Verification

Synopsys Submissions to SystemVerilog 3.1a

Arturo Salz

September 18, 2003

Synopsys, Inc.

Testbench Submissions (SV-EC)

- Randomization
  - Constraints enhancements
  - Weighted random case: randcase
  - Random Sequence Generation
- Foreach statement
- Fine-Grain Process Control
- Queues
- Bit-streams: Pack/Unpack
- Reacting to Assertions
  - Sequence events
  - expect statement
- Virtual Interfaces
- Functional Coverage

Constraints Enhancements

- Object handles as guards

```
class SList;        // sorted linked list
  rand int n;
  rand Slist next;
  constraint sort { if( next != null ) n < next.n;};
endclass
```

- Functions in Constraints

```
class B;
  rand int length, v;
  ...
  constraint C1 { length == count_ones( v ) };
endclass
```

function

Constraints Enhancements

- **Iterative Constraints : foreach**

```
class C;
  rand byte a[];
  constraint C2 {foreach ( a[j] ) a[j] > 2 * j; };
```

- **Constraints on scope variables (outside class)**

```
task stimulus( int length );
  int a, b, c, d, success;
  success = randomize( a, b, c )   // random variables
       with { a < b ; a + b < length };
  ...
  success = randomize(d) with {d inside {[a:b]}};
endtask
```

Weighted Random Case

- Randomly select one statement
  - Label expressions specify distribution weight

```
randcase
   3 : x = 1;                    // branch 1
   1 : x = 2;                    // branch 2
   a : x = 3;                    // branch 3
endcase
```

  - If a == 4:
    - branch 1 taken with 3/8 probability (37.5%)
    - branch 2 taken with 1/8 probability (12.5%)
    - branch 3 taken with 4/8 probability (50.0%)

Random Sequence Generation

- Grammar-directed random sequences


```
class DSL; ... endclass   // creates valid DSL packets

randsequence (STREAM)
  DSL PACKET;

  STREAM : GAP DATA := 80
         | DATA := 20 ;

  DATA : PACKET(0) := 94 { txmit( PACKET ); }
       | PACKET(1) := 6 { txmit( PACKET ); };

  PACKET(bit bad) : { DSL d = new;
                      if( bad ) d.crc ^= 23;
                      return d;
                    };

  GAP: { ## $urandom_range( 1, 20 ); };
endsequence
```

Traffic consists of data packets & gaps

6% are bad packets

A bad packet mangles its checksum

Gaps are random cycle waits: 1..20

DSL Traffic Generator

foreach Statement

- Iteration over array dimension(s)

```
string words [2] = { "hello", "world" };
int prod [1:8] [1:3];

      // print index and each word
foreach( word [j] ) $display( j , word[j] );

      // initialize to index product
foreach( prod[ k, m ] ) prod[k][m] = k * m;
```

- Multiple dimensions supported
- Any dimensions may be skipped:
  - `foreach( arr[ j, , k] ) ...`

Fine-Grain Process Control

- Built-in class: process

```
class process
  enum state { RUNNING, WAITING, SUSPENDED, ... };
  static function process self();
  function state status();
  task kill();
  task await();
  task suspend();
  task resume();
endclass
```

- Can control any process via its *process-id*
  - A process retrieves its own *id* with: process::self

Queues

- Variable-sized Array: *data_type name* [$]
  - Uses array syntax and operators

```
int q[$] = { 2, 4, 8 };  int e, pos, p[$];

e = q[0];              // read the first (leftmost) item
e = q[$];              // read the last (rightmost) item
q[0] = e;              // write the first item
p = q;                 // copy entire queue
q = { q, 6 };          // append: insert '6' at the end
q = { e, q };          // insert 'e' at the beginning
q = q[1:$];            // delete the first (leftmost) item
q = q[1:$-1];          // delete the first and last items
q = {};                // clear the queue: {} empty queue
q = { q[0:pos-1], e, q[pos,$]}; // insert 'e' at pos
```

Bit-stream casting: Pack / Unpack

- Bit-stream casts convert unpacked ↔ packed data

```
typedef struct
{
  shortint address;
  reg [3:0] code;
  byte command [2];
} Control;

typedef bit Bits [36:1];
```

```
Control s, r;
Bits q[$];  // Bits queue
   // pack and append
q = {q, Bits'(s)};
  // unpack 1st 'Control'
r = Control'(q[0]);
q = q[1:$]; // remove r
```

- Useful if conversion can be expressed as a cast
  - Data is always packed left to right (msb -> lsb)
  - Order and organization of packed is predefined
  - Packet size is known

Stream Operators: Pack / Unpack

- Streaming operators extend bit-stream casts:
  - pack / unpack in any arbitrary order / organization
    - bit-reversed, little-endian, byte-swapped, nibbles, etc…
  - `{>>{a,b,c}}` Stream a,b,c by bits left to right
  - `{<< byte{a,b,c}}` Stream a,b,c by bytes right to left
  - `Z = {<<{a,b,c};` Pack a,b,c into Z (right to left)
  - `{>>{a,b,c} = Z;` Unpack Z into a,b,c (left to right)

```
typedef struct {
  int header, len;
  byte payload[];
  byte crc;
} Packet;
```

```
Packet p;
byte q[$];  // queue of bytes
```

```
q = {<< byte {p.header, p.len, p.payload, p.crc}};
{<< byte {p.header, p.len p.payload, p.crc}} = q;
```

Reacting to Assertions: Sequence Events

- Allow a sequence as an event control
- Enables procedural code to wait for the end-point of a sequence

```
sequence abc;
  @(posedge clk) a ##1 b ##1 c;
endsequence

program test;
  initial begin
    @ abc $display( "a-b-c happened" );
    ... // continue here
  end
endprogram
```

Reacting to Assertions: expect

- Procedural blocking statement
  - Property may be declared on-the-fly
  - Statement blocks until property succeeds or fails
    - First attempt
  - Pass-fail block: similar to property assertion syntax

```
program test;
  initial begin
    #200ms;
    expect( @(posedge clk) a ##[1:4] b ##1 c )
      else $error( "expect failed" );
    ... // continue here
  end
endprogram
```

Virtual Interfaces

- Allows interface instances to be:
  - Passed to tasks and functions
  - Stored in a variable (typically a transactor class)

```
interface SBus; logic req, grant; ... endinterface

class SBusTransctor;
  virtual SBus bus;  // virtual interface type Sbus
  function new( virtual SBus s );
    bus = s;     // initialize the virtual interface
  endfunction
  task request(); bus.req <= 1'b1; endtask
  task wait_for_bus(); @(posedge bus.grant); endtask
endclass
```

- Same code can interact with multiple interfaces

Functional Coverage

- New covergroup container allows declaration of
  - coverage points
    - variables
    - expressions
    - transitions
  - cross coverage
  - sampling expression : clocking event

```
enum { red, green, blue } color;
bit [3:0] pixel_adr,

covergroup g1 @(posedge clk);
  c: coverpoint color;
  a: coverpoint pixel_adr;
  AxC: cross color, pixel_adr;
endgroup;
```

Functional Coverage

- Automatic or used-defined coverage bins
  - Coverage-point bins (values and transitions)
  - cross coverage
- Filtering (guard) expressions at any level
- Options to regulate and query coverage

```
bit [7:0] v_a, b;

covergroup cg @ev1;
  a: coverpoint v_a {
    bins a1 = { [64:127],200 };  // user-defined bins
    bins a2 = { 0,10,100,220 } iff( !reset );
    bins bad = default;          // all other values
  }
  c : cross a, b;
endgroup
```

Design Submissions (SV-BC)

- Operator Overloading
- Hardware Functions
- Port Expressions
- Array Query Functions
  - bits, size, …

Operator Overloading

- Binds arithmetic operators to user-defined functions
  - Not for built-in operators
- Allows users to create their own math packages

```
typedef struct {
   real R;
   real I;
} Complex;

Complex X,Y,Z;

function Complex CMULT( Complex A, B );
  CMULT.R = A.R * B.R - A.I * B.I;
  CMULT.I = A.R * B.I + A.I * B.R;
endfunction

function * Complex CMULT( Complex, Complex );

assign X = Y * Z;
```

Hardware Functions

- Preserves mathematical representation
  - Increase language expressiveness to capture design intent
- Eliminates top level temporary signals

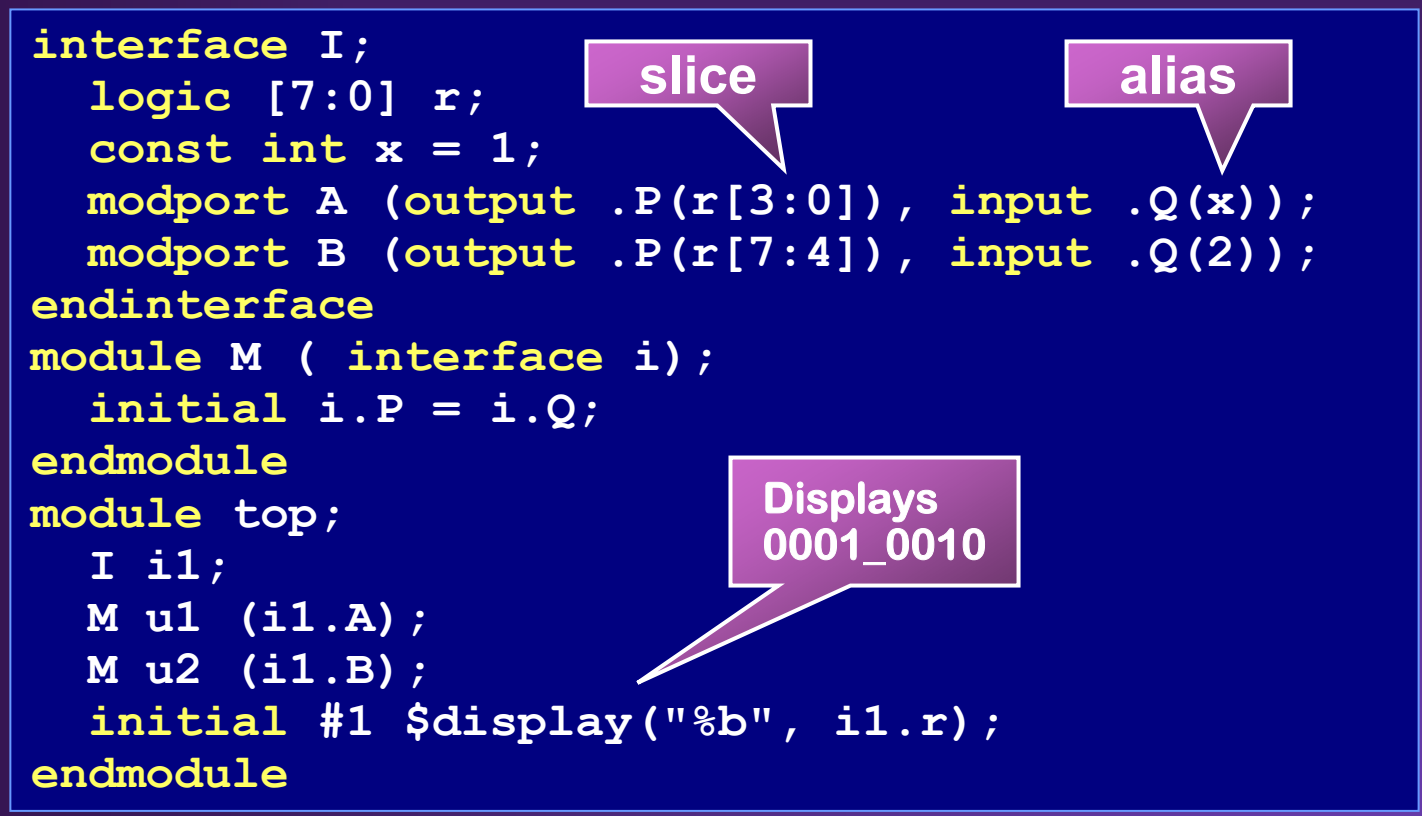
```
 // 1 mux and 3 primitives
assign w = a ? and(b,c) : or(d, xor(e,f));
```

```
and (and_1, b, c);
xor (xor_1, e, f);
or (or1, d, xor_1);

assign w = a ?
     and_1 : or_1;
```

```
module adderchain #(parameter n=1)
(output [15:0] sum, input a[n]);
 logic [15:0] s [n];
 generate for ( genvar I = 0; I < a::size; I++)
   begin : loop
     if (I > 0) assign s[I] = s[I-1] + a[I];
     else assign s[0] = a[1] + a[0];
   end
 endgenerate
endmodule

logic [15:0] S = adderchain #(4)(a,b,c,d);
```

Port Expressions

- Allows elements of structures and arrays to be connected to ports of modules and interfaces




```
interface I;
  logic [7:0] r;
  const int x = 1;
  modport A (output .P(r[3:0]), input .Q(x));
  modport B (output .P(r[7:4]), input .Q(2));
endinterface
module M ( interface i);
  initial i.P = i.Q;
endmodule
module top;
  I i1;
  M u1 (i1.A);
  M u2 (i1.B);
  initial #1 $display("%b", i1.r);
endmodule
```

- Complete ANSI-style port list definition for modules

Array Query Functions and Methods

- All $*array* query and $bits system functions now:
  - Provide a method version
  - Work on all types of array dimension types
    - Fixed, dynamic, associative, queue
  - Can be used on type identifiers (in addition to variables)
  - Can be used as a constant function
    - When applied to any fixed-size type

```
typedef logic [16:1] Word;
Word w1, Ram[0:9];

$size(w1)
$size(Ram,1)
Word::size
Ram.size(2)
```

All return 16

Thank You