Enhancement proposals for System Verilog 3.1

6/5/02
Jayant Nagda
Synopsys Inc.
New Technologies in Verification

- Test bench language to create tests and verification environments
- Assertions to create checkers (dynamic) and properties (Formal)
- C++ for high level of abstraction and representing algorithms
- Coverage Tools to Improve test quality
- New technologies are interacting with the simulator through PLI
Accellera with System Verilog 3.1

• Accellera is looking at most of these new technologies or interfaces for standardization

• System Verilog 3.0 is a major milestone in bringing higher level of abstraction to Verilog

• Successful language standard need to meet current and future requirements

• A unique opportunity to make lasting impact Support of Users
  • Support of Users
  • Support of tool vendors
  • Conduit to IEEE-standard
Proposed Enhancements
For System Verilog 3.1

• Test Bench Features
• Unified Assertion language
• Interface to C/C++
• Extensive API

• The language will be comprehensive and complete
• Higher Simulation Performance
• Ease of Use
• Easier for new technologies to interface with System Verilog simulators
Test Bench Features : Motivation

• At RTL level test benches have evolved.
• Test Benches were part of Verilog languages at gate level.
• Large portion of time is spent in creating tests and test environment
• Performance of test benches is become more important
Test Bench Features

- **Dynamic Objects**
  - Test bench users are not sophisticated programmers
  - Dynamic objects like classes automatically created and removed
- **Build in Test bench primitives**
  - Protocols, handshakes without implementation details
  - Semaphores, lists, mail boxes etc.
- **Advance Control constructs for complex scenarios**
  - Fork-joins: Fork –join all; join-one; join-none
  - Triggers: passing of events
- **Interactions to not just DUT**
  - To assertions, To Coverage, formal tools
  - Test generation reactive and more productive
Object declaration and allocation

Generate packet for all ports

Packets are used without worrying about freeing memory

```java
class Packet {
    nib inp, outp;
    int count;
    byte data_array[];
    task new (nib inport, nib outport, int byteCount) {
        inp = inport; outp = outport;
        count = byteCount;
        while (byteCount--)
            data_array[byteCount] = random();
    }
    task Send()...
}
```

```java
task generator(int size){
    int l, o;
    for (l = 1; l <= 16; l++) {
        for (o =1; <= 16; o++) {
            Packet testPacket = new (l, o, size);
            testPacket.send();
        }
    }
} 
```

```java
task monitor(Packet curPacket){
    nib inport = curPacket.inp;
    nib outport = curPacket.outp;
    ....
}
```
Concurrent and Synchronization

Dynamic concurrent execution

```
task generate( )
{
}
task check( )
{
}
task monitor( )
{
}
```

```
fork
for (i=0; i < 4; i++)
generate port[i] ;
join none
```

```
fork
for (i=0; i < 4; i++)
monitor port[i] ;
join none
```

```
fork
for (i=0; i < 4; i++)
monitor port[i] ;
join none
```

thread 1

thread 2

thread 3
module ...

program mailboxExample {
    Transfer t;
    Bus b = new();

    repeat(10) {
        t = new();
        b.transfer(t);
    }
}

class Bus {
    integer mbId;
    task new() {
        mbId = alloc (MAILBOX, 0, 1);
        fork
            transactor();
        join none
    }

    task transfer(Transfer t) {
        mailbox_put(mbId, t);
    }
}

task transactor() {
    Transfer t;
    while(1) {
        mailbox_get(WAIT, mbId, t);
        @(posedge CLOCK);
        bus.addr = t.addr;
        bus.size = t.size;
        bus.type = t.type;
        if (t.type == 1) {
            bus.data = t.data;
            @(posedge bus.ack);
        } else {
            @(posedge bus.ack);
            t.data = bus.data;
        }
    } // end while
}

.. end module
Unified Assertions

• Assertions are single interpretation of specifications. User writes assertions only once.
  ▪ For Dynamic simulation
  ▪ For Formal Property Checking
  ▪ For Functional Coverage

• Main requirements for Assertions
  ▪ Provide temporal language
  ▪ Provide Modeling aspects
  ▪ Provide System Verilog compatible expression semantics
Assertions Usage Model

- **RTL Source**
  - Module-level Assertions
  - Chip-level Assertions

- **Assertions**

- **Test Bench**

- **RTL sim**

- **Formal**

- **Proven Properties**

- **Runtime Report Coverage and Debug**
Proposal for Unified Assertion Based Verification

• Consider in System Verilog 3.1:
  ▪ Temporal expressions with Boolean expressions, syntactically and semantically identical to Verilog
  ▪ regular expressions for temporality
  ▪ multiple clocks with simple synchronization

• Discussion at 2 pm
C/C++ Interface

• Simpler interface for Calling C/C++ functions
  ▪ Use of PLI Require understanding of PLI and complexity
  ▪ Direct interface for calling C functions from Verilog
  ▪ Direct interface to call Verilog tasks from C

• Interchanging complex data structures across C and System Verilog boundaries
  ▪ PLI capabilities require data conversion

• Easy usage C/C++ code in Verilog:
  ▪ PLI does not allow creating ports to a C/C++ algorithm
  ▪ PLI does not allow mixing of C/C++ code fragments with Verilog
C Function call Example

• No true strings in Verilog
  ```verilog
  reg [1000*8:1] name; /* for a string up to 1000 characters */
  ... name = "tests/stimulus.dat";;
  Task T; input [1000*8:1] status; ... endtask
  ... T("passed");
  ```

• Memory & time inefficient
• string literal as the actual argument for a type `string` will be interpreted as a C-style, i.e. `char *`
  ▪ Better solution:
  ```c
  external string aString(string);
  Module top;
  reg [31:0] name; // string pointer; for strings of all sizes
  .... name = aString("tests/stimulus.dat");
  Task T; input [31:0] status; ... endtask
  .... T(aString("passed")); ...
  ```

char *aString(char *s){return s;}

© 2001 Synopsys, Inc. (14)
#include "complex.h"

`timescale 1ns/10ps

cmodule cmod(clock, rqst, out)
    input reg clock; inout reg rqst;
    output bit [31:0] out;
{
    bool odd_clock;
    void wait_for_2_clocks() {
        @(posedge clock); @(posedge clock);
    }
    initial { odd_clock=false; }
    always @(clock) {
        complex* temp = new complex;
        vc_delay(10);
        if (odd_clock && rqst) {
            wait_for_2_clocks();
            rqst = 0;
        }
    }
} // cmod()
Extended API

• Extending language have implications on simulator interface. New Technologies like Coverage and Assertions are part of simulator.

• Need API to access important information like Coverage, Assertions to interface other tools with System Verilog simulators.

• Comprehensive API Allows new tools and flows to be created and easily interfaced with System Verilog simulators.

• Non standard simulator interfaces can delay adoption and have high overhead