Design Creation & Synthesis Division

Avoid FPGA Project Delays by Adopting Advanced Design Methodologies

Alex Vals, Technical Marketing Engineer
Mentor Graphics Corporation
June 2008

Mentor Graphics®

Introduction

Over 40% of FPGA design projects fall behind schedule. In order to reduce risk of delay of product delivery, changes need to be made not just in verification but also in the design process itself. Design simplification must be a principle that starts at the beginning of the project life cycle – before verification of complex code has become the bottleneck that delays project delivery.

We will examine two concepts here that can simplify the implementation of a complex FPGA design: the use of SystemVerilog and vendor-independent modeling.

No Time for Change

Most design teams are reluctant to change their design process even if they offer more efficiency. They fear that that attempts to adopt new methodologies to avoid project delays can be delays in and of themselves. With such rigid tape-out schedules, they consider it safer to stay with "what works".

But current methodologies are, in fact, not working as well as they could. Nearly 20% of project delays are caused by design-and-verification issues. Another 60% are caused by system-integration, test, and verification issues. The same customer surveys show that today's engineers spend over 30% of their time on verification and this is expected to grow to roughly 40% in the next two years.

One way to look at both design and verification as an interrelated problem is to ask to what degree the number of bugs correlate to the number of lines of RTL code and the code's complexity. In the past we were able to perform detailed design reviews within a few days. For today's designs, because of the greater amount of RTL code, this process is more laborious and takes much more time—and it is expected to get worse. More than 60% of designers today expect the total number of lines in their RTL code to increase by 20% in their next project. Even taking into account design reuse methodologies, more potential bugs per design are expected due to greater size and complexity.

The design review process is only one area where the length of the code impacts schedule. Larger designs also cause longer simulation runs, and lengthen project schedules even further. These lengthened design reviews and simulation runs are only meant to identify bugs. Unfortunately, even more time must be consumed in correcting these bugs and verifying the fixes. As a result, for larger designs, project schedule increases considerably, as shown in the bug histogram in Figure 1.

So how should this problem be addressed? In conjunction with improving the verification process, design engineers can reduce the number of bugs in their designs by adopting SystemVerilog for design and vendor-independent modeling.

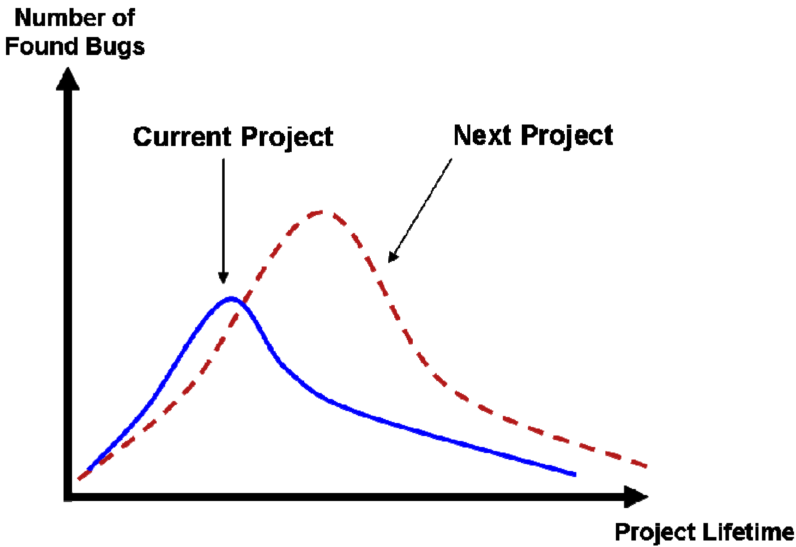


Figure 1: Bug Histogram

SystemVerilog for Reducing Complexity

A wise man once said “make things simple”. So how is it possible to simplify RTL code while the complexity of the design itself is increasing? The answer is to use higher level language structures. Studies have shown that using SystemVerilog for design entry may reduce the size of the design by a factor of two to five.

SystemVerilog for design has experienced slower adoption than the language’s verification and assertion constructs. Among the reasons for this includes the effort of changing design methodology, training design engineers, and, of course, finding adequate tool support.

One design team leader of a well known semiconductor company stated that his engineers have not adopted SystemVerilog for two primary reasons. First and foremost, they simply have no time. They release a new chip every few months and can not allocate time between projects to research the new language extensions. Secondly, they claim that the available tool support is inadequate.

This latter claim is no longer the case — today’s market leading design tools have comprehensive SystemVerilog support that allow for seamless usage of the language. Precision FPGA Synthesis gives designers comprehensive support for the SystemVerilog IEEE-1800 standard. Many users today implement their SystemVerilog designs into FPGAs using Precision.

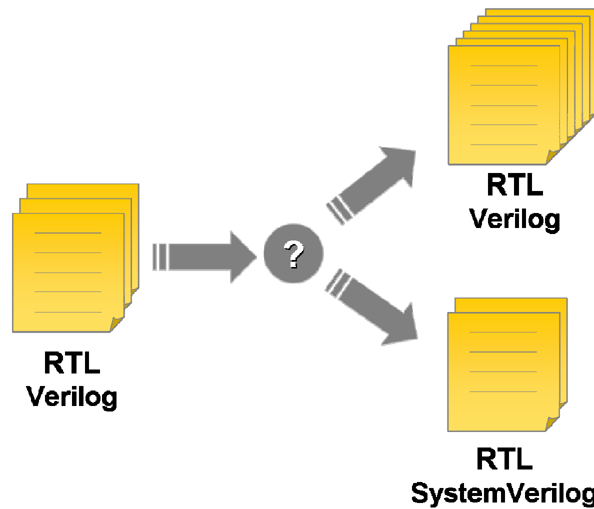


Figure 2: Impact of SystemVerilog on Code Complexity

There are many useful design constructs in SystemVerilog, but only a sub-set are examined here. The following is an example of parameterized interfaces, where I/O ports are bundled and reused among different modules, reducing the risk of error during top level integration.

In this example, a behavioral model of Single Port RAM is instantiated under the top level module. The use of parameterized interfaces prevents the common mistakes of port mismatch and wrong interface interconnections.

```
/*---------------------------------------------------------------------
--  Copyright (c) 2007 Mentor Graphics Corporation.
--
--  All rights reserved. The following information has
        been
--  generated by Mentor Graphics and may be freely
--  distributed and modified.
--
--  Purpose : This design shows how the interface
        feature
--           can he used to bundle and reuse I/O
        among
--           different modules in the design.
--           It also illustrates how modport is used to
--           define port directions.
--
--  The following constructs in SV are used:
--     - interface, typedef, always_comb
--
--  Version: 1.0
--  Precision:  2007a
----------------------------------------------------------------------*/
interface address_bus;
  parameter p1 = 7;
  wire [p1:0] read_address;
  wire [p1:0] write_address;
  modport address_port_in (input  read_address, input
        write_address);
endinterface

interface data_bus;
  parameter p2 = 7;
  wire [p2:0] data_in;
  wire [p2:0] data_out;
  modport data_port (input data_in, output data_out);
endinterface

interface control_bus;
  wire reset;
  wire clk;
  wire wren;
  wire ren;
  modport memory_control (input reset, input clk,
        input wren, input ren);
endinterface

//-----------------------------------------------------

module memory (address_bus.address_port_in addr,
                control_bus.memory_control   ctrl,
                data_bus.data_port           data);
  parameter p1 =7;
  parameter p2 =7;

  typedef logic [7:0] BYTE;
    BYTE memory[(2*p1+1):0][(2*p1+1):0];
    BYTE latch;

assign  data.data_out = latch;

  always @ (posedge ctrl.clk)
  if(ctrl.reset)
    for(int i = (2*p1+1); i >= 0; i--)
      for(int j = (2*p1+1); j >= 0; j--)
         memory[i][j] = '0;
    latch = '0;
  else if(ctrl.ren)
      latch =
         memory[addr.read_address[3:0]][addr.read_add
         ress[7:4]];
  else if(ctrl.wren)

         memory[addr.write_address[3:0]][addr.write_ad
         dress[7:4]] = data.data_in;

endmodule

//-----------------------------------------------------

module top (clk, reset, wren, ren, raddr, waddr, wdata,
        rdata);
  parameter p1 =7;
  parameter p2 =7;

  input clk, reset, wren, ren;
  input  [p1:0] raddr, waddr;
  input  [p2:0] wdata;
  output [p2:0] rdata;
  logic  [p2:0] rdata;

  // Instantiate interfaces
  address_bus #(.p1(p1)) addr();
  data_bus     #(.p2(p2)) data();
  control_bus             ctrl();

  // Instantiate memory module
  memory #(.p1(p1), .p2(p2)) mem_i (.*);

  assign ctrl.clk   = clk;
  assign ctrl.reset = reset;
  assign ctrl.wren  = wren;
  assign ctrl.ren   = ren;
  assign addr.read_address  = raddr;
  assign addr.write_address = waddr;
  assign data.data_in       = wdata;

  always_comb
    rdata = data.data_out;

endmodule
```

As seen from the above example, once the ***interface*** named ***address_bus*** is defined, it may be reused in different modules in the following way:

```
module memory (address_bus.address_port_in addr)
```

where the hierarchical port declaration ***address_bus.address_port_in*** is used for the ***addr*** port. Using such unified and parameterized declarations can help solve port width mismatch problems in designs.

Another useful construct is the implicit port connection. The ***memory*** module is instantiated using these type of connections:

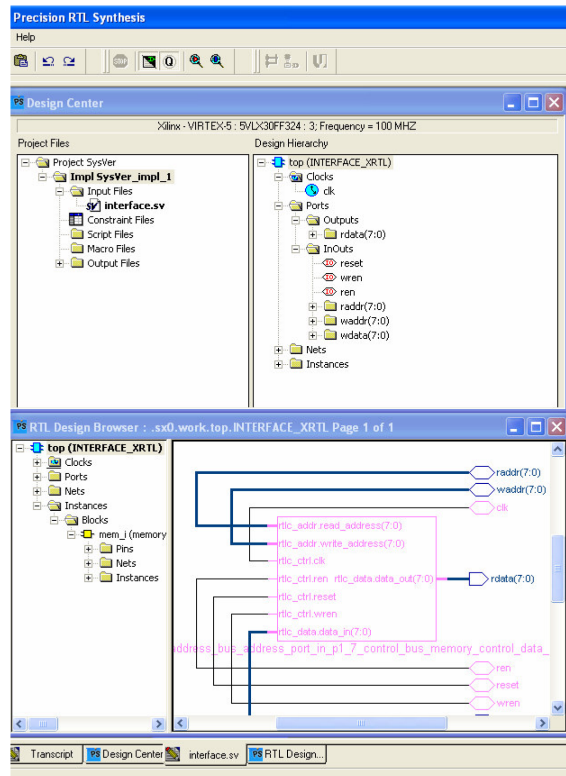
```
memory #(.p1(p1), .p2(p2)) mem_i (.*);
```

The **(.*)** notation automatically connects instance ports to the ***top*** level signals with the same names and data types. This technique can significantly reduce integration problems, particularly when naming rules are used as part of an in-company convention.

Vendor-Independent RTL Modeling of FPGA cores

Note that in the above example, the memory is inferable by Precision Synthesis and may be implemented using dedicated FPGA memory resources depending on the selected FPGA vendor and device family, as shown in Figure 3. This brings us to the next design methodology that can reduce project delay—the use of vendor-independent RTL models.

Figure 3: Design Implementation



Much has been written about vendor-independent design and its advantages versus instantiation of FPGA vendor's IP cores. While there are the advantages of portability, other advantages include faster simulation times, code-re-use, and flexibility for hardware implementation. All of these make the design and verification flow more efficient.

Most FPGA vendor's IP cores are instantiated as black or gray boxes. These are either linked by synthesis or place-and-route (P&R) tools to generated gate-level netlists or predefined functional blocks. As a result, simulation times drastically increase even at the RTL level versus the same design that uses behavioral models. In fact, the simulation time of a design using inferable RTL models may be two to twenty times faster than the same design using instantiated IP cores.

Vendor-independent RTL models also allow for scalability. The memory block shown above may be used many times using different parameters to implement memories throughout the design. A single parameterized module can be enough to describe several configurations, such as a single-port or dual port memory, using either write-first, read-first, or no-change write-modes. These configurations cover the majority of requirements of mainstream FPGA design. This methodology would require four files versus tens or hundreds of files when using vendor IP blocks.

Vendor-independent models also leverage the flexibility of Precision Synthesis to implement the logic using different FPGA resources. Designers may control resource allocation of memory blocks and arithmetical functions during synthesis using either synthesis attributes or via a graphical

resource management interface. Controlling resource allocation without changing code or regenerating many IP blocks may considerably shorten the design process.

Conclusion

SystemVerilog and vendor-independent modeling are just two simple ways to ease the design effort . Because inefficient coding styles can affect all downstream design activities, improving them can have considerable impact on overall schedule. While adopting these methods requires a time investment, it is relatively minor and the time savings gained will pay-off many times over in terms of easier debug, shorter simulation times, code re-usability, and flexibility in hardware implementation. Project delivery can finally be predictable by efficiently generating reliable, bug free code for complex designs.