Orthogonality in SystemVerilog

Cadence Design Systems

October 10, 2004

Overview

- What is the concept of orthogonality?
- What is the impact of applying this concept to SystemVerilog ?
- What are the resultant issues?
  - IEEE-1364 data types working group
  - Implementation of SystemVerilog
  - Interaction with customers
- Which issues are easily resolved?
- Which issues are postponed?
  - For these we only seek to leave room for future enhancement

Concept of orthogonality

- Orthogonality is a standard principle of language design
  - First, define basic properties objects can have
  - Then, combine the basic properties to define objects
- Verilog 1364-2001 begins to distinguish object “kind” and “type”
  - The “kind” of an object defines how an object behaves
  - The “type” of an object defines the set of values it can have
- SystemVerilog introduces a third characteristic “allocation policy”
  - The “allocation policy” defines how an object is allocated and deallocated
- Most Verilog objects can be characterized by combining these properties

Object kinds

- Verilog defines the basic kinds net, variable, and parameter
  - Port and task/function argument can be characterized as subsets of these primary objects
  - There are many kinds of parameters with minor semantic changes
- Nets
  - Provide bi-directionality, multiple drivers, and delay
  - SystemVerilog did not add complex types to nets, main focus of 1364 DT group
- Variables
  - Are updated immediately through assignment
  - SystemVerilog added complex types to variables
  - SystemVerilog added continuous assign to variables (single driver)
- Parameters
  - Post-elaboration constants
  - SystemVerilog added optional data types to parameters

Object types

- Constructs that are definitely types
  - 2-State
  - “Packed”
  - “Unpacked”
- Constructs that may be types or kinds (or both)
  - Named Event
  - Class

Allocation policy

- SystemVerilog defines “safe” allocation
  - Explicit allocation with **new()**
  - Implicit deallocation through garbage collection
- Only defined allocation on classes
- Lack of orthogonal definition precludes dynamic allocation of other types
  - ‘C’-like linked lists of structures
  - Class-like allocation applies to built-in or “near” classes
    - mailbox, semaphore
    - covergroup

Adding types on nets

- Simple, backward compatible, extension by allowing a wire kind declaration in front of a data type

  int I;

  wire int I;

- Allows ports that act as nets to have data types
- Provides abstract modeling of interconnect
  - Preserves bi-directionality, resolution and delay
- Performance advantage of port collapsing
  - Don’t need intermediate continuous assigns
  - Collapsing variable ports can be explicit today with **ref**

Issues with net types

- Short term addressable issues
  - Semantics of nets declared as 2-state?
  - When is a port a net or variable?
  - What happens when new ports with a type are connected to old style ports?
  - Can we still enforce having a net with a single driver?
- Postponed issues
  - Real-valued objects
  - Unpacked structures containing events
  - Dynamically allocated objects
- Only solving the short-term issues adds vast power to Verilog
  - Other issues can be errors for now and allowed later if desired

What are the semantics of nets declared as 2-state?

- Significant effort spent in 1364
- Consensus was that nets always maintain a 4-state value
- Conversion to 2-state happens in “procedural” contexts
  - During assignment to a 2-state variable
  - When displayed in a “procedural” context
- NO conversion happens in “network” contexts
  - Continuous assign, primitives, specify blocks
- Similar to treatment of mixed 2-state/4-state packed struct

Example of 2-state/4-state

- Example

```
module top;
    wire my_net;
    bottom b1 (my_net);
endmodule

 module bottom ( input wire bit my_port );
     initial $monitor ("my_port = %b, my_net = %b", my_port, top.my_net);
 endmodule
```

- Result

my_port = 0, my_net = z

When is a port a net or variable?

- Output ports
  - SV allows: **output** bus_type p; // this is a variable
  - Extension: **output wire** bus_type p; // this would be a net
- Inout ports
  - SV disallows because they are obviously nets
  - Could allow:

    inout wire bus_type p; // this would be a net

    inout bus_type p; // could default inout to net because it is “obvious”
- Input ports
  - SV allows: **input** bus_type p; // In SV this is a variable
  - Could use: **input wire** bus_type p; // this would be a net
- But why are inputs with type implicitly variables?

Change input ports to nets

- **input** p; // this is a net internally to Verilog today
- **input** TypeName p; // Should also be a net
  - Some force/release wording makes them a little like nets already
  - Since this would be new code, port direction could be enforced w/ this form
- An input port that is a net offers the following advantages:
  - You can apply timing (SDF) to the port
  - Tool providers can implement port collapsing to get performance and capacity improvements for a deep hierarchy of connections.
  - Behavior between new SystemVerilog ports and Verilog 2001 ports is closer to what the user expects.
  - Input ports that are nets are suitable for connection to other language domains, like VHDL.

Resultant net or variable rules

- An **input** ports is a net (change from SV)
  - The net type in the declaration is optional
  - Enforce port direction when a type name is present
  - For huge majority of users this would be a transparent change
- An **inout** ports is a net (V2K)
  - The net type in the declaration is optional
- An **output** port
  - Declared without a datatype name is a net (V2K)
  - Declared with a datatype name is a variable (SV)
  - An explicit net type can make it a net (Addition to SV)

What happens when a new port is connected to an old model?

- Multiple drives to a variable can still occur w/legacy models

```
module top;   // New design in SystemVerilog
        logic mout;
        mux m1(sel, in0, in1, mout);
endmodule
module mux (sel, in0, in1, mout);  // Legacy Verilog code
        input sel, in0, in1;
        output mout;
        bufif0 b0(mout, sel, in0);
        bufif1 b1(mout, sel, in1);
endmodule
```

- The above code is legal (in our interpretation)
- Such abuse, incorrect declaration of port modes is incredibly common, enforcing modes in legacy code would break most existing designs.

New “wone” type for a single driver

- Use the **wone** type when a net should only have one driver and it is used in a “network” context

```
module top;   // New design using wone wire kind
        wone logic mout;
        mux m1(sel, in0, in1, mout);
endmodule
module mux (sel, in0, in1, mout);  // Legacy Verilog code
        input sel, in0, in1;
        output mout;
        bufif0 b0(mout, sel, in0);
        bufif1 b1(mout, sel, in1);
endmodule
```

- The above would be an error

Issues with allocation

- The allocation policy of SystemVerilog combines the kind of an object and its allocation policy
  - Covergroups are always dynamic
  - Classes are always dynamic
  - All other object kinds are static
- Simple issues
  - Make classes and covergroup instances explicitly dynamic today
  - Allow static instances of covergroups because it is powerful and easy
- Possibly postponed issues
  - Dynamic allocation of structures
  - Static allocation of classes
- If we don’t make the simple changes now, the postponed changes can never be cleanly done

Make dynamic nature of an object explicit

- Introduce keyword in front of an object that is dynamic

  (Note: The word **dynamic** is used here but is arbitrary)

  class C … **endclass**;

  dynamic C c1; // This object will be dynamically allocated

  initial begin …; c1 = **new**(…); …; **end**

  – In practice this will be buried in a typedef somewhere and mostly transparent

  typedef dynamic C C_d;

  C_d c1 = new(…);

- Without this change the following future changes are more difficult
  - Covergroups that are allowed to be static
  - Structures that are allowed to be dynamic (like in ‘C’, C++)
  - Classes instances that are allowed to be static (like in C++)

Example of future structure allocation

- With proposed change it looks just like classes

  struct { … } S; **class** C … **endclass**;

  dynamic S dynamic_s = **new**({ … });

  dynamic C dynamic_c = **new**( );

- Without change dynamic things look different

  dynamic S s1 = **new**(…); // dynamic structure

  C c1 = **new**(…); // dynamic class

- And without this change there would still be no way to express a static class instance without another keyword use

  static C c1 = **new**(…); // **new**() must have constant expr args for elab

  S s1 = { … }; // static structure

Static Covergroups

- Covergroups evolved as a dynamic testbench construct
  - Covered the stimulus generated randomly
  - Covered the design through XMR/OOMR
- Covergroup are powerful embedded statically

```
module m;
        enum { …. } e_t;      // define the enumerated states
        e_t state_var;        // a variable that holds the states
        covergroup CG @(posedge clk)    // a covergroup for the machine
                sv : coverpoint state_var; endgroup;
        CG cg1;               // a static instance of the covergroup
        // The state machine
        always @(posedge clk) begin … case (state_var) … endcase end;
endmodule
```

Advantage of static covergroups

- Covergroup is embedded in the module so all instances automatically have embedded coverage
  - Same principle as putting assertions embedded in the design
  - Could even extend **bind** statement to allow external specification and insertion of such covergroups
- Static declaration creates a static fanout for the sensitivity of the covergroup
  - Can have a major performance advantage
  - Allows covergroup to participate in global optimization

Proposed change summary

- Allow data types on (at least) scalar and packed nets
- Allow data types on ports acting as nets with default on inputs being changed to net
- Adopt **wone** wire type to express single driver to a net
- Make specification of dynamic objects explicit
- Allow static declaration of covergroups in modules/interfaces


cadence

Advantages of 2-state/4-state

- Accurate hardware behavior is maintained.
- The rules are simple.
- The approach builds upon the treatment of packed structs and unions that contain mixed logic types, providing consistency for users.
- Mixed two-state/four-state network simulation matches synthesis results in "copies", e.g.

```
wire logic a,c;
wire bit   b;
assign a=b;
assign b=c;
```

- The implementation choice of implicit continuous assignments at port connections vs. port collapsing (for performance and capacity) does not produce any new visible differences.