# Constrained Random Data Generation Using SystemVerilog

Tim Pylant, Cadence Design Systems, Inc.



cadence™

# Ideal Stimulus Generation

- Need a way to generate stimulus without long, manual process
- Data should be random to generate patterns not considered
- Data should be legitimate

# Data Item Implementation

- Data-items
  - Represent the main transaction input to the DUT
  - Consistent values are generated and sent
  - Examples include packets, transactions, instructions, and so on
- A wide spectrum of the data-item aspects should be generated by default
  - Almost all legal ranges should be able to be generated
  - A default distribution and preferences could be specified
- Implemented as classes

# Classes

- OO constructs
- Allows encapsulation of attributes, procedural code, constraints, events, etc.
- Allows specification of random variables
- Allows specification of constraints
- Can be extended to further constrain generation
- Raw data can be modeled as SystemVerilog struct and the class has a method for creating a packed struct (useful for working with synthesizable BFMs)

# Class Example

**Spec:**
**A CPU instruction consists of an opcode and two operands. Each operand has 16 bits**

```
package ex_cpu_pkg;
 typedef enum bit [1:0] {CLEAR, WRITE, READ} opcode_e;
 typedef bit [15:0] operand_t;

class instruction_c;
  rand bit        valid;     // valid input instruction
  rand opcode_e   opcode;    // instruction opcode
  rand operand_t  op_a;      // operand a
  rand operand_t  op_b;      // operand b

  function void do_print();
    $display("Instruction: valid=%b opcode=%s op_a=%h
              opb=%h", valid, opcode.name(), op_a, op_b);
  endfunction : do_print

endclass : instruction_c
endpackage
```

This is a control field for generation purposes only, that is not sent to the DUT

A class can combine data and functionality. A struct cannot.



cadence

# Constraints

- There are three different types of constraints
  - Spec driven constraints
  - Default preference constraints
  - Test/config constraints

```
class instruction_c;
  rand bit       valid;
  rand opcode_e  opcode;
  rand operand_t op_a;
  rand operand_t op_b;
  constraint a1 {op_a <= 'hff; }
endclass
```

**Spec:**
**First operand can not be larger than 'hff**

No semi colon at end



cadence

# Generating Random Data

- In System Verilog one can disable a constraint block explicitly

```
program test;
  initial begin
    instruction_c instr;
    instr = new();
    assert(instr.randomize());
    instr.do_print();
    // disable the c1 constraint block
    instr.c1.constraint_mode(0);
    assert(instr.randomize());
    instr.do_print();
 end
 …
```

constraint c1 {op1 <= ‘hff; }

For sure smaller than ‘hff (default constraint)

Not necessarily smaller (constraint disabled)

*This can also be achieved in a declarative way using inheritance (will be discussed later)*



cadence™

# Variable Sized Data Items

- Data items are generated and sent to the DUT
- Some protocol requires variable payload length

```
class idt_packet_c;
rand bit [5:0] length;
local data_t payload [];  // Dynamic array

function void randomize_payload();
  for (int i=0; i<length; i++)
    payload[i] = $urandom;
endfunction

function void post_randomize();
  payload = new[length];
  randomize_payload();
endfunction
endclass
```

```
class small_idt_packet_c extends idt_packet_c;
function void randomize_payload();
  data_t list_item;
  for (int i=0; i<length; i++) begin
    assert(randomize(list_item) with
      {list_item < 5; list_item >= 0;});
    payload[i] = list_item;
  end
endfunction
endclass
```

Use of inheritance

# The Concept of Virtual Control fields

- It is useful to have extra control fields to data-items
  - Not part of the design spec and are not sent to the DUT
  - Simplify test creation and are useful for coverage
  - E.g. a packet might have a control field that determines if a cyclic redundant check (CRC) result is legal

```
typedef enum {ARITH, FLOW} inst_kind_e;
class instruction_c;
   rand inst_kind_e kind;
   rand opcode_e opcode;
   constraint kind_knob {
     (kind == ARITH) -> opcode inside {ADDI, ADD, SUB,SUBI};
     (kind == FLOW)  -> opcode inside {JMP, JMPC, CALL, RETURN};}
   constraint kind_dist {kind dist {ARITH := 3, FLOW := 1};}
endclass: instruction_c
```

Allows things like:
- Specify distribution of ARITH versus FLOW generation
- Coverage of the above
- Cross this coverage with other coverpoints

# Constraints Layering

- Randomness is effective in exploring unanticipated areas
- At some point, users may want to change the default distribution of the generated data in order to nail down a corner case
  - Changes could be required all over the environment
  - Changes could be required for a specific driver
- Object Oriented calls for deriving a new class from the parent class and adding the constraints in the new class
- Polymorphism allows overriding a parent class usage in the environment
- Still a problem occurs at initialization
- Let’s look at example …

# Constraints Layering Example 1

```
class packet_c; // part of the reusable package
    rand int data;
    rand int length;

    constraint length_const {length >= 0; length < 100;}

endclass

// the user may initialize the class in multiple locations in the code
packet_c cur_packet = new;

// How would a test call for packets with short length only?

class short_packet_c extends packet_c;
    constraint new_delay_const {length < 20;}
endclass
```

Now we need to visit all the locations in the code that define packet_c to use short_packet_c

# Constraints Layering Example 2

- The driver clones a random data-item upon request

```
// In a test1.v
program test ();

class short_packet_c extends packet_c;
constraint short_length {length < 20;}
endclass

initial begin
  short_packet_c pkt;
  pkt = new();
  top.driver.set_packet(pkt);
  top.driver.gen_packet(10);
end

endprogram
```

Step1:define a new class

Step2:set the desired Driver to use the new class

```
module master_driver(…);

packet_c packet;

function void set_packet(packet_c p);
  packet = p;
endfunction

initial begin
  // initialize the abstract struct if needed
  if (!packet) packet = new();
     …
end

endmodule
…
// generates the required sub-class
Assert(packet.randomize());
```

Step3:This driver will generate short_packets

# Constraints Layering Example 3

- Add additional constraints when calling randomize
  - Use a macro to simplify the call

```
class packet_c; // part of the reusable package
    rand int data;
    rand int length;

    constraint length_const {length >= 0; length < 100;}
endclass

packet_c packet = new();

initial begin
  assert(packet.randomize() with {length < 20;});
end
```

```
`define GEN_PKT(constr) \
  assert(packet.randomize() with {constr});

initial begin
  `GEN_PKT(length < 20;)
end
```

# Summary

- Classes provide a powerful way to easily generate random data
- Use built-in functions (methods) to add additional flexibility to manipulate the data
- Use class extensions and generic object creation to provide a way to add additional constraints