Mantis 696

P1800-2012

Motivation

This Mantis item enables the use of parameterized functions and tasks. This is done primarily through the use of static methods inside of classes. The committee debated enhancing the existing syntax of functions and tasks to enable this; we realized pragmatically that using static class methods already enabled this feature.

Add sub-clause 13.8 to the Task and Function Clause as follows:

13.8 Parameterized Tasks and Functions

SystemVerilog provides a way to create parameterized tasks and functions, also known as parameterized subroutines. A parameterized subroutine allows the user to generically specify or define an implementation. When using that subroutine one may provide the parameters that fully define its behavior. This allows for only one definition to be written and maintained instead of multiple subroutines with different array sizes, data types, and variable widths.

The way to implement parameterized subroutines is through the use of static class methods (see 8.9 Static Methods and 8.24 Parameterized classes). The following generic encoder and decoder example shows how to use static class methods along with class parameterization to implement parameterized subroutines. The example has one class with two subroutines that, in this case, share parameterization. The class may be declared virtual in order to prevent object construction and enforce the strict static usage of the method.

```
class C#(parameter DECODE_WIDTH = 16, parameter ENCODE_WIDTH = 4);
  static function logic [ENCODE_WIDTH-1:0] ENCODER_f(input  logic [DECODE_WIDTH-1:0]
DecodeIn);

      ENCODER_f = '0;
      for (int i=0; i<DECODE_WIDTH; i++) begin

         if(DecodeIn[i]) begin
           ENCODER_f = i[ENCODE_WIDTH-1:0];
           break;
         end

      end

  endfunction

  static function logic [DECODE_WIDTH-1:0] DECODER_f(input  logic [ENCODE_WIDTH-1:0]
EncodeIn);

      DECODER_f = '0;
      DECODER_f[EncodeIn] = 1'b1;

  endfunction
endclass
```

The class contains two static subroutines, ENCODER_f and DECODER_f. Each subroutine is parameterized by the class using the parameters DECODE_WIDTH and ENCODE_WIDTH, both of which have default settings. These parameters are used within each subroutine to define the size of the encoder and decoder.

```
module top ();
   logic [7:0] encoder_in;
   logic [2:0] encoder_out;
   logic [1:0] decoder_in;
   logic [3:0] decoder_out;
```

```
   // Encoder and Decoder Input Assignments
   assign encoder_in = 8'b0100_0000;
   assign decoder_in = 2'b11;

   // Encoder and Decoder Function calls
   assign encoder_out = C#(8,3)::ENCODER_f(encoder_in);
   assign decoder_out = C#(4,2)::DECODER_f(decoder_in);

   initial begin
      #50;
      $display("Encoder input = %b  Encoder output = %b\n", encoder_in, encoder_out );
      $display("Decoder input = %b  Decoder output = %b\n", decoder_in, decoder_out );
   end

endmodule
```

The top level module first defines some intermediate variables used in this example, and then assigns constant values to the encoder and decoder inputs. The subroutine call of the generic encoder, ENCODER_f, uses specialized class parameter values of 8 and 3 that represent the decoder and encoder width values respectively for that specific instance of the encoder while at the same time passing the input encoded value. This expression uses the static class scope resolution operator '::' (see 8.22) to access the encoder subroutine. The expression is assigned to an output variable to hold the result of the operation. The subroutine call for the generic decoder, DECODER_f, is similar using the parameter values of 4 and 2 respectively.