Testbench Architecture & Implementation with SystemVerilog

Module #2: SystemVerilog

Michael A. Warner
Worldwide Consulting Manager

Mentor Graphics®

Agenda

- Verification Planning
- Testbench Architecture
- **Testbench Implementation**
  - **SystemVerilog Basics**
  - OOP with SystemVerilog
  - OVM Introduction

SystemVerilog Basics

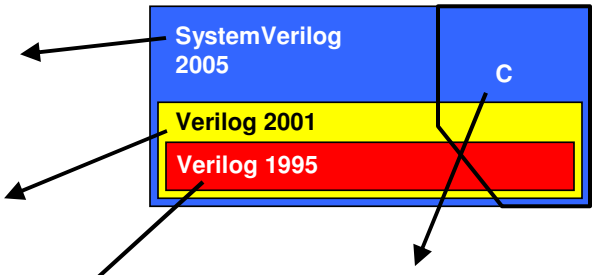
- Introduction to SystemVerilog
- Data Types, Arrays & Literals
- Behavioral Modeling
- Design Structure & Hierarchy

What is SystemVerilog?

- SystemVerilog is a standard set of extensions to the IEEE 1364-2001 Verilog standard
- SystemVerilog was developed by Accellera
  - Many donations including SUPERLOG & VERA
  - Features borrowed from other standard languages such as VHDL, C, PSL
- Current standard IEEE 1800-2005
  - Developed from Accellera 3.1a SystemVerilog donation
- Development of next version nearing completion
  - Draft P1800-2009
  - Combined standard including both 1800 & 1364

SystemVerilog "It's Alive"

- Complete design & verification features unified into a single language
- Added additional constructs designers wanted
- Hardware Design from Behavioral & RTL to gate
- Limited verification capabilities in the current Verilog language



- C like data types, loops, & operators

SystemVerilog Basics

- Introduction to SystemVerilog
- **Data Types**
- Behavioral Modeling
- Design Structure & Hierarchy

Integer Data Types

shortint	2-state SV type, 16-bit signed integer
int	2-state SV type, 32-bit signed integer
longint	2-state SV type, 64-bit signed integer
byte	2-state SV type, 8-bit signed integer or ASCII character
bit	2-state SV type, user-defined vector size
logic	4-state SV type, user-defined vector size
reg	4-state Verilog type, user-defined vector size
integer	4-state Verilog type, 32-bit signed integer
time	4-state Verilog type, 64-bit unsigned integer

String Type

- String literals in SystemVerilog same as Verilog
- SV also provides ***string*** type
  - Strings can be arbitrary length
  - No truncation

```
string str = "Hello World!";
```

- Operations allowed on strings

str1 == str2	**Equality**
str1 != str2	**Inequality**
str1 [<,>,<=,>=] str2	**Comparison**
{str1, str2, …, strn}	**Concatenation**
{multiplier{str1}}	**Replication**
str1[index]	**Indexing – Returns ASCII code at index (byte)**

String Methods

- SystemVerilog has a robust set of string methods

str.len()	function int len()	Returns length of string
str.putc()	task putc(int i, byte c)	Replace ith character with c
str.getc()	function byte getc(int i)	Returns the ith ASCII code
str.Toupper()	function string Toupper()	Returns string in upper case
str.Tolower()	function string Tolower()	Returns string in lower case
str.compare()	function int compare(string s)	Compares str with s
str.icompare()	function int icompare(string s)	Case insensitive compare
str.substr()	function string substr(int i, int j)	Returns sub-string from index i to j
str.atoi()	function int atoi()	Returns int corresponding to ASCII decimal representation of string

String Methods Cont.

str.atohex()	function int atohex()	Interprets string as hexadecimal
str.atooct()	function int atooct()	Interprets string as octal
str.atobin()	function int atobin()	Interprets string as binary
str.atoreal()	function real atoreal()	Returns real corresponding to ASCII decimal representation of string
str.itoa()	task itoa(integer i)	Store ASCII decimal representation of i in string (inverse of atoi)
str.hextoa()	task hextoa(integer i)	Inverse of atohex
str.octtoa()	task octtoa(integer i)	Inverse of atooct
str.bintoa()	task bintoa(integer i)	Inverse of atobin
str.realtoa()	task realtoa(real i)	Inverse of atoreal

User Defined Types

- In addition to built-in types, user can create their own types

```
// create a new type
typedef int inch;
// declare two new variables of type inch
inch foot = 12, yard = 36;
```


Type Operator

- The SystemVerilog ***type*** operator improves parameterized models

```
module DFF #(parameter type data_t = int)(
    input data_t D,
    input clk, rst,
    output data_t Q);
    always @(posedge clk)
        if (rst == 1'b0) Q <= data_t'(0);
        else Q <= D;
endmodule : DFF
```

Flip-flop with the default type set to ***int***

Ports specified using the type parameter

Cast literal to the type

Change type during instantiation

```
DFF #(.data_t(logic [7:0])) r0 ( ... );
DFF #(.data_t(real)) r1 ( ... );
```

Data Organization

- Desire to organize data
  - Like other high-level programming languages
  - explicit, meaningful relationships between data elements
- Verilog provides only informal relationships
- SystemVerilog provides new data types for this purpose
  - Structures, unions & arrays used alone or combined better capture design intent

Data Organization - Structs

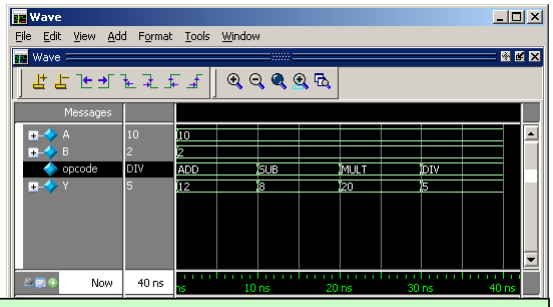
- Structs Preserve Logical Grouping
- Not restricted to elements of same size and type as with arrays
- Reference to Struct yields longer expressions but facilitates more meaningful code

```
struct {
   addr_t SrcAdr;
   addr_t DstAdr;
   data_t Data;
} Pkt;

Pkt.SrcAdr = SrcAdr;
if (Pkt.DstAdr == Adr)
    local_data = Pkt.Data;
```

Data Organization - enum

- Enums formally define symbolic set of values
- Use as symbolic indexes to make array references more readable
- Provides better self-documentation & debug




```
typedef enum {ADD, SUB, MULT, DIV} opc_t;

module ALU (input opc_t opcode, input int A, B output int Y);
    always_comb
        case (opcode)
            ADD: Y = A + B;
            SUB: Y = A - B;
                ...
        endcase
endmodule : ALU
```

Arrays

- SystemVerilog adds many new types and operations on arrays
  - Packed/Unpacked arrays
  - Array querying functions
  - Dynamic arrays
  - Associative arrays
  - Queues
  - Array manipulation methods

Packed & Unpacked Arrays

- Verilog 1995 only allows one dimensional arrays, vectors, and memories:

```
reg ARRAY [0:127];
reg [63:0] VECT;
reg [7:0] MEM [0:127];
```

- Verilog 2001 enhances the language with multidimensional arrays and part/bit selects:

```
reg [7:0] MULTI [0:127] [0:15];
assign Q = MULTI [109] [8] [7:4];
```

  - Although an improvement, still very restrictive access to arrays
- SystemVerilog allows much more versatile access with packed & unpacked arrays

Dynamic Arrays

- A ***dynamic*** array is a one-dimensional array with NO range specified at declaration
  - No space allocated until sized during runtime
  - Increase or decrease size any time during simulation
  - Check memory size any time during simulation

```
logic [7:0] mem [];                    // dynamic array of 8-bit vectors
initial begin
  mem = new[128];                      // allocate space for 128 vectors
  #10 mem[62] = 8'h8f;                 // legal assignment
  #10 mem[131] = 8'h16;                // illegal - size is currently 128
  #10 mem = new[mem.size * 2] (mem); // increase size of array by factor
                                       // of 2 & seed memory with
                                       // previous values
  #10 mem[131] = 8'h16;                // now this assignment is legal
  #10 mem.delete;                      // delete all elements of array
end
```

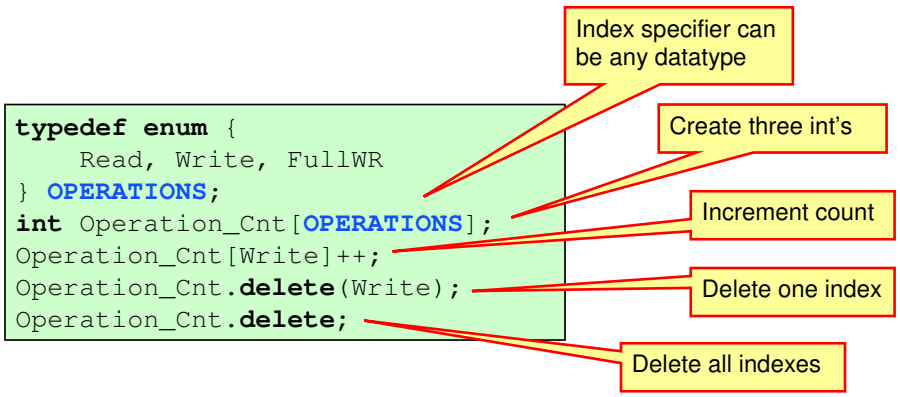
Queues (Dynamic Lists)

- List of like items that grows & shrinks dynamically
  - A list is a variable length array
- List manipulation syntax is similar to concatenation and bit select in packed arrays
- Queues are very useful during verification
  - In-order scoreboards
    - Data received in order it is sent
  - Stack of packets on ports
    - Test is over when queues are empty

Associative Arrays

- User-defined index type
- Memory allocated as elements are written
- Associative arrays are very useful during verification
  - Out of order scoreboards
    - Random access read/write
  - Model Sparse memories
    - Conserve memory
  - Simple coverage information

Associative Arrays




```
typedef enum {
    Read, Write, FullWR
} OPERATIONS;
int Operation_Cnt[OPERATIONS];
Operation_Cnt[Write]++;
Operation_Cnt.delete(Write);
Operation_Cnt.delete;
```

```
int Operation_Cnt[OPERATIONS] = '{default:0};
```

Set default for all nonexistent elements, does NOT explicitly allocate storage

SystemVerilog Basics

- Introduction to SystemVerilog
- Data Types
- **Behavioral Modeling**
- Design Structure & Hierarchy

Increment Operators

- The ++ and -- operators have been added to SystemVerilog

j = i++	Post-Increment. j is assigned the value of i, and then i is incremented by 1
j = ++i	Pre-Increment. i is incremented by 1 , and then j is assigned the value of i
j = i--	Post-Decrement. j is assigned the value of i, and then i is decremented by 1
j = --i	Pre-Decrement. i is Decremented by 1 , and then j is assigned the value of i

- Synthesizable, but only when used in a separate statement

```
i++; // synthesizable
sum = i++; //not synthesizable
```


Assignment Operators

- All assignment operators behave as blocking assignments
- RHS = Right-hand side – LHS = Left-hand side

+=	Add RHS to LHS and assign
-=	Subtract RHS from LHS and assign
*=	Multiply RHL from LHS and assign
/=	Divide LHS by RHS and assign
%=	Divide LHS by RHS and assign the remainder
&=	Bitwise AND RHS with LHS and assign
|=	Bitwise OR RHS with LHS and assign
^=	Bitwise exclusive OR RHS with LHS and assign

```
a[1] += 2; //same as a[1] = a[1] + 2;
```

Assignment Operators Cont.

<<=	Bitwise left-shift the LHS by number of times indicated on RHS and assign
>>=	Bitwise right-shift the LHS by number of times indicated on RHS and assign
<<<=	Arithmetic left-shift the LHS by the number of times indicated by the RHS and assign
>>>=	Arithmetic right-shift the LHS by the number of times indicated by the RHS and assign

```
bit signed [5:0] a;
a <<<= 2; //Shift left 2 bits, retain sign
```

```
bit [5:0] a;
a <<= 2; //Shift left 2 bits, including sign
```


Equality Operators

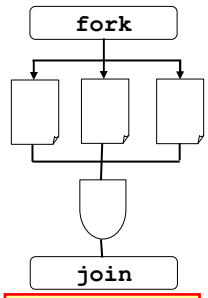
== !=	Case equality. Return 'X' if either operand has an 'X' or 'Z'
=== !==	Identity. Exact bitwise match of 0,1,X and Z
=?= !?=	Equality. Bitwise match, masking all 'X' as wildcards

```
010Z == 010Z  //Unknown, returns X
010Z === 010Z //True
010Z === 010X //False
```

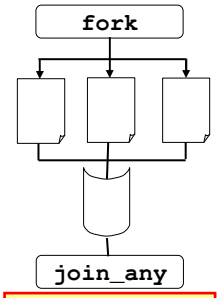
```
010X =?= 0101 //True
010Z =?= 010X //True
1110 =?= XXX0 //True
0011 =?= X1XX //False
```

Processes

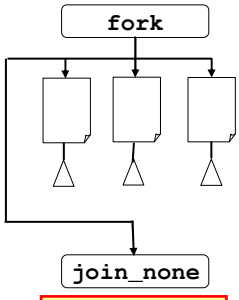
- Verilog provides a simple process spawning capability via the ***fork-join*** statement
- SystemVerilog adds 2 additional mechanisms ***join_any*** and ***join_none***



Blocks until ***ALL*** threads complete



Blocks until ***ANY*** thread completes



Doesn't block

Process Control Methods

- Process methods
  - ***status()*** - returns the process status
    - FINISHED, RUNNING, WAITING, SUSPENDED, KILLED
  - ***await()*** - allows one process to wait for the completion of another process
  - ***suspend()*** - allows a process to suspend either its own execution or that of another process
  - ***resume()*** - restarts a previously suspended process
  - ***kill()*** - terminates the given process and all its sub-processes

```
if ( job[1].status() != process::FINISHED )
    job[1].kill();
```

Check the status of job #1 & kill it if it did not finish

```
job[1].await();
```

Wait for Job #1 to finish

Tasks and Functions

- SystemVerilog allows you to mix automatic & static
  - Static variables within automatic tasks
  - Automatic variables within static tasks

```
task automatic review_results;
   static int total_results
   total_results++;
endtask : review_results
```

```
task inc;
   automatic int i = 0;
   do
      $display(i);
      i++;
   while(i < 20);
endtask : inc
```

NOTE:
No longer require ***begin/end*** on tasks or functions

Match names for ***task*** & ***endtask***

Functions

- In SystemVerilog functions now use the 'return' statement to return a value
  - Old:

```
function int mult (input int a, b)
   mult = a * b;
endfunction
```

Returns variable that shares the name of the function

  - New:

```
function int mult (input int a, b)
   return (a * b);
endfunction : mult
```

Return type inferred by function type

- Using 'return' you can exit functions before 'endfunction'

```
function int mult (input int a, b)
   if (a < 0) return 0;
   return (a * b);
endfunction : mult
```

Enhanced Loops

- The foreach construct allows you to easily iterate over the elements of an array

```
byte a[4] = '{0,1,2,3};
foreach (a[i]) begin
  $display("Value is %d",i);
end
```

Provide the name of the array and the variable name you wish to use to iterate.

- Enhanced "for" loop

Can have multiple loops with same index name

```
for (int i = 0; i < NUM_LOOPS; i++) d[i] += i;
for (int i = 0; i < 8; i++) y[i] = a[i] & b;
```

Loop control variable declared directly in loop & is local to that loop

Bottom Testing Loop

- SystemVerilog has added a do..while loop. This allows similar functionality to a while loop, except the checks are done after each loop execution

```
do
begin
    $display("Node Value: %d", node.value);
    node = node.next;
end
while(node != 0);
```

- Use when you know you will execute the loop at least once
- Same synthesis rules that apply to while loops apply to do..while loops. Generally are not synthesizable due to their dynamic nature.

Mailboxes

- Used for inter-process communication

```
mailbox #(packet_t) m_channel = new(25);
```

"m_channel" accepts up to 25 objects of type "packet_t"

- Behaves like a FIFO
- Mailbox shared between processes

```
task sender (packet_t P);
    m_channel.put(P);
task receiver (packet_t P);
    m_channel.get(P);
```

Put the packet "P" into the mailbox

Get the next packet on the stack out of the mailbox & place it in "P"

Mailbox Methods

new()	Create a mailbox with a specified number of slots
num()	Return number of items in mailbox
get()	Retrieve an item from the mailbox, if empty block until an item is available
try_get()	Retrieve an item from the mailbox, if empty do not block
peek()	Copy an item from the mailbox, if empty block until an item is available
try_peek()	Copy a item from the mailbox, if empty do not block
put()	Put an item in the mailbox, if full block until an space is available
try_put()	Put an item in the mailbox, if full do not block

Semaphore

- Provides control of shared resources for multiple processes
  - system bus or memory
- Conceptually a bucket with a fixed set of keys
- Processes using a ***semaphore*** need to procure a key or keys before they can execute

```
semaphore want_to_control;
want_to_control = new();
```

Create the semaphore

Specify the number of keys to the constructor, default is 0

```
want_to_control.get();
```

Request control – default is 1 key

```
want_to_control.put();
```

Release control – default is 1 key

```
want_to_control.try_get();
```

Nonblocking get – default is 1 key, return positive integer if successful else return zero

SystemVerilog Basics

- Introduction to SystemVerilog
- Data Types
- Behavioral Modeling
- **Design Structure & Hierarchy**

Packages

- Provides support for sharing throughout design
  - nets
  - variables, types, package imports
  - tasks, functions, dpi_import_export
  - classes, extern constraints, extern methods
  - parameters, local parameters, specparams
  - properties, sequences
  - anonymous program
- Use instead of global **`defines**
- Questa allows packages to be shared between VHDL and SystemVerilog
  - Compile with ***vcom/vlog -mixedsvvh***

Package Declaration

```
package ComplexPkg;
    typedef struct {
        float i, r;
    } Complex;
    Complex C_data = '{i:2.0, r:5.7};
    function Complex ADD (Complex a, b)
        ADD.r = a.r + b.r;
        ADD.i = a.i + b.i;
    endfunction : ADD
    function Complex MULT (Complex a, b)
        MULT.r = (a.r * b.r) + (a.i * b.i);
        MULT.i = (a.r * b.i) + (a.i * b.r);
    endfunction : MULT
endpackage : ComplexPkg
```

Define new types

Create global variables

NOTE: Assignments are done before any initial or always blocks are started

Define functions to operate on the new types

Using Packages

- SystemVerilog provides several ways to use the contents of packages
  - Scope resolution operator

```
ComplexPkg::C_data = ComplexPkg::MULT(a, b);
```

  - Explicit import

```
import ComplexPkg::C_data;
initial C_data = '{i:6.0,r:2.7};
```

  - Wildcard import

```
import ComplexPkg::*;
Initial C_data = ADD(a, b);
```


Named Blocks & Statement Labels

- Verilog allowed naming blocks without matching end
  - Verilog developers sometimes used comment

```
initial begin : simulation_control
   ...
end // simulation_control
```

- SystemVerilog now allows matching names at the end of blocks including task/functions, modules, interfaces, classes, etc.
  - Allows tool to catch mistakes with nested blocks

```
initial begin : simulation_control
   ...
end : simulation_control
```