Using SystemVerilog for FPGA Design

A tutorial based on a simple bus system

Doulos January 2010

SystemVerilog includes a number of enhancements to the Verilog language that are useful for FPGA design. Synthesis tools from FPGA vendors and EDA tool vendors enable SystemVerilog designs to be described using easier-to-understand styles and higher levels of abstraction than were possible in Verilog, speeding up the coding process and easing reuse. This article looks at how synthesisable SystemVerilog can be written for various functional blocks commonly used in FPGAs. An example design based around a generic bus with multiple arbitrated masters and multiple slaves is used to illustrate synthesisable coding styles. Some of the techniques described here have been borrowed from VHDL while others are only possible with SystemVerilog.

Representing Logic Values

A standard 4-valued type named logic is defined in the SystemVerilog language. This represents the “type” that was implicitly used for Verilog variables (reg) and wires (0,1,X and Z). You should use this type for single bit ports and variables in your synthesisable code. Multi-bit ports and variables can be defined by vectors of type logic. The following example shows part of a counter module with 1-bit wide input and output ports and an 8-bit wide internal variable. Note that SystemVerilog allows variables to be set from a single continuous assignment statement (see output q in the example), thus removing the requirement for internal wires that would have been required in Verilog.

```
module counter(input logic clk,reset,enable,output logic q);
    logic [7:0] count;
    ...
    assign q = count[7];
endmodule: counter
```

SystemVerilog allows users to define new types (e.g. a logic vector of a particular length) and to give these types meaningful names: this can make the code easier to read and increases reusability if changes can be made to a few common typedefs rather than many individual variable declarations. Type definitions are therefore often placed within a package that can be imported into other modules. User-defined types are supported by SystemVerilog synthesis tools provided the underlying types are themselves syntheisable. The following example shows the definition of a package containing a new type that represents an 8-bit logic vector and a packed struct that consists of two 4-bit logic vectors. The significance of declaring a struct to be packed is that the whole struct is treated by the synthesis tool exactly like a vector (logic [7:0] in this case) but its elements can be accessed by name in the code.
package types;
  typedef logic [7:0] wdata_t;
  typedef struct packed {logic [3:0] data_h;
                  logic [3:0] data_l;
            } wdata_struct_t;
endpackage: types

Combination Logic

Most designers find it easiest to describe combinational logic functions procedurally (rather than as a set of continuous assignments). SystemVerilog always_comb processes are designed for this purpose and automatically work out what signals will trigger the process, so the designer does not have to add a sensitivity list in the code. Designers sometimes inadvertently infer latches when they write HDL code for their combinational logic: the always_comb statement tells the synthesis tool that latches should not be inferred by the process so it will issue a warning if it finds any.

SystemVerilog also includes operators such as ++ and break (borrowed from C/C++) that can simplify the code while making it easier to understand. The following code shows part of the logic to multiplex the bus master signals in our example design. It uses a for loop with a break statement to select the master signals with the highest priority. Note that in SystemVerilog, the loop index variable of a for loop can be declared as part of the loop statement (so it is local to the for loop and cannot have unexpected side-effects elsewhere).

always_comb begin
  //default assignments prevent latches
  addr_a = m_addr[0];
  wdata_a = m_wdata[0];
  RE_a = m_RE[0];
  WE_a = m_WE[0];

  //check all m_gnt signals - start at 0 in case only 1 master
  for (int i=0; i < nmasters; i++) begin
    if (m_gnt[i]) begin //lowest index has priority
      addr_a = m_addr[i];
      wdata_a = m_wdata[i];
      RE_a = m_RE[i];
      WE_a = m_WE[i];
      break;
    end
  end
end

Sequential Logic

The registers (flip-flops) in a synthesisable Verilog design are usually defined by an always process that is sensitive to a clock edge (and often an asynchronous reset too). SystemVerilog adds the always_ff construct for describing processes that represent sequential logic. Synthesis tools are required to check that always_ff processes really do represent sequential logic (only one timing control, no blocking timing control statements and no assignments to variables that are also written to by other processes). It is usual to use non-blocking assignments to variables within an always_ff process to avoid simulation race conditions between multiple processes that are
triggered by the same clock edge (just as it is with clocked always processes in Verilog). The following example shows the process for a counter module. Note that the counter value has not been incremented by writing `count++` since this is treated as a blocking assignment and some synthesis tools do not allow blocking and non-blocking assignments to the same variable within a process.

```verilog
always_ff @(posedge clk or posedge reset) begin
  if (reset == 1'b1)
    count <= 0;
  else
    if (enable == 1'b1) count <= count + 1'b1;
end
```

### Memories

Arrays with multiple dimensions may be declared in SystemVerilog. These can have packed dimensions (dimension specified immediately before the variable name) or unpacked dimensions (dimension specified immediately after the variable name). A two-dimensional array of logic with one packed dimension and one unpacked dimension can be synthesised as an embedded block RAM if the array is accessed in a process whose controls match those of the actual RAM. The following example is a complete module that is synthesised as synchronous RAM with a size set by the module parameter.

```verilog
module slave #(parameter asize = 8)(interface bus);
  import types::*;

  wdata_t [(2**asize)-1:0] memory;

  always_ff @(posedge bus.clk) begin
    if (bus.WE)
      memory[bus.addr] <= bus.wdata;
    if (bus.RE)
      bus.rdata <= memory[bus.addr];
  end
endmodule: slave
```

### Finite State Machines

Many FPGA designs contain finite state machines (FSMs). Writing synthesisable code for these in SystemVerilog is more flexible than in Verilog since enumerated types can be used to represent the set of states. When an enumerated type is used for the state register, the compiler will check that only legal enumerated values are assigned to the state register. By default, FPGA synthesis tools will pick an appropriate encoding pattern to represent each enumerated value. In many cases this will be one-hot encoded to produce the most efficient implementation in the FPGA (the encoding actually used can be controlled by setting options in the synthesis tools). Several styles of writing the code for a FSM are popular (e.g. a single clocked process or one clocked process plus one combinational process). The following example shows part of a single-clocked-process FSM in our bus master module.
typedef enum logic[5:0] {IDLE, REQ, WAITING, READ0, READ1, WRITE0} bus_trans_t;
bus_trans_t bus_trans_state;

always_ff @(posedge bus.clk or posedge bus.reset) begin
  //asynchronous reset
  if (bus.reset) begin
    acount = 0;
    bus.req <= 0;
    bus_trans_state <= IDLE;
    bus.WE <= 0;
    bus.RE <= 0;
    bus.addr <= 0;
    bus.wdata <= 0;
  end
  else begin
    case(bus_trans_state)
      IDLE: begin
        bus_trans_state <= REQ;
        bus.WE <= 0;
        bus.RE <= 0;
      end
      REQ: begin
        bus.req <= 1;
        bus_trans_state <= WAITING;
      end
      WAITING: begin
        if (bus.gnt) begin
          acount++;
          bus_trans_state <= READ0;
          bus.WE <= 0;
          bus.RE <= 1;
          bus.addr <= {1'b0,acount};
        end
      end
      READ0: begin
        bus_trans_state <= READ1;
        bus.WE <= 0;
        bus.RE <= 0;
      end
      READ1: begin
        //set data struct from rdata vector
        data = bus.rdata;
        //manipulate data struct members
        data.data_h[2:0] = {data.data_h[2:0],data.data_h[3]};
        data.data_l[2:0] = {data.data_l[2:0],data.data_l[3]};
        bus_trans_state <= WRITE0;
        bus.WE <= 1;
        bus.RE <= 0;
        bus.addr <= {1'b1,acount};
        bus.wdata <= data;
      end
      WRITE0:begin
        bus.req <= 0; //clear request
        bus_trans_state <= IDLE;
        bus.WE <= 0;
      end
  end
Hierarchical Design

There are two main reasons why creating a hierarchy in a SystemVerilog design is simpler than in Verilog. The first reason is that SystemVerilog lets you use variables to connect module instance ports, you don’t have to declare wires. This means that FPGA designers can stop worrying about when they should use wires and when they should use variables. A general rule is to use variables all the time (except for a few special cases, such as tri-states and bi-directional pins). The second reason is implicit .name port connections. These allow the name of the signal connected to a specified port to be omitted from the port list if its name is exactly the same as the port name (e.g. if the port and signal were both named clk). It is even possible to use the wildcard (‘*’) instead of the port names: this means connect every port to a signal with the same name as the port. However, wildcard port connections are disliked by some designers since it is harder to trace port connections down through the hierarchy if the port names are not listed in the module instantiations. The following code shows the top level of our bus-based design and includes a mix of implicit and explicit port connections. The masters and slaves are connected to interface ports. Interfaces are explained in the next section.

module top(input logic clk, reset, enable, output logic RE, output logic WE, output types::addr_t addr, output types::rdata_t rdata, output types::wdata_t wdata);

logic en_data_gen, en_data_gen_b;

intf #(.nmasters(2), .nslaves(2)) bus(.clk,.reset);
data_gen m1(.enable(en_data_gen_b),.bus(bus.master[0].mport));
master m0(.bus(bus.master[1].mport));
slave #(.asize(7)) s0(.bus(bus.slave[0].mport));
slave #(.asize(7)) s1(.bus(bus.slave[1].mport));
counter c0(.clk,.reset,.enable,.q(en_data_gen));

assign addr = bus.addr_a;
assign rdata = bus.rdata_a;
assign wdata = bus.wdata_a;
assign RE = bus.RE_a;
assign WE = bus.WE_a;
assign en_data_gen_b = !en_data_gen;

endmodule: top
Bus Interfaces and Bus Fabric Logic

One of the most attractive reasons for using SystemVerilog for FPGA synthesis is its ability to represent the complex connections and logic associated with on-chip busses in an easy-to-manage block known as an interface. Unlike most of the topics discussed so far, this does not have any equivalent construct in VHDL (not even the 2008 version).

A SystemVerilog interface is similar to a module in so far as it can have ports and contain processes, but unlike a module it can connect to a port. An interface that represents all of the wires within an on-chip bus therefore only requires a single port connection to each master and slave on the bus. Furthermore, modports within the interface allow master ports and slave ports to have different characteristics.

The following code segments show the main points of the interface that represents the arbitrated bus in our simple example system. The first segment shows the interface's ports for the clock and reset required by the bus logic. It also shows how parameters are used to set the size of the internal signals to reflect the number of master and slaves connected to the bus.

```systemverilog
interface intf #(parameter int nmasters=2, nslaves=1) (input logic clk, reset);
import types::*;
localparam mvec_size = $clog2(nmasters);
localparam svec_size = $clog2(nslaves);

//arbitrated signals
addr_t addr_a;
word_t wdata_a;
read_t rdata_a;
logic RE_a;
logic WE_a;

//signals to/from each master
addr_t[mvec_size:0] m_addr;
word_t[mvec_size:0] m_wdata;
logic[mvec_size:0] m_req;
logic[mvec_size:0] m_gnt;
logic[mvec_size:0] m_RE;
logic[mvec_size:0] m_WE;

//signals to/from each slave
read_t[svec_size:0] s_rdata;
logic[svec_size:0] s_RE;
logic[svec_size:0] s_WE;
```

Next, modports need to be defined for each master and slave that will be created. In the second segment, we have used a generate statement that uses the nslaves and nmasters parameters to create the right number of modports. At the time of writing, not all synthesis tools support the use of generate within an interface. An alternative to using generate would be to create an interface with internal signals and modports to support the maximum number of masters and slaves expected. Any unconnected signals and modports will then be removed automatically by the
optimiser during synthesis. Ideally, each master and slave that is connected to the bus should not be aware of the internal structure of the bus (e.g. which elements of multiplexed signals are associated with each connection). Every master and slave modport (remember generate will create arrays of modports) can be given a generic set of port names by using "modport expressions", for example each master can access a port named req in its modport that is internally connected to the interface signal \( m\_req[0] \) or \( m\_req[1] \), etc.

```verilog
generate
genvar m,s;
for (s=0; s < nslaves; s++) begin:slave
    modport mport(input clk, reset,
                  input .addr(addr_a[(slog2(nslaves))-0]),
                  output .rd(s_rdata[s]),
                  input .wd(wdata_a),
                  input .RE(s_RE[s]),
                  input .WE(s_WE[s]));
end

for (m=0; m < nmasters; m++) begin:master
    modport mport(input clk, reset,
                  output .req(m_req[m]),
                  input .gt(m_gnt[m]),
                  output .adr(m_addr[m]),
                  input .rd(rdata_a),
                  output .wd(m_wdata[m]),
                  output .RE(m_RE[m]),
                  output .WE(m_WE[m]));
end
dendgenerate
```

The bus logic for arbitration, address decoding and multiplexing the master and slave outputs is specified in several processes within our interface. Here is the process that defines the simple arbitration used in this example: it sets a bit in the \( m\_gnt \) register that corresponds to the master that has been granted access to the bus (the master with the lowest index that has a request pending). The code for the process that multiplexes the master outputs has already been shown: the address decoder logic process is similar.

```verilog
always @(posedge clk or posedge reset) begin
    if (reset) begin
        m_gnt <= 0;
    end
    else begin
        if (m_gnt) begin //gnt persists until master drops req
            for (int i=0; i < nmasters; i++)
                if (m_gnt[i]) m_gnt[i] <= m_req[i];
        end
        else begin //no master currently has gnt
            for (int i=0; i < nmasters; i++) begin
                if (m_req[i]) begin //master i has priority
                    m_gnt[i] <= 1;
                    break;
                end
            end
        end
    end
end
```

Copyright © 2010 by Doulos Ltd. All rights reserved. All information is provided “as is” without warranty of any kind.
Conclusions

If you are currently using Verilog for FPGA design, there are clear advantages in updating your design flow to use SystemVerilog instead (it will still be backwards compatible with all of your existing Verilog code). If you are just starting out, it makes sense to choose SystemVerilog as your first language to learn for FPGA design. If you currently use VHDL you might be wondering if the effort in switching to SystemVerilog could be justified? If your FPGAs contain on-chip multiplexed busses with multiple masters and slaves, it probably is!

Once you know the basics of using SystemVerilog for design, you might be tempted to have a look at the testbench features in the SystemVerilog language. SystemVerilog is a "big" language and most of the constructs listed in the LRM are there to support advanced verification methodologies using techniques such as constrained random generation of test vectors, functional coverage to see what aspects of your design have been fully tested, assertions to catch design errors, etc. In this respect, SystemVerilog is much more powerful than either VHDL or Verilog but that is another story!

All of the SystemVerilog features used in this example are covered in-depth in our Comprehensive SystemVerilog course (together with many other features not discussed here). See our web site pages for further details.