# System Verilog
# OPERATORS, SUBPROGRAMS

# Operators

- Included from Verilog
  - Arithmetic + - * / % **
  - Logical ! && ||
  - Relational > < >= <=
  - Equality == != === !==
  - Bitwise ~ & | ^ ~^ ^~
  - Reduction & ~& | ~| ^ ~^ ^~
  - Shift >> << >>> <<<
  - Concatenation { op1, op2, op3, .. , opn }
  - Replication { no_of_times { a } }
  - Conditional cond ? True_Stm : False_Stm

# Operators

- Additions to System Verilog
  - Arithmetic += -= *= /= %=
  - Increment/Decrement ++ --
  - Logical -> <->
  - Bitwise &= |= ^=
  - Shift >>= <<= >>>= <<<=
  - Wildcard Equality ==? !=?
  - Set Membership inside
  - Distribution dist

# Example1

```
int a, b, c=2, d=6, e=10;
initial begin
a=d++;
b=++d;
c*=d;
c>>=1;
e%=3;
e+=2;
end
```

Result:

a = 6

b= 8

c= 8

d= 8

e= 3

# Example2

```
int a, b, c, d;
initial begin
b=3;
if((a=b))   //brackets compulsory
$display("a=%d b=%d", a, b);
a=(b=(c=4));
end
```

Result:

a=3 b=3  //Display

c=4

b= 4

a=4

if ((a=b)) is same as

a=b;

if (a)

# Example3

a->b is same as !a || b

a<-> b is same as (!a || b) && (!b || a)

```
int a=1, b=2;
initial begin
if(a->b)
$display("a implies b");
if (a<-> b)
$display("a is logically equivalent to b");
end
```

Result:

a implies b

a is logically equivalent to b

# Example4

```
int i=11, range=0;
bit a=5'b10101;
initial begin
if(i inside { [1:5], [10:15], 17,18})      // i is 1-5 or 10-15 or 17or 18
range+=1;
if(a==?5'b1XZ01)                           //X and Z acts like don't care
range+=1;
end
```

Result:

range=2

# Loops

- Included from Verilog
  - for
  - repeat
  - while
- Additions to System Verilog
  - foreach
  - do while

# Loops

```
inital begin
int a [8] [5];
foreach ( a [i, j] )
a[i] [j]=$random;
end
```

Used to access all elements in an array

```
inital begin
int i=10;
do begin
i -=1;    //statements
end
while (i >5)
end
```

Statements executed first an then execution depends upon condition

# package

- Packages provide ways to have common code to be shared across multiple modules.
- A package can contain any of the following:
  - Data Types
  - Subprograms (Tasks/Functions)
  - Sequence
  - property
- Elements of a package can be accessed by:
  - :: (Scope Resolution Operator)
  - import keyword

# `include vs import

- `include is used to include the content of specified file to the given location.

- It is equivalent to copying the content and pasting at the given location.

```
`include “xyz.v”
```

- Import is used to access elements defined inside the package without copying them to current location.

```
import :: element_name;
import :: *;
```

# Example

“file1.sv”

```
function int add (input int a, b );
add= a + b;
endfunction
```

“file2.sv”

```
function int add (input int a, b, c );
add= a + b + c;
endfunction
```

```
`include “file1.sv”
`include “file2.sv”
 module test;
 initial begin
 int x=add(1, 2, 3);
 int y=add(3, 4);
 end
 endmodule
```

Compilation error add already exists

# Example

“pack1.sv”

```
package mypack1;
int x;
function int add (input int a, b );
add= a + b;
endfunction

endpackage
```

“pack2.sv”

```
package mypack2;
int y;
function int add (input int a, b, c );
add= a + b + c;
endfunction

endpackage
```

# Example

```
`include “pack1.sv”
`include “pack2.sv”
 module test;
 import mypack1::*;
 import mypack2::*;
 initial begin
 x=mypack1 :: add(3, 6);          //x=9
 y=mypack2 :: add(4, 5, 3);       //y=12
 end
 endmodule
```

# unique and priority

- Improperly coded case statements can frequently cause unintended synthesis optimizations or unintended latches.
- System Verilog unique and priority keywords are designed to address improperly coded case and if statements.
- unique and priority keywords can be placed before an if, case, casez, casex statement.

# unique

- A unique keyword performs following checks:
  - Each choice of statement is unique or mutually exclusive.
  - All the possible choices are covered.

- A unique keyword causes simulator to perform run time checks and report warning if any of the following conditions are true:
  - More than one case item matches the case expression.
  - No case item matches the case expression, and there is no default case

# unique

```
always @ *
unique case (sel)
2’b00: y=a;
2’b01: y=b;
2’b01: y=c;
2’b10: y=d;
2’b11: y=e;
endcase
```

Result:

| Inputs | Outputs |
| --- | --- |
| 00 : | y=a; |
| 01 : | y=b; warning issued |
| x1 : | Latch; warning issued |
| 11 : | y=e; |

# unique

```
always @*
casez (ip)
4’b1??? : y=2’b11;
4’b?1?? : y=2’b10;
4’b??1? : y=2’b01;
4’b???1 : y=2’b00;
default: y=2’b00;
endcase
```

Synthesis Result:
Priority Encoder

# unique

```
always @*
unique casez (ip)
4’b1??? : y=2’b11;
4’b?1?? : y=2’b10;
4’b??1? : y=2’b01;
4’b???1 : y=2’b00;
default: y=2’b00;
endcase
```

Synthesis Result:
Encoder

# priority

- A priority instruct tools that choices should be evaluated in order they occur.

- A priority case will cause simulation to report a warning if all possible choices are not covered and there is no default statement.

- A priority if will cause simulators to report a warning if all of the if…if else conditions are false, and there is no final else branch.

# priority

```
always @ *
priority case (sel)
2'b00: y=a;
2'b01: y=b;
2'b01: y=c;
2'b10: y=d;
2'b11: y=e;
endcase
```

Result:

| Inputs | Outputs |
|---|---|
| 00 : | y=a; |
| 01 : | y=b; |
| x1 : | Latch; warning issued |
| 11 : | y=e; |

# priority

```
always @ *
priority if (sel==2'b00) y=a;
else if (sel==2'b01) y=b;
else if (sel==2'b10) y=c;
else if (sel==2'b10) y=d;
else if (sel==2'b11) y=e;
```

Result:

| Inputs | Outputs |
|---|---|
| 00 : | y=a; |
| 01 : | y=b; |
| 10 : | y=c; |
| 11 : | y=e; |
| 1x : | Latch; warning issued |
| z1 : | Latch; warning issued |

# Procedural Statements

- If there is label on begin/fork then you can put same label on the matching end/join.
- User can also put label on other System Verilog end statements such as endmodule, endfunction, endtask, endpackage etc.

```
module test;
initial
for (int i=0; i<15; i++)
begin : loop
……..
end : loop
endmodule : test
```

# Scope and Lifetime

- System Verilog adds the concept of global scope. Any declaration and definitions which are declared outside module, interface, subprograms etc has a global scope.
- These declaration and definitions can be accessed by any scope that lies below the current scope including the current scope.
- All global variables have static lifetime i.e. they exist till end of simulation. Global members can be explicitly referred by $unit.

# Example

```
int i;
//task increment
module test;
//task decrement
initial begin : label
i=5;
#6 $unit::i=3;
#3 increment;
#4 decrement;
end : label
endmodule : test
```

```
task
   increment;
i+= 1;
endtask

task decrement;
$unit::i-=1;
endtask
```

# Scope and Lifetime

- Local declarations and definitions are accessible at scope where they are defined or scopes below it.
- By default all the variables are static in a local scope.
- These variables can be made automatic.
- Static variables can be accessed by hierarchal names.

# Scope and Lifetime

- automatic variables cannot be accessed by hierarchical name.
- automatic variables declared in an task, function, or block are local in scope, default to the lifetime of the call or block, and are initialized on each entry to the call or block.
- Static variables are initialized only once during the simulation period at the time they are declared.
- Advantage of defining variables local to scope is that there is no side effect i.e the variable is getting modified by operations local to it.

# Example1

```
int i;                          //Global Declaration
module test;
int i;                          //Local to module
initial begin
int i;                          //Local to initial block
for (int i=0; i<5; i++)          //Local to for loop
test.i=i;                        //Modifies i inside test
i=6;                            //Modifies i inside initial
end
endmodule : test
```

# Example2

```
int svar1 = 1;                          // static keyword optional
initial begin
for (int i=0; i<3; i++) begin : l1
    automatic int loop3 = 0;            // executes every loop
    for (int k=0; k<3; k++) begin : l2
    loop3++;
    $display(loop3);
    end : l2                            //loop3 destroyed here
end : l1
end
```

Result: 1 2 3 1 2 3 1 2 3

# Example3

```
initial begin
for (int i=0; i<3; i++) begin : l1
    static int loop3 = 0;                    // executes once
    for (int k=0; k<3; k++) begin : l2
    loop3++;
    $display(loop3);                         //loop3 stays till end of
    end : l2                                 //simulation
end : l1
end
```

Result: 1 2 3 4 5 6 7 8 9

# Type Parameter

- A parameter constant can also specify a data type.
- This allows modules, interfaces, or programs to have ports and data objects whose type can be set for each instance.

```
module test #( parameter p1=1, parameter type p2= logic)
                    ( input p2 [p1:0] in, output p2 [p1:0] op);
p2  [p1:0] x;
endmodule

test u0 (15, bit);    //Module instance example
```

# Subprograms

- Following advancement has been done to System Verilog Subprograms (Functions and Task) :

  - Default Port Direction : default port is input, unless specified. Following types of ports are allowed:
    - input : value captured during subprogram call.
    - output: value assigned at end of subprogram.
    - inout : value captured at start assigned at the end.
    - ref : a reference of actual object is passed, the object can be modified by subprogram and can also respond to changes.

# Subprograms

- Following advancement has been done to System Verilog Subprograms (Functions and Task) :

  - Default Data Type : Unless declared, data types of ports is logic type.
  - Default Value : Input ports can have default values. If few arguments are not passed, there default values are taken.
  - begin..end : begin end is no longer required.
  - Return : return keyword can be used to return value in case of functions and to exit subprogram in case of tasks.
  - Life Time : Variables can be defined as static or automatic.

# Function and Tasks

- Both Functions and Tasks can have zero or more arguments of type input, output, inout or ref.
- Only Functions can return a value, tasks cannot return a value.
- A void return type can be specified for a function that is not suppose to return any value.
- Functions executes in zero simulation time, where as tasks may execute in non zero simulation time.

# Example1

```
function int add (int a=0, b=0, c=0);
return a + b+ c;
endfunction

initial begin
int y;
y=add(3, 5);                    //3+5+0
#3 y=add();                     //0+0+0
#3 y=add(1, 2, 3);              //1+2+3
#3 y=add(, 2, 1);               //0+2+1
end
```

# Example2

```
function void display (int a=0, b=0);    //void function
$display("a is %0d b=%0d", a, b);
endfunction

initial begin
display(3, 5);                  //a=3  b=5
#3 display();                   // a=0  b=0
#3 display(1);                  // a=1  b=0
#3 display( , 3);               // a=0  b=3
end
```

# Example3

```
function int initialize(ref int a [7:0]);
foreach( a[ i ] )
a[ i ]=$random;
return 1;
endfunction

int b[7:0], status;
initial begin
status=initialize(b);          //same as pointer concept in c
#3 void'(initialize(b));       // ignore return value
end
```

# Example4

```
//If argument is const then subprogram cannot modify it
function void copy(const ref int a [7:0], ref b [7:0]);
foreach( a[ i ] )
b[ i ]=a[ i ];
endfunction

int a[7:0], b [7:0];
initial begin
foreach (a [i] ) a [ i ]=$random;
copy(a, b);
end
```

# Example5

```
task check (int a, output b);
if (!a) begin
b=1;
$display(“error”);
return; end
b=0;
endtask

initial begin
#3 check(5, error);              // error=0
#3 check(0, error);              // error=1
end
```

# Example6

```
task add (int a=0, b=0, output int z);  //Variables are static by
                                        //default
#2 z=a + b;
endtask

int x, y;

initial fork
add(3, 5, x);
#1 add(2, 4, y);
join
```

Result:

x=6

y=6

# Example6

```
task add (int a=0, b=0, output int z);  //Variables are static by
                                        //default
#2 z=a + b;
endtask

int x, y;

initial begin
add(3, 5, x);
#1 add(2, 4, y);
end
```

Result:

x=8

y=6

# Example7

```
task automatic add (int a=0, b=0, output  int z);
#2 z=a + b;
endtask

int x, y;

initial fork
add(3, 5 , x);
#1 add(2, 4 , y);
join
```

Result:

x=8

y=6