



Introduction to System Verilog.

Sarath Valapala.
Sep 07, 2009.



Table of Contents, Arial Bold, 24pt.

- 01 Standard Data Types and Literals
- 02 Operators
- 03 Arrays and Queues
- 04 User-Defined Data Types and Structures
- 05 Hierarchy and Connectivity
- 06 Procedures and Procedural Statements
- 07 Tasks and Functions
- 08 Simple Verification Features
- 09 Interfaces
- 10 Verification Blocks
- 11 Inter-process Synchronization

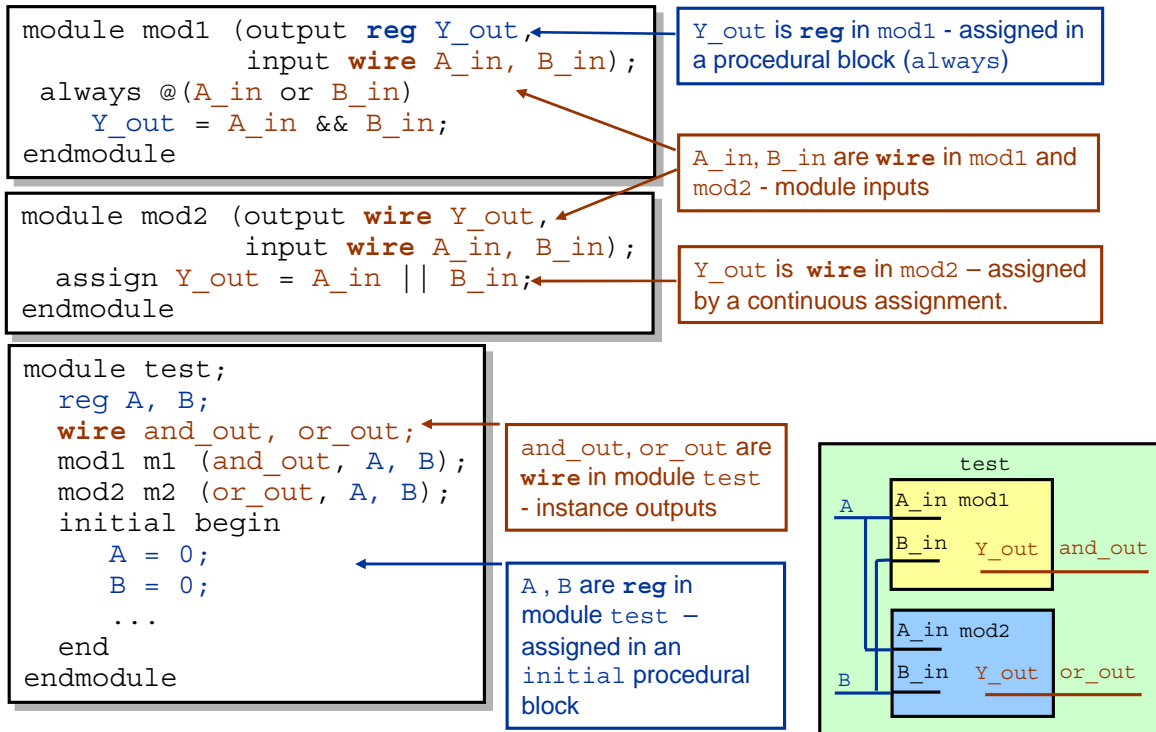
Standard Data Types and Literals

Chapter 2

July 31, 2007



Verilog Data Type Rules Example



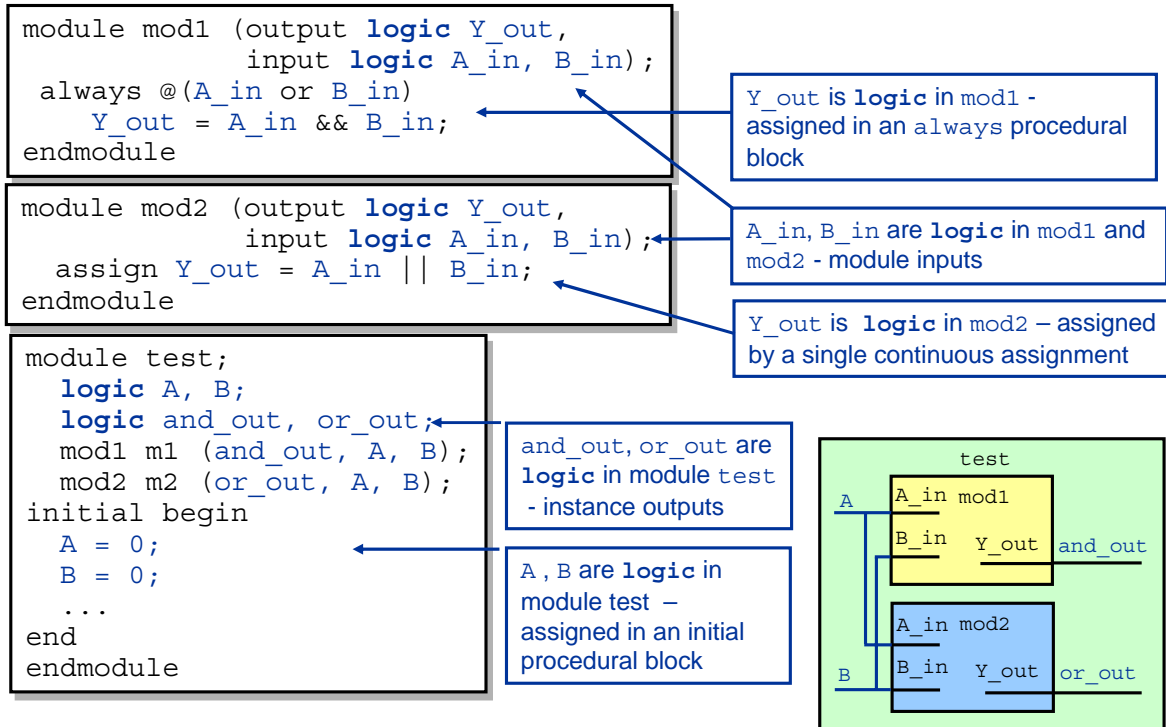
7/31/07

SystemVerilog for Design

25

For Verilog, you need to appropriately declare your ports as either net or register types. Pause here for a moment and review the use of the Verilog net and register port types.

Relaxed Data Type Rules Example



7/31/07

SystemVerilog for Design

27

These relaxed rules permit you to use a logic variable where Verilog would require a net, making the RTL more consistent and thus easier to write. Pause here for a moment and review the use of the SystemVerilog logic port type.

2-State Data Types

The Verilog value set consists of four basic values: (0, 1, X, Z)

SystemVerilog adds 2-state value types based on `bit`

- ◆ Values 0 and 1 only
- ◆ Direct replacements for `reg`, `logic` or `integer`
- ◆ Greater efficiency at higher-abstraction level modeling (RTL)

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

7/31/07

SystemVerilog for Design

30

All Verilog data types except `real` are 4-state logic. 4-state logic is essential for gate-level modeling, but rarely used at RTL and higher abstraction levels, so at times unnecessarily degrades performance. To enhance simulation performance at the higher abstraction levels, SystemVerilog adds 2-state types with only the 0 and 1 values, similar to those of the C language, but occasionally renamed to avoid confusion. The SystemVerilog `shortint` is 16 bits, the `int` is 32 bits, and the `longint` is 64 bits, on all platforms. Objects of the `bit` type are by default unsigned, and objects of the other integral types are by default signed.

SystemVerilog also renames the 32-bit C float to `shortreal` to differentiate it from the 64-bit Verilog `real`, which corresponds to a C double.

2-State Value Objects

2-state variables initialize to 0

- ◆ Hides test failure to initialize design

4-state variables initialize to X

- ◆ Exposes test failure to initialize design

Mix 2-state and 4-state logic as needed

```
bit [7:0] twostate;  
logic [7:0] fourstate;  
...  
twostate <= fourstate;
```

```
bit [7:0] test_number;  
bit [7:0] mem8x32 [0:31];  
byte char;  
int i;  
...  
for (i = 0; i < 100; i=i+1)  
    ...  
char = test_number;  
char = "E";
```

4-state to 2-state

1	1
0	0
x	0
Z	0

Objects with 2-state values are initially in the 0 state, thus may hide a failure to initialize the design.

Objects with 4-state values are initially in the unknown (X) state, thus promote detection of a failure to initialize the design.

Un-sized Literals

Verilog syntax
`<size>'<value>`

SystemVerilog syntax
`'<value>`

Size and value need *not* fit the target

- ◆ Value is padded or truncated to fit target

Verilog-1995 extends to 32 bits

- ◆ Extended to 32 bits

- ☐ with 0 if leftmost bit is 0 or 1
- ☐ with leftmost bit if Z or X

Verilog-2001 extends to size of expression

New SystemVerilog *single-bit* literal syntax

- ◆ Fills whole target with that single value

```
reg [5:0] databus;

databus = 6'hF;    // 001111
databus = 6'b0;    // 000000
databus = 6'b1;    // 000001
databus = 6'bz;    // zzzzzz
databus = 6'bx;    // xxxxxx
databus = 4'bx;    // 00xxxx
```

```
reg [5:0] databus;

databus = '0;      // 000000
databus = '1;      // 111111
databus = 'x;      // xxxxxx
databus = 'z;      // zzzzzz
```

Verilog-1995 extends an unsized literal to 32 bits. If the leftmost bit is the high impedance value or the unknown value, it extends to 32 bits with that value. It extends beyond 32 bits with zeroes.

Verilog-2001 extends the leftmost high impedance value or unknown value to the size of the enclosing expression.

SystemVerilog has a new syntax for single-bit unsized literals that are that notwithstanding their name are both unsized and unbased.

When you assign an unsized literal, SystemVerilog fills all bit positions of the target with the specified single-bit 4-state value.

This makes it very easy to set a logic vector to for example all ones, all high impedance, or all unknowns.

Time Literals

```
100ns  
3.55ps  
1step
```

SystemVerilog allows units when specifying time literals

- ◆ More control over time values

Format:

- ◆ Integer or fixed point value
- ◆ Time unit (fs ps ns us ms s step)
 - ❑ step advances simulation one precision unit
- ◆ No space between the value and unit

Literal is interpreted as a realtime value scaled to the current time unit and rounded to the current precision

You write a time literal as an integer or fixed-point number, followed immediately by a time unit (fs ps ns us ms s step) without an intervening space character. The new step time unit refers to the simulation step, that is, the simulation precision. SystemVerilog scales the time literal to the current time unit and rounds it to the current time precision. This implies that you must have a current time unit and precision specified for the interface, module or program in which the time literal appears.

Operators

Chapter 4

July 31, 2007



Assignment Operators

The C-like assignment operators combine a *blocking* assignment and an operator

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

7/31/07

SystemVerilog for Design

61

Assignment operators are blocking assignments. Use them only where you would use a blocking assignment, such as for combinational logic, temporary variables within a sequential block, or possibly in your testbench.

IEEE Standard 1800 for SystemVerilog — Section 8.3

Pre-/Post Increment/Decrement Operators

These combine a *blocking* assignment and an increment or decrement operator

- ◆ Pre- form (`++a`, `--a`) adds or subtracts then uses new value
- ◆ Post form (`a++`, `a--`) uses value then adds or subtracts
- ◆ You can use these as a stand-alone statement or as part of an expression

statement

```
--total;
```

statement

```
for (i=0; i < 13; i++)  
    ...
```

expression

```
b = 1;  
a = b++; // post a=1, b=2  
a = ++b; // pre  a=3, b=3  
a = b--; // post a=3, b=2  
a = --b; // pre  a=1, b=1
```



Synthesis

statement form
synthesizable

Pre and post increment and decrement operators are blocking assignments. Use them only where you would use a blocking assignment, such as for combinational logic, temporary variables within a sequential block, or possibly in your testbench.

Assignment Patterns

Define a list of values for an assignment to:

- ◆ Array elements
- ◆ Structure fields

Subtly different from concatenation

- ◆ Equivalent to individual assignments
 - so unsized literals are okay

Can use pattern “keys”

- ◆ Array element index
- ◆ Structure field name or type
- ◆ 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};
```

7/31/07

SystemVerilog for Design

64

You can make pattern assignments to unpacked arrays and unpacked structures.

An assignment pattern is very similar to a concatenation, but unlike a concatenation, does not need to be sized. SystemVerilog pads or truncates each element individually as needed.

You can make pattern assignments by position or by key, but not by both in the same assignment. For arrays, the key can be an element index or the default keyword, which assigns that default value to all elements not otherwise specified in the pattern. For structures, the key can be a field name, a field type, or the default keyword. See the chapter on “User-Defined Data Types” for more information.

IEEE Standard 1800 for SystemVerilog — Section 8.13

Wild Equivalence Operator

Indicates “don’t-care” bit position (like `casex`)

Asymmetric – only right side can have wildcard bits

Also wild inequality (`!=?`)



Synthesis

Not for synthesis

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

if (a == b) // unknown
    ...

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

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

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

if (a ==? 4'b01??) // true
    ...
```

7/31/07

SystemVerilog for Design

65

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. These operators return a 1-bit result, that can be 0, 1 or unknown. Remember that you can use the “Z” and question-mark (?) characters interchangeably in a Verilog 4-state literal.

IEEE Standard 1800 for SystemVerilog — Section 8.5

Arrays and Queues

Chapter 9

July 31, 2007



Verilog Array Review

An array is a collection of nets or variables of one declared type

- ◆ Verilog supports arrays of any Verilog net or register type
 - ❑ variable types are integer, real, reg, time
 - ❑ also vector e.g. reg [7:0] mems [0:3] [0:31] // 4 32X8 mems
- ◆ Verilog supports an unlimited number of dimensions
 - ❑ each from *left* bound to *right* bound
- ◆ Verilog syntax can access only individual elements
 - ❑ cannot directly take a bit-select or part-select of memory element

```
reg [5:0] count;           // 1 element X 6 bits wide
reg flags [0:5];           // 6 elements X 1 bit wide
reg [7:0] dmem [0:255];    // 256 elements X 8 bits wide
integer intmem [1:3] [1:50]; // 3X50 integer
```

You use arrays to group elements of one type into multi-dimensional objects. Each dimension that you declare is a constant integer expression. For Verilog, you can reference only one complete element at a time, by providing an integer expression index for each dimension.

Packed and Unpacked Arrays

- ◆ Unpacked arrays are the dimensions *after* the array name
 - Traditional array – any type
- ◆ Packed arrays are the dimensions *before* the array name
- ◆ You can use both – an unpacked array of packed arrays

```
// packed array (aka vector)
bit [7:0] vector1;
// 2-D packed array - new in SV
logic [3:0][15:0] p_array;
```

```
// unpacked array of reals
real myReals [7:0];
// unpacked 2-D array of integers
integer up_array [3:0][3:0];
```

```
bit v[8]; // equivalent to: bit v[0:7];
bit [3:0] v1 [7:0]; // v1 is an array of 1-D packed arrays
bit [3:0] [5:1] v2 [7:0]; // v2 is an array of 2-D packed arrays
// v2[3] is a 2-D packed array[3:0][5:1]
// v2[3][2] is a 1-D packed array[5:1]
// v2[3][2][1] is a scalar

logic [2:1][2:0] v3 [1:0][4:1]; // v3 is 2-D array of 2-D packed
// arrays
```

7/31/07

SystemVerilog for Verification

175

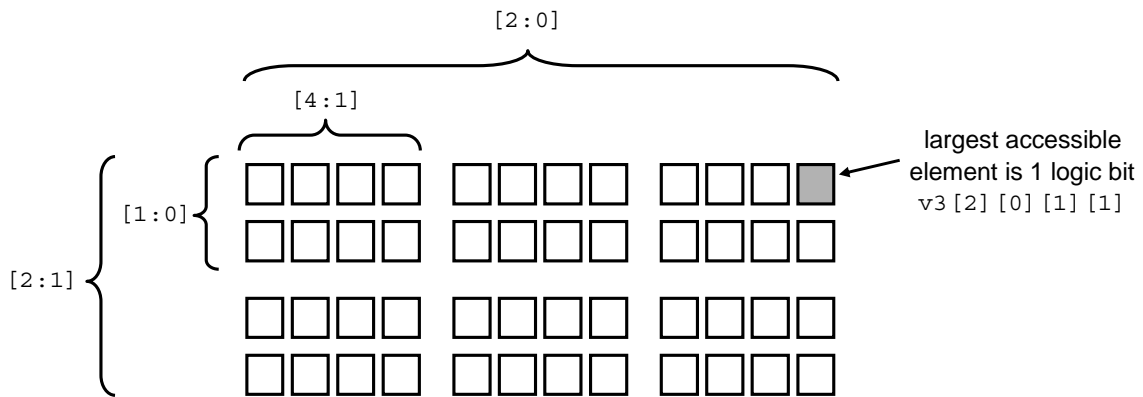
The SystemVerilog unpacked array is what we traditionally think of as an array. A SystemVerilog unpacked array can be of any type.

The packed dimensions are those you declare before the array identifier. Where the array name appears as a primary, SystemVerilog treats it as a single one-dimensional vector. You can pack only the single-bit types (bit, logic, reg), and recursively, other packed objects (array, structs) of those types. A packed array is essentially just a vector with fields you predefine for your convenience. 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.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 5.2

4-D Unpacked Array

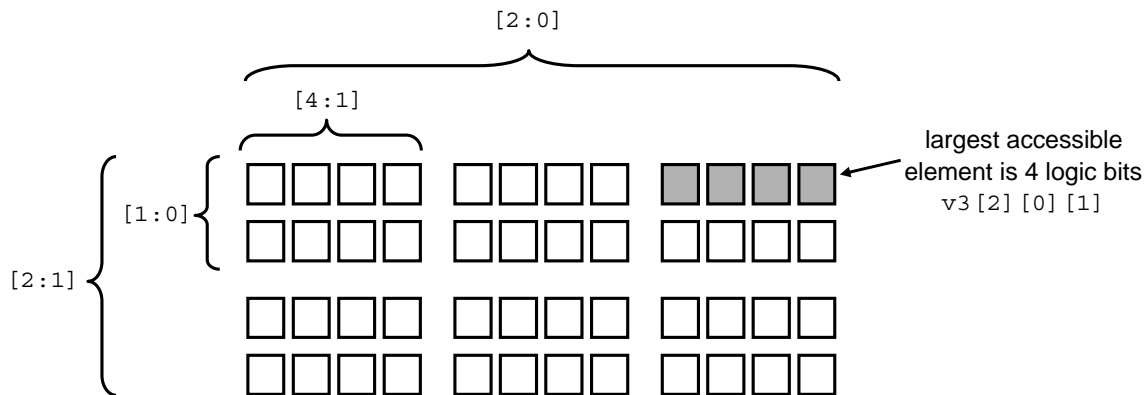
```
logic v3 [2:1][2:0][1:0][4:1]; // v3 is 4-D unpacked array
```



This array has four unpacked dimensions and no packed dimensions. It is a 2X3X2X4 unpacked array of logic elements. The “packed” part of this array is a single logic element, which you reference by providing an integer expression index for each of the four unpacked dimensions.

3-D Unpacked Array of 1-D Packed Arrays

```
logic [4:1] v3 [2:1][2:0][1:0]; // 3-D unpacked, 1-D packed
```



This array has three unpacked dimensions and one packed dimension. It is a 2X3X2 unpacked array of 4-bit logic vectors. The packed part of this array is a 4-bit logic vector, which you reference by providing an integer expression index for each of the three unpacked dimensions. You can use that packed element as a 4-bit unsigned integer in any expression in which an integer may appear.

Indexing and Slicing of Arrays

You can select one or more contiguous elements (*slice*) of an array.

```
logic [3:0][7:0] p_reg, p_reg1;           // packed logic array
byte  subreg;                             // 8-bit vector
logic [3:0] sub_subreg;                   // 4-bit vector
logic [1:0][7:0] mini_reg;               // sub-array of p_reg

initial begin
    p_reg[0]      = 7;                    // array assignment
    p_reg[3:1]    = {8'h00, 8'h10, 8'hff}; // array slice - [3:1]
    p_reg1[1:0]   = p_reg[3:2];           // number of elements must match
    subreg        = p_reg[2];             // array assignment to a vector
    sub_subreg    = p_reg[2][3:0];        // array slice - [2][3:0]
    mini_reg      = p_reg[1:0];           // array slice - [1:0]
end
```



Support

A slice is a selection of one or more contiguous elements of an array. To select a slice, you simply provide a range of indices instead of a single index. You can slice only one dimension.

Pause here, and examine these array slices.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 5.4

Dynamic Arrays

Use a dynamic array when the array size must change during the simulation

- ◆ Declare a dynamic array by leaving one unpacked dimension unsized
- ◆ A dynamic array doesn't exist until it is explicitly created during runtime
- ◆ Use the `new[size]` operator to create the array or change its size
 - Use the `new[size] (array)` syntax to copy content from an existing array (including itself) when setting or changing the array size
- ◆ Dynamic arrays have the `size()` and `delete()` methods
 - `function void delete() // delete all elements (sets size to 0)`

```
logic [7:0] dynarr[]; // dynamic array of 8-bit vectors
int index;
...
dynarr = new[8];           //initialize
for (int i=0; i<8; i++) dynarr[i] = i+1;
index = dynarr.size();     // 8
dynarr = new[16] (dynarr); // resize array keeping values
index = dynarr.size();     // 16
dynarr.delete();          // 0
```

7/31/07

SystemVerilog for Verification

184

Use a dynamic array when the array size changes during the simulation. Declare the dynamic array by leaving one unpacked dimension unsized. Create, and re-create, the array during run time using the `new` operator. As a parameter to the `new` operator, you can provide an existing array of the same type to initialize the new array. Dynamic arrays have the `size()` and `delete()` methods, as well as the standard array manipulation methods (see 5.15) and standard array query system functions (see 22.6).

IEEE Std. 1800-2005 SystemVerilog LRM, Section 5.6

Associative Arrays

Use an associative array when the data space is unbounded or sparsely populated

- ◆ Declare the array by specifying a type instead of a size for its one dimension
 - ❑ The key can be any type for which a relative order can be determined
 - ❑ Array elements do not exist until you assign to them
 - ❑ Array elements are “pairs” of associated key and data values
- ◆ Associative arrays have the `num()`, `delete()`, `exists()`, `first()`, `last()`, `next()`, and `prev()` methods

```
bit [3:0] aa1 [int];           // associative array of 4-bit 2-state
                               // with index type of int
logic [7:0] aa2 [integer];    // associative array of 8-bit logic
                               // with index type of integer
```

Use an associative array when the address space is unbounded or sparsely populated. Declare the associative array by specifying a type instead of a size for its one dimension. The key can be any type for which a relative order can be determined, that is, any type to which SystemVerilog can apply a relational operator. Associative array elements do not exist until you assign them. You assign “pairs” of associated key and data values.

IEEE Std. 1800-2005 SystemVerilog LRM, Sections 5.9 – 5.13

Associative Array Lookup Table

```
logic [7:0] assoc[int];           // associative array declaration
bit  [4:0] rand_a;
logic [7:0] rand_d, rdat;
initial begin
    for (int i=0; i<32; i++) begin
        success = randomize(rand_a, rand_d); // random data gen
        write_mem (rand_a, rand_d);
        assoc[rand_a] = rand_d;             // associative array assign
    end
    $display("Memory locations to check:%d", assoc.num());
    for (int i=0; i<32; i++)
        if (assoc.exists(i)) begin         // only read address if written
            read_mem(i, rdat);
            if (rdat != assoc[i])          // check against array data
                $display("Error at Addr:%h" i);
            assoc.delete(i);               // Deallocate memory at index
        end
    $display("array size:%d", assoc.num()); // should display 0
end
```

7/31/07

SystemVerilog for Verification

188

This example declares an associative array to store 8-bit logic vectors with keys of the int type. In a loop, it randomizes address and data 32 times, writes the data to a memory component, and stores the address/data pairs in an associative memory. As the address vector is only 5 bits wide, it is very likely that some addresses are written multiple times. In a second loop, the example iterates through the potential address space. For each address, if the associative array has an entry for that address, it verifies that the memory component has the correct data, then deletes the entry for that address. Deleting entries individually is inefficient.

More Efficient Lookup Table

```
logic [7:0] assoc[int];           // associative array
bit  [4:0] rand_a;
logic [7:0] rand_d, rdat;
int ai;                           // index for methods
initial begin
  for (int i=0; i<32; i++) begin
    success = randomize(rand_a, rand_d); // random data gen
    write_mem (rand_a, rand_d);
    assoc[rand_a] = rand_d;           // associative array assign
  end
  $display("Addresses Assigned:%d", assoc.num());
  if (assoc.first(ai))               // assign first index
  do begin
    read_mem(ai, rdat);
    if (rdat !== assoc[ai])         // check data against array
      $display("Error at Addr:%h", ai);
  end
  while(assoc.next(ai));             // assign next index
  assoc.delete();                   // deallocate all memory
end
```

7/31/07

SystemVerilog for Verification

189

This algorithm is more efficient. It utilizes associative array methods to iterate through only the assigned entries. It deletes the entire associative array at once when finished with it.

Queues

Use a queue “array” where insertion and extraction order are important

- ◆ Declare a queue by using \$ as the size of its one dimension:

```
int q_int[$];  
int q_int[$:200]; // optionally limit the queue size
```

- ◆ Use \$ to reference the end of the queue: `myInt = q_int[$];`
- ◆ Queues have the `size()`, `insert()`, `delete()`, `pop_front()`, `pop_back()`, `push_front()` and `push_back()` methods

```
integer      q_integer[$];    // queue of integers  
logic [15:0] q_logic [$];     // queue of 16-bit logic  
int          q_int[$:2000];   // queue of int - max size of 2000  
time         q_time [$:10];   // queue of time - max size of 10
```

Use a queue when insertion and extraction order are important. Declare the a queue by using the “dollar” (\$) character as the size of its one dimension. You can optionally limit the queue size by appending a colon (:) followed by a constant expression after the “dollar” character. You can use the “dollar” character to represent the end of the queue in subsequent array subscript operations.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 5.14

Queue Methods Example

```
int q_int[$];           // queue declaration syntax
bit [7:0] q_bit[$:100]; // queue with maximum size of 100

initial begin
    q_int.push_front(0); // {0}
    q_int.push_back(1);  // {0,1}
    q_int.push_front(2); // {2,0,1}
    q_int.insert(1, 3);  // {2,3,0,1}
    q_int.insert(3, 4);  // {2,3,0,4,1}
    q_int.delete(2);     // {2,3,4,1}
    q_int.insert(2,5);   // {2,3,5,4,1}
    data = q_int.pop_back(); // {2,3,5,4} data = 1
    data = q_int.pop_front(); // {3,5,4} data = 2
    while (q_int.size() > 0) // checking queue size
        data = q_int.pop_back(); // loop executes 3 times
    q_bit.push_front(8'h01); // {'h01}
    q_bit.push_back(8'h45);  // {'h01,'h45}
    q_bit.push_front(8'h89); // {'89,'h01,'h45}
end
```

Queue methods support inserting and extracting elements. This example illustrates use of several of the methods.

Pause here and examine this example using queue methods.

Queue Indexing Example

```
int q_int[$];           // queue declaration syntax
bit [7:0] q_bit[$:100]; // queue with maximum size of 100

initial begin
  q_int = {0,q_int};           // {0}
  q_int = {q_int,1};           // {0,1}
  q_int = {2,q_int};           // {2,0,1}
  q_int = {q_int[0],3,q_int[1:$]}; // {2,3,0,1}
  q_int = {q_int[0:2],4,q_int[3]}; // {2,3,0,4,1}
  q_int = {q_int[0:1],q_int[3:4]}; // {2,3,4,1}
  q_int = {q_int[0:1],5,q_int[2:3]}; // {2,3,5,4,1}
  data = q_int[$]; q_int = q_int[0:$-1]; // {2,3,5,4} data = 1
  data = q_int[0]; q_int = q_int[1:$]; // {3,5,4} data = 2
  while (q_int.size() > 0) begin // checking queue size
    data = q_int[$];           // loop executes 3 times
    q_int = q_int[0:$-1]; end
  q_bit = {8'h01,q_bit};       // {'h01}
  q_bit = {q_bit,8'h45};       // {'h01,'h45}
  q_bit = {8'h89,q_bit};       // {'89,'h01,'h45}
end
```

7/31/07

SystemVerilog for Verification

193

A queue is an array, so if you really want to, you can still use array indexing to insert and extract elements. As this example illustrates, you probably don't really want to. The queue methods are much more friendly.

We won't bother examining this slide too closely.

Quiz

Match the array that best meets each design need:

- | | |
|---|----------------------|
| 1. I want to concisely code an RTL instruction processor to strip the opcode, address and data fields off the instruction word... | A. Packed array |
| 2. I need to validate the architecture of my processor stack... | B. Unpacked array |
| 3. I need to validate the architecture of my packet data temporary storage... | C. Dynamic array |
| 4. I need to validate the architecture of my instruction cache... | D. Associative array |
| 5. I want to perform matrix operations on fixed-size “chunks” of data of the real type... | E. Queue |

Solutions in Appendix A

User-Defined Data Types and Structures

Chapter 5

July 31, 2007



Primitive and User-Defined Types

Primitive data types

- ◆ Built-in types not constructed from other data types

- e.g. `bit`, `logic`, `int`, `real`

User-defined data types

- ◆ Types named with `typedef`
- ◆ Types constructed from other types
 - e.g. vectors and arrays
- ◆ 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



Support:
`union`

A primitive data-type is a built-in type that is not constructed from other types. Any other data type is user-defined. This includes types that you named with the `typedef` keyword and types you construct from other types, such arrays, enumerations, structures and unions.

Type Definition

You can declare and name a type using `typedef`

- ◆ Verilog standard types
- ◆ SystemVerilog user-defined types

You can use your type name anywhere you can use a type

- ◆ e.g. variable or port declaration

Must declare type before use!

Also C-like anonymous types

- ◆ Any *not* declared with `typedef`

```
typedef logic [7:0] vec8_t;  
  
input vec8_t op1;  
vec8_t busa, busb;  
function vec8_t checksum  
    ...
```

```
// typed logic type  
typedef logic [7:0] vec8_t;  
vec8_t byte_reg;
```

```
// anonymous type  
logic [7:0] byte_reg;
```

The `typedef` construct declares and names a user-defined type, allowing the type to be used in the declaration of variables and other types. You must declare a type before you attempt to use it, either:

- In the compilation unit scope, which makes it visible in all lexically following design units in the compilation unit scope.
- Inside a package, which makes it visible in all design units that import the type from the package or reference the type definition from that package.
- Inside any declarative scope (interface, module, program), which makes it visible only within that declarative scope, and any further nested declarations.

IEEE Standard 1800 for SystemVerilog — Section 4.9

Enumerated Types

Type with user-defined values

- ◆ Values must be unique in current scope

Declared with `enum` keyword

- ◆ Typed or anonymous

Enumerated type variables are strongly typed

Should only be *directly* assigned:

- ◆ An enum value, or...
- ◆ From a variable of the same enum type

Naming type permits typecasting

- ◆ `type_name' (value)`

```
// anonymous enum type
enum {idle, start, done} astate;
```

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

```

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

7/31/07

SystemVerilog for Design

75

SystemVerilog enumerated types are much like their C counterparts. SystemVerilog permits you to assign a variable of an enumerated type only from another variable of the same type or from an enumeration value of that type. To assign any other value requires you to cast the value to the enumerated type. Statically casting the value to the enumerated type does not check the validity of the value. As Verilog does not require such strong type checking, some compilers permit the direct assignment of an integral value to an enumerated type variable.

IEEE Standard 1800 for SystemVerilog — Section 4.10

Enumerated Type in Expressions

The base type defaults to `int`

Values are by default integers incrementing from zero in declaration order

You can use a variable of the enumerated type in an expression

```
enum {idle, start, done} astate;  
  
// idle = 0  
// start = 1  
// done = 2
```

astate is type
int by default

- ◆ SystemVerilog automatically casts *from* the enumerated type
- ◆ You must do any casting *to* the enumerated type



Tip

Naming (typedef) permits typecasting

doesn't work

try this

```
typedef enum {idle, start,  
             done} state_t;  
state_t tstate;  
int aint;  
  
tstate = start;  
aint = tstate + 1;    // = 2  
tstate = tstate + 1;  
tstate = state_t'(tstate + 1);
```

7/31/07

SystemVerilog for Design

76

SystemVerilog automatically casts an enumeration value to the base value of the enumerated type when the enumeration value is used in an expression. You must explicitly cast the expression value back to the enumerated type to assign it to a variable of the enumerated type.

IEEE Standard 1800 for SystemVerilog — Section 4.10

Encoding of Enumerated Types

Values by default increment from zero in declaration order
You can define values

- ◆ Allows one-hot coding etc

```
enum {idle, start, pause, done} mstate;  
// idle = 0; start = 1  
// pause = 2; done = 3
```

You can mix explicit and implicit value encoding

- ◆ Implicit values increment from previous explicit value
- ◆ Compilation error if encodings overlap

```
enum {idle = 1, start = 2,  
      pause = 4, done = 8} mstate;
```

```
enum {S0 = 1, S2, S3, S4,  
      S5 = 9, S6, S7, S8} encs;
```



Error

P6 duplicates
P5 encoding

```
enum {P0 = 1, P2, P3, P4,  
      P5, P6 = 5, P7, P8} badencs;
```

7/31/07

SystemVerilog for Design

77

Value encodings for enumerated types by default start at 0 and then increment. You can explicitly specify any encoding that does not duplicate values already used. For 2-state types, you can use only the binary values (no Z or X). You can mix explicit and implicit value encoding in the enumerated type declaration. At any point where you do not specify a value, the value simply increments from the previous value.

IEEE Standard 1800 for SystemVerilog — Section 4.10

Explicit Enumerated Types

You can specify the enumeration base type

- ◆ Base type `int` by default

Default encoding will match specified type

Explicit encoding must match type (and length!)

```
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);
```

The base type of an enumerated type defaults to `int`. You need to declare any other base type. You can declare base types of the atomic integer types (`byte`, `shortint`, `int`, `longint`, `integer`, `time`) or integer vector types (`bit`, `logic`, `reg`).

IEEE Standard 1800 for SystemVerilog — Section 4.10

Enumerated Type Access Methods

SystemVerilog methods enable iteration over enumeration values.

Method	Description
<code>first()</code>	Returns first value
<code>last()</code>	Returns last value
<code>next(N)</code>	Returns next Nth value from current
<code>prev(N)</code>	Returns previous Nth value from current
<code>num()</code>	Returns number of values
<code>name()</code>	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
```

You can use enumeration methods to iterate over the valid values of an enumeration type and to obtain the name associated with a specific valid value.

IEEE Standard 1800 for SystemVerilog — Section 4.10.4

Struct Type

Declare data structure type with fields of different types

- ◆ Can be typed or anonymous

Use “dot” notation to access individual structure fields

You can make pattern assignments to struct objects

- ◆ Ordered or keyed
 - not both in same pattern
- ◆ Keys: name, type, default
 - can mix keys

You can declare arrays of a 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}};
```

7/31/07

SystemVerilog for Design

85

SystemVerilog adopts from C the struct construct for grouping fields of potentially different types, and the union construct for interpreting the same area in memory as different types. You can make pattern assignments to SystemVerilog structures. The next page describes pattern assignments in detail.

IEEE Standard 1800 for SystemVerilog — Section 4.11

Keyed Pattern Assignment

Order of precedence:

1. NAME

Assign fields by name

2. TYPE

Assign remaining fields of a specific type

3. DEFAULT

Assign remaining fields using **default** (value must be type compatible with all remaining fields)

```
typedef struct {
    bit      id, par;
    int      addr;
    logic[7:0] data;
} frame_t;
frame_t frame1;

// assignment by order
frame1='{1'b0, 1'b0, 42, 8'hff};

// assignment by name
frame1='{data:8'hff,addr:42,par:0};

// assignment by name and type
frame1='{data:8'hff,addr:42,bit:0};

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

7/31/07

SystemVerilog for Design

86

Assignment patterns may be ordered or may use keys, but not both in the same pattern. Key precedence is: The field name; All other fields of the specified type; The default keyword, which applies to all remaining fields. All fields to which the default value applies must be of a type compatible with the default value specified.

IEEE Standard 1800 for SystemVerilog — Section 3.8

Packed Structures

Struct type can be packed

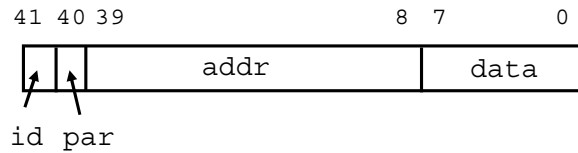
All fields must be “packable”

- ◆ Integral values only
- ◆ Any field that is array or struct must also be packed
- ◆ Can operate on whole struct as if one dimensional bit vector

Still use “dot” notation to access individual structure fields

If any item is 4-state then the whole struct is stored as 4-state

- ◆ Conversions done automatically upon read/write



```
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;
```

SystemVerilog allows you to declare a struct packed if it contains only integral fields. Like a packed array, you can later access the packed struct as if it was a one-dimensional bit vector, performing bit and part selects and other integral operations on it.

IEEE Standard 1800 for SystemVerilog — Section 4.11

New Type Declaration Regions

You can declare ports of user-defined types, – but –

You must declare types before you use them

- ◆ How do modules see the type definition?

SystemVerilog has two new name spaces:

- ◆ compilation unit scope
- ◆ packages

The coming chapter explains this in more detail...

```
typedef enum {start, done} mode_t;
```

declaration

```
typedef enum {start, done} mode_t;
module mone(
    input  mode_t mode,
    output logic [7:0] out);
```

compilation unit scope

```
package mytypes;
    typedef enum {start,done} mode_t;
endpackage : mytypes
```

package

```
import mytypes::*;
module mtwo(
    input logic [7:0] out,
    output  mode_t mode);
```

package reference

7/31/07

SystemVerilog for Design

88

You must declare types before you attempt to use them. This presents a problem when you need to declare ports of a user-defined type. To resolve this, SystemVerilog adds two new name spaces in which you can declare types:

- The Compilation Unit Scope is a declarative region that is by default immediately before the module definition in the same SystemVerilog file. The standard requires vendors to provide a means to expand this scope to all files compiled during the same session.
- The package is a new design unit for shared declarations. You reference declarations from the package by using import statements or by using resolved pathnames.

A later chapter on “Hierarchy” explores both mechanisms in more detail.

IEEE Standard 1800 for SystemVerilog — Sections 19.2, 19.3

Hierarchy and Connectivity

Chapter 6

July 31, 2007



Port Connection

Verilog offers two port connection options:

- ◆ Ordered port connections
 - ❑ Risk of incorrectly ordered connections
- ◆ Named port connections
 - ❑ Safer but very verbose

```
module counter (  
    input  logic clk, rst, ld,  
    input  logic [7:0] data,  
    output logic [7:0] cnt);  
    ...  
endmodule
```

SystemVerilog adds two options to simplify port connections:

- ◆ .name (dot-name)
- ◆ .* (dot-star)

```
counter c1 (clk, rst, ld, data, cnt);
```

ordered

```
counter c2 (clk, ld, rst, data, cnt);
```

badly ordered

```
counter c3 (.clk(clk), .rst(rst), .ld(ld),  
            .cnt(cnt), .data(data));
```

named

For Verilog, you can use implicit ordered port connections or explicit named port connections. Use of ordered port connections exposes you to the risk that you may connect a signal to the wrong port. Use of named port connections greatly reduces that risk, so is the preferred method, but to use named port connections is very verbose. Following pages describe new SystemVerilog implicit port connection options.

IEEE Standard 1800 for SystemVerilog — Section 19.11.1 – 19.11.4

.name Implicit Port Connection

Simplifies port connection
where variable name and port
name match

`.clk = .clk(clk)`

As safe, but not as verbose, as
named connection

Can be mixed with named
connection

- ◆ For ports where names don't
match

```
module count (  
    input  logic clk, rst, ld,  
    input  logic [7:0] data,  
    output logic [7:0] cnt,  
    output logic val);  
...  
endmodule
```

```
count c4 (.clk, .rst, .ld,  
          .data, .cnt, .val);
```

dot-name

```
count c5 (.clk(clk), .rst(rst), .ld(ld),  
          .data(data) .cnt(cnt), .val(val));
```

named equivalent

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

dot-name and named

7/31/07

SystemVerilog for Design

96

For nets or variables that connect to ports of the same name, you can use a “dot name” (`.portname`) implicit connection for the subset of those you want automatically connected. You connect any remaining unmatched ports using the Verilog port name connection. You can use a “dot name” (`.portname`) implicit port connection only if the name, size and type of the port matches that of the connecting net or variable.

A SystemVerilog “dot name” port connection is equivalent to a Verilog named port connection (`.portname(name)`) with the following exceptions:

- It does not implicitly declare previously undeclared nets.
- It does not allow explicit or implicit type-casting between the port and the connected net or variable, not even padding or truncation, with two exceptions:

It automatically converts between 2-state and 4-state types of the same length,
and it automatically converts between a net and a variable of the same length.

IEEE Standard 1800 for SystemVerilog — Section 19.11.3

. * Implicit Port Connection

Automatically connects
matching variable and port
names

Safety of named connection
without verbosity

- ◆ Can lose some
port list readability

Can be mixed with named
connection

- ◆ For ports where names
don't match

```
module count (  
    input  logic clk, rst, load,  
    input  logic [7:0] data,  
    output logic [7:0] cnt,  
    output logic val);  
...  
endmodule
```

```
count c7 (. *);
```

dot-star

```
count c8 (.clk, .rst, .ld,  
          .data, .cnt, .val);
```

dot-name
equivalent

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

dot-star and named

You can use a “dot star” (.*) implicit port connection for those ports whose name, size and type matches that of the connecting net or variable. This connects all such matching ports not otherwise explicitly connected. You connect any remaining unmatched ports using the Verilog port name connection. The “dot star” implicit connection is subject to the same rules as the “dot name” (.portname) implicit connection.

IEEE Standard 1800 for SystemVerilog — Section 19.11.4

.name and .* Rules

Cannot mix ordered connections with .name or .* in the same instantiation

```
count c10 (.*, .ld, .cnt);
```



Can mix named and .name connections in the same instantiation

Can mix named and .* connections in the same instantiation

```
count c11 (.data, .clk, .cnt, .val,  
          .rst(reset), .ld(load));
```

dot-name and named

Named connections are required for:

- ◆ Port/signal name mismatches
- ◆ Port/signal width mismatches
- ◆ Unconnected ports

```
count c12 (  
    .*,  
    .ld(load),  
    .data(data[15:8]),  
    .val()  
);
```

dot-star and named

7/31/07

SystemVerilog for Design

98

SystemVerilog port connections are subject to rules that include the following:

- You cannot use Verilog implicit ordered connections with any other kind of port connection in the same instance.
- You can use Verilog explicit named port connections with either the SystemVerilog implicit “dot name” (.portname) connection or the SystemVerilog implicit “dot star” (.*), in fact, you often have to, to connect ports to signals that have a different name or are expressions of signals.

IEEE Standard 1800 for SystemVerilog — Section 19.11.3, 19.11.4

New Type Declaration Regions

You can declare ports of user-defined types, – but –

You must declare types before you use them

- ◆ How do modules see the type definition?

SystemVerilog has two new name spaces:

- ◆ compilation unit scope
- ◆ packages

```
typedef enum {start, done} mode_t;
```

declaration

```
typedef enum {start, done} mode_t;
module mone(
    input  mode_t mode,
    output logic [7:0] out);
```

compilation unit scope

```
package mytypes;
    typedef enum {start,done} mode_t;
endpackage : mytypes
```

package

```
import mytypes::*;
module mtwo(
    input logic [7:0] out,
    output  mode_t mode);
```

package reference

You must declare types before you attempt to use them. This presents a problem when you need to declare ports of a user-defined type. To resolve this, SystemVerilog adds two new name spaces in which you can declare types:

- The Compilation Unit Scope is a declarative region that is by default immediately before the module definition in the same SystemVerilog file. The standard requires vendors to provide a means to expand this scope to all files compiled during the same session.
- The Package is a new design unit for shared declarations. You reference declarations from the package by using import statements or by using resolved pathnames.

IEEE Standard 1800 for SystemVerilog — Section 19.2, 19.3

Compilation Unit Scope

New scope for declarations

- ◆ Outside of modules, packages, interfaces etc

Default scope is a single file

- ◆ Tool options can change this
 - e.g. define scope as multiple files compiled at same time

Scope is unnamed

- ◆ 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'b1111100;

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'b0011111;
    ...
    assign oa = eseg;           //local
    assign ob = $unit::eseg;    //unit
endmodule : mtwo
```

modules.sv



Beware

Scope of compilation unit is tool specific and can change between compile sessions.

Local declarations override compilation unit scope declarations without warning.

7/31/07

SystemVerilog for Design

101

The compilation unit scope is a new scope for declarations that is outside any design unit.

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. Vendors may also provide other options.

A compilation unit is equivalent to an anonymous package. As it is anonymous, there is no name for referencing a compilation unit scope, so you cannot hierarchically reference its further nested declarations.

A type declaration in the compilation unit scope is visible to any design units defined within the same compilation unit scope. Design unit declarations with the same name hide those in the compilation unit without warning. You can use \$unit with the scope operator to explicitly reference the compilation unit declarations.

IEEE Standard 1800 for SystemVerilog — Section 19.3

Packages

New separately-compilable definition name space unit (as are interfaces, modules and programs)

Contains declarations to be shared between modules

- ◆ Types, variables, subroutines...

You can import the declarations

- ◆ Explicit – specifically named
- ◆ Implicit – all using wildcard (*)

You can directly access a declaration by using the resolution operator (::)

- ◆ For this import is not required



Beware: imports in compilation unit

```
package mytypes;
    typedef enum {start,done} mode_t;
endpackage : mytypes
```

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

explicit import

```
import mytypes::*;
module mone(
    input  mode_t mode,
    output logic [7:0] out);
```

wildcard import

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

resolved name

7/31/07

SystemVerilog for Design

102

The package is a new separately-compilable definition name space unit.

You can declare types, variables, tasks, functions and assertion sequences and properties within a package.

You can reference packages from within modules, interfaces, programs, and other packages. You can explicitly import declarations or implicitly import the entire package contents. You can also reference a package declaration using the package name and the scope resolution operator.

The first example explicitly imports the “mode_t” type. This import is before the module definition to allow use of the “mode_t” type in the port declarations.

The second example implicitly imports the entire contents of the package. Local declarations can override such implicitly imported declarations.

The last example uses a resolved pathname to the package declaration and does not import the declaration.

IEEE Standard 1800 for SystemVerilog — Section 19.2

Explicit versus Wildcard Import

Explicit import directly loads declaration into module

- ◆ Compilation error if local or other explicitly imported declarations have same name
- ◆ Imports only the symbols specifically referenced

```
package P1;
  localparam int c1 = 10;
  typedef enum {start,stop} mode_t;
endpackage : P1
```

Wildcard import makes declaration *candidate* for import

- ◆ Local or explicitly imported declarations can override declaration *not yet used*

local declaration
clashes with
explicit import

```
module mone(...);
  import P1::c1;
  import P1::mode_t;
  logic [7:0] c1;
  mode_t mode;
  if (mode==stop) ...
  ...
endmodule
```

Error

undeclared
identifier

Imports in a compilation unit follow compilation unit rules

- ◆ Local declarations override compilation unit scope declarations without warning.
 - Even for an explicit import

Local declaration
overrides implicit
import

```
module mone(...);
  import P1::*;
  logic [7:0] c1;
  ...
endmodule
```

✓



Beware

Local or explicitly imported declarations take precedence over wildcard imports

7/31/07

SystemVerilog for Design

103

An explicit import directly loads the imported type definition into the design unit, as if you had declared it in the design unit. It is an error to locally redeclare such an explicitly imported identifier. Your explicit import imports exactly those identifiers you specify. For example, if you explicitly import an enumerated type, that does not automatically also import the enumeration value names!

An implicit import makes the package types candidates for import. It imports each type only when it is used. You can redeclare or explicitly import from another package an identifier that has been only implicitly imported and not yet used.

An import statement in the compilation unit scope places those type declarations in the compilation unit scope. Such declarations are then subject to the rules of any other declaration in the compilation unit scope, most notably that you can locally override them without warning.

IEEE Standard 1800 for SystemVerilog — Section 19.2.1

Ambiguity and Resolved Names

Be careful of ambiguous references

- ◆ Multiple declarations in separate packages using same name
- ◆ Compilation error

```
package P1;
  typedef enum {start,done} mode_t;
  int c = 5;
  ...
endpackage : P1
```

```
package P2;
  int c = 8;
  ...
endpackage : P2
```

Solutions

- ◆ Try to avoid duplicate names
- ◆ Structure packages carefully
 - ❑ Only required declarations visible
- ◆ Use resolved names
- ◆ Use explicit imports

```
module mone(...);
  import P1::*, P2::*;
  logic [7:0] d = c;
  logic [7:0] e = P2::c;
  ...
endmodule
```

☒ ambiguous

☒ resolved

```
module mtwo(...);
  import P2::c;
  import P1:: mode_t;
  logic [7:0] d = c;
  ...
endmodule
```

☒

7/31/07

SystemVerilog for Design

104

It is an error to import the same identifier more than once. For implicit imports, the error does not manifest until you actually attempt to use the identifier.

In this example, packages P1 and P2 both declare the identifier “c”. Implicitly importing the contents of both packages gives a compilation error only upon the first use of “c”.

- You can use resolved names to disambiguate the duplicated identifier.
- You can alternatively explicitly import only parts of each package.

You should where possible avoid ambiguity by carefully naming declarations and partitioning packages. You can better control declaration visibility by using more smaller packages rather than fewer larger ones.

IEEE Standard 1800 for SystemVerilog — Section 19.2.1

Import Placement

import is a statement not a declaration

- ◆ Can be used at any time

Declarations are visible from the import onwards

Explicitly imported declarations can clash with earlier local declarations or imports

Tip: avoid import into compilation unit scope

Tip: place package references at beginning of design unit

- ◆ Fewer visibility issues
- ◆ Better readability

```
package P1;
  int c = 5;
  ...
endpackage : P1

package P2;
  int c = 8;
  ...
endpackage : P2
```

```
module mthree(...);
  logic [7:0] d = c;
  ...
  import P2::c;
  ...
endmodule
```

✗ c undefined

```
module mthree(...);
  import P1::*;
  logic [7:0] d = c;
  ...
  import P2::c;
  ...
endmodule
```

forces P1::c

✗ clash

7/31/07

SystemVerilog for Design

105

An import statement is not a declaration. You can place an import statement at any point in an interface, module, program or other package.

Imported declarations are visible from the point of the import statement onward, so you must import a declaration before you attempt to use it.

An explicitly imported declaration always clashes with any other local declaration or explicitly imported declaration of the same name.

An implicitly imported declaration is made a candidate 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.

IEEE Standard 1800 for SystemVerilog — Section 19.2.1

\$root

Reference to top-most design unit in hierarchical paths

In Verilog, local paths take precedence

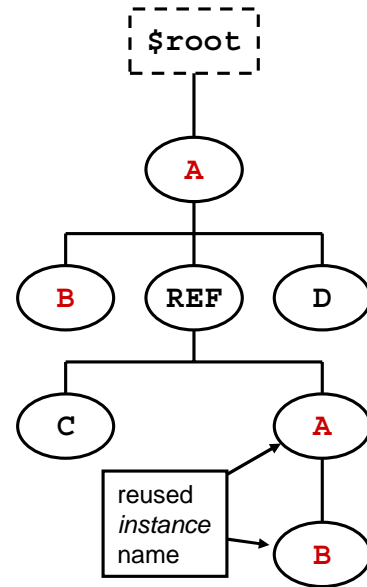
- ◆ From REF, relative path `A.B` is always path `A.REF.A.B`

`$root` can be used to start path references at top-most design unit

- ◆ From anywhere, path `$root.A.B` is always the absolute path `A.B`

Remember hierarchical paths can be used:

- ◆ To read or write remote variables or events
- ◆ In task or functions calls



Verilog does not differentiate between an absolute hierarchical path name and a relative hierarchical path name. If the path names happen to be identical, Verilog assumes you mean the relative path. To avoid such confusion, 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.

IEEE Standard 1800 for SystemVerilog — Section 19.4

Procedures and Procedural Statements

Chapter 3

July 31, 2007



Named Ends and Joins

Verilog 2001 allows `begin...end` and other code blocks to be named

SystemVerilog allows a matching name to be specified with the block end

Allowed for the following code blocks:

```
module...endmodule
begin...end
task...endtask
function...endfunction
fork...join (all forms)
primitive...endprimitive
interface...endinterface
```

Verilog2001

```
module MyTest;
initial
  begin : blockA
    ...
    begin : blockB
      ...
    end
  end
endmodule
```

SystemVerilog

```
module MyTest;
initial
  begin : blockA
    ...
    begin : blockB
      ...
    end : blockB
  end : blockA
endmodule : MyTest
```

For any named block, SystemVerilog allows you to again specify the name after the block end, join, join_any, or join_none keyword, preceded by a colon. For nested blocks, this helps document which block is ending. The name at the end is not required, but if present, is an error if it is not the correct name.

IEEE Standard 1800 for SystemVerilog — Section 10.8

For Loop Enhancements

SystemVerilog provides two **for** loop enhancements:

- ◆ You can declare loop variable within the `for` statement

Verilog

```
// loop variable declared outside loop
int i;
for (i=0; i<10; i=i+1)
...
```

SystemVerilog

```
// loop variable declared in loop
for ( int i=0; i<10; i=i+1)
...
```

- ◆ You can use multiple initial and step assignments

```
for (int i=0, int j=0; i*j <= 250; i=i+1, j=j+1)
-- 1st loop i = 0, j = 0
-- 2nd loop i = 1, j = 1
...
```

7/31/07

SystemVerilog for Design

41

An obvious benefit of allowing variables in unnamed blocks is that you can now declare for loop iterators within the `for` statement.

In fact, you can declare multiple variables in the loop, separately initialize each variable, and separately reassign each variable. Thus, SystemVerilog for loops can be a function of more than one loop variable.

IEEE Standard 1800 for SystemVerilog — Section 10.5.2

Syntax: `for (for_initialization ; expression ; for_step) statement_or_null`

Do ... While Loop

SystemVerilog adds the C-style `do...while` loop

- ◆ Condition is checked after statements execute
- ◆ Statement block executes at least once
- ◆ Makes certain loop functions easier to create

if enable false on loop entry,
count incremented

```
do
    count = count + 1;
while (enable);
```

if enable false on loop entry,
count **not** incremented

```
while (enable)
    count = count + 1;
```

The syntax of the SystemVerilog `do while` loop is identical to that of its C counterpart.

IEEE Standard 1800 for SystemVerilog — Section 10.5.1

Syntax: `do statement_or_null while (expression);`

Foreach Loop

Iterates over the elements of an array

Loop variable characteristics:

- ◆ Does not have to be declared
- ◆ Read only
- ◆ Only visible inside loop

Use multiple loop variables for multi-dimensional arrays

- ◆ Equivalent to nested loops

Useful for initializing & array processing

```
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;
```

```
int arr2d [7:0] [2:0];  
...  
foreach (arr2d [k, l])  
    arr2d[k] [l] = k*l;
```



Support

7/31/07

SystemVerilog for Design

43

The SystemVerilog foreach loop does not have a C counterpart and is syntactically and semantically different than the foreach loop construct of scripting languages such as Tcl [pronounce “T C L”]. The SystemVerilog foreach loop iterates over the elements of an array. You can declare an iterator variable for each dimension. This construct is useful for simple loop-based array initialization.

IEEE Standard 1800 for SystemVerilog — Section 10.5.3

Syntax: foreach (array_identifier [loop_variables]) statement

Break and Continue Loop Statements

SystemVerilog adds the `break` and `continue` keywords for loop constructs

◆ `break`

- ❑ Jumps to end of the loop
- ❑ Usually under conditional control

```
for (int i=0; i<=7; i=i+1)
begin
    data = {data[6:0], data[7]};
    if (data[7])
        break;
end
```

◆ `continue`

- ❑ Jumps to the next iteration of a loop
- ❑ Usually under conditional control

```
for (int i=0; i<=7; i=i+1)
begin
    if (data[i])
        continue;
    count = count + 1;
end
```

7/31/07

SystemVerilog for Design

44

You can use `break` and `continue` statements only within loop statements (`do`, `for`, `foreach`, `repeat`, `while`). With these rather abrupt flow control statements, you can produce code that may be more readable than that produced strictly with branching statements or with the Verilog `disable` statement.

The “`break`” example rotates the data vector left until the rightmost bit is 1.

The “`continue`” example counts the number of zeros in the data vector. The `for` loop iterates through the data bits and jumps over the increment statement if the current data bit is 1.

IEEE Standard 1800 for SystemVerilog — Section 10.6

Syntax: `return [expression] ; | break ; | continue ;`

iff Event Control

The **iff** keyword qualifies a procedural event control

The event expression triggers only if the **iff** condition is true

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

hardware equivalent to:

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

- ◆ Example: block executes when **a** changes and **enable** is 1

- not when **enable** changes

iff has precedence over **or**

- ◆ Can add parenthesis for clarity.

```
always @(posedge clk iff (rst == 0) or posedge rst)
    if (rst)
        data_out <= 8'h00;
    else
        data_out <= data_in;
```

You can detect any change in a variable or net by using the “at” (@) event control.

For SystemVerilog, you can add an “if and only if” (iff) qualifier expression to each event, net, or variable in the expression. The event guard applies only to the immediately preceding identifier.

In this example, the “enable” signal guards the “a” signal. Transitions of the “enable” signal have no effect. Transitions of the “a” signal trigger block execution only when the “enable” signal is 1.

IEEE Standard 1800 for SystemVerilog — Section 10.10

Tasks and Functions

Chapter 7

July 31, 2007



Verilog-2001 Tasks and Functions Review

Tasks

- ◆ any number of input, output or inout arguments – can be ANSI-C style
- ◆ May contain timing: @(...), #delay, wait

```
task write_mem (  
    input  [7:0]  waddr, wdata;  
    output [7:0]  indata, addr);  
begin  
    indata  <= wdata;  
    addr    <= waddr;  
    write   <= 1'b0;  
    #5 write <= 1'b1;  
    #10 write <= 1'b0;  
end  
endtask
```

Verilog2001

Functions

- ◆ one or more input arguments only – can be ANSI-C style
- ◆ No timing allowed – function must execute in zero-time
- ◆ Returns single value – vector, real or integer

```
function [7:0] flip(  
    input [7:0] word);  
    reg [7:0] tempword;  
begin  
    for (int i=7; i>=0; --i)  
        tempword[i] = word[7-i];  
    flip = tempword;  
end  
endfunction
```

Verilog2001

7/31/07

SystemVerilog for Design

115

Verilog tasks may have input, output and inout formal arguments, or even no arguments at all. You may declare the arguments in an ANSI-C style port list. Tasks may declare local variables. Tasks may have sequential and concurrent blocks and may contain timing controls. Tasks do not have a return type.

Verilog functions must have at least one input formal argument, and must have no output or inout arguments. You may declare the arguments in an ANSI-C style port list. Functions may declare local variables. Functions may have sequential blocks and concurrent blocks, and must not contain timing controls or schedule assignments. Verilog functions always return a single type through their name.

Verilog-2001 Static and Automatic Subprograms

Verilog-1995 subprograms are static

- ◆ Only one copy exists
- ◆ Concurrent calls cause clobber arguments and variables

Verilog-2001 added automatic subprograms

- ◆ New copy for every call
- ◆ Concurrent calls each have their own set of arguments and variables

```
function automatic [63:0] factorial;  
input [7:0] n;  
    if (n == 1)  
        factorial = 1;  
    else  
        factorial = n * factorial(n-1);  
endfunction
```

Verilog-2001

```
task automatic count_clocks;  
    input [31:0] edges;  
begin  
    repeat(edges)  
        @(posedge clk);  
end  
endtask  
  
initial  
begin  
    count_clocks(10);  
    ...  
end  
  
always @(posedge trig)  
begin  
    count_clocks(6);  
    ...  
end
```

Verilog-2001

7/31/07

SystemVerilog for Design

116

Verilog-1995 subprograms are static. One set of arguments and local variables exists that all concurrent or recursive calls share. For Verilog-2001, you can declare your subprogram to be automatic. 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.

IEEE Standard 1800 for SystemVerilog — Section 12.1

SystemVerilog Optional Begin/End and Named Ends

You can omit the `begin / end` keywords

You can repeat the subprogram name with `endtask/endfunction`

◆ SystemVerilog verifies it is the same name

```
task write_mem (  
    input  [7:0]  waddr, wdata;  
    output [7:0]  indata, addr);  
//begin  
    indata <= wdata;  
    addr   <= waddr;  
    write  <= 1'b0;  
#5 write  <= 1'b1;  
#10 write <= 1'b0;  
//end  
endtask : write_mem
```

SystemVerilog

```
function [7:0] flip  
(  
    input [7:0] word  
);  
    logic [7:0] tempword;  
//begin  
    for (int i=0;i<=7;++i)  
        tempword[i] = word[7-i];  
    flip = tempword;  
//end  
endfunction : flip
```

SystemVerilog

7/31/07

SystemVerilog for Design

117

For SystemVerilog, you do not need to group multiple subprogram statements between `begin` and `end`, or `fork` and `join`, statements.

SystemVerilog allows you to again specify the name after the `endfunction` or `endtask` keyword, preceded by a colon. This is purely for readability purposes. The name at the end is not required, but if present, is an error if it is not the correct name.

IEEE Standard 1800 for SystemVerilog — Section 12.1

Void Functions

You can declare functions that do not return a value

- ◆ Function return type `void` represents no return data

You call a `void` function as a statement

- ◆ All restrictions of functions still apply

You can “cast away” a function's return value with `void' (func)`

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

7/31/07

SystemVerilog for Design

118

Logic synthesis tools ignore delay controls in subprograms and prohibit event controls in subprograms.

For code to be synthesized, you may want to use void functions rather than tasks. As functions cannot have delays or event controls, using a void function rather than a task ensures that your subprogram to be synthesized has no such controls.

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 ())`

IEEE Standard 1800 for SystemVerilog — Section 12.3.1

Function Output Arguments

You can provide `input`, `output` and `inout` function arguments

- ◆ Argument default is `input` of type `logic`
- ◆ `output`/`inout` arguments let functions return multiple values
- ◆ `output`/`inout` arguments let void functions return values

Function becomes a general-purpose synthesizable subroutine

```
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);
```



Synthesis

void functions enforce the “no timing in tasks” synthesis rule

```
// 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);
```

7/31/07

SystemVerilog for Design

119

Input arguments are passed in when the function is called, and outputs are passed out when the function returns.

To remain compatible with Verilog, an undefined argument direction defaults to input and an undefined argument type defaults to `logic`.

Output arguments allow conventional functions to return more than one value – the return type of the function plus any output arguments.

Output arguments also allow void functions to return values.

You can use functions with output or inout arguments only within procedural blocks and not in continuous assignments.

IEEE Standard 1800 for SystemVerilog — Section 12.3

Return Statement

Executing a `return` statement immediately exits a subprogram

- ◆ Allows easier structuring of subprograms

In a function, you can include an expression to return a function value

```
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
```

SystemVerilog permits you to explicitly exit a subprogram through a return statement. For a function, the return statement can provide a value that overrides any value previously assigned to the function name variable. 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.

IEEE Standard 1800 for SystemVerilog — Sections 12.2, 12.3.1

Default Argument Values

You can specify default values for subprogram parameters

- ◆ SystemVerilog uses default values for those not passed – with rules!
- ◆ Must have unambiguous mapping of actuals to formals – use commas
- ◆ It is an error for a formal parameter to not have an unambiguous value



```
// 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
```

7/31/07

SystemVerilog for Design

121

SystemVerilog allows a subprogram declaration to specify a default value for each singular argument that is not an output port. You can use default values only with ANSI-C style port declarations. When you call the subprogram, you can omit any arguments that have default values. If necessary, you can use commas as placeholders for default arguments.

IEEE Standard 1800 for SystemVerilog — Section 12.4.3

Task/Function Optional Argument List

You can omit passing arguments if all arguments have default values

- ◆ You can even omit the parentheses !

```
// 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
```

You can omit the parentheses when calling a void function or class function method that has no arguments, or when calling a task, void function or class method for which all arguments have default values. For a directly recursive nonvoid function method call, you need to either include the parentheses, or hierarchically qualify the function called. This is to differentiate between the recursive function call and the function name variable.

IEEE Standard 1800 for SystemVerilog — Section 12.4.5

Argument Binding 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));   //
```

SystemVerilog allows you to bind subprogram argument actuals to their formals by name. This is especially useful with default values, for which you would otherwise need to use commas as position placeholders.

IEEE Standard 1800 for SystemVerilog — Section 12.4.4

Argument Passing by Value

Default argument passing

Same as Verilog-2001 – but

- ◆ Can also pass new data types: arrays, structures, unions, classes, etc!


Copies values *in* on call

- ◆ Subprogram then unaware of input changes

Copies values *out* on return

- ◆ Environment meanwhile unaware of output changes

Issue for testbench tasks

 **Error**

```
task cpu_drive_bad
(input logic [7:0] write_data,
 input logic data_read,
 output logic data_valid,
 output 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_drive_bad
...
cpu_drive_bad(8'hff, rd, valid, d_cpu);
```

input changes not detected

only last output updates seen

7/31/07

SystemVerilog for Design

124

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.

Copying large arguments is wasteful if not really needed. The conventional solution for stimulus tasks is to not declare task inputs as arguments, but rather to access module nets and variables either directly, or by using out-of-module references. This prohibits the reuse of a task definition for different sets of such nets or variables.

IEEE Standard 1800 for SystemVerilog — Section 12.4.1

Argument Passing by Reference

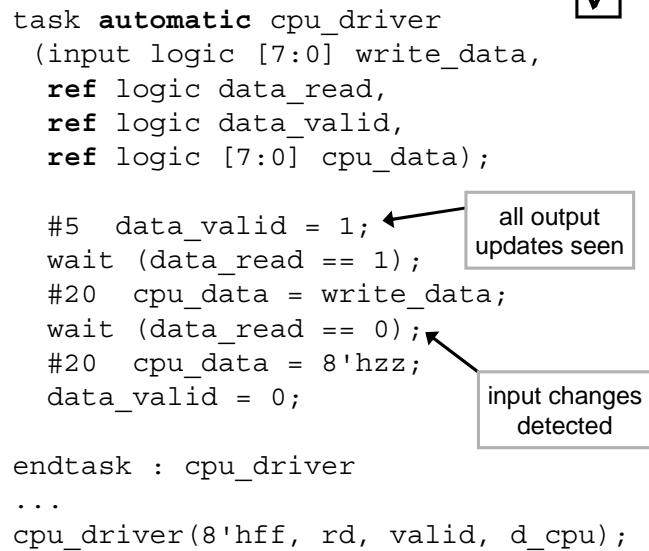
Declare reference parameters using **ref** instead of direction

- ◆ Use **ref** *instead of* parameter direction
- ◆ Only variables not nets

Creates a link to the actual arguments

- ◆ Changes in inputs can be seen in task
- ◆ Changes in outputs are immediately updated

Useful for stimulus tasks



```
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);
```

all output updates seen

input changes detected

SystemVerilog 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. The executing subprogram is aware of transitions of the object's value and the environment is immediately aware of updates the subprogram makes to the object's value. Not copying large arguments boosts simulation performance.

IEEE Standard 1800 for SystemVerilog — Section 12.4.2

More on Reference Parameters

Can use only with automatic (not static) subprograms

- ◆ Risk of outdated reference

More efficient when arguments occupy large amounts of memory

Without direction, parameters may be written in error

- ◆ Use **const ref** to prevent subprogram from modifying parameter

```
task automatic create_crc
  (ref logic [255:0] payload [1:0]
   ...);
...
payload = '0;
```

should the data be modified?

Error

```
task automatic create_crc
  (const ref logic [255:0] payload [1:0]
   ...);
...
payload = '0;
```

const parameter cannot be modified

7/31/07

SystemVerilog for Design

126

You can hierarchically access the arguments of a static subprogram throughout the simulation. The static reference would be valid only while the referenced variable exists. For this reason, SystemVerilog does not permit passing static arguments by reference. You must in general be careful to not use a reference that might not be valid, for example, in an automatic task that consumes time and returns after the calling process goes out of scope.

When the arguments are large, it can be undesirable to pass arguments by value. Pass by value creates multiple copies of the same data, and is thus inefficient. Passing large arguments by reference does not create additional copies every time the subprogram is called. This can be much more efficient.

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.

IEEE Standard 1800 for SystemVerilog — Section 12.4.2

Simple Verification Features

Chapter 2

July 31, 2007



Strings

- ◆ New data type `string`
 - Dynamic array of bytes
 - Grows/shrinks automatically to hold contents
- ◆ Indexed as a normal array
- ◆ Can be concatenated, replicated and compared
- ◆ Can be assigned a string literal
 - Some additional special characters
 - `\f` – form feed
 - `\v` – vertical tab
 - `\a` – bell
 - `\x0A` – hex ASCII

```
string message = "test ";  
  
if (pass)  
    message = {message, "passed"};  
else  
    message = {message, "failed"};  
$display("%s", message);
```

```
string repstr;  
repstr = {2{"go "}};
```

7/31/07

SystemVerilog for Verification

23

For Verilog, you can assign a character string literal to a reg vector.

SystemVerilog provides a string type, which is essentially a dynamic array of byte elements with some methods defined to manipulate it as a character string. You can still do the operations you would expect to do on a vector reg that happens to hold the ASCII representation of a character string. Just remember that indexing a string is selecting a byte instead of a single bit.

Verilog provides escaped character sequences for including newline (`\n`), tab (`\t`), backslash (`\\`) and quote (`\`) characters, and any octal representation of a character, within a string literal.

To these, SystemVerilog adds the form feed, vertical tab and bell characters, and any hexadecimal representation of a character.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 4.7

String Operators

Operator	Usage	Function	Description
<code>==</code>	<code>s1 == s1</code>	Equality	Returns 1 if strings are equal
<code>!=</code>	<code>s1 != s2</code>	Inequality	Logical negation of <code>==</code>
<code><</code> <code><=</code> <code>></code> <code>>=</code>	<code>s1 < s2</code> <code>s1 <= s2</code> <code>s1 > s2</code> <code>s1 >= s2</code>	Comparison	Return 1 if condition is true.
<code>{Str,Str}</code>	<code>s = {s1,s2,s3}</code>	Concatenation	Returns concatenated string
<code>{N{Str}}</code>	<code>s = {5{s1}}</code>	Replication	Returns string replicated N times
<code>[]</code>	<code>b = s1[index]</code>	Indexing	Returns the byte at index

All of these string operators are just Verilog operators, so they should look familiar to you.

String Access Methods

Method	Description	Syntax & Usage
<code>len()</code>	Returns length of the string (length excludes any terminating character)	function int len() str.len();
<code>putc()</code>	Replaces i^{th} character of the string with the integral value c	task putc(int i, byte c) str.putc(i, c);
<code>getc()</code>	Returns ASCII code of the i^{th} character in the string	function byte getc(int i) x = str.getc(i);
<code>compare()</code>	Compares strings with regard to lexical ordering	function int compare(string s) str.compare(s); // compares str to s
<code>icmpare()</code>	Compares strings with regards to lexical ordering and is sensitive to character case	function int icmpare(string s) str.icmpare(s); // compares str to s
<code>substr()</code>	Returns new string that is a substring formed by characters in position i through j of str .	function string substr(int i, int j) str.substr(i,j);

7/31/07

SystemVerilog for Verification

25

Pause here and on the next slide to examine these string methods.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 4.7

String Access Methods (continued)

Method	Description	Syntax & Usage
<code>toupper()</code> <code>tolower()</code>	Returns the upper/lower case of the specified string. String is unchanged.	function string toupper () x = str.toupper();
<code>atobin()</code> <code>atooct()</code> <code>atoi()</code> <code>atohex()</code>	Returns integer value corresponding to ASCII binary / octal / decimal / hexadecimal representation. Scans all leading digits and underscore (_) characters and stops when any other characters are encountered.	function int atobin() str = "21334"; int i = str.atoi(); // i = 21334;
<code>atoreal()</code>	Returns real value corresponding to ASCII decimal representation of a number	function real atoreal() str = "3.1416"; real r = str.atoreal(); // r = 3.1416;
<code>bintoa()</code> <code>octtoa()</code> <code>itoa()</code> <code>hextoa()</code>	Stores the ASCII representation of a binary / octal / decimal / hexadecimal number into a string	task hextoa (int i) i = 16'h5356; str.hextoa(i); // str = "5356";
<code>realtoa()</code>	Stores the ASCII representation of a real number into the string	task realtoa(real r) r = 3.1416; str.realtoa(r); // str = "3.1416";

Immediate Assertions

- ◆ A procedural `assert` statement

- ❑ Similar to an `if` statement

- ❑ Ignored by synthesis

```
always @(negedge clock)
    A1: assert !(wr_en && rd_en);
```

immediate assertion

- ◆ May be labeled

- ❑ Access label via `%m` formatter

- ◆ When executed, verifies a boolean expression

- ❑ *Success* if evaluates to 1

- ❑ *Failure* if evaluates to 0, Z, X

```
always @(negedge clock)
    if (wr_en && rd_en)
        $display("error");
```

similar verilog

- ◆ Reports *error* message by default upon assertion failure

- ❑ You can change this behavior

7/31/07

SystemVerilog for Verification

29

Immediate assertions are similar to if statements. The assertion succeeds if the asserted expression is unambiguously true, and fails if the assertion expression is false or unknown. Upon assertion failure, the tool must display the following minimum information:

- The file name and line number of the assertion.
- The assertion label, or if unlabelled, the scope of the assertion.
- The severity level of the assertion.
- The simulation time at which the assertion failed.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 17.2

Action Blocks


You can provide an “action” block to execute upon success, failure or both

- ◆ `else` branch executes on assertion failure

```
always @(negedge clock)
  rw_chk: assert !(wr_en && rd_en)
    $display ("%m: success");
  else
  begin
    $display("read/write fail");
    err_count++;
    -> rw_err_event;
  end
```

```
always @(negedge clock)
  assert (valid)
  // pass block omitted
  else $display("valid inactive");
```

```
always @(posedge clock)
  limit_check: assert (err_count < 14)
  else $display("errors >= 15");
```

 delays in
action
blocks
Support

7/31/07

SystemVerilog for Verification

30

You can associate an action block with the assertion. In the action block, you place code to execute on either the success or the failure of an assertion. The pass block executes if the assertion succeeds. The fail block executes if the assertion fails. You have the option to include neither, either or both. It is very common to omit the pass block and just include the fail block.

Synthesis tools ignore assertions, including their action blocks. Do not place code representing design hardware in an action block.

Note in this example the use of the “percent m” (%m) formatter to display the assertion label.

Severity Levels

The action block may override the default reporting behavior

- ◆ \$info
- ◆ \$warning
- ◆ \$error
 - Default
- ◆ \$fatal
 - Terminates simulation

Severity level is reported.

You can append additional information, using syntax identical to that for \$display

```
always @(negedge enable)
    assert (valid)
    else
    begin
        $warning("invalid enable");
        err_count++;
    end
```

```
always @(negedge clock)
    rw_chk : assert !(wr_en && rd_en)
    else
        $error("wr_en && rd_en");
```

Assertion severity is error by default. The simulator reports assertion failure as an error and “should” return from execution with a non-zero error code. You can override the default severity by using these system tasks in the action block. You may use these system tasks only in an action block. The fatal severity has the effect of terminating the simulation. These system tasks accept the same arguments you would use with the \$display system task. Just remember that prior to your own output you will also see all the other assertion failure information.

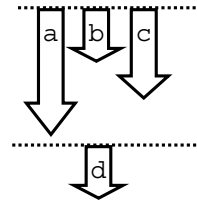
Fork Join Enhancements

Verilog-2001 `fork...join` completes when all spawned blocks complete

- ◆ This blocks further execution until the `fork...join` completes

Verilog fork-join

```
fork
  a;b;c;
join
d;
```

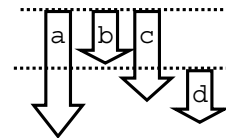


SystemVerilog adds two join variants to control when `fork...join` completes

- ◆ `join_any` completes the fork as soon as *any* of the blocks complete
 - ❑ Other blocks left running
- ◆ `join_none` completes the fork *immediately*
 - ❑ All blocks left running

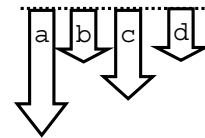
SystemVerilog

```
fork
  a;b;c;
join_any
d;
```



SystemVerilog

```
fork
  a;b;c;
join_none
d;
```



With the conventional Verilog `fork...join`, all forked processes must complete before the procedure can continue. SystemVerilog adds the `join_any` and `join_none` variants:

- With `join_any`, the parent procedure continues when any forked process completes.
- With `join_none`, the parent procedure continues without waiting for any forked process to complete. As `join_any` allows the procedure to continue while forked processes are still running, SystemVerilog adds statements to control forked processes from outside the `fork-join` statement.

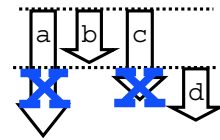
IEEE Std. 1800-2005 SystemVerilog LRM, Section 11.6

Disable and Wait Fork

The **disable fork** statement terminates all remaining forked blocks in the current scope

- ◆ Use after `join_any` to ensure only one forked block completes

```
fork
  a;b;c;
join_any
disable fork;
d;
```



The **wait fork** statement stops further execution until all forked blocks complete

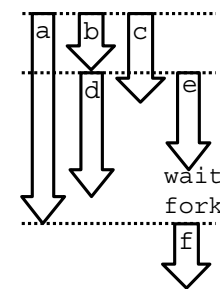
- ◆ Affects all forks in current scope

```
fork
  a;b;c;
join_any

fork
  d;
join_none

e;

wait fork;
f;
```



7/31/07

SystemVerilog for Verification

35

The **disable fork** statement terminates all sub-processes of the procedure that executes the statement.

Using a `fork...join_any` with a later `disable fork` allows a SystemVerilog testbench to spawn several concurrent processes to wait for separate events, for example separate system interrupts, and as soon as one event is received, terminate the other outstanding processes to handle the received event.

The **wait fork** statement causes the executing procedure to wait until all of its sub-processes have completed.

With the combination of `join_any`, `join_none` and `wait fork`, you can model complex concurrent and serial behavior. The lower example illustrates this. The parent procedure spawns three subprocesses, and upon the earliest to complete, spawns another subprocess and continues with its own execution. At a later point where it needs the results from all subprocesses, it waits for all to complete before continuing its own execution.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 11.8

Interfaces

Chapter 8

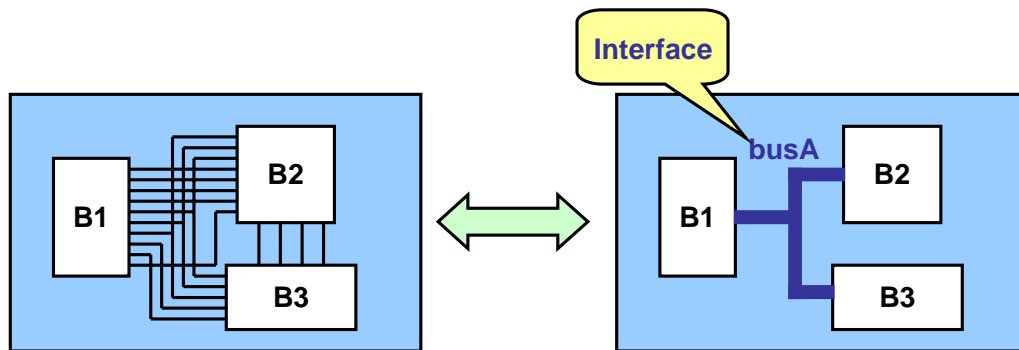
July 31, 2007



SystemVerilog Interfaces

SystemVerilog provides the interface construct to:

- ◆ Encapsulate communication between hardware blocks
- ◆ Provide a mechanism for grouping together multiple signals into a single unit that can be passed around the design hierarchy
- ◆ Enable abstraction in RTL design



7/31/07

SystemVerilog for Design

133

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.

IEEE Standard 1800 for SystemVerilog — Section 20.1.

What is an Interface?

You declare an interface as a hierarchical object, much like a module

- ◆ You instantiate it in a module, like any other hierarchical block
- ◆ You use the instance like a type in port lists and port maps

A simple interface is just a named bundle of nets or variables

- ◆ Similar to a `struct` type
- ◆ You can reference the nets or variables where needed

Interfaces can also contain module-like features for defining signal relationships

- ◆ Continuous assignments, tasks, functions, initial/always blocks, etc
- ◆ Can further instantiate interfaces
- ◆ Cannot declare or instantiate module-specific items
 - modules, primitives, specify blocks, configurations

7/31/07

SystemVerilog for Design

134

A SystemVerilog interface is a design object, like a module, but not allowed to have some module-specific things like specify blocks.

A SystemVerilog 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.

IEEE Standard 1800 for SystemVerilog — Section 20.1

Application of Interfaces

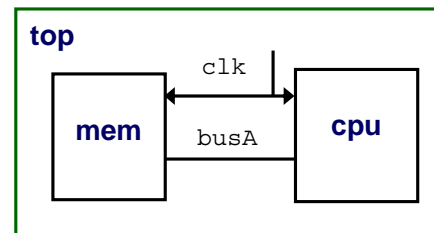
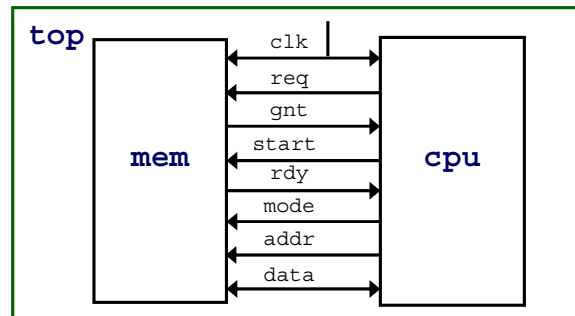
One Verilog hierarchical connection requires 5 declarations

- ◆ Two port declarations in modules `mem` and `cpu`
- ◆ Signal declaration in `top`
- ◆ Signal added to each instantiation of `mem` and `cpu`

Creating and maintaining multiple connections is tedious

With a SystemVerilog interface:

- ◆ Define a simple interface – a named bundle of signals
- ◆ Use it as a port type throughout the design



7/31/07

SystemVerilog for Design

135

To add a single connection between two modules in one level of hierarchy, you:

- add a port to each module,
- declare a net or variable in the parent module...
- and map the net or variable to the port of each submodule instance.

A standard communications bus may be composed of many signals, which may be connected to multiple modules. Creating and maintaining such a bus connection can be difficult in Verilog.

SystemVerilog supports interfaces. In its simplest form, an interface is a separately declared and named group of signals. All connections associated with a specific interface are declared and maintained in one place.

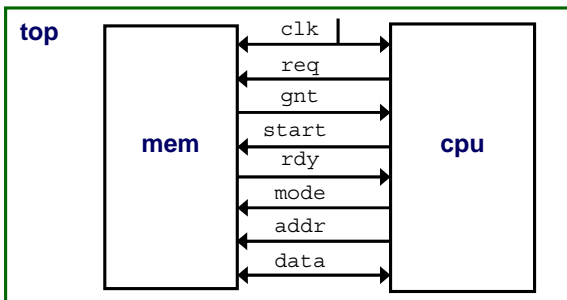
More complex interfaces can have their own ports through which they can share external nets and variables with the connected modules. They can be parameterized to, for example, allow different bus widths, and can contain subprograms to, among other things, convert highly abstract data communications to low level signal transitions.

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
```



7/31/07

SystemVerilog for Design

136

Declaration of the bus ports and signals is time-consuming and error-prone. Maintaining bus connections is also difficult, particularly if the bus is connected to many separate modules. To modify one signal connection requires you to:

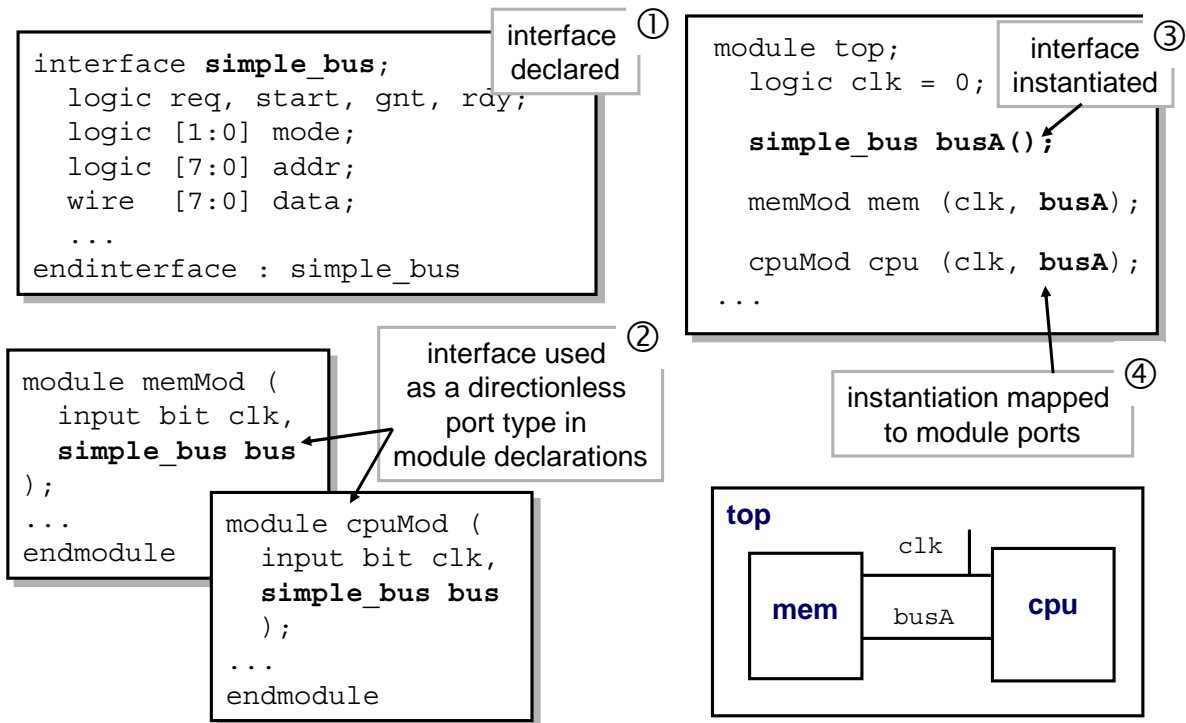
- modify the port list of every module that connects to the bus,
- modify every instantiation of the changed modules...
- and modify signal declarations in the parent module.

A typical module may have many port connections, representing perhaps several buses. In such cases it can be difficult to remember which ports belong to which bus.

In this example, a connection between the “mem” and “cpu” modules is composed of the “req”, “gnt”, “start”, “rdy”, “mode”, “addr” and “data” signals. Each of these connections is declared in the “mem”, “cpu” and “top” modules.

IEEE Standard 1800 for SystemVerilog — Section 20.2.1

Same Example Using Interfaces



7/31/07

SystemVerilog for Design

137

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.

This example defines the “simple_bus” interface containing the “req”, “gnt”, “start”, “rdy”, “mode”, “addr” and “data” signals. The “mem” module and “cpu” module each have a port of that “simple_bus” type. The parent module instantiates the “simple_bus” interface and the submodules, mapping the submodule interface port to the interface instance.

All of the signals for a given bus type are declared once in one easily edited interface definition.

IEEE Standard 1800 for SystemVerilog — Section 20.2.2

Interfaces in Module Instantiations

SystemVerilog port mapping rules also apply to interfaces

You can map by position or name

You can use implicit (.name) mapping if instance name matches port name

You can use implicit (.*) mapping if all names match

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

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

```
module top;  
    logic clk = 0;  
  
    simple_bus bus();  
    simple_bus if_bus();  
  
    modone mem_pos (clk, bus);  
    modone mem_nam(.clk(clk), .busA(bus));  
    modtwo mem_dotnam(.clk, .if_bus);  
    modtwo mem_dotstr(.*);  
...
```

The diagram illustrates four different port mapping styles in SystemVerilog. The 'positional' label points to the instance `mem_pos` where ports are connected by position. The 'named' label points to the instance `mem_nam` where ports are connected by name using dot notation. The 'implicit' label points to two instances: `mem_dotnam` (using implicit name mapping `.if_bus`) and `mem_dotstr` (using implicit all-name mapping `.*`).

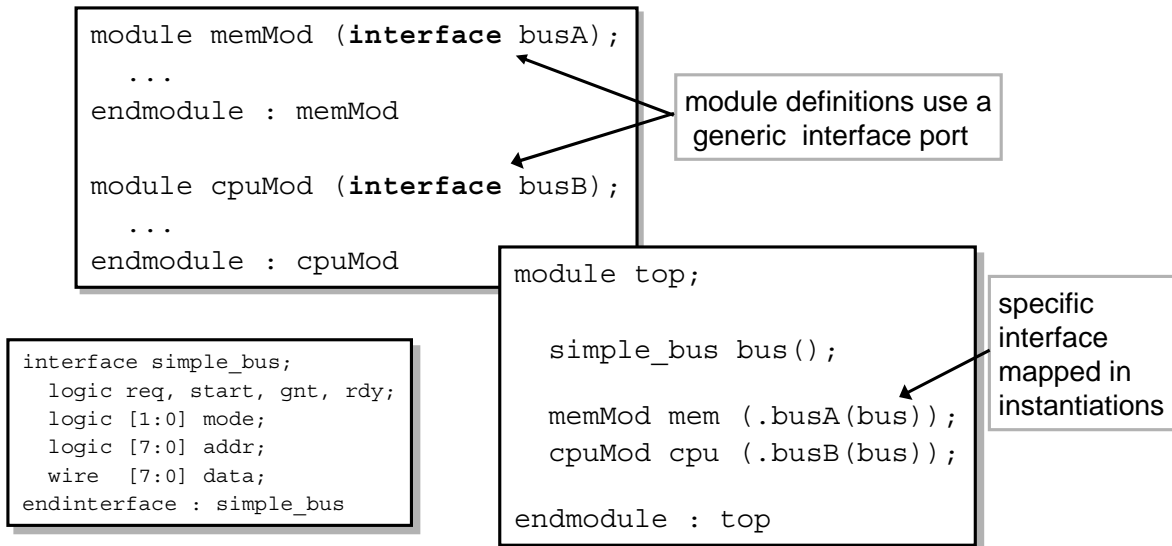
The port map syntax for ports of an interface type is identical to that for ports of any other type. You can use ordered connections, named connections, “dot name” (.portname) connections, and “dot star” (.*) connections.

IEEE Standard 1800 for SystemVerilog — Section 20.1

Generic Interfaces

You can declare a module port of a generic interface type

- ◆ This defers actual interface selection until module instantiation



7/31/07

SystemVerilog for Design

139

You can use the interface keyword rather than a specific interface type in your module declaration's list of ports. This serves as a placeholder for an interface type you select upon instantiating the module, so that a module definition can access, for example, interface tasks that are implemented differently in different interface definitions. This unspecified interface is called a "generic interface reference". You can use it only with the Verilog named port connections.

IEEE Standard 1800 for SystemVerilog — Section 20.2.3

Interface Port References

For a module with a *port* of the interface type, you access interface objects through the port name

- ◆ This module has a *port* of the `simple_bus` interface type

`<port_name>.<interface_item>`

```
module memMod ( input bit clk, simple_bus bus );
    reg [31:0] mem [0:31];
    wire read, write;
    assign read  = (bus.gnt && (bus.mode == 0) );
    assign write = (bus.gnt && (bus.mode == 1) );
    always @(posedge clk)
        if (read)
            bus.data = mem[bus.addr];
        else if (write)
            mem[bus.addr] = bus.data;
endmodule
```

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

From within a module which has a port of an interface type, you reference the contents of the interface hierarchically through the port name (`port_name.interface_signal_name`).

IEEE Standard 1800 for SystemVerilog — Section 20.1

Interface Instance References

For a module with an *instance* of the interface type, you access interface objects through the instance name

- ◆ This module has an *instance* of the `simple_bus` interface type

`<instance_name>.<interface_item>`

```
module top;
    logic clk;

    simple_bus busA();

    memMod mem (clk, busA);
    cpuMod cpu (clk, busA);
    ...
    always @(busA.rdy)
        if (busA.mode == 0)
            ...
endmodule : top
```

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

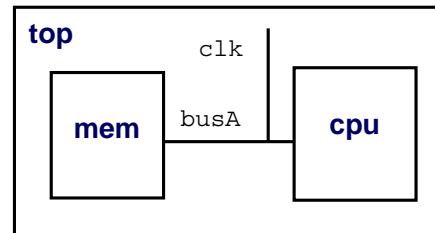
From within a module which has an instance of an interface type, you reference the contents of the interface hierarchically through the instance name (`instance_name.interface_signal_name`).

IEEE Standard 1800 for SystemVerilog — Section 20.1

Interface Ports

An interface can have its own ports

- ◆ Connect it like any module port
- ◆ Use it to share an external signal



```
module top;
  logic clk = 0;

  simple_bus busA(clk);

  memMod mem (busA);

  cpuMod cpu (busA);
  ...
endmodule
```

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

```
module memMod (
  simple_bus bus);
  ...
endmodule
```

```
module cpuMod (
  simple_bus bus);
  ...
endmodule
```

7/31/07

SystemVerilog for Design

142

Modules connected through their interface ports to an interface instance share the interface nets and variables. How would the modules share an external net or variable? One way is for both modules to each have a port connected to the external signal. You may find it more practical to connect the external net or variable to a port of the interface, thus permitting the interface itself as well as the connected modules to share the external net or variable.

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.

IEEE Standard 1800 for SystemVerilog — Section 20.3

Interface Port Applications

```
interface pci_if (input clk);
    logic [7:0] address;
endinterface

interface ahb_if (input clk);
    logic [7:0]
    wire [7:0]
    ...
endinterface

module top;
    logic clk, clk1 = 0;

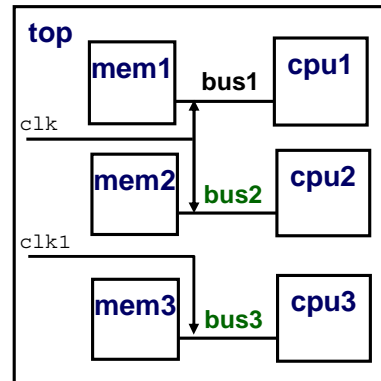
    pci_if bus1(clk);
    memMod mem1 (bus1);
    cpuMod cpu1 (bus1);

    ahb_if bus2(clk);
    memMod mem2 (bus2);
    cpuMod cpu2 (bus2);

    ahb_if bus3(clk1);
    memMod mem3 (bus3);
    cpuMod cpu3 (bus3);

endmodule : top
```

- ◆ Common signals for instances of different interface definitions
- ◆ Different signals for different instances of one interface definition



7/31/07

SystemVerilog for Design

143

Connections from mem1 to cpu1, and from mem2 to cpu2 utilize different interface types, but must be synchronized to the same clock signal. This example passes the common “clk” signal to the input port of both the “bus1” and “bus2” interface instances.

The connection from mem3 to cpu3 uses the second interface type again, but with a completely different clock signal.

IEEE Standard 1800 for SystemVerilog — Section 20.3

Parameterized Interfaces

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  
    .....
```

Parameterization of an interface is exactly like parameterization of a module.

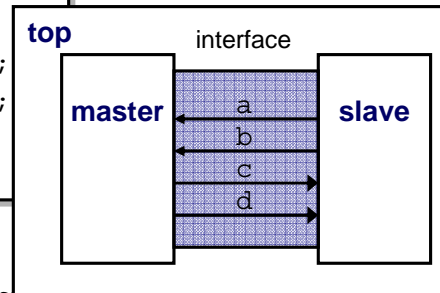
IEEE Standard 1800 for SystemVerilog — Section 20.7

Interface Modports

Modports create different views of an interface

- ◆ Specify a subset of interface signals accessible to a module
- ◆ Specify direction information for those signals

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



You can specify a modport view for a specific module in two ways.

- ◆ In the module definition
- ◆ In the module instantiation

7/31/07

SystemVerilog for Design

145

A simple interface does not contain any direction information for the interface signals. Therefore a module to which the interface is connected could read or write any interface signal. Typically a bus connection between two modules will have a direction associated with it, there will be a transmitter or master to send data over the connection, and a receiver or slave to read the information. This gives us two views of the bus connection. You can define these views using the modport keyword.

An interface can have any number of modports, and each define a different view of the interface contents. The view can make available a subset of the interface contents and restrict how connected modules use those contents.

This example provides a modport through which a module reads signals “a” and “b” and writes signals “c” and “d”, and provides another modport for exactly the opposite use.

A module definition can specify a modport in its port list, or a parent module can specify a modport when making the port connection.

IEEE Standard 1800 for SystemVerilog — Section 20.4

Modport Selection: Module Definition

You can select the interface modport in your module definition port list

- ◆ Module mmod declares port mbus of type mod_if.master
- ◆ Module smod declares port sbus of type mod_if.slave

Testbench instantiates mod_if instance busA

Testbench maps instance mmod1 port mbus to interface mod_if instance busA

```
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
```

7/31/07

SystemVerilog for Design

146

This example defines an interface type with master and slave modports.

The definition of the master module specifies that its port of the interface type must access only the “master” modport.

The definition of the slave module specifies that its port of the interface type must access only the “slave” modport.

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.

IEEE Standard 1800 for SystemVerilog — Section 20.4.1

Modport Selection: Module Instantiation

You can select the interface modport in your module instance port map. Module definitions declare ports of type `mod_if` interface.

Testbench instantiates `mod_if` instance `busB`.

Testbench maps instance `mmod1` port `mbus` to interface instance `busB` modport `master` etc.

```
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 mbus);
endmodule

module smod (mod_if sbus);
endmodule
```

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

7/31/07

SystemVerilog for Design

147

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.

IEEE Standard 1800 for SystemVerilog — Section 20.4.2

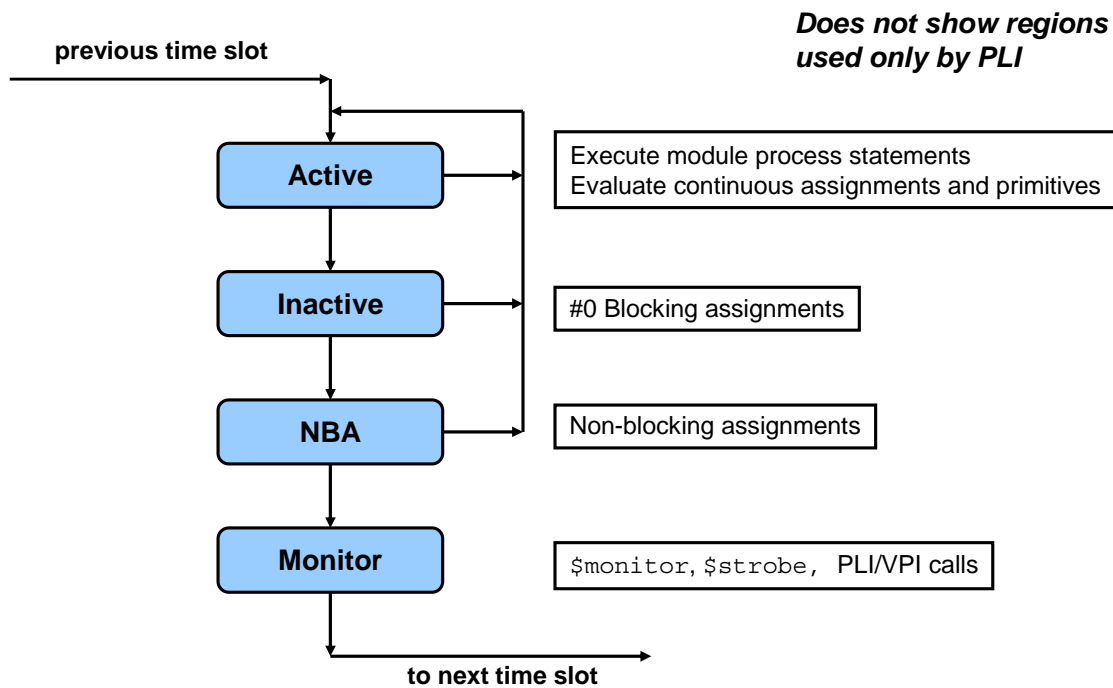
Verification Blocks

Chapter 3

July 31, 2007



Simplified Verilog Event Scheduler



7/31/07

SystemVerilog for Verification

43

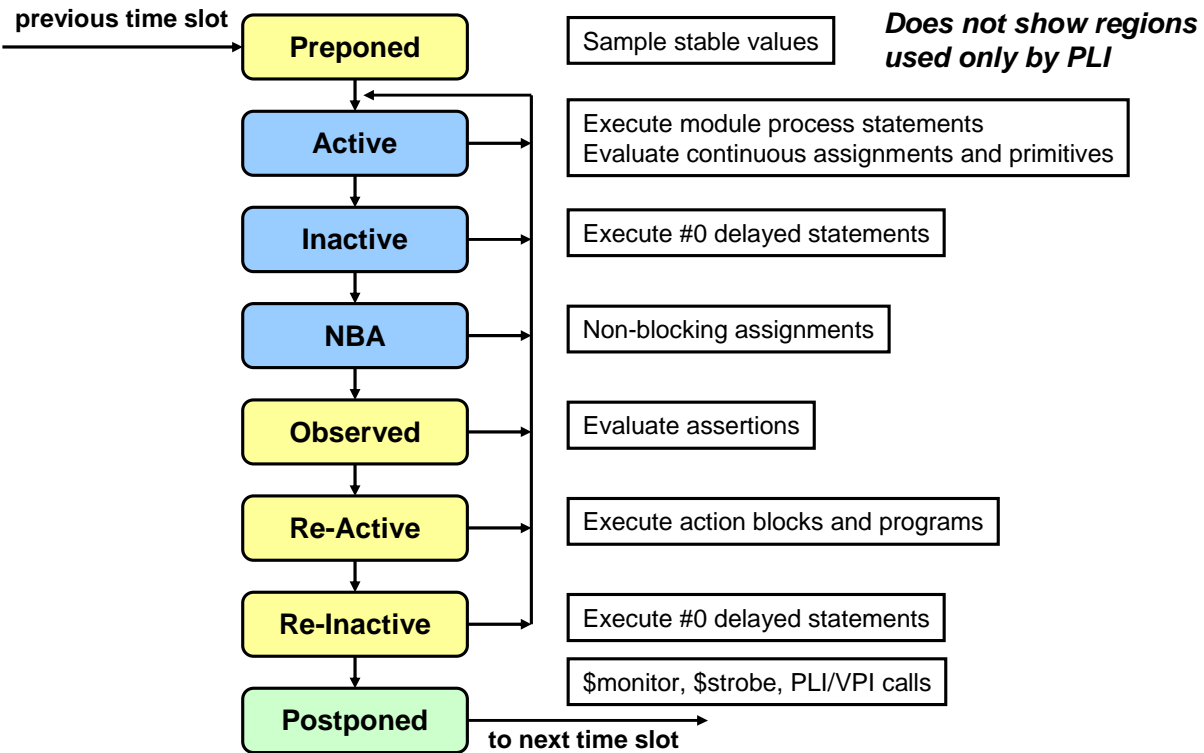
A simulation timeslot is divided into ordered regions to provide a predictable interaction between design constructs.

The Verilog event schedule has four regions for each simulation time:

- The Active region is for executing process statements.
- The Inactive region is for executing process statements postponed with a “pound zero” (#0) procedural delay.
- The NBA region is for updating non-blocking assignments.
- The Monitor region is for executing \$monitor and \$strobe and for calling user routines registered for execution during this read-only region. You cannot within this region create additional events.

The first three of these regions are iterative – they can schedule events that require return to the Active region. When no more events exist for the current simulation time, the simulator executes Monitor statements and then advances simulation time to the next time for which events are scheduled. The simulation terminates when no such future events exist.

SystemVerilog Event Scheduler



7/31/07

SystemVerilog for Verification

44

SystemVerilog adds regions to provide a predictable interaction between assertions, design code and testbench code. It adds the Observed region in which to evaluate assertions, and the Re-Active and Re-Inactive regions in which to execute assertion action blocks and testbench programs.

The SystemVerilog event schedule has 13 regions for each simulation time. Some regions used only by the PLI are not shown on the slide and not described here:

- The Preponed region is for sampling signal values before anything in the time slice changes their values.
- The Active region is for executing interface and module process statements. It is the same as for Verilog.
- The Inactive region is for executing interface and module process statements postponed with a “pound zero” (#0) procedural delay. It is the same as for Verilog.
- The NBA region is for updating non-blocking assignments. It is the same as for Verilog.
- The Observed region is for assertion evaluation.
- The Reactive region is for executing program process statements and assertion action blocks.
- The Re-inactive region is for executing program process statements postponed with a “pound zero” (#0) procedural delay.
- The Postponed region is for system tasks that record signal values at the end of the time slice.

The Active through Re-inactive regions are iterative. Most of these regions can schedule events that require return to the Active region.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 9.3

Final Blocks

A `final` procedural block executes at the end of simulation.

- ◆ Executes after explicit or implicit call to `$finish`
- ◆ Execute exactly once
 - Similar to an initial block, but execute at the *end* of simulation
- ◆ Cannot consume simulation time (no blocking delays)
- ◆ You can use to calculate and display simulation statistics

```
final begin
    if (timeout_error)
        $display ("ERROR: %0t: Test Timed Out", $time);
    else
        $display ("INFO: %0t: Test Complete", $time);
        $display("Error Count: %d", error_count);
        $display("Fifo Overflow Count: %d", fifo_overflow);
end
```

A final block cannot consume simulation time, but is otherwise similar to an initial block. You can place a final block anywhere you can place an initial block. SystemVerilog executes all final blocks exactly once at the end of simulation, in an unspecified but repeatable order.

SystemVerilog executes the final blocks upon encountering `$finish` or upon running out of events to process. Any final block that itself calls `$finish` or calls a user-defined system task or function that terminates the simulation (invokes `tf_dofinish()` or `vpi_control(vpi_finish...)`), aborts execution of the final blocks. After SystemVerilog completes or terminates execution of the final blocks, it calls any PLI routines you have scheduled for execution at the end of the simulation.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 10.7

Programs

Programs are somewhat similar to modules, but:

- ◆ Are intended for the testbench
 - ❑ To clearly separate design and test
 - ❑ Cannot have an `always` block
 - ❑ Cannot instantiate hierarchy
 - ❑ Other restrictions – see next slide
- ◆ Execute in the *reactive* region of the time slot
 - ❑ To avoid user-induced races
- ◆ Can use `$exit` system task

```
program memtest (  
    output wire [7:0] data;  
    output bit [4:0] addr;  
    output bit read, write);  
    ...  
    initial begin  
        ...  
    end  
endprogram : memtest
```

```
module top;  
    wire [7:0] data;  
    wire [4:0] address;  
    wire      read, write;  
    memtest test (.*);  
    memory mem8x32 (.*);  
endmodule : top
```

7/31/07

SystemVerilog for Verification

46

A program cannot contain hierarchy and cannot have an `always` block, but is otherwise similar to a module, with some additional restrictions. It is intended to remove testbench stimulus from the module construct and instead execute those statements in the REACTIVE region separately from the execution and update of design code. To further reduce the potential for user-induced clock/data races, the program construct requires the user to make only blocking assignments to program variables and only non-blocking assignments to module and interface variables.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 16

Program Block Assignments

Compilation error reported if these rules are not followed:

- ◆ Use blocking assignment (=) to update program variables
 - Any declared local to a program
- ◆ Use non-blocking assignment (<=) to update non-program variables.

```
module design (logic dA);  
    logic dB;  
    int    dC;  
endmodule : design
```

```
module top;  
    logic top_A;  
    design d1 (top_A);  
    test   t1 (top_A);  
endmodule: top
```

```
program test (logic tA);  
    logic temp;  
    int number;  
    initial begin  
        tA    <= 0; // ERROR  
        temp  <= 1; // ERROR  
        number = 15; // okay  
        top.d1.dB = 0; // ERROR  
        top.d1.dC <= 5; // okay  
    end  
endprogram : test
```

Within a program, you can make only blocking assignments to program variables and only non-blocking assignments to module and interface variables. This restriction ensures that all program variables are updated in the REACTIVE region of the SystemVerilog event scheduler, and then all non-program variables are updated in the next NBA region, thus avoiding race conditions in the application of stimuli and sampling of design response.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 16.2

Referencing Program Block Declarations

Only programs can reference program variables.

Nothing outside a program can reference program variables.

```
program test1 (output reg t1A);  
    bit [3:0] addr;  
    initial repeat (20)  
        #5 addr++;  
    ...  
endprogram : test1
```

```
program test2 (output reg t2A);  
    wire [3:0] save;  
    // GOOD program block variable access  
    assign save = top.t1.addr;  
    ...  
endprogram : test2
```

```
module top;  
    wire top_A, temp_A;  
    design d1 (top_A); //design  
    test1 t1 (top_A); //program 1  
    test2 t2 (top_A); //program 2  
  
    // ILLEGAL program variable access  
    assign temp_A = t1.addr[3];  
endmodule: top
```

A SystemVerilog philosophy is that the testbench should know about the design, but the design should not be aware of the testbench. Programs can access variables anywhere in the design or testbench. Interfaces and modules cannot access program variables.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 16-2

“References to program variables from outside any program block shall be an error.”

Clocking Blocks

- ◆ Specifies timing relative to a specific clock for a set of signals
 - Sample skew and drive skew
- ◆ Specifies timing – does not *declare* signals
- ◆ Separates signal timing from signal function
- ◆ Helps avoid user-induced clock/data races
- ◆ You can write tests in terms of cycles and transactions

```
clocking cb @(posedge clk);  
    input dout;  
    output reset, enable, data;  
endclocking
```

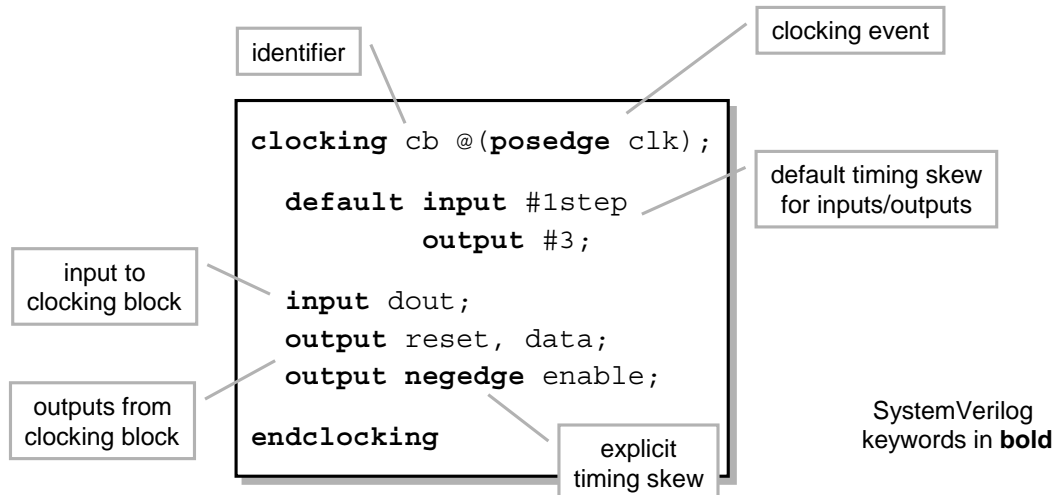
A clocking block defines a set of timing, relative to a specified clock, for a set of signals. Any number of signals may appear in any number of clocking blocks. You declare a clocking block as an interface, module or program item. You can declare one clocking block in any scope to be the default clocking block for that scope. You apply the timing by referencing the signal hierarchically through the clocking block name.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 15

Clocking Block Syntax

Input and output directions are relative to the clocking block

- ◆ Input *to* the block and output *from* the block



7/31/07

SystemVerilog for Verification

53

A clocking block is both a declaration and instance of that declaration. You do not instantiate a clocking block.

Pause here for a moment, and examine the clocking block syntax.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 15.2

```
clocking_declaration ::=  
    [ default ] clocking [ clocking_identifier ] clocking_event ;  
    { clocking_item }  
    endclocking [ : clocking_identifier ]
```

Input and Output Skews

- ◆ Input skew designates sample time for signal *before* the clocking event
- ◆ Output skew designates driving time for signal *after* the clocking event
- ◆ You can specify skew:
 - ❑ For all inputs/outputs with a default
 - ❑ Explicitly in the signal identifier, overriding the default
 - ❑ As an edge, number or time literal
 - Number uses current timescale

```
clocking cb @(posedge clk);  
    default input #1step  
            output #3;  
    input dout;  
    output reset, data;  
    output negedge enable;  
endclocking
```

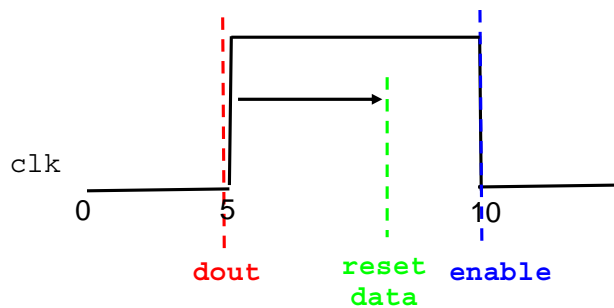
A skew must be a constant expression, which may be a parameter.

An input skew specifies the sampling time before the clock event. The default input skew is 1step, which samples the input during the preceding Postponed region. An explicit #0 skew samples the input during the Observed region.

An output skew specifies the drive time after the clock event. The default output skew is #0, which drives the output during the NBA region.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 15.3

Clocking Skew Example



```
clocking cb @(posedge clk);
  default input #1step
           output #3;
  input dout;
  output reset, data;
  output negedge enable;
endclocking
```

Assume clock period of 10ns, timescale of 1ns and timeprecision of 10fs

dout	sampled #1step before (posedge clk) = 10fs before 5ns
reset, data	driven #3 after (posedge clk) = 8ns
enable	driven on (negedge clk) = 10ns

For this example:

- The positive edge of the clock occurs at 5ns.
- Inputs are sampled one simulation time step, in this case 10fs, before the clock event.
- Most outputs are driven 3ns after the clock event, in this case, at 8ns.
- The “enable” signal is driven on the negative edge of the clock after the clock event, in this case, at 10ns.

Multiple and Default Clocking Blocks

You can define multiple clocking blocks in a scope

- ◆ For multiple clocks, different signals, or simply different timing
- ◆ Only one block in a scope can be the default clocking block
 - ❑ Add `default` to the beginning of the clocking block declaration
 - ❑ Or use a `default` statement separate from the declaration

```
clocking cb1 @(posedge clk1);
  default input #2 output #3;
  input cdout;
  output reset, data;
endclocking

clocking si1 @(posedge clk2);
  input #2 ip1;
  output #2 op1;
endclocking

default clocking si2 @(posedge clk2);
  input #2 ip2;
  output #5 op2;
endclocking
```

```
clocking si2 @(posedge clk2);
  input #2 ip2;
  output #5 op2;
endclocking

default clocking si2;
```

7/31/07

SystemVerilog for Verification

56

You can use either of two methods to declare a clocking block to be the default clocking block of the scope:

- You can directly declare a default clocking block.
- You can declare an already declared clocking block to be the default clocking block.

Only one default clocking block can exist in a module, interface or program.

A default clocking block is valid only in the scope declaring it and any nested declarations.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 15.11

Accessing a Clocking Block Signal

To apply timing, access the signal through the clocking block

- ◆ `cb.enable <= 1;`
 - Must use only non-blocking assignments to clocking block signals
- ◆ `dreg <= cb.dout;`
- ◆ `@(cb) // @(posedge clk)`
- ◆ `@(cb.dout) // sampled values`

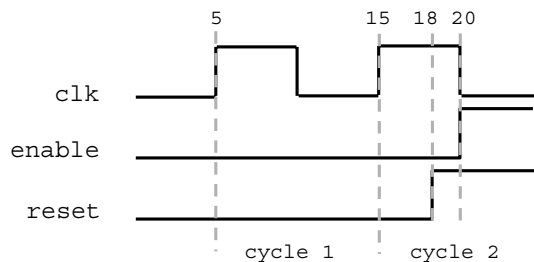
```
bit reset, enable;
clocking cb @(posedge clk);
    default input #1step output #3;
    input dout;
    output reset, data;
    output negedge enable;
endclocking

initial begin
    @(cb);
    cb.reset <= 1'b1;
    cb.enable <= 1'b1;
    @(cb.dout);
```



Beware

Timing applies only when signal accessed through clocking block



7/31/07

SystemVerilog for Verification

57

To apply the timing of a clocking block, you hierarchically reference the signal through the clocking block name.

To synchronize a process to a clocking block, you can either:

- Directly use the clocking block name in a sensitivity list. This triggers the sensitive process on the clocking block clock event.
- Use clocking block signal inputs, or slices of them, in a sensitivity list. This triggers the sensitive process on the clocking block signal event.

IEEE Std. 1800-2005 SystemVerilog LRM, Sections 15.9, 15.13

Cycle Delay and Synchronous Drive

##N is delay by N number of clocking cycles

- ◆ N is *positive* expression calculated at run time
- ◆ When used in a statement making an assignment to a clocking block signal
 - ❑ Refers to specified block
 - ❑ `##1 cb.data <= d_in;`
 - ❑ `cb.data <= ##1 d_in;`
- ◆ When used anywhere else
 - ❑ Refers to default clocking block (which must exist)
 - ❑ `##1; cb.data <= d_in;`



Beware

```
default clocking def @(negedge clk);
endclocking
clocking cb @(posedge clk);
    input #1step dout;
    output #3 reset, data;
endclocking

// Wait 2 def cycles
@(def);
@(def);
// Wait 2 default clocking cycles
##2;

// Drive data after 1 cb cycle
##1 cb.data <= 2'b01;

// drive data after 1 def cycle
##1; cb.data <= 2'b10;

// Drive with current tmpdat value
// after 3 cb cycles
cb.data <= ##3 tmpdat;
```

7/31/07

SystemVerilog for Verification

58

The semantics of the cycle delay (**##N**) operator differ depending upon how you use it:

- When used as a procedural or intra-assignment delay in an assignment to a clocking block signal, it refers to the clock event of the specified clocking block. These assignments must use the non-blocking operator.
- When not used as a procedural or intra-assignment delay in an assignment to a clocking block signal, it refers to the default clocking block. For this situation, the compiler shall issue an error if no default clocking block has been declared.

IEEE Std. 1800-2005 SystemVerilog LRM, Sections 15.10, 15.14

```
clocking_drive ::=
    clockvar_expression <= [ cycle_delay ] expression
  | cycle_delay clockvar_expression <= expression
```


Hierarchical Expressions

A clocking block signal cannot *be* a hierarchical expression

You can *associate* a clocking block signal with a hierarchical expression

```
clocking cb @(posedge clk);  
    default input #1step  
            output #3;  
    input dout;  
    //data associated with hierarchical expression  
    output reset, data = top.dut.data; // Valid  
    // output top.dut.data;                // Invalid  
    output negedge enable;  
endclocking
```

You cannot within a clocking block directly declare a hierarchical signal. You must instead declare a clocking block virtual signal and associate it with an expression that references the hierarchical signal. The semantics of this association are much like those for a port connection – you can use any expression, including slices, concatenations and part selects, that you can legally connect to a port of that direction.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 15.4

Virtual Interfaces

```
virtual [interface] interface_type identifier;  
identifier = interface_instance;
```

A virtual interface is a *variable* that represents an interface instance

- ◆ Your test code assigns actual interface instances to the interface variable at various time throughout the test
 - ❑ assignment (=) of interface instance, interface variable, or `null`
 - ❑ comparison (==, !=) to interface instance, interface variable, or `null`
- ◆ You test code can pass interface variables to tasks and functions
- ◆ Your test code procedurally accesses methods and variables (but not nets) of that interface instance through the interface variable
 - ❑ Each interface instance drives its own nets, through clocking blocks or continuous assignments, that can reflect variable changes made by test
- ◆ Use of virtual interfaces promotes code reuse (write once, use often)

7/31/07

SystemVerilog for Verification

72

A virtual interface is an interface variable that at various times throughout the test can represent different interface instances. By iterating through an array of interface variables and passing each element in turn to a test task, your one testbench task can test multiple instances of similar design blocks, each through its own interface instance.

You can declare virtual interfaces in modules, programs, classes and packages. You can pass virtual interfaces by value, but not by reference, to tasks and functions.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 20.8

Virtual Interface Example

```
// Interface
interface inv_if;
    var logic in_var, out_var;
endinterface
```

```
// Inverter
module inv(interface intf);
    assign intf.out_var = !intf.in_var;
endmodule
```

```
module top;
    inv_if intf1(), intf2();
    inv inv1(.intf(intf1));
    inv inv2(.intf(intf2));
    virtual inv_if intf;
    task check (exp);
        ...
    endtask
    task do_test;
        intf.in_var = 0;
        #1 check(1);
        intf.in_var = 1;
        #1 check(0);
    endtask
    initial begin
        intf = intf1; do_test;
        intf = intf2; do_test;
        $display("TEST DONE");
        $finish(0);
    end
endmodule
```

This example defines an inverter interface and an inverter module and a test module. The test module declares two interface instances and two inverter instances and connects each inverter to its respective interface. It declares an interface variable and defines a test task to test the inverter connected to the interface that the interface variable currently represents. The test block, in turn, assigns each interface to the interface variable, and calls the test task to test the inverter connected to that interface.

Interprocess Synchronization

Chapter 12

July 31, 2007



Non-Blocking Event Trigger (continued)

You use the non-blocking event trigger for the same reason you use the non-blocking assignment – to prevent races between processes.

- ◆ Non-blocking event triggers occur in the NBA region:

```
module test();
  event e;
  integer i = 0;

  always @e
    $display("i is %d",i);

  initial
  begin
    i <= 1;
    -> e; // blocking
  end
endmodule
```

```
sim> i is 0
```

```
module test();
  event e;
  integer i = 0;

  always @e
    $display("i is %d",i);

  initial
  begin
    i <= 1;
    ->> e; //non-blocking
  end
endmodule
```

```
sim> i is 1
```

The non-blocking event trigger serves the same general purpose as the non-blocking assignment – to prevent a race between a block that triggers the event and a block that waits for the event in the same delta cycle.

Persistent Event Trigger

A process can wait for the `triggered` property of the event

- ◆ The `triggered` state persists to the end of the time step

“hangs”

```
module test;
  event e1, e2;
  initial
  begin
    fork
      @e1;
      -> e1;
      -> e2;
      @e2;
    join
    $display("fork over!");
  end
endmodule
```

completes

```
module test;
  event e1, e2;
  initial
  begin
    fork
      @e1;
      ->> e1;
      -> e2;
      wait(e2.triggered);
    join
    $display("fork over!");
  end
endmodule
```

7/31/07

SystemVerilog for Verification

233

An event is not persistent. If a process “waits” for an event after it occurs, the process can go on waiting forever.

The first example illustrates this. The forked blocks can be scheduled in any order. It is very likely that one of the event controls is too late – its associated event has already occurred.

The second example utilizes the non-blocking event trigger, which schedules the event for the NBA region, after the event control has had a chance to “wait” for that event. The example also illustrates use of the `triggered` event property, which persists to the end of the time step. If the “e2” event has already occurred when the “wait” statement executes, the `triggered` property is still true.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 14.5.4

Mailboxes

- ◆ Mailboxes are a message-based synchronization mechanism
- ◆ Used for passing messages where order is important (FIFO)
 - ❑ Any process puts mail into the mailbox
 - Blocking: `put(anytype message)`
 - Non-Blocking: `int try_put(anytype message)`
 - ❑ Any process gets mail from the mailbox
 - Blocking: `get(ref variable)`
 - Non-blocking: `int try_get(ref variable)`
 - User is responsible for type compatibility!
 - ❑ Any process can alternatively use `peek()` and `try_peek()` to check for mail without removing it

```
mailbox hugebox = new; // mailbox of unlimited size
mailbox fourbox;
fourbox = new(4);      // mailbox of size 4
```

7/31/07

SystemVerilog for Verification

238

Mailboxes are a class-like message-based process synchronization mechanism.

A mailbox is conceptually like a mailbox, with a delivery slot on one side, and a door on the other side for retrieving mail.

When you create the mailbox, you pass to its constructor the maximum capacity of the mailbox. If you do not pass a number, the mailbox has unlimited capacity and can never be full.

A mailbox by default accepts messages of any type. Later slides show you how to restrict a mailbox to just one type.

Any process can place messages in the mailbox. The blocking `put()` task attempts to put a message in the mailbox and blocks if the mailbox is full. The non-blocking `try_put()` function attempts to put a message in the mailbox and returns 0 if the mailbox is full.

Messages become available for retrieval in the same order they are placed in the mailbox.

Any process can retrieve messages from the mailbox. The blocking `get()` task attempts to get a message from the mailbox and blocks if the mailbox is empty. The non-blocking `try_get()` function attempts to get a message from the mailbox and returns 0 if the mailbox is empty.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 14.3

Mailbox Methods

Method	Description	Syntax
<code>new()</code>	Mailbox constructor which optionally specifies maximum size	<code>function new(int bound = 0);</code> Note by default no maximum size.
<code>num()</code>	Returns the number of messages currently in the mailbox	<code>function int num();</code>
<code>put()</code>	Places a message in the mailbox – blocks if mailbox full	<code>task put (<message>;</code>
<code>try_put()</code>	Places a message in a mailbox – returns 0 if mailbox full	<code>function int try_put (<message>;</code>
<code>get()</code>	Retrieves a message from the mailbox – blocks if mailbox empty – error if type mismatch	<code>task get (ref <variable>;</code>
<code>try_get()</code>	Retrieves a message from the mailbox – returns +int if successful – returns 0 if empty – returns -int if type mismatch	<code>function int try_get (ref <variable>;</code>
<code>peek()</code>	<i>Copies</i> a message from the mailbox – blocks if mailbox empty – error if type mismatch	<code>task peek (ref <variable>;</code>
<code>try_peek()</code>	<i>Copies</i> a message from the mailbox – returns +int if successful – returns 0 if empty – returns -int if type mismatch	<code>function int try_peek (ref <variable>;</code>

7/31/07

SystemVerilog for Verification

239

The `new()` constructor constructs a mailbox of an optionally limited size.

The `num()` function returns the number of messages currently in the mailbox.

The blocking `put()` task attempts to put a message in the mailbox and blocks if the mailbox is full.

The non-blocking `try_put()` function attempts to put a message in the mailbox and returns 0 if the mailbox is full.

The blocking `get()` task attempts to get a message from the mailbox and blocks if the mailbox is empty. It is an error to attempt to get a message of a type incompatible with the type of the message at the head of the mailbox queue.

The non-blocking `try_get()` function attempts to get a message from the mailbox and returns 0 if the mailbox is empty and returns a negative integer on an attempt to get a message of a type incompatible with the type of the message at the head of the mailbox queue.

The `peek()` task and `try_peek()` function operate similarly to the `get()` task and `try_get()` function but copy the message instead of removing it.

Use the `try_get()` and `try_peek()` functions when you do not know what type of message is at the head of the mailbox queue.

Process Synchronization with Mailbox

```
mailbox #(int) mbox = new(4);
```

```
initial
  for (int i=0; i<6; i++)
    #1ns mbox.put(i);
```

sender

```
int rdat;
initial
  if (mbox.try_get(rdat)==0)
    #6ns for (int i=0; i<6; i++)
      #1ns mbox.get(rdat);
```

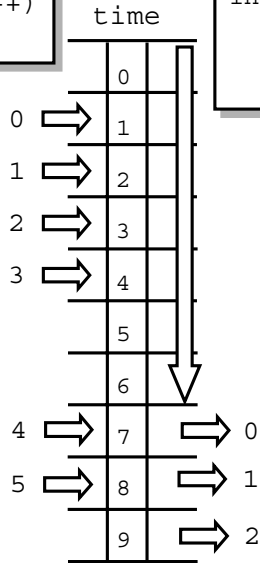
receiver

sender blocks at
time 4 – mbox full

sender continues as
receiver gets mail

receiver unsuccessful
at time 0 – waits a while

receiver gets
mail at time 7



At time 1, the sender starts putting six messages into the mailbox, at intervals one time unit apart. The mailbox can hold at most four messages, so the sender blocks at time 5.

At time 0, the receiver attempts to retrieve a message from the mailbox. At that time, the mailbox is empty, so the receiver waits for seven time units, and then retrieves six messages, at intervals one time unit apart.

As the receiver takes messages out of the mailbox, the sender resumes putting messages into the mailbox.

Typeless Mailbox

common declarations

```
class pkt;
  rand bit    [4:0] addr;
  rand logic [7:0] data;
endclass
mailbox mbox = new;
typedef enum
{pass, fail, hung} state_t;
...
```

A mailbox can hold messages of different types

- ◆ Blocking `get()` or `peek()` with incompatible variable type results in runtime error
- ◆ Non-blocking `try_get()` or `try_peek()` with incompatible variable type returns a negative integer
 - Possible solution – try all types?

sender

```
string pstring;
pkt ppkt = new;
state_t pstatus;
...
pstring = "test one";
assert(ppkt.randomize);
mbox.put(pstring);
mbox.put(ppkt);
mbox.put(pstatus);
...
```

receiver

```
string gstring;
pkt gpkt;
state_t gstatus;
...
mbox.get(gstring);
mbox.get(gpkt);
mbox.get(gstatus);
...
```

??

```
ok = mbox.try_get(gstring);
if (ok < 0) begin
  ok = mbox.try_get(gpkt);
  if (ok < 0) begin
    ok = mbox.try_get(gstatus);
  end
end
...
```

7/31/07

SystemVerilog for Verification

241

A mailbox by default accepts messages of any type.

To retrieve a message using the blocking `get()` or `peek()` methods requires you to know the type of the message at the head of the mailbox queue. Unless you always put and get the messages in a preordained order, you are unlikely to know the type of the message at the head of the mailbox queue.

You can alternatively write code that uses the non-blocking `try_get()` or `try_peek()` methods that return a negative integer upon attempt to retrieve a message of a different type than that at the head of the mailbox queue. This code can iterate through all the types that you place in the mailbox.

You may elect to more simply use a separate mailbox for each message type.

Parameterized Mailboxes

common declarations

```
class pkt;
  bit   [4:0] addr;
  logic [7:0] data;
endclass
typedef enum
  {pass, fail, hung} state_t;
mailbox #(string)  smbox = new;
mailbox #(pkt)     pmbox = new;
mailbox #(state_t) tmbox = new;
...
```

sender

```
string pstring;
pkt ppkt = new;
state_t pstatus;
...
pstring = "test one";
assert(ppkt.randomize);
smbox.put(pstring);
pmbox.put(ppkt);
tmbox.put(pstatus);
...
```



Support

receiver

```
string gstring;
pkt gpkt;
state_t gstatus;
...
smbox.get(gstring);
pmbox.get(gpkt);
tmbox.get(gstatus);
...
```

You can type-parameterize a mailbox

- ◆ Define type upon declaration
- ◆ Holds messages of that and equivalent types
- ◆ Compiler detects type mismatch

You can simply use a separate mailbox for each message type.

You specify the one type a mailbox may accept, by overriding its type parameter when you declare the mailbox variable.

The compiler can now detect any attempt to store or retrieve messages of an incompatible type.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 14.4

Semaphores

Semaphores are a key-based synchronization mechanism.

Used for mutual exclusion, controlling access to shared resources, and process synchronization.

- ◆ A process requests semaphore key(s) before accessing the resource

- ❑ Requesting more keys than a semaphore currently has can *block* until sufficient keys are returned

- ❑ Blocking: `get(int keyCount=1)`

- ❑ Non-blocking: `int try_get(int keyCount=1)`

- ◆ A process returns semaphore key(s) after accessing the resource

- ❑ `int put(int keyCount=1)`

- ◆ The user assumes responsibility for key management

```
semaphore keybox = new(5); // semaphore with 5 keys
semaphore sync;
sync = new(4);           // semaphore with 4 keys
```

7/31/07

SystemVerilog for Verification

243

Semaphores are a class-like key-based process synchronization mechanism.

A semaphore is conceptually like a basket of keys.

When you create the semaphore, you pass to its constructor the number of keys you want the basket to initially hold. If you do not pass a number, there will be no keys.

When a process needs to access a resource, it requests a number of keys, by default just one, from the basket. This can be a blocking request, if insufficient keys are available the process blocks, or a non-blocking request, if insufficient keys are available an error flag is returned. The order in which processes block is the order in which they are later serviced.

The process returns some or all of its keys when it no longer needs a portion, or any, of the resource.

This is all the semaphore does. It is up to the user to define, manage and utilize the resource.

IEEE Std. 1800-2005 SystemVerilog LRM, Section 14.2

Semaphore Methods

Method	Description	Syntax
<code>new()</code>	Semaphore constructor which specifies number of keys	<code>function new(int keyCount = 0);</code> Note default number of keys set is 0.
<code>get()</code>	Extracts a set number of keys from the semaphore – blocks if the keys are not available	<code>task get(int keyCount = 1);</code>
<code>try_get()</code>	Extracts a set number of keys from the semaphore without blocking – returns 0 if the keys are not available	<code>function int try_get(int keyCount=1);</code>
<code>put()</code>	Returns a set number of keys to the semaphore	<code>task put(int keyCount=1);</code>

7/31/07

SystemVerilog for Verification

244

The `new()` constructor constructs a semaphore with the specified number of keys. The default value of its parameter is 0 – no keys.

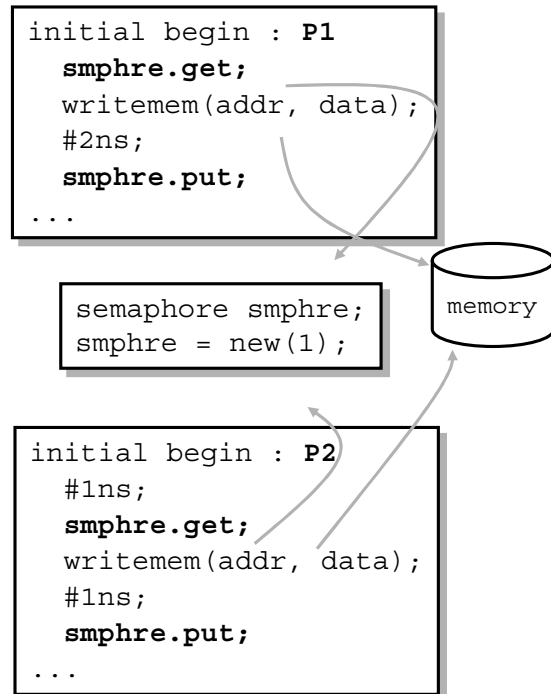
The blocking `get()` requests one or more keys and suspends the calling process if the key or keys are not available. The blocked process resumes when sufficient keys to meet its needs are put into the semaphore.

The non-blocking `try_get()` returns 0 if the key or keys are not available.

The `put()` method puts keys into the semaphore. The user is responsible for key management, for example, to ensure that a process returns only those keys that it previously retrieved.

Process Synchronization with Semaphore

- ◆ `smphre` is a single-key semaphore
- ◆ Processes access the resource between calls to `get()` and `put()`
- ◆ Process `P1` requests a key at time 0 and gets it and accesses the resource
- ◆ Process `P2` requests a key at time 1 and blocks waiting for a key
- ◆ Process `P1` completes its resource access and returns the key
- ◆ Process `P2` unblocks, gets the key, and accesses the resource
- ◆ Process `P2` completes its resource access and returns the key



7/31/07

SystemVerilog for Verification

245

In this example:

- At time 0, process `P1` requests a key, and gets it, and accesses the resource.
- At time 1, process `P2` requests a key and blocks, waiting for the key.
- At time 2, process `P1` completes its resource access and returns the key.
 - Process `P2` now unblocks, gets the key, and accesses the resource.
- At time 3, process `P2` completes its resource access and returns the key.

Remember the user is responsible for defining, managing and utilizing the resource. Common user errors include:

- Writing procedural statements that access the resource without first obtaining the requisite number of keys.
- Writing procedural statements that fail to return keys after completing the resource access.
- Writing procedural statements that return keys to the semaphore that they never took out of the semaphore.