

# SYSTEM VERILOG

## (For Design)

### Chapter 1: General overview

variables can be declared almost anywhere, and the operator “++” is supported:

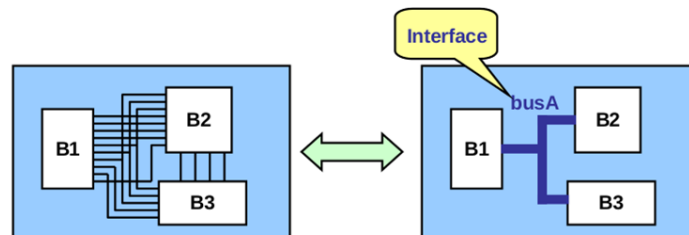
```
initial
begin : myBlock
...
for (int i=0; i<12; ++i)
    intArray[i] += i*5;
...
end : myBlock
```

```
always_comb
unique case (sel)
    2'b00: mux_out = in_a;
    2'b01: mux_out = in_b;
    2'b10: mux_out = in_c;
    default: mux_out = 8'hx;
endcase
```

In this example is shown how to use enum and typedef:

```
module test;
    enum logic [1:0] {TRUE = 1, FALSE = 0, NOTSURE = X } answers;
    int error_count;          // 2-state int type init to 0 at time 0
    typedef enum logic[2:0] {IDLE, FIRST, SECOND, FINISH} MyStates;
    MyStates state, nstate;    // uses user-defined MyStates type
    ...
endmodule : test
```

- **Interface:** is a new kind of design block that captures inter-module communications in one place. Rather than declare many ports and signals at a hierarchy level and connect them, you can declare the signals in one interface and declare ports of an interface type, which when connected to the interface, automatically make all those individual signals connections for you.



Here is an example of assertions in system Verilog:

```
// immediate assertion
always_comb
    if (!sel) mux_out = in0;
    else if (sel) mux_out = in1;
    else
        assert ('0)
            else $error("Bad mux select");
```

```
// concurrent assertion
property abcd;
    @(posedge clk)
        a ##1 b |>= c ##1 d;
endproperty

A1: assert property (abcd);
```

Note the use of the unsized and unbased literal ('0) in the assertion expression. As this expression value is always false, the assertion always fails, causing System Verilog to execute the else branch of the associated action, which calls the \$error system task. The \$error system task issues an assertion failure error message, and then displays any additional user arguments.

- **Coverage:** in system Verilog there are two types of functional coverage: data-oriented and control-oriented, here there is an example of each of them:

```
// SV coverage group:
covergroup cg1 @(posedge clk);
  Addr: coverpoint addr
  { bins low = { [0:'h0F], 19 };
    bins mid[] = { 16, 17, 18 };
    bins high = { ['h14:'hFF] }; }
  AddrXvalid : cross Addr, valid;
endgroup : cg1
```

```
// SVA functional coverage
property full2empty;
  @(posedge clk)
    fifo_full ##[1:$] fifo_empty;
endproperty
C1 : cover property (full2empty);
```

## chapter 2: Standard data types and literals

Topics:

- 1) A new data type logic
- 2) Relaxed data type rules
- 3) 2-state value types
- 4) UN-sized literals
- 5) Time literals
- 6) Module-specific time units and time precision

1) logic is reg replacement, since reg is somewhat confusing and it can be synthesized as to net or reg depending on use.

The formal keyword is: “var logic”. However, var can be omitted.

2) System Verilog relaxes the rules for variables. You can assign a System Verilog variable:

- > In any number of initial or always blocks
  - > As in Verilog currently
  - > From a single continuous assignment
  - > From a single module out or inout port
  - > From a single primitive output
  - Thus you can declare most design signals to be variable
  - >As the var keyword is optional ...
  - >you can simply declare most signals type logic
- Example: logic myVariable;

Only **net (wire logic or logic)** types can have multiple drivers, with other types, you cannot put the same variable in two output ports. With this System Verilog resolves multiple drivers. However, Verilog allows assignments to a variable from multiple procedures. The solution to this problem is the use of special blocks (always\_comb, always\_ff, always\_latch), which allow only one single assignment to a variable within the procedures.

3) To enhance simulation performance at the higher abstraction levels, system Verilog adds 2-state data types considering only ‘1’ and ‘0’, discarding ‘x’ and ‘z’. It’s default value is 0, so it hides the failure of variable initialization, **Be careful!** They can be perfectly replaced with logic, reg and integer. Those new types are:

bit	single bit, scalable to vector	default <b>unsigned</b>
byte	8-bit integer or ASCII character	default signed
shortint	16-bit integer	default signed
int	32-bit integer	default signed
longint	64-bit integer	default signed

4-state to 2-state  
→

0	0
1	1
z	0
x	0

4) In system Verilog the value is truncated to fit the target so you just need to specify the value: '0, '1, 'x and 'z.

5) System Verilog allows units when specifying time literals (fs, ps, ns, us, ms, step), when you say "1step" it refers to one time the time precision unit. Remember there should be no space between the number and the precision unit. For example, if you declare the directive: 'timescale 1ns / 100ps at the top of your code, then if you specify later in the code: 1step, it will refer to 100ps.

**6) Time unit and time precision declarations are equivalent to timescale directive, but without the file-order dependency of compiler directives. An example of this is:**

```
module testbench;
timeunit 1ns;
timeprecision 100ps;
```

These declarations must be written right after the interface module.

### Chapter 3: Procedures and procedural statements

Topics:

- 1) **Block names: can also appear at block end**
- 2) **Local variables: block name unnecessary**
- 3) **Loop enhancements: do...while, foreach**
- 4) **Branch statements: priority and unique case and if**
- 5) **Event control qualifier: iff**
- 6) **Synthesis blocks: always\_comb, always\_latch, always\_ff**

1) Every block can be named, placing the name after the reserved keyword "begin", is optional to place the name after the end statement, with block we refer to: modules, tasks, procedures, etc.

2) In system Verilog, block named and unnamed can have variable declaration. These variables are visible to nested blocks, but hierarchical reference to these variables is not possible.

3) the enhancements in system Verilog are:

- You can declare variables within for statement (for(int i, i<maximum, i++))
- You can declare multiple initial and step assignments (i,j,k,etc)

the loop: do...while (); is supported in system Verilog.

Use of foreach loop. It iterates over the elements of an array. The variables specified there does not have to be previously declared and are only seen from inside the loop. It uses multiple variables for multi-dimensional arrays.

```
int intarr [7:0];
...
foreach (intarr [i])
    intarr[i] = 7-i;
```

equivalent

```
for (int i = 7; i>=0; i=i-1)
    intarr[i] = 7-i;
```

Break (jumps to the end of the loop) and continue (jumps into the next iteration) loop statements are supported.

4) Normally when a case statement is used, it is inferred a priority structure. However, to avoid this declaration: (\* synthesis, parallel\_case \*).

The modifier "**priority**" that could precede the case keyword, defines that the branches are complete and prioritize, it will trigger a run-time warning if no branch found for a case value.

It also can be used with casex and casez. The overlapping is allowed.

The modifier “**unique**” defines branches as complete and mutually exclusive. Only one branch can be executed. No overlapping is allowed and it can be applied to casex and casez. It is equivalent to parallel\_case and full\_case attributes.

These modifiers can be applied to if statements having the same effects. If necessary, to complete an if statement, append an unqualified last else and null statement (else ;).

**Indiscriminate use of priority and unique will cause problems!**

- 5) The iff keyword qualify the procedural event control, the event expression only triggers if the condition is true. It has precedence over the “or” expression and parenthesis can be added to for clarity. The following example will clarify this:

hardware equivalent to:

```
always @(a iff enable == 1)
    y <= a;
```

```
always @(a)
    if (enable)
        y <= a;
```

- 6) System Verilog adds implementation-specific procedural blocks (always\_comb, always\_latch, always\_ff). These blocks reduce design ambiguity by clearly indicating the hardware intent for a procedural block.

Always\_comb: Implied, complete sensitivity list. Variables assigned in an always\_comb cannot be assigned by another procedure. Procedure first executes at time 0 without waiting for trigger (After all initial and always blocks have executed).

Always\_latch: similar features to always\_comb, but is used is limited to special cases when its used is essentially required.

Always\_ff: Variables assigned in always\_ff cannot be assigned by another procedure. Contains one and only one event control. Cannot contain any block timing.

## Chapter 4: Operators

### Topics

- 1) C-like assignment operators (always blocking)
  - 2) C-like pre-/post- increment/decrement operators
  - 3) C-like assignment patterns (plus enhancements!)
  - 4) Wild equality/inequality operators
  - 5) Set membership (inside) operator
  - 6) Operator precedence and associativity
- 1) In this table the assignment operators’ syntaxes are described: (Use them only where you would use blocking assignments)

Symbol	Usage	Meaning	Description
+=	a += b	a = a + b	add
-=	a -= b	a = a - b	subtract
*=	a *= b	a = a * b	multiply
/=	a /= b	a = a / b	divide
%=	a %= b	a = a % b	modulus
&=	a &= b	a = a & b	logical and
=	a  = b	a = a   b	logical or
^=	a ^= b	a = a ^ b	logical xor
<<=	a <<= b	a = a << b	left shift logical
>>=	a >>= b	a = a >> b	right shift logical
>>>=	a >>>= b	a = a >>> b	right shift arithmetic
<<<=	a <<<= b	a = a <<< b	left shift arithmetic

- 2) Pre- form (++a, --a) adds or subtracts then uses new value. Post form (a++, a--) uses value then adds or subtracts. (Used as blocking assignments)
- 3) Define a list of values for an assignment to: array elements or structure fields. Can use pattern “keys”: array element index, structure field, name or type, and the “default” tag.

```
int arr1 [3:0];
arr1 = '{0,1,2,3};
```

```
// equivalent to
arr1[3] = 0;
arr1[2] = 1;
arr1[1] = 2;
arr1[0] = 3;
```

```
int arr1 [3:0];
arr1 = '{3:1, default:0};
```

```
// equivalent to
arr1[3] = 1;
arr1[2] = 0;
arr1[1] = 0;
arr1[0] = 0;
```

```
reg [15:0] mem [0:1023] = '{default:0};
```

- 4) The wild equality operator (==?) and wild inequality operator (!=?) treat unknown (X) and high impedance (Z) values in the given bit position of the right operand as “don’t-care” bits to ignore when doing the comparison

```
a = 4'b0101;
b = 4'b01XZ;
```

```
if (a === b) // false
...

if (a ==? b) // true
...

if (a !=? b) // false
...
```

- 5) The keyword “inside” does a comparison. True if expression value is contained within a value list. List values can be variables (including arrays), ranges. List values can be wildcards. List values can overlap. For non-integral expressions, uses the equality operator (==) (result ‘1’ or ‘0’). For integral expressions, uses a wild equality operator (==?) (result ‘1’, ‘0’ or ‘x’).

```
int bar [1:0]
if ( a inside {bar, [0:3]} )
...
```

```
// equivalent to:
if (a inside {bar[1], bar[0], 0, 1, 2, 3})
```

```
if ( a inside {2'b0?} )
...
```

```
// equivalent to
if ( a inside {2'b00, 2'b01, 2'b0X, 2'b0Z} )
```

6)

() [] :: .	left
+ - ! ~ & ~&   ~  ^ ~^ ^~ ++ -- (unary)	right
**	left
* / %	left
+ - (binary)	left
<< >> <<< >>>	left
< <= > >= inside <b>dist</b>	left
== != === !== ==? !=?	left
& (binary)	left
^ ~^ ^~ (binary)	left
(binary)	left
&&	left
	left
?: (conditional operator)	right
->	right
= += -= *= /= %= &= ^=  = <<= >>= <<<= >>>= := :/ <=	none
{ } { }	concatenation

## Chapter 5: User-defined data types and structures

Topics:

- 1) Primitive and user-defined data types
- 2) Defining (typedef) a type
- 3) Enumerated types
- 4) Arrays: unpacked and packed
- 5) Structures: unpacked and packed
- 6) Package and compilation unit scope name spaces

- 1) Primitive data types: built-in types such as: int, bit, logic, real, etc.  
User-defined data types: use of the keyword “typedef”
  - C-like enum for user-defined values sets
  - C-like struct to bundle multiple variables into one object
  - C-like union to store different data types in the same space
- 2) You can use your type name in variable or port declaration. The use this keyword is as follows: (typedef logic [3:0] new\_type;)
- 3) With enumerated types, the values are restricted to the ones declared. This type of variable types allows you to use type-casting:

```
// typed enum type
typedef enum
    {idle, start, done} state_t;
state_t tstate, next_tstate;
```

```

tstate = next_tstate;
don't work → tstate = 2'b00;
              → tstate = 2;
try this → tstate = state_t'(2);
```

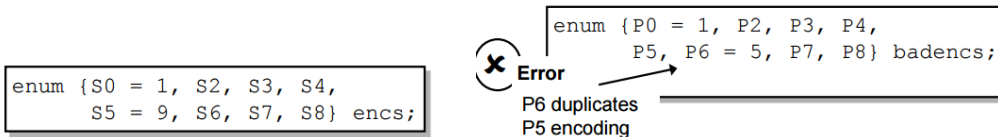
Examples of type-casting:

```
typedef enum {idle, start,
              done} state_t;
state_t tstate;
int aint;
```

```

tstate = start;
doesn't work → aint = tstate + 1; // = 2
              → tstate = tstate + 1;
try this → tstate = state_t'(tstate + 1);
```

Encoding of enumerated types: You can mix explicit and implicit value encoding, Implicit values increment from previous explicit value, Compilation error if encodings overlap.



Explicit enumerated types you can specify the enumeration base type, and the values will be assign by default starting on zero, or you can specify a different encoding, considering the appropriate type:

```
enum bit[2:0] {idle, start,
               pause, done} stbit;

// idle = 3'b000
// start = 3'b001
// pause = 3'b010
// done = 3'b011
```

```
typedef enum bit[2:0]
{idle = 3'b000,
 start = 3'b001,
 pause = 3'b010,
 done = 3'b100} onehot0_t;
onehot0_t state;

state <= onehot0_t'(3'b001);
```

You also can specify some sequences in the value list:

```
typedef enum {go, R[3:5], stop} seqb_t;
```

equivalent to

```
typedef enum {go, R3, R4, R5, stop} seqb_t;
```

For 2-state types, the default value is '0 and for 4-state types the default value is 'x

```
enum logic[2:0] {LOG[4]} var_l;
```

```
initial begin
  $display("value %b", var_l);
  $display("name %s", var_l.name());
end
```

value xxx  
name <invalid>

```
enum logic[2:0] {LOGX='X, LOG0=0 ...} var_l;
```

```
initial begin
  $display("value %b", var_l);
  $display("name %s", var_l.name());
end
```

value xxx  
name LOGX

You can encode the 'x to more easily detect assign failure.

System Verilog methods enable iteration over enumeration values:

Method	Description
first()	Returns first value
last()	Returns last value
next(N)	Returns next Nth value from current
prev(N)	Returns previous Nth value from current
num()	Returns number of values
name()	Returns string equivalent of value

```
typedef enum {
  idle, start, done
} state_t;
state_t st = st.first();

forever begin
  $display ("%s = %d", st.name(), st);
  if (st == st.last()) break
  st = st.next();
end
```

- 4) **Unpacked dimensions:** declared after the name, this can be used with any type even user-defined ones, implementation may store elements of unpacked arrays **non-contiguously**.  
**Packed dimensions:** declared before the name, this can only be used with: logic, reg, and bit. Implementations must store elements of packed arrays **contiguously**.

You can read and write any element or any set of contiguous elements or the arrays as a whole.

```
unpackarr = '{8'h00, 8'hff};
unpackarr = '{default:8'h00};
```

Unpacked multidimensional arrays: you can read and write any slice from element to whole array. You can use pattern assignments.

```

int intc [2:0][0:1] = '{2,1},{5,4},{3,2}';
// intc[2][0] = 2
// intc[2][1] = 1
// intc[1][0] = 5
...
int intd [1:3][2:1] = '{3(1,2)}';
// intd[1][2] = 1
// intd[1][1] = 2
// intd[2][2] = 1
...
int inte [0:15][0:255] = '{default:100};
// inte[0][0] = 100...

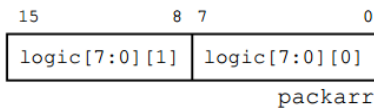
```

Packed multidimensional arrays: can operate on whole array as if it were a one-dimensional vector, use only for: reg, bit, logic. Obey standard rules for vector assignment (padding and truncate).

```

// 2-d packed array
logic [1:0][7:0] packarr;

```



```

packarr = 16'h0;
packarr = packarr << 1;
packarr = ~packarr;

```

You can use the packed array name as an integer in any expression in which an integer may appear. Remember that a packed array is by default unsigned and all multiple bit integer types are by default signed.

## 5) Struct type:

```

typedef struct {
    bit    id, par;
    int    addr;
    logic[7:0] data;
} frame_t;

frame_t frame1, two_frame[1:0];
logic[7:0] data_in;

// individual field assignment
frame1.id <= 1'b1;
data_in <= frame1.data;

```

```

// ordered assignment pattern
frame1 <= '{1'b1, 1'b1, 15, 8'hff};

// named assignment pattern
frame1 <= '{id:0,par:1,addr:0,data:0};

// nested ordered assignment pattern
two_frame = '{0,0,0,255}, '{1,1,1,0};

```

You can combine keywords to make a pattern assignment, but you cannot use index at the same time:

```

// assignment by name, type & default
frame1 = '{data:8'hff,int:42,default:0};

```

Packed structures: all fields must be packable, can operate on whole struct as if one-dimensional bit vector. If any element is 4-state, then the whole struct is stores as 4-state.

```

typedef struct packed {
    logic    id, par;
    int    addr;
    logic[7:0] data;
} frame_t;

```

```

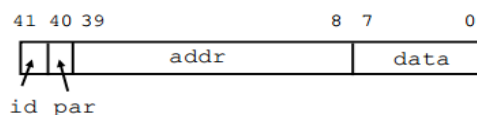
frame_t frame1, frame2;
int addrbuf;

```

```

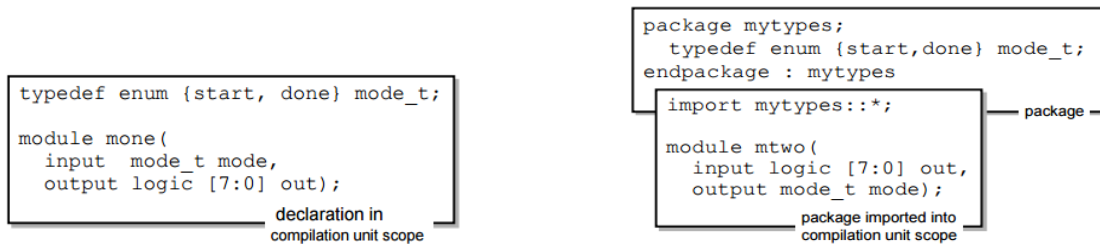
addrbuf = frame1.addr;
addrbuf = frame1[39:8];
frame1 = frame2 >> 2;

```





- 6) You can declare ports of user-defined types, but you must declare the type previously. In this sense, System Verilog have 2 new name-spaces: Compilation unit scope and packages.



## Chapter 6: Hierarchy and connectivity

### Topics

- 1) Implicit .name and .\* port connections
- 2) Compilation unit scope name space and \$unit
- 3) Package name space and importing declarations
- 4) Hierarchy root reference using \$root
- 5) Nested module declarations

- 1) Port connections, can be a problem if you order the signals badly. However, System Verilog gives you 2 simple options to make it easier and safer.

- .name (Example: counter u0 (.S\_in(A), S\_out(B));)
- .\* (Example: counter u0 (.\*);) only valid if the name signals have the same name

If only some names are equal, then you can use a mix using implicit name connection and name connection:

```
count c6 (.data, .clk, .rst(reset),
          .ld(load), .cnt);
```

dot-name and named

use .name only if both names match (variable name and port name). Also there is an easier way to do it, mixing .\* connection with .name connection, just for the variables names that do not match:

```
count c9 (.*, .rst(reset), .ld(load));
```

dot-star and named

Always remember that mixing .name or .\* with ordered connections is prohibited. Besides, .name connections are required for width mismatches, name mismatches or unconnected ports.

- 2) **Compilation unit scope:** Cannot make hierarchical references to declarations within a compilation unit scope. Local declarations over-ride compilation unit scope. Use **\$unit::** to access compilation unit declaration

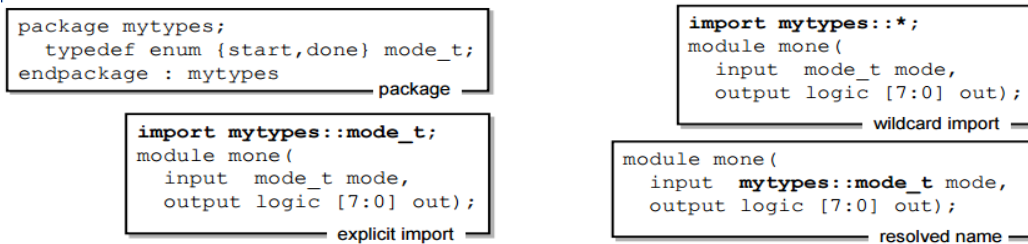
```
typedef enum {start, done} mode_t;
localparam eseg = 7'b11111100;

module mone(input mode_t mode,
            output logic [7:0] out);
...
endmodule : mone
```

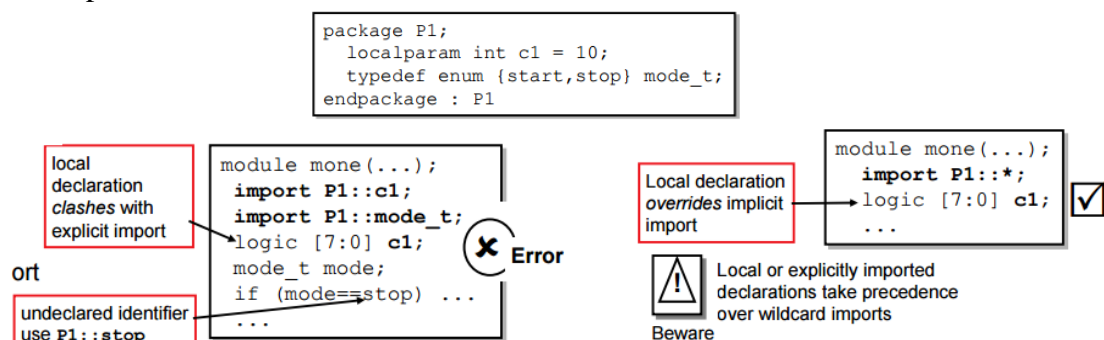
```
module mtwo(input logic [7:0] out,
            output mode_t mode,
            output [6:0] oa, ob);
  localparam eseg = 7'b00111111;
  ...
  assign oa = eseg;           //local
  assign ob = $unit::eseg;    //unit
endmodule : mtwo
```

A compilation unit scope is by default restricted to one file. Compliant compilers must provide a means to extend the compilation unit to all files compiled at the same time.

- 3) **Package:** Contains declarations to be shared between modules. You can declare types, variables, tasks, functions and assertion sequences and properties within a package. You can import the declarations as implicit (\*) or Explicit (name). You can directly access a declaration by using the resolution operator (::).



**Explicit import:** Compilation error if local or other explicitly imported declarations have same name. Local declarations override compilation unit scope declarations without warning, even for an explicit import. Local or explicitly imported declarations take precedence over wildcard imports.

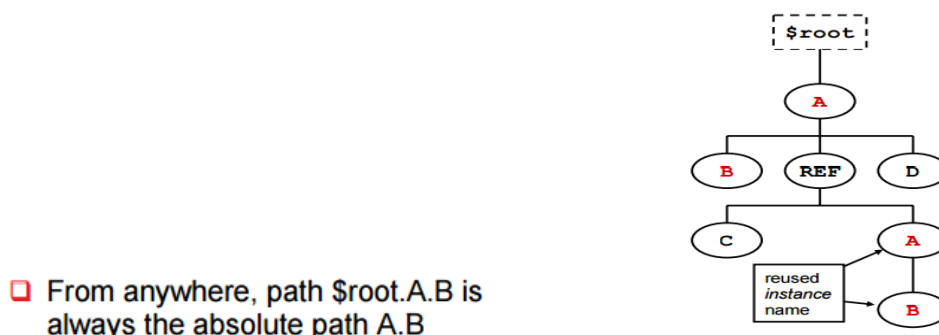


To avoid ambiguity in the references, you can avoid using the same names in different packages or simply use resolved names, specifying the package where the declaration comes from.

**Implicit import:** its declaration makes candidates for import – it is not imported until it is actually used. Until it is used, you can safely redeclare it locally or explicitly import it from another package.

**Import placement:** Explicitly imported declarations can clash with earlier local declarations or imports. They show the declarations onward its placement. **TIPS:** avoid import into compilation unit scope, Tip: place package references at beginning of design unit.

- 4) Reference to top-most design unit in hierarchical paths. \$root can be used to start path references at top-most design unit. hierarchical paths can be used: To read or write remote variables or events and in task or functions calls.



most users name their top level module something obvious like "top" and do not reuse that

instance name lower in the hierarchy. For SystemVerilog, you can use “dollar root” (\$root) to refer to a virtual simulation root, thus unambiguously specifying an absolute path.

- 5) **Module references:** the issues are: Different modules with same name cause naming conflicts and Module access cannot be restricted (all modules from an IP block can be accessed externally).

A nested module definition is visible only in the module where it is declared or in any subhierarchy of the module where it is declared. You can declare modules in the top level of your own design component, and thus restrict their use to just that component. Such local declarations override declarations in the definitions name space. This allows use of different module definitions of the same name. Deeply nesting module definitions can make your code less readable. You may want to judiciously utilize “include” (`include) directives and place each module definition in a separate file. The FSM module in the example may be instantiated before it is declared. This is consistent with such declarations in the definitions name space.

```
module FSM (...);
...
endmodule : FSM

module IP(...);

  D m1 (...);
  FSM m2 (...); // local FSM

  module FSM (...);
  ...
  endmodule : FSM

  `include dmod.v

endmodule : IP
```

```
module D (...);
...
endmodule : D
```

dmod.v

## Chapter 7: Task and Functions

### Topics

- 1) **Functions and Tasks review**
- 2) **Optional begin...end and named ends**
- 3) **Functions can return void**
- 4) **Function output arguments**
- 5) **Return statements**
- 6) **Default parameter values**
- 7) **Passing arguments by name**
- 8) **Pass by value and pass by reference**

- 1) **Function:** one or more inputs, just one output. No timing allowed, function must execute in time zero.

**Task:** any number of inputs, outputs or inout arguments, may contain timing

**Static subprograms:** Only one set of arguments and local variables exists that all concurrent or recursive calls share.

**Automatic subprograms:** create a new copy of all arguments and local variables for each call. As they are non-persistent, you cannot access these automatic subprogram items using hierarchical references.

**To the Verilog standard, SystemVerilog adds that:**


- You can declare a local variable of a static subprogram automatic. This allocates a new non persistent stack variable for each invocation of the subprogram, so each concurrently executing thread has its own copy.
  - You can also declare a local variable of an automatic subprogram static. This allocates one persistent variable shared by all invocations of the subprogram, so all concurrently executing threads use the same copy
- 2) You can omit the begin / end keywords. You can repeat the subprogram name with endtask/endfunction. Example: task write\_mem (); ..... endtask: write\_mem  
You can choose whether or not repeating the name and the end.

### Void function:

```
function void printerr (input int status);
  if (status == 0)
    $display ("No Errors");
  else
    $display ("%0d Errors", status);
endfunction
...
always @(test_done)
  printerr(status);
```

- 3) You call a void function as a statement rather than as an expression. You can “cast away” the value of a function that returns a value, thus calling any function as a statement while avoiding compiler errors. To do this, use the static type cast operator to cast the function return to the void type: **void’ (nonVoidFunctionCall ())**;
- 4) In functions argument default is input of type logic. However, we can define output arguments, the following example shows the syntax:

### ◆ Function becomes a general-purpose synthesizable subroutine

<pre>function [7:0] addcarry (input [7:0] a, b,                         output carry);   {carry, addcarry} = a + b; endfunction logic [7:0] a, b, sum; logic cry; assign sum = addcarry(a, b, cry);</pre>	<div style="display: flex; align-items: center;">  <div> <p>void functions enforce the "no timing in tasks" synthesis rule</p> <pre>// a and b default to inputs function void add ( integer a, b,                   output integer sum);   sum = a + b; endfunction  always @(a or b)   // void function call   add (a, b, sum);</pre> </div> </div>
---	--

- 5) Executing a return statement immediately exits a subprogram. It can be used to terminate a subprogram without sending a value back or sending back a value, in tasks or functions.

You must remember to assign any output or inout arguments before executing the return statement. Output and inout arguments that you do not assign will have either default initialization values, or for static subprograms, values from the previous call, if any.

```
function integer mult (input integer num1, num2);
  if ((num1 == 0) || (num2 == 0)) begin
    $display("Zero multiply");
    return ('hx);
  end
  else
    mult = num1*num2;
endfunction
```

```
task printstatus (input int errors);
  if (errors == 0) begin
    $display("Zero Errors");
    return;
  end
  else begin
    $display("%d Errors", errors);
    case (errors)
      ...
    endcase
  end
endtask
```

- 6) Default values for subprograms (tasks and functions) can be defined. You can omit passing arguments if all arguments have default values

```
// default argument values in task definition
task read (int j = 0, int k, int data = 1);
  ...
endtask

int val = 21;

// invocation of task with default arguments
read ( , 5); // equivalent to (0, 5, 1)
read (2, val); // equivalent to (2, 21, 1)
read ( , 5, 7); // equivalent to (0, 5, 7);
read (2); // ERROR - k not defined
```

✗ Error

```
// default value for every argument
task read (int j = 100, int k = 200, int data = 300);
  ...
endtask

// invocation of task with default arguments
read (); // equivalent to (100, 200, 300)
read ; // also legal in SV
```

- 7) **Binding arguments by name:** You can bind subprogram arguments by name instead of position. Similar to port connection by name. Simplifies calling subprograms having many arguments. Simplifies calling subprograms having default arguments

```
task read (int j = 0, int k, int data = 1);
  ...
endtask

int val = 21;

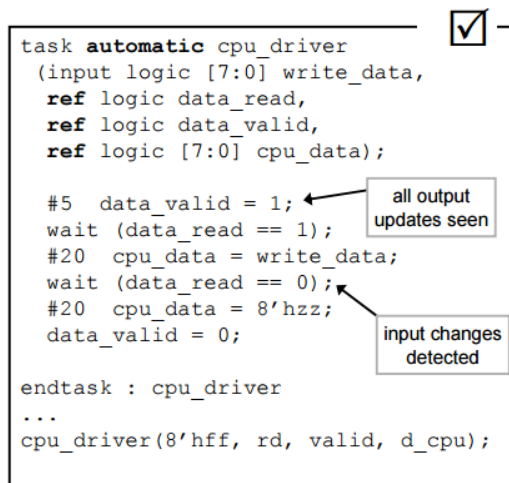
// invocation of task with default arguments and name passing
read (.j(4), .k(val), .data(7)); //
read (.j(2), .k(val)); // equivalent to (2, 21, 1)
read (.k(3)); // equivalent to (0, 3, 1)
read (.data(7), .j(4), .k(3)); //
```

- 8) Verilog supports only the pass-by-value mechanism:

It copies each input into the subprogram area.

- Calls to a static subprogram all share the same local copy.
- Calls to an automatic subprogram each make their own local copy. Subprograms are unaware of argument value changes.

System Verilog supports pass-by-reference (**ref**) of variables. It passes only the values of nets. This passes a handle to the actual argument object, rather than its value. The subprogram directly accesses the variable whose reference is passed. Changes in inputs can be seen in task. Changes in outputs are immediately updated. Not copying large arguments boosts simulation performance. **Can use only with automatic** (not static) subprograms. More efficient when arguments occupy large amounts of memory.



```

task automatic cpu_driver
(input logic [7:0] write_data,
  ref logic data_read,
  ref logic data_valid,
  ref logic [7:0] cpu_data);

  #5 data_valid = 1;
  wait (data_read == 1);
  #20 cpu_data = write_data;
  wait (data_read == 0);
  #20 cpu_data = 8'hzz;
  data_valid = 0;

endtask : cpu_driver
...
cpu_driver(8'hff, rd, valid, d_cpu);

```

With the use of ref keyword, the direction information of the parameter is lost. This may lead to inadvertently updating parameters meant only as inputs. To prevent a parameter from being updated, declare it as a “**const ref**” parameter.

## Chapter 8: Interfaces

### Topics

- 1) What are interfaces?
- 2) Simple interfaces
- 3) Generic interfaces
- 4) Interface ports
- 5) Parameterized interfaces
- 6) Modports
- 7) Tasks/functions in interfaces

- 1) It encapsulates communication between hardware blocks, provides a mechanism for grouping together multiple signals into a single unit that can be passed around the design hierarchy. Enable abstraction in RTL design.

An interface is a new kind of design block that captures inter-module communication in one place. Rather than declare many ports and signals at each hierarchy level and connect them, you can declare the signals in one interface and declare ports of that interface type, which when connected to the interface, automatically make all those individual signal connections for you.

A System Verilog interface is, at its most basic level, simply a bundling of external nets and/or variables normally shared by one or more modules by virtue of port connections. The bundling of these nets and variables into an interface type that you then instantiate, permits the modules each to connect one port to the entire bundle, rather than several ports, one to each net and variable. Interfaces can also contain much more complex features.

- 2) With a simple interface, you declare all the signals for a specific bus once as an interface type. You use the interface type as the port type in a module port list. As the modules can both read and write the interface contents, the port has no direction. You instantiate the interface at the same level you instantiate the modules to be connected, and map the module instance ports to the interface instance.

## Example Without Using Interfaces

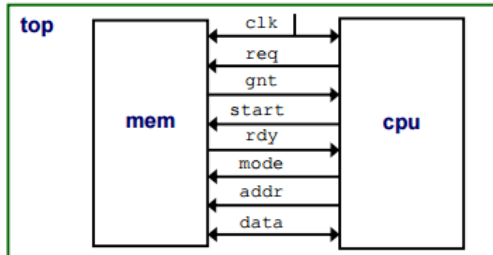
```
module memMod (
    input  logic clk, req, start,
           logic [1:0] mode,
           logic [7:0] addr,
    inout  wire [7:0] data,
    output logic gnt, rdy
);
...
endmodule
```

```
module cpuMod (
    input  logic clk, gnt, rdy,
    inout  wire [7:0] data,
    output logic req, start,
           logic [7:0] addr,
           logic [1:0] mode
);
...
endmodule
```

```
module top;
    logic req, gnt, start, rdy;
    logic clk = 0;
    logic [1:0] mode;
    logic [7:0] addr, data;

    memMod mem (clk, req, start,
               mode, addr, data, gnt, rdy);

    cpuMod cpu (clk, gnt, rdy, data,
               req, start, addr, mode);
    ...
endmodule
```



## Same Example Using Interfaces

```
interface simple_bus;
    logic req, start, gnt, rdy;
    logic [1:0] mode;
    logic [7:0] addr;
    wire [7:0] data;
    ...
endinterface : simple_bus
```

① interface declared

```
module top;
    logic clk = 0;

    simple_bus busA();

    memMod mem (clk, busA);
    cpuMod cpu (clk, busA);
    ...
endmodule
```

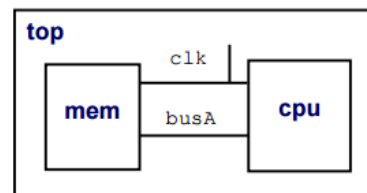
③ interface instantiated

```
module memMod (
    input bit clk,
    simple_bus bus
);
...
endmodule
```

② interface used as a directionless port type in module declarations

```
module cpuMod (
    input bit clk,
    simple_bus bus
);
...
endmodule
```

④ instantiation mapped to module ports



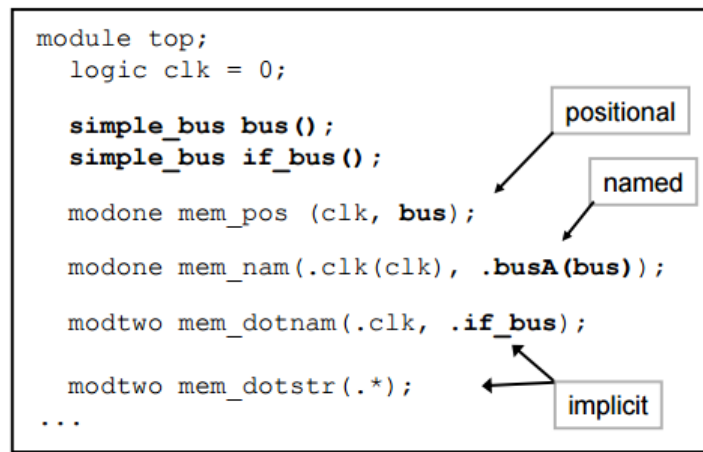
**Interface module instantiation:**

```
module modone (
    input bit clk,
    simple_bus busA);
    ...
endmodule
```

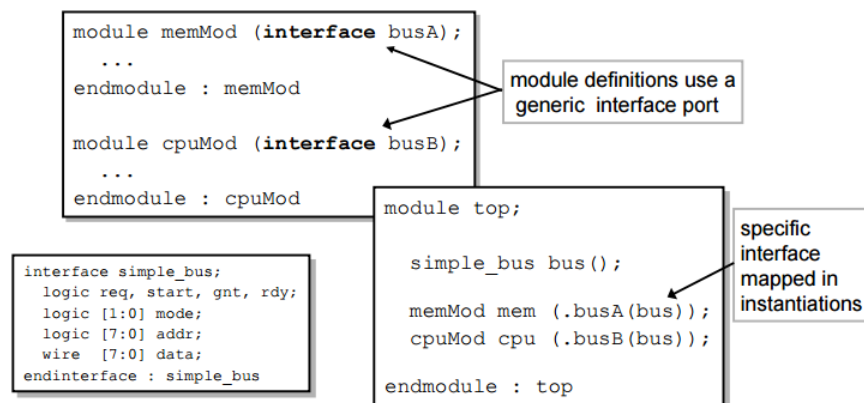
```
module modtwo (
    input bit clk,
    simple_bus if_bus);
    ...
endmodule
```

The port map syntax for ports of an interface type is identical to that for ports of any other type. You can use ordered connections, named connections, “dot name” (.portname) connections, and “dot star” (.\* ) connections.

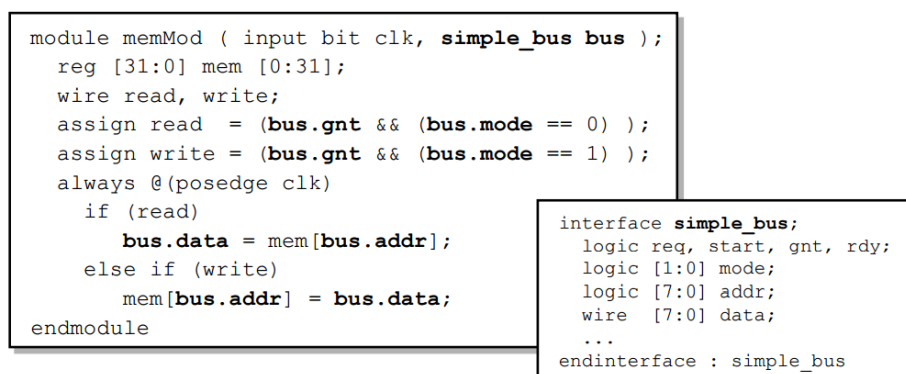




- 3) You can declare a module port of a generic interface type. This defers actual interface selection until module instantiation. You can use it only with the Verilog named port connections.



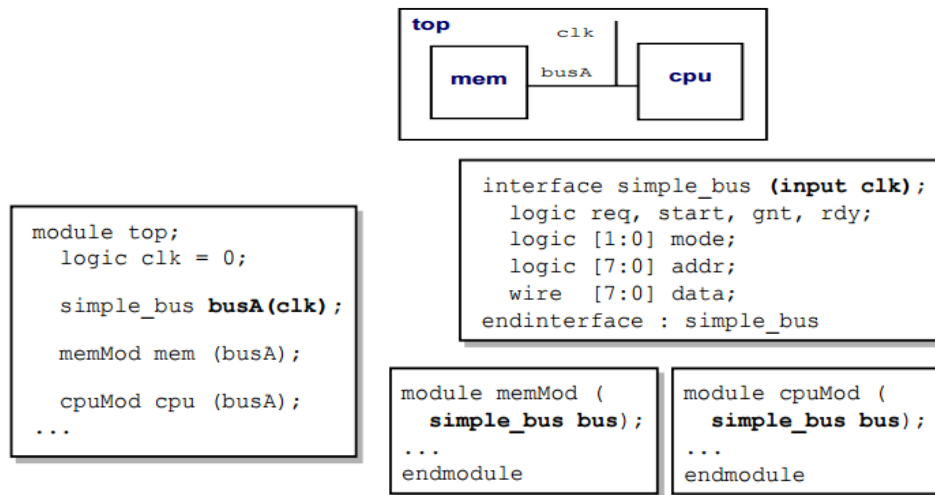
This module has a port of the `<simple_bus>`. `<interface type>`. An example of this:



Inside the top, to reference the items inside the interface instances made for submodules, the syntax is: `<instance_name>.<interface_item>`.

- 4) This example adds the “clk” port to the “simple\_bus” interface definition. The “mem” module and “cpu” module no longer require the “clk” port, as it is now part of the interface. The parent module maps its own “clk” signal to the “clk” port of the interface when it instantiates the interface. The modules can access this interface signal the same way they access any other interface signal





Common interface port applications: Common signals for instances of different interface definitions, Different signals for different instances of one interface definition.

- 5) You can parameterize an interface just like a module.

```

interface fastbus #(DBUS = 32, ABUS = 8) (input clk);
    wire [DBUS-1:0] data;
    logic [ABUS-1:0] address;
    ...
endinterface

interface slowbus;
    parameter WIDTH = 16;
    wire [WIDTH-1:0] a, b;
    ...
endinterface

module test;
    fastbus #(8, 5) bus8x5(clk); // 8-bit DBUS and 5-bit ABUS
    fastbus #(8) bus8x8(clk);    // 8-bit DBUS and 8-bit ABUS
    slowbus #(.WIDTH(8)) bus2(); // 8-bit a, b variables
    .....
endmodule

```

- 6) Modports create different views of an interface: Specify a subset of interface signals accessible to a module and specify direction information for those signals. You can specify a modport view for a specific module in two ways: In the module definition or in the module instantiation.

#### Module definition

```

interface mod_if;
    logic a, b, c, d;
    modport master (input a,b, output c,d);
    modport slave (output a,b, input c,d);
endinterface

```

```

module mmod (mod_if.master mbus);
endmodule

module smod (mod_if.slave sbus);
endmodule

```

```

module testbench;
    mod_if busA();
    mmod mmod1 (.mbus(busA));
    smod smod1 (.sbus(busA));
endmodule

```

The parent module instantiates the interface and maps it to the ports of the submodule instances. This modport selection method works well when all instances of a submodule definition use an interface in the same way.

## Module instantiation

```
module mmod (mod_if mbus);
endmodule

module smod (mod_if sbus);
endmodule
```

```
module testbench;
  mod_if busB();
  mmod mmod1 (.mbus(busB.master));
  smod smod1 (.sbus(busB.slave));
endmodule
```

The definition of the interface stays the same. In this example, the parent module itself specifies which view of the interface a submodule instance will have. This modport selection method works well when different instances of a submodule definition use an interface in different ways.

**Complex interfaces:** You can instantiate interfaces individually, in arrays and as a generate item. **Constructs allowed:** < Variable and net declarations < Continuous assignments < always, initial and final blocks < assertions < task and function definitions < Declarations and instances of interfaces and programs. **Constructs not allowed:** < Declarations or instances of modules or primitives < specify blocks, specparams, defparam.

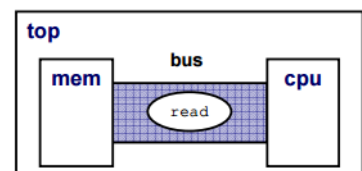
- 7) **Interface tasks:** Writing/reading interface signals can be handled by tasks. However, every module must contain copies of the tasks (Maintainability issues). Solution is to declare tasks once as part of the interface (methods).

**Interface methods:** A task defined within an interface is called an interface method. Same syntax and statements as in a module task definition. Declare once – visible to any module connected to interface. Call methods through interface port or instance.

```
module cpuMod (simple_bus bus);
...
  bus.read(8'hFF);
...
endmodule
```

```
interface simple_bus (input clk);
  logic req, start, gnt, rdy;
  logic [1:0] mode;
  logic [7:0] addr;
  wire [7:0] data;

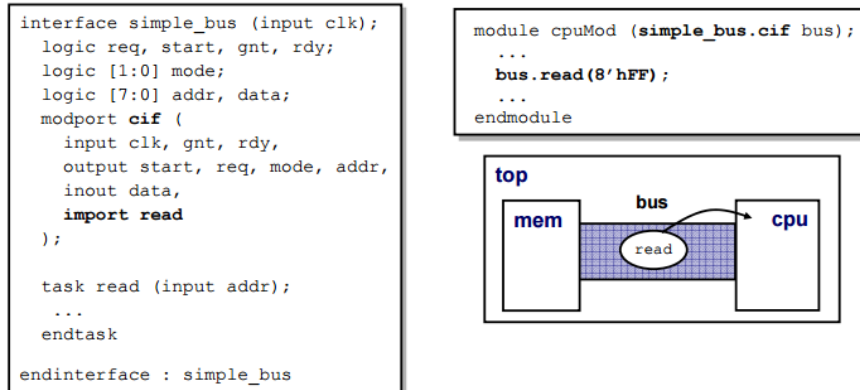
  task read (input address);
    @(posedge clk);
    addr = address;
    ...
  endtask
endinterface : simple_bus
```



You can place subprogram definitions within the interface definition itself. Connected modules call the subprograms using the same sort of hierarchical references they use to access other interface items. Conversely, interfaces may use subprograms declared in modules. An interface that uses an external subprogram must first make a forward declaration of the subprogram, that is, a subprogram prototype prefixed with the extern keyword.

If connected to an interface modport, a module sees only: ports defined in that

modport. Methods imported from that modport.



Within a modport, you can import a subprogram from the interface to make it available to the connected module, and export a subprogram from a connected module to make it available to the interface. Remember that all modport references are from the perspective of the connected module. You do not need to make a forward declaration (using `extern`) of an external subprogram that is exported through a modport. You do not typically need to fully prototype subprograms imported or exported through modports, providing just the identifier is usually sufficient. The only situation where you need to fully prototype an external subprogram is in the modport import statement that imports into a module a subprogram that another module through a modport exports to the interface.

### Some Interesting annotations from Verilog 2001:

- You can disable default ``default_nettype none` < Compiler will instead issue error upon attempted use of undeclared identifier < Prevents inadvertent typographical errors from causing connectivity problems < `none` is not a reserved word.

- **Ways to overwrite parameters:**

Verilog2001

```
defparam u2.width_b = 7;

us_mult #(5,7) u1 (.a (a_net),
    .b (b_net), .product (a_b_mult));

us_mult #(.width_a(5)) u1 (.a (a_net),
    .b (b_net), .product (a_b_mult));
```

- Assignments between signed reg and integer types maintain sign information
- **Signed support:** Arithmetic shift operators maintain sign information `Right >>>` `Left <<<` < Literal values can be qualified as signed with `s` `%04'sb1001`. New system functions cast a value as signed or unsigned `signed()` `unsigned()`.
- **Variable system tasks / Random number system tasks**

- **type name functions : \$typename** The \$typename system function returns a string that represents the resolved type of its argument.
- **expression size system functions : \$size** The \$bits system function returns the number of bits required to hold an expression as a bit stream.
- **range system functions : \$isunbounded** The \$isunbounded system function returns true if the argument is \$.
- **\$urandom\_range** : The \$urandom\_range() function returns an unsigned integer within a specified range.
- **\$urandom** : The system function \$urandom provides a mechanism for generating pseudo-random numbers. The function returns a new 32-bit random number each time it is called. The number shall be unsigned.
- **\$srandom** : The srandom() method allows manually seeding the RNG of objects or threads.
- **\$set\_randstate** :The set\_randstate() method sets the state of an object's RNG.
- **\$get\_randstate** : The get\_randstate() method retrieves the current state an object's RNG.

## - Array system tasks / Assertions system tasks

- **\$dimensions** : Returns' the total number of dimensions in the array (packed and unpacked, static or dynamic) 1 for the string data type or any other nonarray type that is equivalent to a simple bit vector, 0 for any other type total number of unpacked dimensions for an array (static or dynamic) 0 for any other type
- **\$unpacked\_dimensions** : Return's the the total number of unpacked dimensions for an array (static or dynamic).
- **\$left** : Return's the left bound (MSB) of the dimension
- **\$right** : Return's the right bound (LSB) of the dimension
- **\$low** : Return's the minimum of \$left and \$right of the dimension
- **\$high** : Return's the maximum of \$left and \$right of the dimension
- **\$increment** : Return's 1 if \$left is greater than or equal to \$right and -1 if \$left is less than \$right
- **\$size** : Returns the number of elements in the dimension, which is equivalent to \$high - \$low + 1
- **\$assertoff** : Stop the checking of all specified assertions until a subsequent \$asserton. An assertion that is already executing, including execution of the pass or fail statement, is not affected.
- **\$assertkill**: Abort execution of any currently executing specified assertions and then stop the checking of all specified assertions until a subsequent \$asserton.
- **\$asserton**: Reenable the execution of all specified assertions.
- **\$fatal** :Run-time fatal assertion error, which terminates the simulation with an error code. The first argument passed to \$fatal shall be consistent with the corresponding argument to the Verilog \$finish system task, which sets the level of diagnostic information reported by the tool. Calling \$fatal results in an implicit call to \$finish.
- **\$error**: Run-time error.
- **\$warning**: Run-time warning, which can be suppressed in a tool-specific manner.
- **\$info**: Indicate that the assertion failure carries no specific severity.

## - Coverage system tasks / Improve Verilog system tasks

- **\$coverage\_control** : This function is used to control or query coverage availability in the specified portion of the hierarchy.
- **\$coverage\_get\_max** : This function obtains the value representing 100% coverage for the specified coverage type over the specified portion of the hierarchy.
- **\$coverage\_get** : This function obtains the current coverage value for the given coverage type over the given portion of the hierarchy.
- **\$coverage\_merge** : This function loads and merges coverage data for the specified coverage into the simulator.
- **\$coverage\_save** : This function saves the current state of coverage to the tool's coverage database and associates it with the given name.
- **\$display, \$write, \$fdisplay, \$fwrite, \$swrite** : The format arguments to these tasks must be string literals.
- **\$fscanf and \$sscanf** :The format arguments to these tasks may be expressions of string data type.
- **%u and %z format specifiers** : For packed data, %u and %z are defined to operate as though the operation were applied to the equivalent vector.
- **\$fread** : which has two variants: a register variant and a set of three memory variants.

## - Default tasks from verilog

### - \$display, \$monitor, \$strobe, \$finish, \$stop, \$readmemh, \$random

- **Indexed (Variable) array selection:** An indexed part select contains %Base expression (variable) %Width expression (constant) %Offset direction o Positive +: o Negative -:

```
reg [63:0] word;
reg [3:0] byte_num;
wire [7:0] byteN;
byteN = word[byte_num*8 +: 8];
```

◆ If byte\_num is 4, then expression is:

```
byteN = word[39:32];
```

- **Generate statements:** Label is required after the keyword “begin” to create generate for instance names. Only in loops, in conditional is not required.

**Note: All Figures were taken from Cadence.**