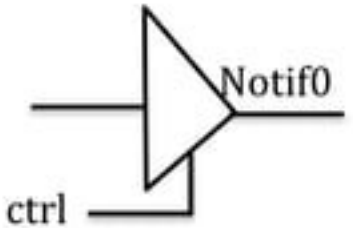


Bufif 0	in \ ctrl	0	1	X	Z
	0	0	Z	H	H
	1	1	Z	L	L
	X	X	Z	X	X
	Z	X	Z	X	X

Basic gate primitive in verilog



Notif1	ctrl 0	1	X	Z
in 0	Z	1	L	L
1	Z	0	H	H
X	Z	X	X	X
Z	Z	X	X	X



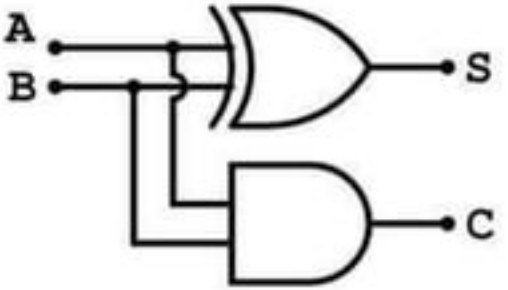
notif 0	ctrl 0	1	X	Z
in 0	1	Z	H	H
1	0	Z	L	L
X	X	Z	X	X
Z	X	Z	X	X

Example of verilog Design using gate level

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

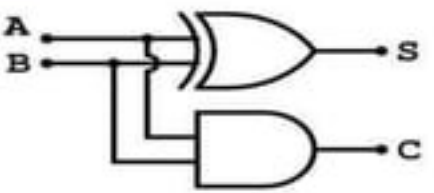
- I/P => A , B
- O/P => Sum & carry

- Boolean Function
- Sum = A^B
- Carry = A.B



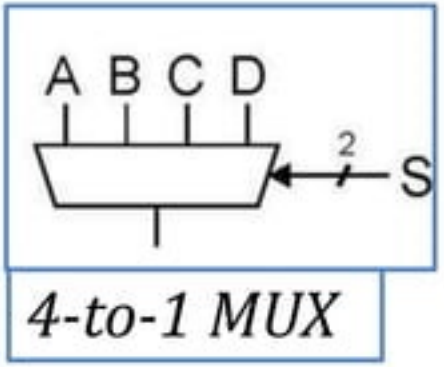
Example of verilog Design using gate level

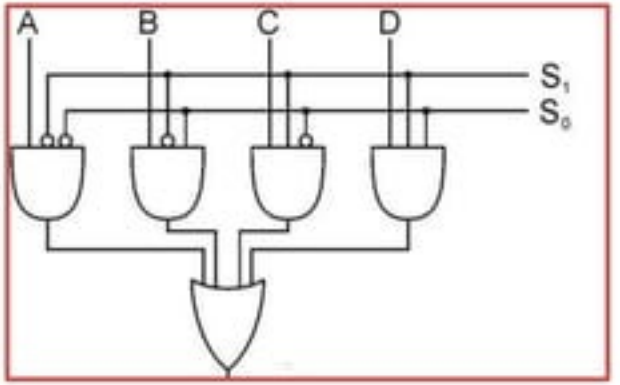
```
// this is half adder verilog code
module half_adder (S, C , A, B )    // module name & port declaration

   output S;                        // output port declarations
   output C;                        // output port declaration

    input A;                        // input port declarations
    input B;                        // input port declarations

  xor  x1 ( S, A, B);               // instantiation of xor gate
  and  a1 (C, A, B);                // instantiation of and gate

endmodule
```



Example of verilog Design using gate level

4 to 1 (MUX)

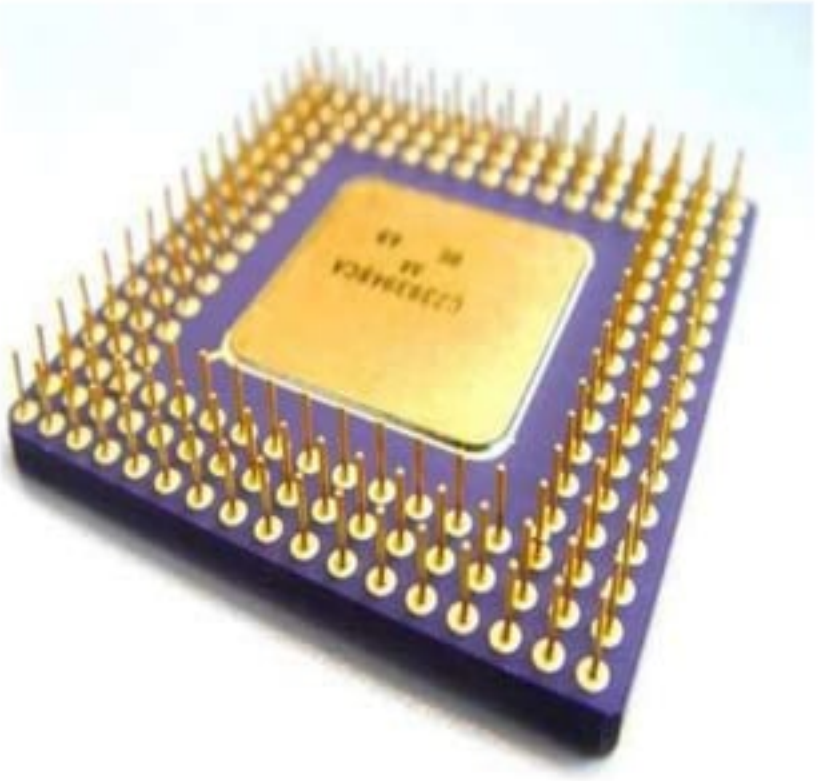





```
// this is 4to1 MUX verilog code
module  mux4to1 (Y, A, B, C, D, S1,S0 )   // module
                                          //declaration

   output Y;                    // output port declarations

   input A, B, C, D;            // input port declarations
   input S0, S1                 // input port declarations
// instantiation of and gates
  and  a1( Y1 , A, S0' , S1')          // instance a1
  and  a2 (Y2, B , S1' , S0);          // instance a2
  and  a3 (Y3 , C , S1 , S0');         // instance a3
  and  a4 (Y4 , D , S1 , S0);          // instance a1

// instantiation of or gate
   or o1(Y, Y1, Y2, Y3, Y4);

endmodule
```

Continues to verilog

1. Introduction to VLSI
2. Introduction to Verilog HDL
3. History
4. Design Methodology
5. Verilog Keywords
6. Lexical Conventions
7. Verilog data types
8. Abstraction
9. Gate level Abstraction
10. Data flow level Abstraction
11. Behavioral level Abstraction
12. Switch level Abstraction
13. Advance verilog Keywords



Data flow level Abstraction

- The gate level approach is very well for small scale logic.
- With the increasing level of structure its difficult to model any design in terms of gates.
- So therefore we move to data flow level.
- With the help of Synthesis Tool we can convert data level code to Gate level code.
- All the Boolean function can be implemented using data flow level.
- Any data flow level can be expressed using continuous assignment.
- Expressions, operators, operands.

Continuous Assignment

- A continuous assignment is the most basic statement in dataflow modeling.
- This continuous assignment is used to drive a value onto a net.
- This continuous assignment is equivalent to gate after synthesis.
- Continuous assignment can be defined by the keyword **assign**.
- Continuous _assign : : = assign {drive_strength} [delay] list_of_net_assignment;
- Drive strength is optional and can be specified in terms of strength level.
- Default value for strength level is strong1 & strong0.
- **Ex :: assign out = in1 | in2;**

Continuous Assignment

- assign out_net = in1 & in2;
- Continuous assignment are always active and can be updated at any time.
- The left hand side value is known as output net and it must be net data type .
- **wire out_net;** // explecitly declared the variable as wire
- **assign out_net = in1 & in2;** // the default declaration of any variable is net type
- The right hand side statement is divided into two parts operands & operator.
- Whenever the value of operands will change it will directly update ssss the output net.
- Operands on right hand side can be reg as well wire.
- Delay can be specified in term of #time_unit >= #10
- assign sum #10 = a + b;
- assign data[15:0] = dataA[15:0 & dataB[15:0];

Implicit Continuous Assignment

- wire data;
- assign data =A | B; // regular continuous assignment

- wire data = A | B ; // implicit continuous assignment

- **Implicit net declaration:**
- **Assign data = A | B ; // by defult.**

Delay Types

- Delay is the time value taken by any continuous expression to transfer its right hand sight value to the left hand sight nets.
- There are three ways to define delay in continuous statement.
- Regular assignment delay.
- Implicit assignment delay.
- Net declaration delay.

Delay Types

- Rise Delay
- Fall Delay
- Turn-Off Delay
- assign #[10,20,30] dout = in1 & in2

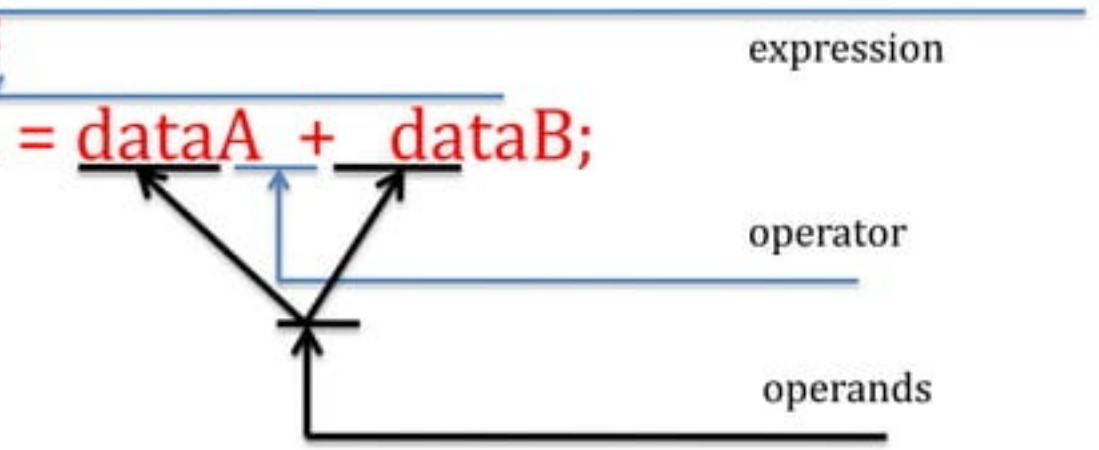
- Min/Typ/Max Delay values
- assign #[20:20:30] dout = in1 & in2

Delays(cont...)

- **Regular assignment delay:**
- assign #10 dout = in1 & in2 ; // delay in the continuous assignment
- **Implicit assignment delay:**
- wire #10 dout = in1 & in2 ; // effect is same as regular assignment
- **Net assignment delay:**
- wire #10 dout ;
- assign dout = in1 & in2; // effect is same in all the expression

Expression, Operator, Operands

- Expression:
- asssign data = dataA + dataB;




```
assign reg_out <= regA + regB ;
assign  data_value = in1 & in2;
```

Operators

- Mathematical :
- (/) divide

```
div = A / B;          // the division of A by B
```

- (*) multiply

```
Mult  = A * B;        // Multiplication of A & B
```

- (%) modulus

```
mod = A %10           //remainder of A div by 10
```

- (+) addition

```
Sum = A + B;          // addition A & B
```

- (-) subtraction

```
Sub = A - B;          // subtraction of A& B
```

- (**)power operator

```
p = A ** 2;           // result is A * A
```

Operators

- Logical:
- ! logical negation

  If(! Reset) // logical invert of Reset

- && logical and

  If (A && B) //logically A &B both should Active

- || logical or

  If (a || b) // logically either a or b should be active

Operators

- Relational:
- > greater than

  If (a> b) // condition a is grater than b
- >= greater than or equal to

  If (a >= b) // condition a is grater than or equal to b
- < less than

  If (a < b) // condition a is less than b
- <= less than and equal to

  If (a <= b) // condition a is less than or equal to b

Operators

- Equality:
- (==) logical equality

  If (A == B) // condition A is Equal to B (result is H ,L X ,Z)

  if value of either A or B is unknown then that result will be X
- (!=) logical inequality

  If (A != B) // condition A is Not Equal to B
- (===) case equality

  If (a === b) // case equality result is H or L

  if value of either A or B is unknown then that result will be 0
- (!==) case inequality

  If (a !== b) // case inequality result is H or

Operators

- Bitwise
- ~ bit-wise negation

```
Out= ~A                                  //inverse  of A
```

- & bit-wise and

```
Out = dataA [7:0] & dataB[7:0]      // bit by bit ANDing
```

- | bit-wise or

```
Out = dataA [7:0] | dataB[7:0]      //  bit by bit ORing
```

- ^ bit-wise XOR

```
Out = dataA [7:0] ^ dataB[7:0]      //  bit by bit XORing
```

- ^~ or ~^ bit-wise xnor

```
Out = dataA [7:0] ~^ dataB[7:0]      //  bit by bit XNORing
```

Operators

- Reduction
- & reduction and

```
dout = & din;          // ANDing of all the bit of din
```

- ~& reduction nand

```
dout = ~& din;         // NANDing of all the bit of din
```

- | reduction or

```
dout = | din;          // ORing of all the bit of din
```

- ~| reduction nor

```
dout = ~| din;         // NORing of all the bit of din
```

- ^ reduction xor

```
dout = ^ din;          // XORing of all the bit of din
```

- ~^ or ^~ reduction xnor

```
dout = ~^ din;         // XNORing of all the bit of din
```

Operators

- Shift operator
- << left shift

```
Out = data << (2)                    // shift the data to left by 2
```

- >> right shift

```
Out = data >>  (2)                   // shift the data to right by 2
```

- >>> Arithmetic right shift

```
Out = data >>  (2)          // if the data is negative unsigned
number
                            // than MSB bit will filled by 1.
```

- <<< Arithmetic left shift

```
Out = data >>  (2)          // if the data is negative unsigned
number
                            //than MSB bit will filled by 1.
```

Operators

❖Conditional

- ?: condition

```
Out = A ? X : Y;          //Ternary operator
```

❖Replication

- {} concatenation

```
Out = { A , B } ;         // concatenate A & B
```

- { { } } Replication

```
Out = { 4{B }} ;          // replicate the value of  B four time
```

Operator Precedence Rules

- / * % + - ! ~ **(unary) highest precedence**
- + - (binary)
- << >>
- < <= > >=
- == != === !==
- &
- ^ ^~
- |
- &&
- ||
- ?: **(ternary operator) lowest precedence**

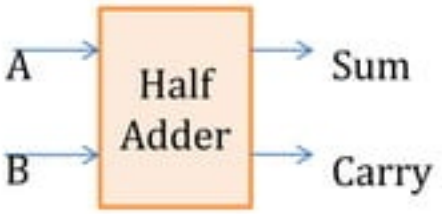
Example of verilog Design using Dataflow level

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- I/P => A , B
- O/P => Sum & carry

- Boolean Function
- Sum = A^B
- Carry = A.B

Half Adder



Boolean Function

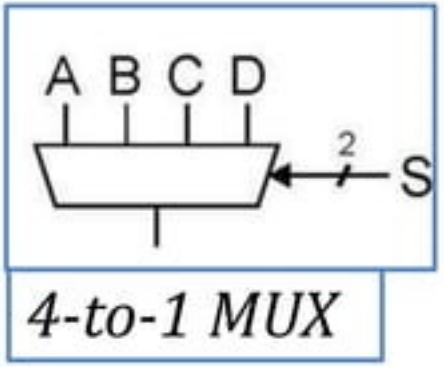
Sum = A^B

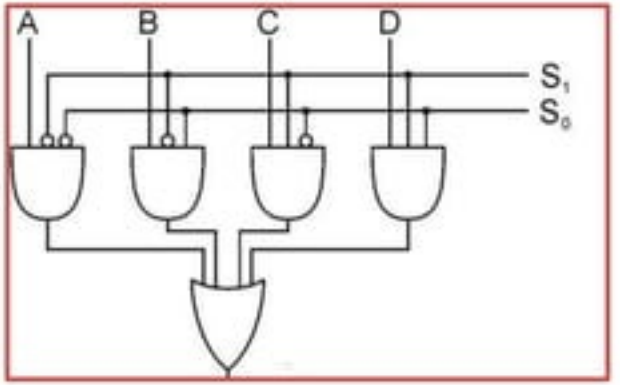
Carry = A.B

```
module half_adder (A , B , Sum , Carry);
  input A;            //first input to Half adder
  input B;            // Second input to Half Adder

  output Sum;         //output from the module
  output Carry;       //output from the module

 wire  Sum , Carry;   //wire declaration of the signal

  assign Sum = A ^ B;  // data flow representation of
    Sum
  assign Carry = A & B;   // data flow representation for
    Carry

endmudule
```

Example of verilog Design using gate level

4 to 1 (MUX)

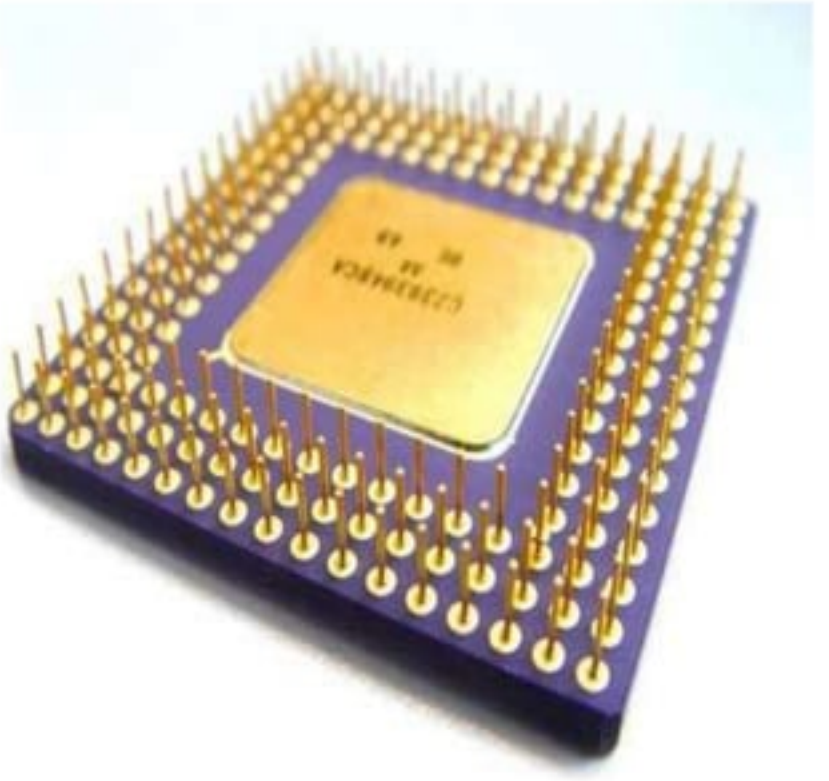





```
// this is 4to1 MUX verilog code
module  mux4to1 (Y, A, B, C, D, S1,S0 )   // module
                                          //declaration

   output Y;                    // output port declarations

   input A, B, C, D;            // input port declarations
   input S0, S1                 // input port declarations
// continuous  assignment for anding
   assign  Y1 =  A &  S0' &  S1';
   assign Y2  =  B & S1' & S0 ;
   assign Y3 =  C  & S1 & S0';
   assign Y4 =  D & S1 & S0 ;
//continuous  assignment for oring
    assign Y =  Y1 | Y2 |  Y3 | Y4 ;

endmodule
```

Continues to verilog

1. Introduction to VLSI
2. Introduction to Verilog HDL
3. History
4. Design Methodology
5. Verilog Keywords
6. Lexical Conventions
7. Verilog data types
8. Abstraction
9. Gate level Abstraction
10. Data flow level Abstraction
11. Behavioral level Abstraction
12. Switch level Abstraction
13. Advance verilog Keywords



Behavioral level Abstraction

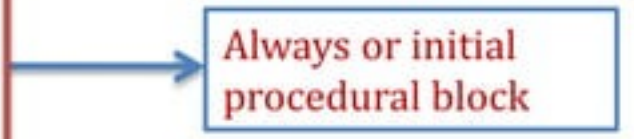
- Modeling a circuit with logic gates and continuous assignments is quite complex.
- Behavioral level is the higher level of abstraction in which models can be defined as its functional behavioral.
- Thus Behavioral procedural constructs are well suited to tackling problems such as describing a microprocessor or implementing complex timing checks.

Behavioral level Abstraction

- Verilog behavioral models contain structured procedural statements that control the simulation and manipulate variables of the data types previously described.
- These statements are contained within procedures.
- Each procedure has an activity flow associated with it.
- There are two types of structured procedural constructs initial and always.

Behavioral level Example

```
module  test;
    integer  dataA;                    // variable declarations
    reg VAR ;                          // variable declarations
type_of_block  (sensitivity_list)
    begin    // beginning of the block
    .....    // multiple statements

    end

endmodule
```



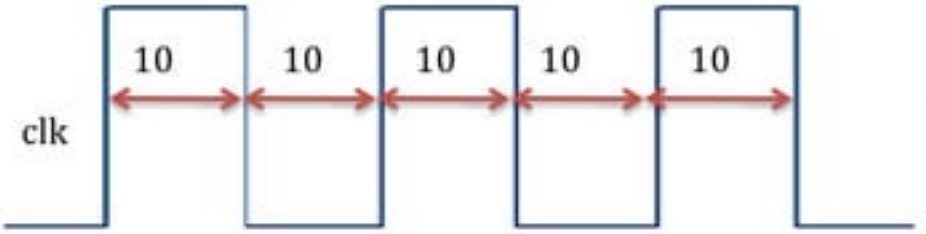
Behavioral level Example

```
module  test;
    integer  dataA;                          // 32 bit integer data
    integer dataB;                           // 32 bit integer data
    integer Result;                          // 32 bit integer data
   initial  begin
        dataA = 7;                           // data assignment
        dataB = 32;                          // data assignment
    end
    always #50 Result  = dataA + dataB ;     //operation of 2 data

    always  monitor("%d  + %d = %d", dataA,dataB,Result);     //monitor All the data

endmodule
```

Structured always block

- All the behavioral statements inside an always block constitute an always block.
- The always block statements start at zero simulation time.
- All the always block continuously execute its statement throughout the entire simulation.
- The value on the statement inside the always block depends its triggering signal and timing control.

always block Example



- The above code will generate a clk signal with 50% duty cycle.
- As always block is that will start at 0 simulation time but wait for 10 time unit and then inverse the clk signal.
- always block keep executing its statements continuously.

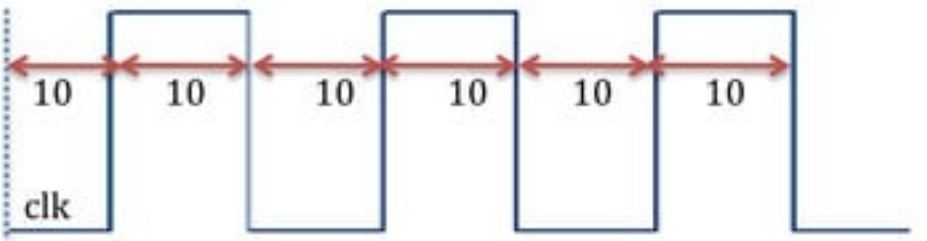
```
module clk_gen;
 reg clk;
     initial clk=1'b0;
     always  #10 clk = ~
  clk;
     initial #60 $finish;
endmodule
```

initial block

- All the behavioral statements inside an initial statement constitute an initial block.
- All the initial statement start concurrently at zero simulation time .
- All the initial block execute its statement only once.
- Each initial block finish execution independently to others.
- Multiple statement are enclosed inside begin end like in C language { } .

initial block example

```
module clk_gen;
 reg clk;
      initial begin
                clk=1'b0;
            #10 clk = 1'b1;
            #10 clk = 1'b0;
            #10 clk = 1'b1;
            #10 clk = 1'b0;
            #10 clk = 1'b1;
            #10 clk = 1'b0;
           end

      initial #60 $finish;
endmodule
```



- The initial block start at 0 simulation time.
- Execute its statement sequentially.
- Two initial block start at 0 simulation time but finish at 60 time unit.

Procedural Assignment

- Procedural Assignments, are used for updating reg, integer, time, and memory variables.
- There is a significant difference between procedural assignments and continuous assignments.
- Continuous assignments drive net variables and evaluate then update whenever an input operand changes its value.

Procedural Assignment

- Procedural assignments update the value of register variables under the control of the procedural flow constructs that surround them.

Blocking Procedural Assignments

- A blocking procedural assignment statement must be executed before the execution of the
- statements that follow it in a sequential block.
- A blocking procedural assignment statement does not prevent the execution of statements that follow it in a parallel block Parallel Blocks.

➢ Syntaxt: <lvalue> = <timing_control> <expression>

➢ Ex: Sum = #10 Data1 + Data2;

Syntax: <lvalue> = <timing_control> <expression>

- Where lvalue is a data type that is valid for a procedural assignment statement.
- = is the assignment operator.
- Timing_control is the optional intra-assignment delay.
- The timing_control delay can be either a delay control (for example, #6) or an event control (for example, @(posedge clk)).
- The expression is the right-hand side value the simulator assigns to the left-hand side.

Non-Blocking Procedural Assignment

- The non-blocking procedural assignment allows you to schedule assignments without blocking the procedural flow.
- We can use the non-blocking procedural statement whenever you want to make several register assignments within the same time step without regard to order or dependance upon each other.

➢ Syntax: <lvalue> <= <timing_control> <expression>

➢ Ex: Sum <= #10 Data1 + Data2;

<lvalue> <= <timing_control> <expression>

- Where lvalue is a data type that is valid for a procedural assignment statementl.
- <= is the non-blocking assignment operator.
- Timing_control is the optional intra-assignment timing control.
- The timing_control delay can be either a delay control (for example, #6) or an event control (for
- example, @(posedge clk)).
- The expression is the right-hand side value the simulator assigns to the left-hand side.

Working process of Non blocking Assignment

- When the simulator encounters a non-blocking procedural assignment, the simulator evaluates and
- executes the non-blocking procedural assignment in two steps.
- 1. The simulator evaluates the right-hand side and schedules the assignment of the new value
- to take place at a time specified by a procedural timing control.
- 2. At the end of the time step, in which the given delay has expired or the appropriate event
- has taken place, the simulator executes the assignment by assigning the value to the left-hand side.

- Ex: a <= b; // take the initial value of a & b and store to temporary
  b <= a; // registrer & at the end of simulation tim assign to RHS.

Working process of Non blocking Assignment

- Multiple non-blocking assignments to the same register in a particular time slot.
- Multiple non-blocking assignments to the same register with timing controls.

Blocking and Non-Blocking Procedural Assignments

- Evaluate the right-hand side of all assignment statements in the current time slot.
- Execute all blocking procedural assignments. At the same time, all non-blocking procedural assignments are set aside for processing.
- Execute all non-blocking procedural assignments that have no timing controls.
- Check for procedures that have timing controls and execute if timing control is set for the current time unit.
- Advance the simulation clock.
- Conditional Statement.

Timing controls

- Delay based Timing Control.
- Intra-Assignment Timing Controls .
- Inter-Assignment Timing Controls
- Zero-Assignment Delay
- Event based Timing Control.
- Regular Event Control.
- Named Event Control.
- Event OR Control.

- Level Sensitive timing Control

Delay Based Timing Control

- Inter-Assignment Timing Controls:
- This is also known as regular assignment delay or inertial delay.
- In this delay expression that statement get that much of time delay to execute.
  - #d dout = din; // execute the statement at d delay
  - #10 dout = din; // execute the statement at 10 time unit
  - #20 Sum = A + B; // execute the statement at 20 time unit

Inter-Assignment Timing Controls:

```
initial
begin
   a = 0;          // executed at simulation time 0
   #10 b = 2;      // executed at simulation time10
   #15 c = a;      // ... at time 25
   #2 c = 4;       // ... at time 27
   b=5;            // ... at time 27
end
```

Delay Based Timing Control

- Intra-Assignment Timing Controls:
- This is also known as propagation or transport delay.
- In this expression the RHS value is stored to temporary register and assign to LHS at that much of time Delay.

```
dout = #d  din;      // assign the RHS value at d delay to LHS
dout = #10 din;      // assign the RHS value at 10 delay to LHS
Sum = #20  A + B;    // assign the RHS value at 20 delay to LHS
```

Delay Based Timing Control

- Zero-Assignment Delay:
- This is Zero Delay expression in this that statement get executed at the end of all the statement before that.

```
o dout = #0  din;        // assign the RHS value at 0 delay to LHS
o  #0  dout = din;       // assign the RHS value at 0 delay to LHS
o #0  Sum = A + B;  // assign the RHS value at 0 delay to LHS
```

Event based Timing Control.

- Regular Event Control:
- Event-based timing control allows conditional execution based on the occupance of a named event.
- Verilog waits on a predefined signal or a user defined variable to change before it executes a block.

```
@ (clock) a = b;                  // when the clock changes value, execute a = b
@ (negedge clock) a = b;          // when the clock change to 0, execute a=b
a = @(posedge clock) b;           // evaluate b immediately and assign to a on
                                  // a positive clock edge.
```

Event based Timing Control

- Named based Event control:

```
event data_in;              // user-defined event
always @(negedge clock) if (data[8]==1)
    -> data_in;             // trigger the event
always @(data_in)           // event triggered block
 mem[0:1] = buf;
```

➢ This can be read as: at every negative clock edge, check if data[8] is 1, if so assert data_in. When data_in is asserted, the statement if the second *always* is executed.

Event based Timing Control

- Event OR Control:

  @ *(event_exp* **or** *event_exp)*
  Event_Expression;

- If we wish to execute a block when any of a number of variables change we can use the sensitivity list to list the triggers separated by **or** in verilog 1995.
- but we can use single comma(,) to separate the sensitivity list variable in verilog 2001.

```
always @ (i_change or he_changes or she_changes)
        somebody_changed = 1;
```

- A change in any of the variables will cause execution of the second statement. As you can see this is just a simple extention to the idea of event based timing control described in the previous section.

Timing Control

- Level Sensitive timing Control:

```
always @ (posedge clk)
    begin
     wait (count_en)
      count = count +1;     // Wait for count_en
    end
```

➢ This will wait for the signal count_en and make an increment on the count value whenever count_en is high.

Blocking_nonblocking Timing Examnple

```
module blocking_nonblocking();
 reg a,b,c,d;
 // Blocking Assignment
 initial begin
       #10 a = 0;
       #11 a = 1;
       #12 a = 0;
       #13 a = 1;
        end
initial begin
      #10 b <= 0;
      #11 b <= 1;
      #12 b <= 0;
      #13 b <= 1;
   end
initial begin
       c = #10 0;
       c = #11 1;
        c = #12 0;
        c = #13 1;
    end
 initial begin
       d <= #10 0;
       d <= #11 1;
       d <= #12 0;
       d <= #13 1;
 end
    initial begin
$monitor("TIME = %g A = %b B = %b C = %b D =
      %b",$time, a, b, c, d);
#50 $finish;
end
endmodule
```

Conditional statement

The *if* statement.

Syntax:

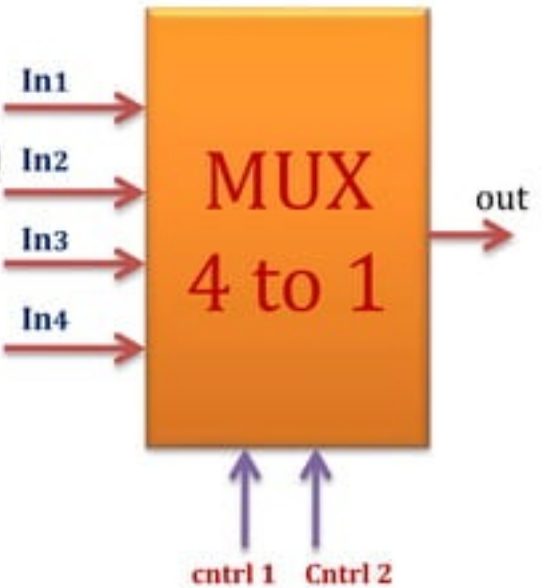
if
(conditional_expression)
statement
else
statement

```
module mux2_1 (out, in1, in2, sel);
   output out;
   input in1, in2;
   input sel;
   reg out;     // reg type variable
always @(in1 or in2 or sel)
      if (sel==1)    // condition Expr
        out = in1;
      else
        out = in2;
endmodule
```

The *if else* statement Ex

- The 4 to 1 multiplexor:

```
module multiplexor4_1 (out, in1, in2, in3 ,in4, cntrl1, cntrl2);
        output out;
        input in1, in2, in3, in4, cntrl1, cntrl2;
        reg out; // Note that this is now a register

          always @(in1 or in2 or in3 or in4 or cntrl1 or cntrl2)
              if (cntrl1==1)
                    if (cntrl2==1)
                       out = in4;
                    else
                       out = in3;
               else if (cntrl2==1)
                       out = in2;
                    else
                       out = in1;
  endmodule
```



The *case* statement

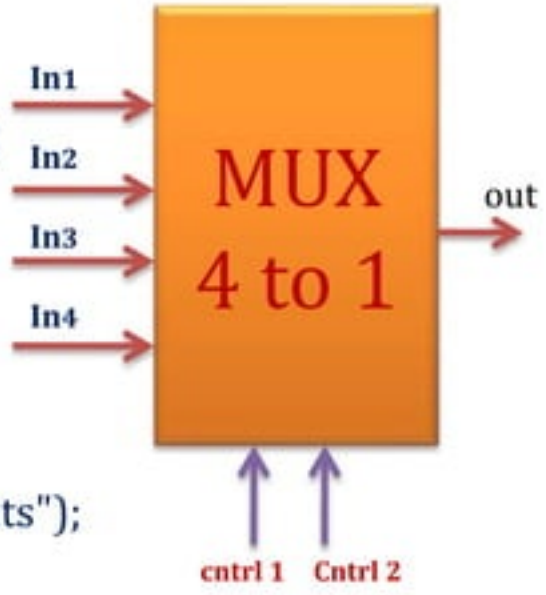
- If else condition method for implementing the multiplexor for more than 4 input lines it would become more tedious.
- A clearer implementation uses a *case* statement .
- This is very similar in syntax to the case statement in C.

Syntax:

```
case (Expression)
   Alternative1 : statement1;
   Alternative2 : statement2;
   Alternative3 : statement3;
   default : statement_default;
endcase
```

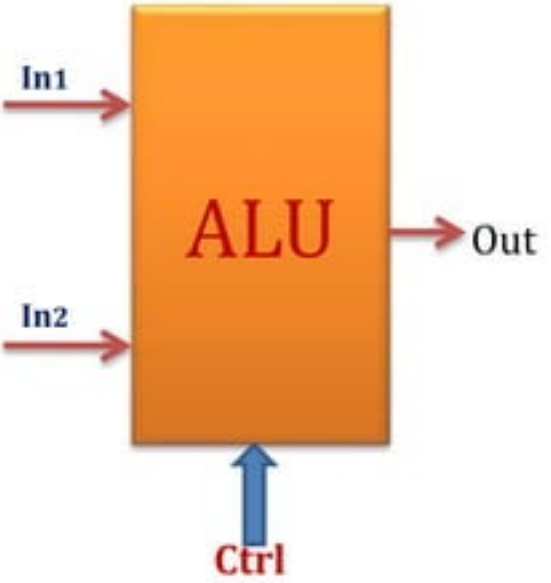
The *case* statement Ex.

```
module multiplexor4_1 (out, in1, in2, in3, in4, cntrl1, cntrl2);
   output out;
   input in1, in2, in3, in4, cntrl1, cntrl2;
  reg out;              // note must be a register
     always @(in1 | in2 | in3 | in4 | cntrl1 | cntrl2)
        case ({cntrl2, cntrl1}) // concatenation

          2'b00 : out = in1;
          2'b01 : out = in2;
          2'b10 : out = in3;
          2'b11 : out = in4;
         default : $display("Please check control bits");

        endcase

endmodule
```



ALU Example using case statement

```
module ALU(In1,In2,Ctrl,Out);
  input  integer In1 , In2;
   input [1:0] Ctrl;
  output integer Out;
   always @*
      begin
       case (Ctrl)
         2'd0 :   Out  =  In1 / In2 ;   // Div operation
         2'd1 :   Out  =  In1 * In2 ;  // Mult operation
         2'd2 :   Out  =  In1 + In2;  // Add Operraion
         2'd3 :   Out  =  In1 -  In2; // Sub Operation
         default : Out = 0;
     endcase
 endmodule
```



The conditional operator

- *conditional_expression* **?** *true_expression* **:** *false_expression*
- This is similar to the conditional operator in C, it can be imagined to be an abbreviated *if-else.*
- The conditional expression is evaluated in the same way, if it evaluates to true the true_expression is evaluated else the *false_expression.*

Ex: assign *Out = Sel ? In1 : In2;*

Looping Constructs

- There are four looping constructs in Verilog are ***while, for, repeat*** and ***forever.***
- All of these can only appear inside **initial** and **always** blocks. The basic idea of each of the loops is :

while: executes a statement or set of statements while a condition is true.

for: executes a statement or a set of statements. This loop can initialise, test and increment the index variable in a neat fashion.

repeat: executes a statement or a set of statements a fixed number of times.

forever: executes a statement or a set of statements forever and ever, or until the simulation is halted.

While Loop

- **Syntax**: ***while*** *(conditional) statement ;*
- The while loop executes while the conditional is true, the conditional can consist of any logical expression.
- Multiple Statements in the loop can be grouped using the keywords **begin ... end.**

```
`define JUGVOL 100
module cup_and_jugs;
  integer  jug;
  initial
    begin
      jug = 0;
      while (jug < `JUGVOL)
       begin
        jug = jug + 10;
$display("It takes %d  jugs", cJug);
       end
  end
endmodule
```

While Loop EXERCISE

How many times does the folowing while loop say hello?

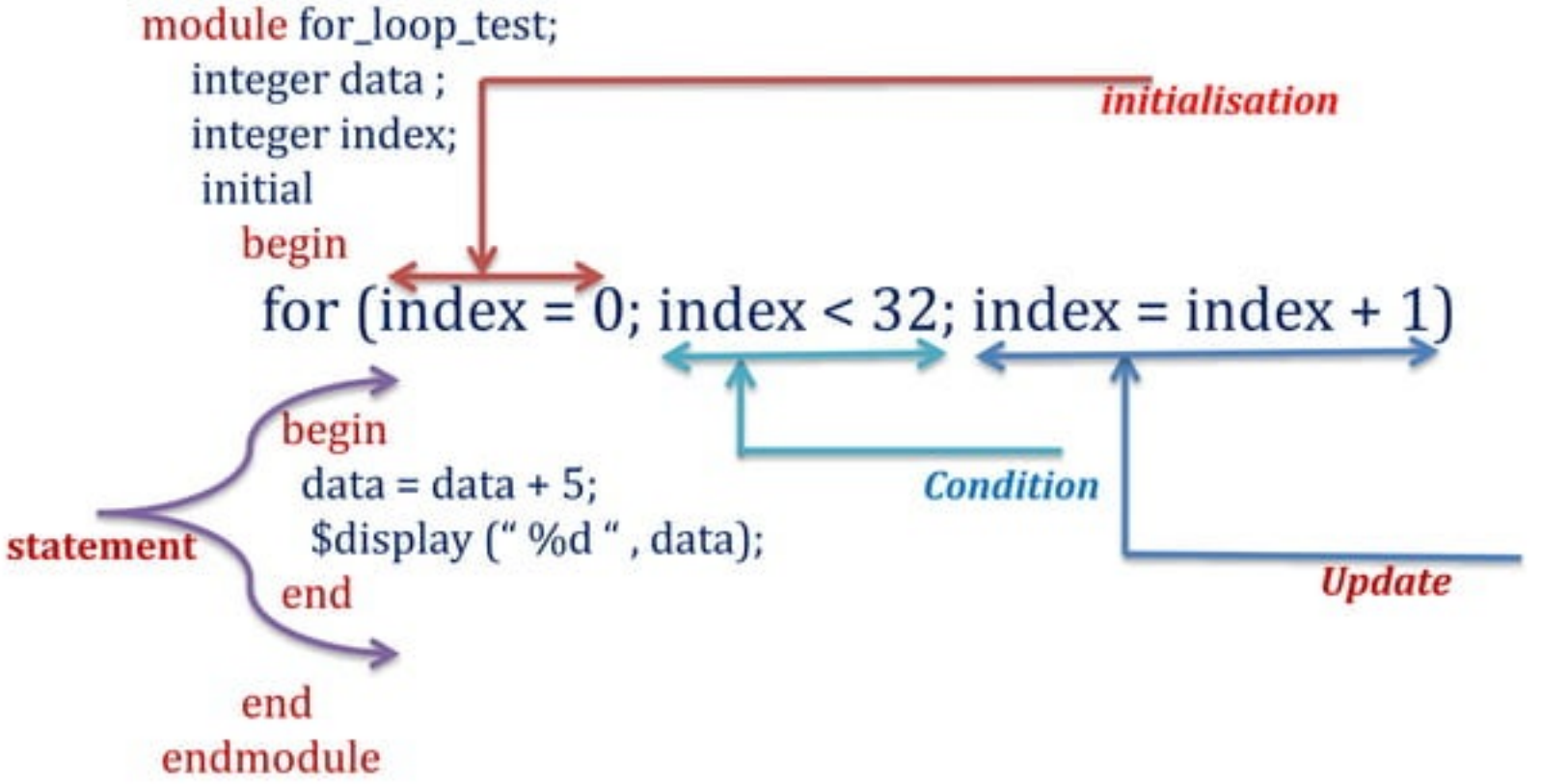
```
initial
    begin
     a = 5;
     c = 0;
      while (a != c)
        begin
          $display("Hello");
          c = c + 1;
        end
end
```

For Loop

- **Syntax:**
  for *(initialisation ; conditional ; update)*
  statement

- The **for** loop is the same as the for loop in C.
- It has three parts:
- the first part is executed once, before the loop is entered.
- the second part is the test which (when true causes the loop to re-iterate).
- The third part is executed at the end of each iteration.

For Loop Ex


module for_loop_test;
integer data ;
integer index;
initial
begin
for (index = 0; index < 32; index = index + 1)
initialisation
Condition
Update
statement
begin
data = data + 5;
$display (" %d " , data);
end
end
endmodule

repeat Loop

- **Syntax**: ***repeat*** *(conditional)*
  statement
- A repeat loop executes a fixed number of times,.
- the conditional can be a constant.
- variable or a signal value but must contain a number.
- If the conditional is a variable or a signal value, it is evaluated only at the entry to the loop and not again during execution.
- If the conditional is X or Z then its treated as Zero.

repeat loop Ex

```
module comply;
 int count; // counting down from 128
  initial
      begin
       count = 128;
        repeat (count)
         begin
            $display("%d seconds to comply", count);
          count = count - 1;
         end
      end
endmodule
```

Forever Loop

- **Syntax:** ***forever*** *statement*
- The **forever** loop executes continuously until the end of a simulation is requested by a **$finish.**
- It can be thought of as a **while** loop whose condition is never false.
- The **forever** loop must be used with a timing control to limit its execution, otherwise its statement would be executed continuously at the expense of the rest of the design.

Forever Loop Ex

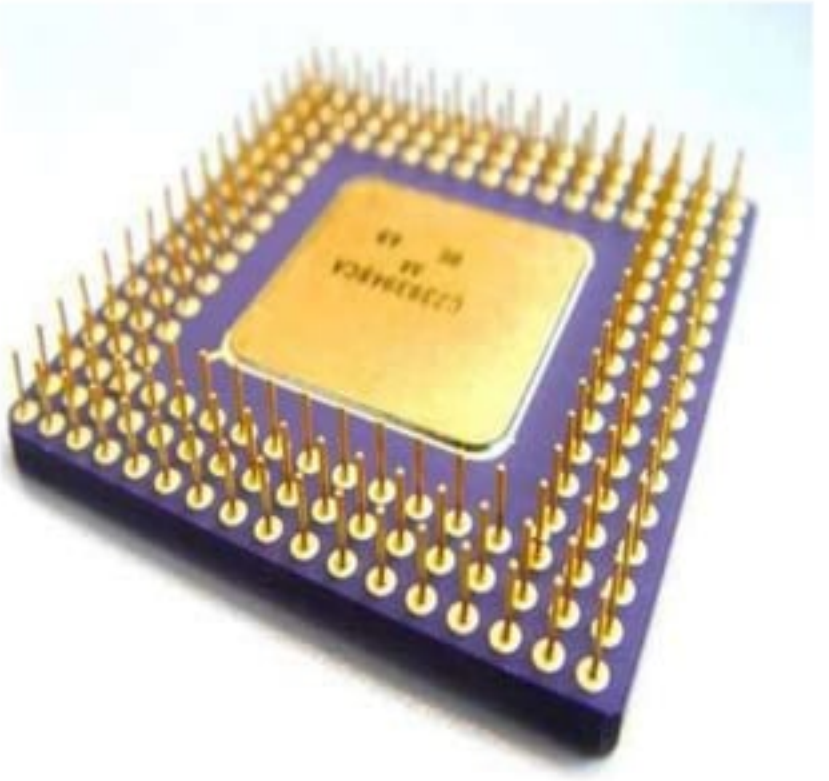
```
reg clock;
 initial
  begin
   clock = 1'b0;
forever #5 clock = ~clock; // the clock flips every 5 time units.
  end
initial
 #2000 $finish;
```

Tasks and Functions

- Tasks and functions provide the ability to execute common procedures from several different places in a description.
- They also provide a means of breaking up large procedures into smaller ones to make it easier to read and debug the source descriptions.
- Input, output, and inout argument values can be passed into and out of both tasks and functions.
- The next section discusses the differences between tasks and functions.

Continues to verilog

1. Introduction to VLSI
2. Introduction to Verilog HDL
3. History
4. Design Methodology
5. Verilog Keywords
6. Lexical Conventions
7. Verilog data types
8. Abstraction
9. Gate level Abstraction
10. Data flow level Abstraction
11. Behavioral level Abstraction
12. Tasks & functions
13. Switch level Abstraction
14. Advance verilog Keywords



Tasks and Functions

- A function must execute in one simulation time unit; a task can contain time-controlling statements.
- A function cannot enable a task.
- A task can enable other tasks and functions.
- A function must have at least one input argument.
- A task can have zero or more arguments of any type.
- A function returns a single value.
- A task does not return a value.
- The purpose of a function is to respond to an input value by returning a single value.
- A task can support multiple goals and can calculate multiple result values. However, only the output or inout arguments pass result values back from the invocation of a task.
- A Verilog model uses a function as an operand in an expression . The value of that operand is the value returned by the function

Tasks and Functions

- switch_bytes (old_word, new_word);

- The task switch_bytes would take the bytes in old_word, reverse their order, and place the reversed bytes in new_word. A word-switching function would return the switched word directly. Thus, the function call for the function switch_bytes might look like the following example:

- new_word = switch_bytes (old_word);

Tasks and Task Enabling

- A task is enabled from a statement that defines the argument values to be passed to the task and the variables that will receive the results.
- Control is passed back to the enabling process after the task has completed.
- Thus, if a task has timing controls inside it, then the time of enabling can be different from the time at which control is returned.
- A task can enable other tasks, which in turn can enable still other tasks—with no limit on the number of tasks enabled.
- Regardless of how many
- Tasks have been enabled, control does not return until all enabled tasks have completed.

Task

```
task <name_of_task>
      <parameter_declaration>
      <input_declaration>
      <output_declaration>
      <inout_declaration>
      <reg_declaration>
      <time_declaration>
      <integer_declaration>
      <real_declaration>
     <event_declaration>
     <task body>
endtask
```

Task Example

```
task clk_gen;
  input integer clk_period;
    begin
      clk=1'b0;
      forever #clk_period/2
      clk = ~clk;
    end
endtask
```

Function Syntax

```
function <range_or_type>? <name_of_function> ;
        <parameter_declaration>
        <input_declaration>
        <output_declaration>
        <inout_declaration>
        <reg_declaration>
        <time_declaration>
        <integer_declaration>
        <real_declaration>
        <event_declaration>
endfunction
```

Function Example

```
function integer funct;
  input integer dataA;
  input ineger dataB;
   begin
     funct = dataA + dataB;
   end
endfunction
```

Calling a task & Function

```
module task_funct_test;
 integer data_type;
  initial
    begin
      task_name (task_parameter);
     data_type=  function_name(funct_parameter);
    end
   assign funct_value=  function_name(funct_parameter);
endmodule
```

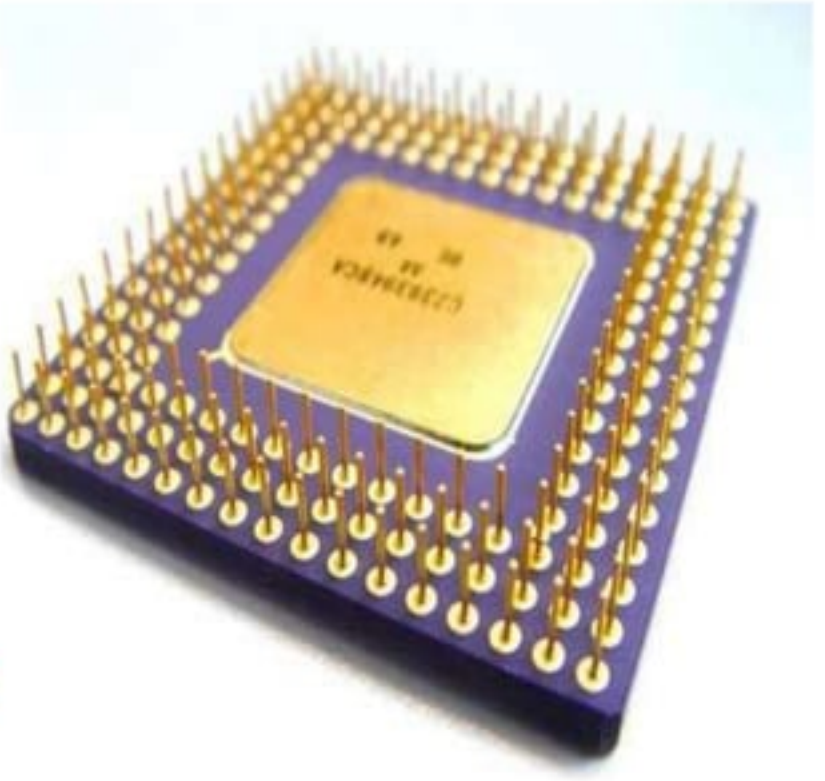
Example for Function

```
module tryfact;
 // define function
 function [31:0] factorial;
   input [3:0] operand;
   reg [3:0] index;
   begin
    factorial = operand ? 1 : 0;
   for (index = 2; index <= operand; index =
index + 1)
            factorial = index factorial;
       end
     endfunction
```

```
//Test the function
        reg [31:0] result;
        reg [3:0] n;
         initial
         begin
              result=1;
               for( n=2; n<=9; n=n+1)
                begin
  $display("Partial result n=%d
result=%d", n,result);
   result = n * factorial (n) / ((n*2)+1);
   end
   $display ("Final result=%d", result);
                 end

                endmodule
```

Continues to verilog

1. Introduction to VLSI
2. Introduction to Verilog HDL
3. History
4. Design Methodology
5. Verilog Keywords
6. Lexical Conventions
7. Verilog data types
8. Abstraction
9. Gate level Abstraction
10. Data flow level Abstraction
11. Behavioral level Abstraction
12. Tasks & functions
13. Verilog useful modeling keywords
14. Switch level Abstraction
15. Advance verilog Keywords



Procedural continuous Assignment

- The procedural continuous assignments are procedural statements that allow expressions to be driven continuously onto registers or nets.

- <statement>
- assign <assignment> ;
- <statement>
- deassign <lvalue> ;

- <force_statement>
- force <assignment> ;
- <release_statement>
- release <lvalue> ;

Assign and deassign Procedural Statements

- The assign and deassign procedural statements allow continuous assignments to be placed on to registers for controlled periods of time.
- The assign procedural assignment statement overrides procedural assignments to a register.
- The deassign procedural statement ends a continuous assignment to a register.
- modeling of asynchronous clear/preset on a D-type edge-triggered flip-flop, where the clock is inhibited when the clear or preset is active.

Assign and deassign Procedural Statements Example

```
// D Flip-Flop model
module d_ff (clk,rst,d,q);      // Design D_FF
  input clk,rst,d;              // input ports
  output q;                     // output port
  reg q;                        // reg type declaration
  always @ (rst)                //sensitivity list
    if (rst)
      assign  q <= 1'b0;        //continuous procedural assignment
    else
      deassign   q;             //release the signal
  always @ (posedge clk)        // assign when posedge of clk is high
      q <= d;
endmodule
```

force and release Procedural Statements

- Another form of procedural continuous assignment is provided by the force and release procedural statements.
- These statements have a similar effect to the assign-deassign pair, but a force can be applied to nets as well as to registers.
- The left-hand side of the assignment can be a register, a net, a constant bit select of an expanded vector net, a part select of an expanded vector net, or a concatenation.
- It cannot be a memory element (array reference) or a bit-select or a part-select of a vector register or non-expanded vector net.
- One can use force and release while doing gate level simulation to work around reset connectivity problems. Also can be used insert single and double bit errors on data read from memory.

force & release Procedural Statements Example

```
// D Flip-Flop model
module d_ff (clk,rst,d,q);      // Design D_FF
  input clk,rst,d;              // input ports
  output q;                     // output port
  reg q;                        // reg type declaration
  always @ (rst)                //sensitivity list
    if (rst)
      force q <= 1'b0;          //continuous procedural assignment
    else
      release  q;               //release the signal
  always @ (posedge clk)        // assign when posedge of clk is high
      q <= d;
endmodule
```

Overriding parameters

- defparam Statement:
- Using the defparam statement, parameter values can be changed in any module instance throughout the design using the hierarchical name of the parameter.
- The defparam statement is particularly useful for grouping all of the parameter value override assignments together in one module.
- The code in Example 12-3 illustrates the use of a defparam.

defparam Statement

```
module top;
 reg clk;
 reg [0:4] in1;
 reg [0:9] in2;
 wire [0:4] o1;
 wire [0:9] o2;
 vdff m1 (o1, in1, clk);
 vdff m2 (o2, in2, clk);

defparam
m1.size = 5,
m1.delay = 10,
m2.size = 10,
m2.delay = 20
endmodule
```

```
module vdff (out, in, clk);
parameter size = 1, delay = 1;
input [0:size-1] in;
input clk;
output [0:size-1] out;
reg [0:size-1] out;
 always @(posedge clk)
 # delay out = in;
endmodule
```

Overriding parameters

- Module Instance Parameter Value Assignment:
- An alternative method for assigning values to parameters within module instances is similar in appearance to the assignment of delay values to gate instances.
- It uses the syntax # (<expression> <,<expression>>) to supply values for particular instances of a module to any parameters that have been specified in the definition of that module.

Module Instance Parameter overriding Example

```
module top;
 reg clk;
 reg [0:4] in1;
 reg [0:9] in2;
 wire [0:4] o1;
 wire [0:9] o2;
 vdff #( .size (5), .delay(10))  m1 (o1, in1, clk);
 vdff #(.size (10), .delay(20)) ) m2 (o2, in2,
     clk);

endmodule
```

```
module vdff (out, in, clk);
parameter size = 1, delay = 1;
input [0:size-1] in;
input clk;
output [0:size-1] out;
reg [0:size-1] out;
 always @(posedge clk)
 # delay out = in;
endmodule
```

Real Numbers in Port Connections

- The real data type cannot be directly connected to a port, but rather must be connected indirectly.
- The system functions $realtobits and $bitstoreal are used for passing the bit patterns across module ports.

Real Numbers in Port Connections

```
module driver
   (net_r);
 output net_r;
 real r;
 wire [64:1] net_r =
   $realtobits(r);
endmodule
```

```
module receiver (net_r);
 input net_r;
 wire [64:1] net_r;
 real r;
 initial assign r =
   $bitstoreal(net_r);
endmodule
```