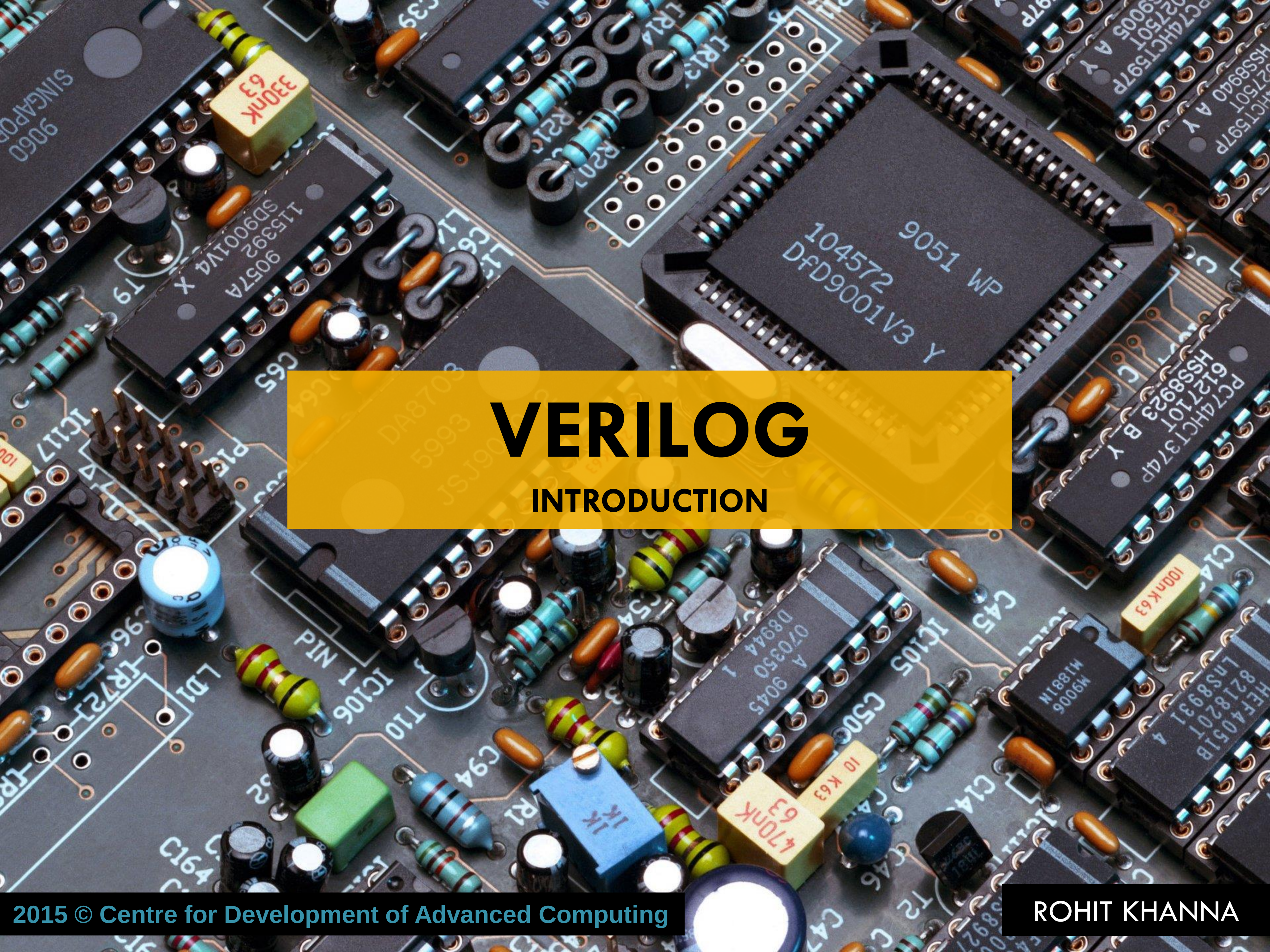


VERILOG

INTRODUCTION

ROHIT KHANNA

History and Evolution of Verilog

1984

Gateway Design Automation introduced Verilog (Prabhu Goel and Phil Moorby)

1989 to 1990

Cadence Design Systems purchased Gateway Design Automation

Cadence Organized Open Verilog International (OVI)

1995

IEEE Standardized Verilog (IEEE 1364 -1995)

History and Evolution of Verilog

2001

New features added and fixed problems with Verilog1995 (IEEE 1364-2001)

2005

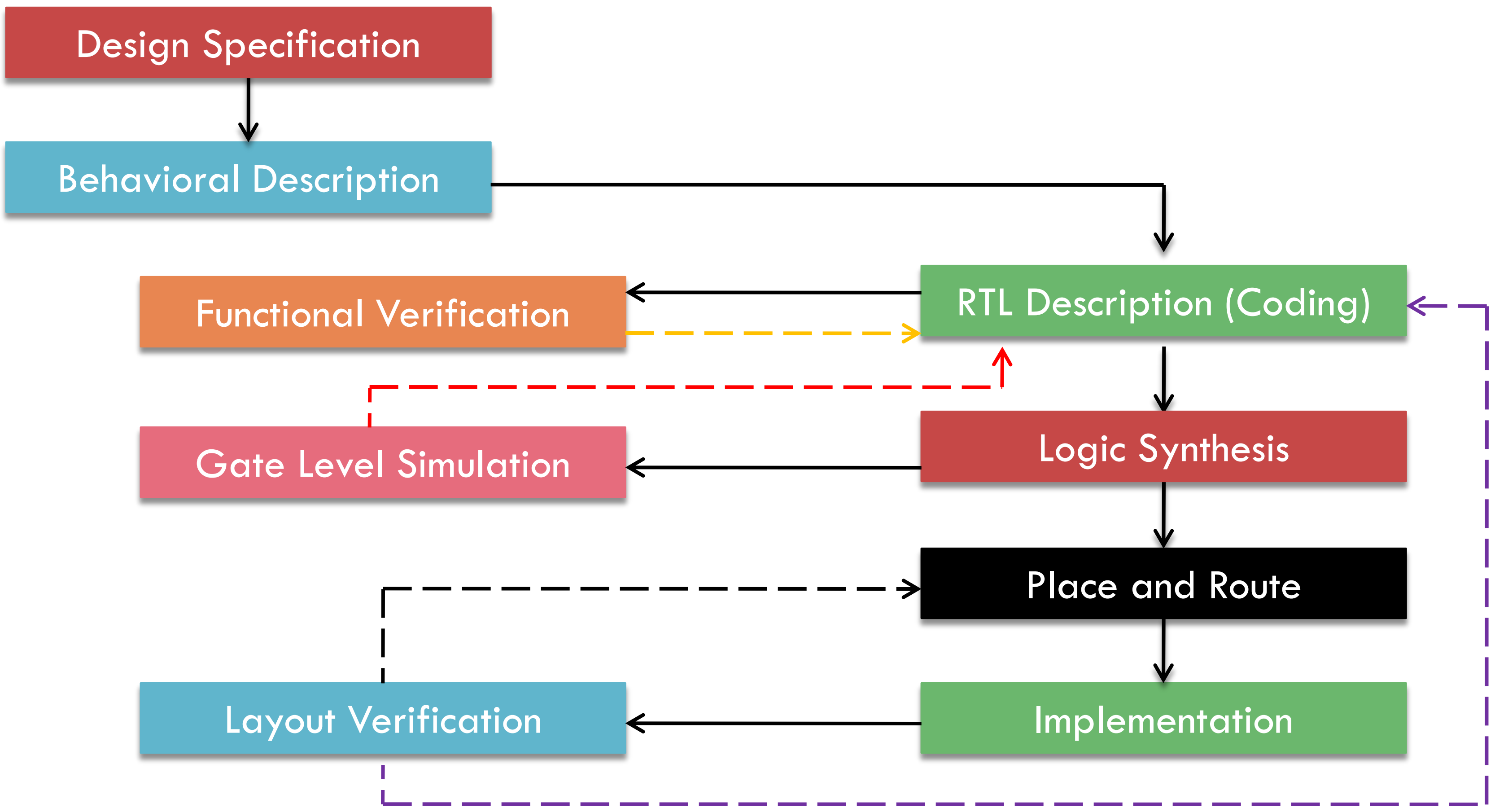
Minor Corrections and few new features added (IEEE 1364-2005)

2009

Super Subset of Verilog 2005 called **System Verilog**

New features added to aid verification and modelling.

Design Flow


Design Specification
Behavioral Description
RTL Description (Coding)
Functional Verification
Logic Synthesis
Gate Level Simulation
Place and Route
Implementation
Layout Verification

Features

- Verilog is case sensitive.
- Most of the syntax is adopted from C language.
- Extension used by verilog files is ".v"
- Verilog can be used to model a digital circuit at Behaviour, Data Flow, Gate and Switch level.
- Verilog supports additional features like Timing Checks, User Defined Primitive(UDP) and Programming Language Interface(PLI).

Comments

- There are two ways to provide comment
  - Single Line Comment

    Begins with double slashes (//)
  - Block Comment

    Comments are enclosed between /* and */

Example

```
// This is a single line comment
 /* This is block comment
     Statement 1
     Statement 2
 */
```

Whitespace

- Blank Spaces(\b), Tabs(\t) and Newlines (\n) constitute whitespaces in verilog.
- White spaces are ignored by Verilog except when it separates two identifier.
- Whitespaces are not ignored in strings.

Identifiers

- Identifiers are name given to objects that can be referred in a design.
- An identifier can contain alphabets(a-z, A-Z), digits(0-1) and special characters(_ , $).
- An identifier must begin with alphabet or underscore(_).
- Identifiers are case sensitive.

Identifiers

- There are identifiers which allow use of any printable ASCII character. They are called as escaped identifier.

- An escaped identifier starts with a \ and ends with a white space.

Example

```
My_Or
my_or                // My_Or and my_or are two different identifiers
_test_bench$
\@pp!e               // Escaped identifier
```

Keywords

- In verilog few identifiers are reserved to define language constructs, they are called as keywords.
- All keywords are in lower case.

Examples

module

always

wire

reg

Number Specifications

- There are two types of number specifications
  - Sized Numbers
  - Unsized Numbers

- Size Numbers

  Syntax : <size>' <base> <number>

  size: decimal value specifying number of bits to represent number.

  base: base represents the format of the number. It can be

decimal (d or D)	hexadecimal (h or H)
octal (o or O)	binary (b or B)

  number: specifying number is chosen base format.

Number Specifications

- Unsized Numbers
  - Numbers are written without size specification.
  - Default number of bits depends upon the machine and simulator, must be at least 32.
  - Decimal is the default base format if not specified.

- Negative Numbers
  - To represent a negative number, add minus sign("-") before the size specification.
  - Negative numbers are stored in 2's Complement.

Number Specifications

Examples

156	// Decimal number
‘hab3	// Hexadecimal number
‘o54	// Octal Number
5’b10110	// 5-bit binary number
12’h7f3	// 12-bit hexadecimal number
-4’d4	// 2’s complement of 4
-3’b010	// 2’s complement of 2
9’o342	// 9-bit octal number
8’h-af	// Invalid syntax

Number Specifications

- X or Z values
  - X is used to represent unknown values.
  - Z is used to represent high impedance.
  - If the MSB bit of the number is x or z, then x or z is padded respectively to fill the remaining most significant bits.
  - If the MSB bit of the number is 1 or 0, then 0 is padded to fill the remaining most significant bits.
- _ or ?
  - _ is allowed anywhere in a number except at first character.
  - _ are used to improve readability. Ignored by the compiler.
  - ? is an alternative for z in verilog.

Number Specifications

Examples

```
12'h14x          // 12 bit hex number with 4 LSB bits X
11'oz32          // 11 bit octal number with 5 MSB bits Z
5'b110           // 5 bit binary number with  2 MSB bits 0

16'b1011_0100_1100_1010
// 12 bit binary number with _ to improve readability.

10'h9??  // 10 bit hex number with 8 LSB bits Z
//Result is truncated to fit in 10 bits by ignoring MSB bits
```

Module

- Module is the name given to a design unit in Verilog.
- Modules are used to provide a common functionality that can be used in others design units.
- A module can be used as a component in other modules.
- Ports of the modules are used to communicate with other modules.

Module

Syntax:

```
 module module_name (port_list);
 <port directions>  port_names;
 //Local declaration
//functionality of module
endmodule          // no semicolon
```

Port_list : specify all ports used to communicate with the module

Port_direction : specify the direction of data flow.

Module

- Verilog allows three types of ports
  - input: Value can be read but cannot be assigned.
  - output: Value can be assigned and can be read internally.
  - inout: Bi-directional port, value is read or assigned depending upon the control pin.

Module

Example: Scalar Port

```
module xor_gate (a, b, c);      // "," is used to separate
                               // port entries

input  a, b ;                   // Direction of each port
output c;
//Local declaration
//functionality of module
endmodule          // no semicolon
```

Module

Example: Vector Port

```
module fulladder  (a, b, c, sum, carry);
//ports with same size are written together
input  [3:0] a, b ;     // [MSB : LSB]
input c;
output [3:0] sum;
output carry;
//Local declaration
//functionality of module
endmodule            // no semicolon
```

Module

Example: Vector Port - ANCI C Style

```
module fulladder ( input [3:0] a, b,
                   input c,
                   output [3:0] sum,
                   output carry);
   //Local declaration
  //functionality of module
endmodule          // no semicolon
```

Module

- The functionality of the module can be defined at four different levels of abstraction

  - Behavioral : In terms of algorithm without concern for hardware.
  - Data Flow : In terms of how data flows between registers.
  - Gate : In terms of Logic gates and their connections.
  - Switch : In terms of switches and their connections.

Data Types

- Verilog supports following data type
  - Value Set
  - Nets
  - Registers
  - Vectors
  - Integer
  - Real
  - Time
  - Arrays
  - Parameters
  - Strings

Data Types

Value Set

- Verilog supports four basic values to model functionality of a hardware.

0 – Logic Zero

1 – Logic one

X – Unknown Logic Value

Z – High Impedance

Value X and Z are case insensitive

Data Types

Nets

- Nets are used to represent connection between elements.
- A net should be continuously driven by some hardware.
- If there is no driver, the value of a net is Z.

Syntax: <net_type> [MSB: LSB] net_name;   Default Value: Z

- wire is most commonly use net type.
- Other net data types are wand, wor, tri etc.

```
Examples: wire temp1;            //1- bit net
          wire [3:0] temp2;      //4- bit net
```

Data Types

Registers

- Registers represent storage elements.
- It maintains value until a new value is assigned.
- Registers don't require a constant driver.

Syntax: `reg [MSB: LSB] reg_name;`   Default Value: X

- Can be assigned only inside always or initial statements.
- It is used to store unsigned quantities.

Examples:
```
reg temp1;                    //1-bit unsigned register
reg [3:0] temp2;              //4-bit unsigned register
reg signed [11:0] temp3;      //12-bit signed register
```

Data Types

Vectors

- Vector is used to represent a group of bits.
- Nets or reg data types can be declared as vectors.

Examples:
```
wire temp1;              //Scalar net
wire [0:7] temp2;        //8 bit vector net
reg [3:0] temp3;         //4 bit vector reg
```

Selecting part of a Vector

```
temp2[5];      //Selecting 5th bit of temp2
temp2[3:6];    //Selecting 3rd to 6th bit of temp2
temp3[2:1];    //Selecting 2nd to 1st bit of temp3
temp3[0:2];    //Invalid because order is different w.r.t declaration
```

Data Types

Integer

- A type of register used to store integer values.
- The default size depends upon host-machine but should be at least 32 bits.
- It is used to store signed quantities.

Syntax: integer variable_name; Default Value: X

Examples:

```
integer int1, int2;        // signed int1, int2
integer [3:0] int3;        // Invalid syntax size not
                           // allowed
```

Data Types

Time

- A type of register used to store simulation time.
- The size depends upon implementation but should be at least 64 bits.
- $time is system function which returns current simulation time.

Syntax: time variable_name; Default Value: X

Examples:

```
time a, b;
a=$time;     //to be used inside initial or
             //always statement
```

Data Types

Real

- A type of register used to store real values.
- Values can be stored in decimal or scientific notation.

Syntax: real variable_name; Default Value: 0

Examples:

```
real a, b;
//Assigned inside initial or always block
a=3.14;     //Decimal
b=5e12;     //Scientific (5 x (10^12))
```

Data Types

Arrays

- Arrays can be used to store same type of data together.

Syntax: <data_type> array_name <array_size>;

Declaration:

```
integer num [0:15];            //1-D array
reg [7:0] data [31:0];
wire temp [1:10] [0:5];        //2-D array
reg [3:0] mem [0:7] [0:7];
```

Accessing:

```
num [3];             //integer at location 3
data [12];           // 8-bit data at location 12
data [10] [2];       // 2nd bit of data at location 10
mem [3] [2];         //data at 3rd row 2nd column
```

Data Types

Parameters

- Constants are defined with keyword parameter.
- Parameter value for each module instance can be overridden at compile time.

Examples:
```
parameter port_size = 8;
parameter [3:0] init = 4'b1010;   //sized parameter
```

Local Parameters

- These are parameters whose value cannot be modified

Examples:
```
localparam [1:0] idle=2'b00, state1=2'b01,
                 state2=2'b10, state3=2'b11;
```

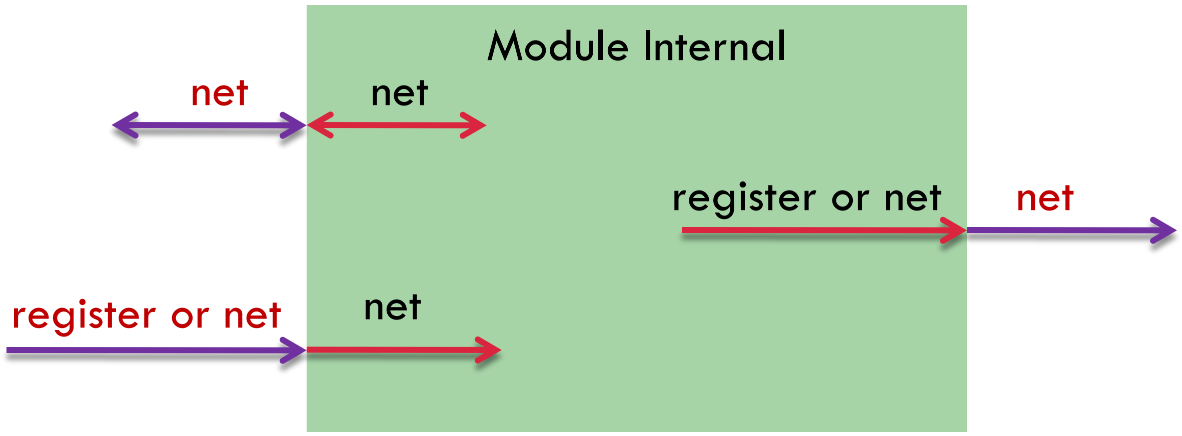
Data Types

Strings

- reg data type is used to store string data.
- Each character requires 8 bits for its storage.
- If reg size is more than size of string, zeros are padded on left of the string.
- If reg size is less, the leftmost bits of string are truncated.

Example:
```
reg [5*8:1] value;
value = "hello";   //Inside initial or always statement
```

Port Data Type



- output port can be of type register or net (Internally) and net(Externally).
- input port must be of type net (Internally) and can be register or net (Externally).
- inout port must be of type net both Internally as well as Externally.

Port Data Type

Syntax: <port_direction> <port_data_type> port_name;

- Port_data_type can be register or net. It is wire by default.

Examples:

```
input a;
output b;                    //wire by default
output reg c;                // output of type reg
output reg [3:0] d;
output integer e;            // output of type integer
inout  f;
input wire g;                // input of type wire
input [31:0] h;              // vector input
input integer i;             // Invalid
```

Continuous Assignment

- Continuous assignment are used to assign values to a net.

  Syntax:

  ```
  assign  target = expression;
  ```

- This assignments are always active, i.e. any change on RHS side leads to assignment of the target.
- An expression can be made up of a net, register or a function call.
- Target has to be of type net.

Examples:

```
input a, b;            //by default all ports
output c;              //are of type net
assign c= a & b;
```

Module Instance

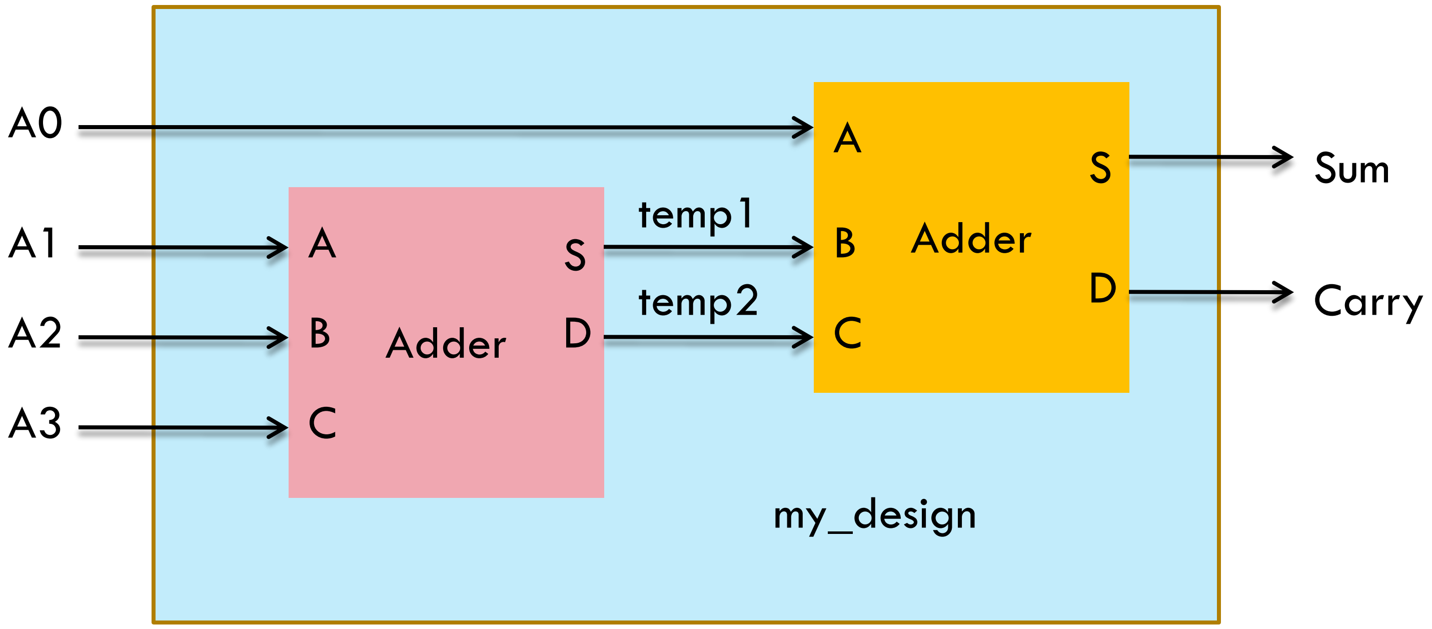
- A module can be used as building block in other modules.
- An instance of a module is used to interact with a given module.
- Each module instance is independent, concurrent copy of a module.

Syntax: module_name instance_name (port_connection);

- Instance name used within a module should be unique.

Module Instance

Example:



Positional Port Connection

```
module my_design (A0, A1, A2, A3, Sum, Carry);
input  A0, A1, A2, A3 ;
output Sum, Carry;
wire temp1, temp2;

 // module adder (A, B, C, S, D);
    adder u0 (A1, A2, A3, temp1, temp2);
// A1→A, A2→B,A3→C, temp1→S, temp2→D
    adder u1 (A0, temp1, temp2, Sum, Carry);
//Order is important
 endmodule
```

Named Port Connection

```
    module my_design (A0, A1, A2, A3, Sum, Carry);
    input  A0, A1, A2, A3 ;
    output Sum, Carry;
    wire temp1, temp2;

     // module adder (A, B, C, S, D);
   adder u0 (.A(A1), .B(A2), .C(A3), .S(temp1), .D(temp2));
           //   .formal ( Actual )
 adder u1  (.A(A0) , .B(temp1), .C(temp2), .S(Sum), .D(Carry));
           // Order is not important
   endmodule
```

Primitives

- Verilog provides basic logic gates in terms of primitives.
- Primitives are instantiated like module.
- Since they are predefined in Verilog, they don’t require module definition.
- It is not compulsory to provide instance name to a primitive.
- There are three class of basic logic gates that verilog provides
    - and/or gates
    - buf/not gates
    - bufif/notif gates

Primitives

- and/or gates
  - They have one scalar output and multiple scalar inputs.
  - The first port is output, other ports are input.
  - The following and/or gates are available in verilog

    and nand or nor xor xnor
- buf/not gates
  - They have one scalar input and one or more scalar outputs.
  - The last port is input, other ports are output.
  - The following buf/not gates are available in verilog.

    buf not

Primitives

- bufif/notif gates

- These are buf/not gates with control input.
- The last port is control input, middle port is input and first port is output.
- The following bufif/notif gates are available in verilog

bufif1 bufif0 notif1 notif0

Examples:

```
and (out, ip1, ip2);
xor x1 (out, ip1, ip2);
or o1(out, ip1, ip2, ip3);
```

Primitives

More Examples:

```
buf (out1, out2, ip);
not n1 (out1, ip);
bufif1 b1(out, ip, ctrl);
notif0 n2 (out, ip , ctrl);
// and primitive for vector ports
and u0 (out[0], ip1[0], ip2[0]);
and u1 (out[1], ip1[1], ip2[1]);
and u2 (out[2], ip1[2], ip2[2]);
and u3 (out[3], ip1[3], ip2[3]);
// Alternative solution
and u [3:0] (out, ip1, ip2);
```

Operators

- Operators in verilog can be classified as following:
- On basis of number of operands
  - Unary(One Operand)
  - Binary(Two Operand)
  - Ternary(Three Operand)
  - Any Number
- On basis of functionality
  - Arithmetic
  - Logical
  - Relational
  - Equality
  - Bitwise
  - Reduction
  - Shift
  - Concatenation
  - Replication
  - Conditional

Arithmetic Operators

- Arithmetic operators are binary operators.
- Verilog provides following arithmetic operations
    - + (Addition)
    - - (Subtraction)
    - * (Multiplication)
    - / (Division)
    - % (Modulus)
    - ** (Exponent)
- If any operand has X or Z value then result of entire expression would be X.

Arithmetic Operators

Examples:

```
A=4'b0010;  B=4'b1101; C=4'b10x1;  // vector reg/wire  type
D=3;        E=5;        F=2;        G=5;   // integer type

A + B;          // evaluates to 4'b1111
A – B;          // evaluates to 4'b0101
B – A;          // evaluates to 4'b1011
A * B;          // evaluates to 8'b00011010
D + E;          // evaluates to 4'b1000
E / D;          // evaluates to 1, i.e. truncates  fractional part
E ** F;         // evaluates to 25
E % G;          // evaluates to 0 result takes sign of 1st operand
E % D;          // evaluates to 2
```

Logical Operators

- Logical operators are binary operators.
- Verilog provides following logical operations
    - ! (Logical not)
    - && (Logical and)
    - || (Logical or)
- It returns single bit 1, 0 or X.
- If operand is non zero, it is equivalent to logic 1.
- If operand is zero, it is equivalent to logic 0.
- If operand is neither zero/non zero, it is equivalent to X. It is normally treated as false condition by simulators.

Relational Operators

- Relational operators are binary operators.
- Verilog provides following equality operations
    - > (greater than)
    - < (less than)
    - >= (greater than equal to)
    - <= (less than equal to)
- It returns logic 1 if expression is true.
- It returns logic 0 if expression is false.
- It returns x if any operand contains X or Z.

Equality Operators

- Equality operators are binary operators.
- Verilog provides following relation operations
    - == (equality)
    - != (inequality)
    - === (case equality)
    - !== (case inequality)
- Logical equality (==, !=) returns 0,1 or X.
- Case equality (===, !==) returns 0 or 1.
- Logical equality returns X if either operands has X or Z data.
- Case equality compares all bits including X and Z.

Examples

```
A=4'b1010;  B=4'b10x0;   C=4'b10x0;   D=4'b1100;
E =4'b0000;  F=4'b1101;

A && D;                                // Returns 1
A || E;                                // Returns 1
(D==4'b1100) && (A==4'b1011);          // Returns 0
F >= A;                                // Returns 1
D > B;                                 // Returns X
B==C;                                  // Returns X
D==E;                                  // Returns 0
A!=F;                                  // Returns 1
B===C;                                 // Returns 1
```

Bitwise Operators

- Verilog provides following bitwise operations
  - ~ (bitwise negation) // unary, others binary
  - & (bitwise and)
  - | (bitwise or)
  - ^ (bitwise xor)
  - ^~ or ~^ (bitwise xnor)
- It performs bit-by-bit operation on two operands.
- If one operand is shorter than other, zeros are padded to match the length of larger operand.
- Return size is equal to that of larger operand.

Reduction Operators

- Reduction operators are unary operators.
- Verilog provides following logical operations
    - & (reduction and)
    - ~& (reduction nand)
    - | (reduction or)
    - ~| (reduction nor)
    - ^ (reduction xor)
    - ^~ or ~^ (reduction xnor)
- Reduction operators performs bitwise operation on a single vector.
- Reduction operators returns 1 bit result.

Shift Operators

- Shift operators are binary operators.
- Verilog provides following shift operations
    - >> (right shift)
    - << (left shift)
    - >>> (arithmetic right shift)
    - <<< (arithmetic left shift)
- Arithmetic shift works with signed data only. Arithmetic right shift pads MSB bits.
- Arithmetic left shift works same as logical left shift.
- Shift by 1'bx or 1'bz result in X.

Examples

```
reg [3:0] A=4'b0101, B=4'b1011, C=4'b1101;
reg signed [5:0] D =6'b101101;

A & B;              // Returns 0001
A | C;              // Returns 1101
& A;                // Returns 0
^  B;               // Returns 1
D >> 2;             // Returns 001011
B << 3;             // Returns 1000
C >>> 2;            // Returns 0011
D >>> 3;            // Returns 111101
B >> -1;            // Returns 0000
A << 1'bx           // Returns XXXX
```

Concatenation Operators

- Concatenation operators can accept any number of operands.
- This operation provides mechanism to append multiple operands.

Syntax: {operand1, operand2, …, operandn};

Examples:

```
A=2'b01;                    // reg or wire data
B=3'b110;
C=2'b10;

{A, C};                     //Result is 4'b0110
{B, A, C};                  // Result is 7'b1100110
{A, 4'b0011, B[2], C};      // Result is 9'b010011110
```

Replication Operators

- Replication operators can accept any number of operands.
- Replication operator are used to replicate an operand specified number of times.

Syntax: { replicate_times {operand} };

Examples:

```
A=2'b01;                    // reg or wire data
B=1'b1;

{3 { A } };                 //Result is 6'b010101
{B , {2{A}}, {2{3'B101}} }; // Result is 11'b1_01_01_101_101
```

Conditional Operators

- Conditional operators are ternary operators.

  Syntax: condition? true_expression : false_expession;

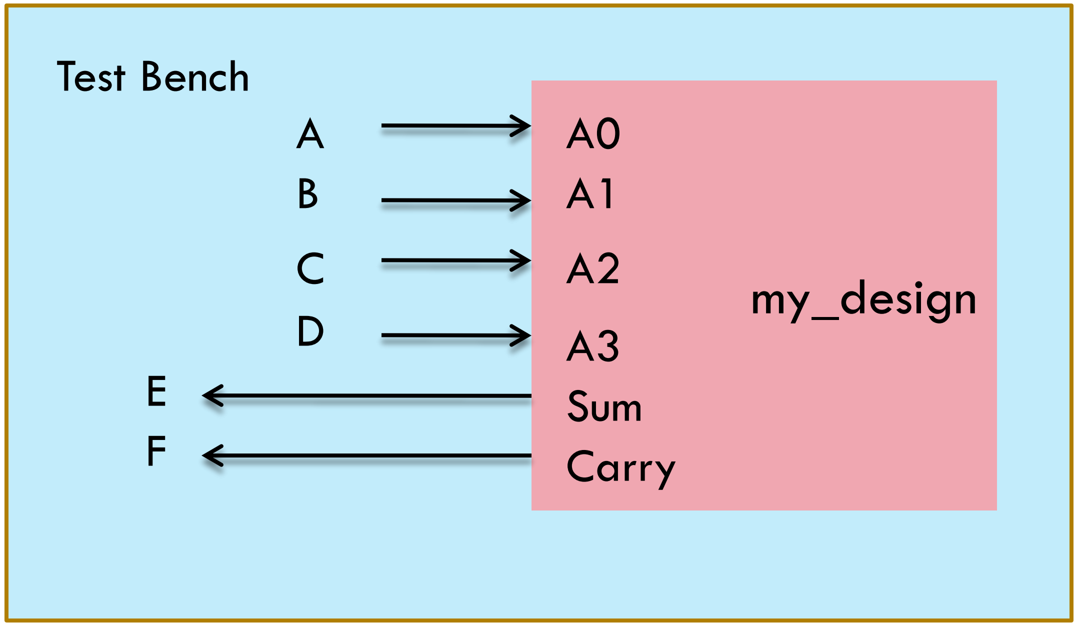
- If condition evaluates to true, then true_expression is returned else false_expression is returned.
- Nesting of conditional operator is also allowed.

  Example:

```
assign y1 = sel ? a : b;
assign y2 = (sel==2) ? in1 : in0 ;
assign y3 = sel1 ?  (sel2? a : b)  :  (sel2? c : d);  //Nested
```

Stimulus (Test Bench)

- Test Benches are used to generate inputs and check response of Design Under Test (DUT).



Test Bench

```
module tb ;
reg  A, B, C, D;
wire E, F;
      my_design  U0  (A, B, C, D, E, F);   //Module Instance
      initial                              // Provide values in initial block
      begin
      A=1; B=0; C=0; D=1;
      #10 A=0;                             // Provide value after 10 time units
      #10 B=1; D=0;
      end
endmodule
```