如何采用 SystemVerilog 来改善基于 FPGA 的 ASIC 原型

关键词:FPGA, ASIC, SystemVerilog

摘要:ASIC 在解决高性能复杂设计概念方面提供了一种解决方案,但是 ASIC 也是高投资风险的,如 90nm ASIC/SoC 设计大约需要 2000 万美元开发成本.为了降低成本,现在可采用 FPGA 来实现 ASIC.但是,但 ASIC 集成度较大时,需要几个 FPGA 来实现,这就需要考虑如何来连接 ASIC 设计中所有的逻辑区块.采用 SystemVerilog,可以简化这一问题.

How to improve FPGA-based ASIC prototyping with SystemVerilog

FPGA prototyping is not without its difficulties; one major obstacle has been connecting all the logic blocks both within an FPGA and across multiple FPGA devices...

By Roger Do, Mentor Graphics

Introduction

ASICs provide a solution for capturing high performance complex design concepts and preventing competitors from simply implementing comparable designs.

However, creating an ASIC is a high-investment proposition with development costs approaching $20M for a 90 nm ASIC/SoC design and expected to top $40M for a 45 nm SoC. Thus, increasingly, only a high-volume product can afford an ASIC.

Besides the increase in mask-set cost, total development cost is also increasing due to the reduced probability of getting the design right the first time. As design complexity continues to increase, surveys have shown that only about a third of today's SoC designs are bug-free in first silicon, and nearly half of all respins are reported as being caused by functional logic error(s). As a result, verification managers are now exploring ways to strengthen their functional verification methodologies.

Before starting on a true ASIC design, to demonstrate that concepts are sound and that designs can be implemented, a lower-cost method of using FPGAs to prototype ASIC designs as part of an ASIC verification methodology has been growing in popularity.

Prototyping ASIC designs in FPGAs, while often yielding different performance, often results in the same logical functionality. Further, running a design at speed on an FPGA prototype with real stimulus allows for a far more exhaustive and realistic functional coverage as well as early integration with embedded software. Thus FPGA prototyping can be used effectively to supplement and extend existing functional verification methodologies.

As ASIC designs have grown larger at a much faster pace than FPGA devices, often multiple FPGA devices must be used to prototype a single ASIC. The obstacle of using multiple devices is the task of connecting all of the logical blocks of the ASIC design across multiple FPGA devices. Physically, with the use of the high speed I/O blocks in FPGA devices, connectivity between physical devices has been simplified. However, methods for logically connecting the design blocks have proven to be manually intensive and error prone. With the introduction of SystemVerilog, an evolutionary RTL language, and advanced mixed language synthesis tools such as Mentor Graphics' Precision Synthesis, the procedure for connection has also been simplified.

SystemVerilog

SystemVerilog is not an entirely new RTL language. With its rich set of extensions to the existing Verilog HDL, SystemVerilog is backward compatible with both Verilog 95 and Verilog 2001. Many of these extensions to Verilog make it easier to create accurate, synthesizable models of designs of any size. These extensions also make SystemVerilog easier to use and are truly beneficial to every engineer currently working with Verilog.

The connectivity advantages of SystemVerilog stem from:

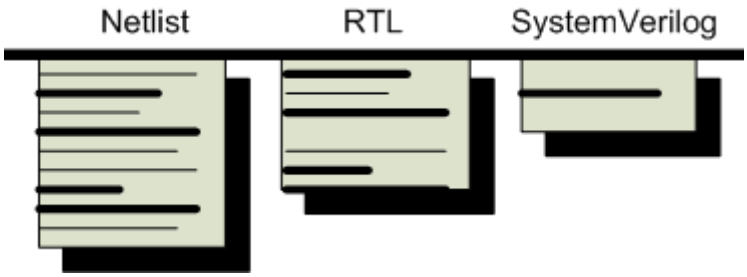
1. More compact RTL descriptions with efficient coding methods.

2. Encapsulation, allowing designers to model at more abstract levels with interface descriptions.

Using SystemVerilog for FPGA prototyping does not necessarily mean that the entire ASIC design needs to be written in SystemVerilog to reap the benefits. The obstacle of connectivity can be simplified by just using SystemVerilog to describe the top level module of each FPGA.

Compacting the code

Increased design sizes have increased the number of lines of RTL code required to represent the design. Design bugs can actually be attributed to the number of lines of code written. SystemVerilog results in improved specification of design, more concise expressions and the unification of verification and design. All of which results in earlier time to market and early detection of design bugs. In fact, SystemVerilog can be two to five times more compact than Verilog RTL.



Both VHDL and Verilog have positional and named port connections. Positional ports can be mis-ordered, while named ports can be too verbose and redundant, especially at the top level modules.

SystemVerilog has **.name** and **.* port** connections. These methods provide a more concise and less error prone method to describe connectivity. Inherent also in this methodology is a stronger typing on port connections. Port sizes must match, ports cannot be omitted, and all ports must be declared.

Implicit Port Connection features provide designers with very important capabilities, which are not currently available from any other HDL languages. These features provide immediate benefits to both ASIC and FPGA designers, especially in the area of FPGA prototyping. Not only can designers save up to 75% of coding for the top-level instantiation, these features also provide strong, VHDL-like and less error-prone coding styles as illustrated by coding examples.

Comparing a simple, top-level design example:

- With Verilog port interfaces:
  - 250 words / 1,770 characters / 122 lines

  .

- With SystemVerilog .* implicit port interfaces:
  - 72 words / 492 characters / 37 lines

Furthermore, the interesting side-effects of the implicit port connection enhancements include the following:

- Significant reduction in code required to model port connections.
- Stronger VHDL-like typing on port connections.
- Reduction in port-size instantiation errors since all port sizes must match.

- Reduction in omission of ports since all unconnected ports must be listed.
- Less repetitive, time consuming, and error-prone than assembling the top-level design in VHDL or Verilog.

Thus, the top-level instantiation of modules to each other can be much simpler by employing SystemVerilog at the top level of each FPGA device. The lower level blocks do not necessarily have to be converted to SystemVerilog.

Encapsulation

Connection of a module to the I/O blocks of the FPGA device presents another dilemma. More often than not, when modules are separated in ASIC designs, the number of total I/O's needed for connection between FPGA devices is now greater than the number available on these devices.

Most often pins must be multiplexed in order to accommodate all of I/O's. SystemVerilog provides a feature that can help in providing this functionality for FPGA prototyping without having to modify the ASIC design.

Verilog connects one module to another through module ports. This requires a detailed knowledge of the intended hardware design, in order to define the specific ports of each module that makes up the design. Several modules often have many of the same ports, requiring redundant port definitions for each module. Every module connected to a data bus protocol, for example, must have the same ports defined.

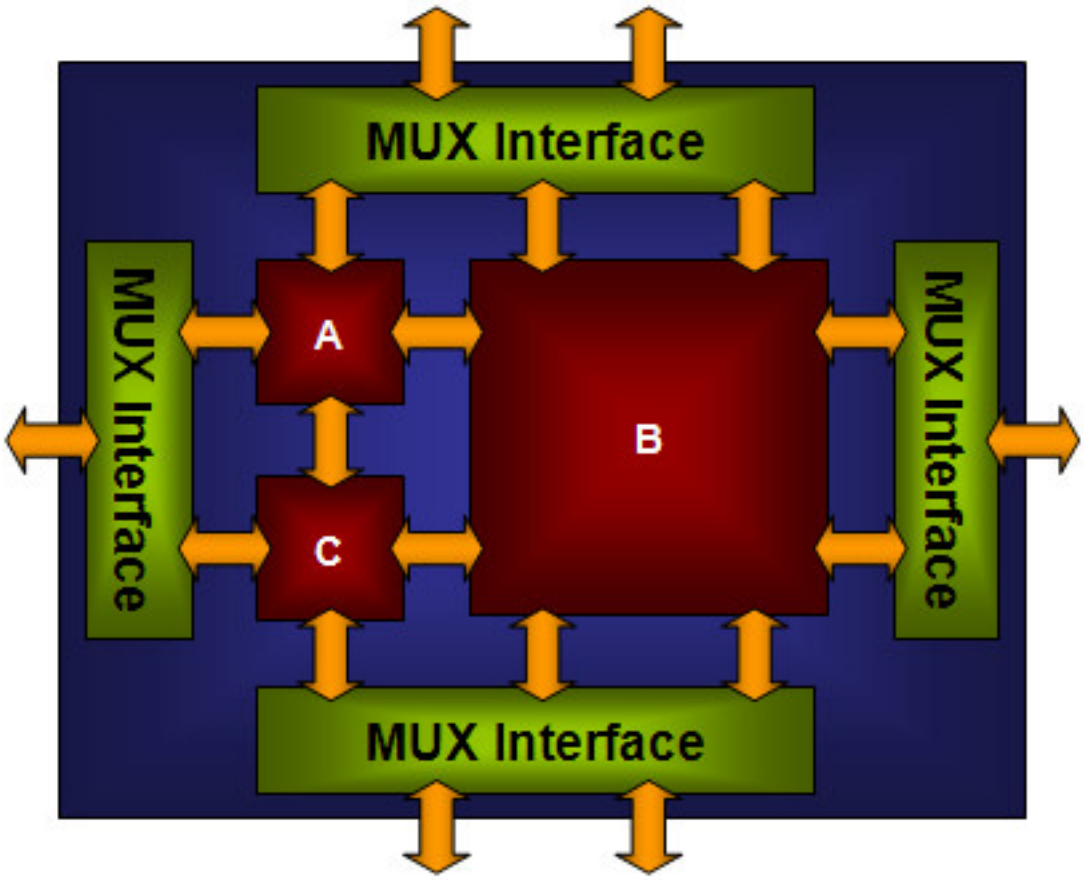
SystemVerilog interfaces offer an object-oriented paradigm for abstraction in communication models by focusing the description in one location. This ability to localize the description of an interface, use it as an abstract port type, and let the synthesis process appropriately spread the hardware through the design, provides a big advantage to the design process.

Many design teams have written a specification for a bus, only to discover in integration testing that the specification was not quite clear enough, and that there were two or more interpretations of it, requiring pieces of the design to be reworked.

An interface is defined independently from modules, between the keywords "interface" and "endinterface". Modules can use an interface exactly like if it were a single port.

In its simplest form, an interface can be considered a bundle of wires. However, interfaces go far beyond just representing bundles of interconnecting signals. An interface can also contain *data type declarations*, *tasks*, *functions*, *continuous assign statements*, and ,i>procedural blocks to specify communication protocols based on bus signals. An interface can also include functionality that is common to each module that uses the interface and can include built-in protocol checking. Thus interfaces can be used to mux signals at the top level of the FPGA design connecting to the I/O blocks.

When interfaces are used, one engineer can own the interface and provide an API that other engineers can use to connect to the bus, hiding the details of the data transfer onto and off of the bus. This provides advantages in terms of scalability, and because the description is in a single location. If another signal needs to be added to the interface, this can be done without requiring every module that passes the bus through it to be modified to add the signal.



Another advantage of interfaces is that they are easily exchangeable with other interfaces supporting the same API. If a design was originally developed with a serial bus, but it is subsequently discovered that a parallel bus is required, for example, the interface can be exchanged

leaving the rest of the design unmodified – a very fast method to retarget a design. Or, in the case of FPGA prototyping, the bus definition can be changed between the ASIC design and the FPGA design without affecting any of the logic blocks.

SystemVerilog allows multiple views of the interface to be defined using *modports*. For example, each module connected to the interface can specify and share direction of the signal local to the interface port. Significant code size reduction is possible when multiple modules refer to the same interface; rather than listing all of the ports on each module, the single port reflecting the interface is sufficient.

The following example shows the basic syntax for defining, instantiating, and connecting an interface:

```
interface addr_bus;
        parameter p1 = 7;
        logic[p1:0] rd_addr;
        logic[p1:0] wr_addr;
        modport addr_port ( input rd_addr, input wr_addr );
endinterface

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

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

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

interface ctrl_bus
        logic reset, clk, wren, ren;
        modport mem_ctrl ( input reset, clk, wren, ren);

endinterface

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

interface io_bus
        logic[7:0] io_data_in;
        logic[7:0] io_data_out;
        logic select;
        logic[3:0] data_in;
        logic[3:0] data_out;
always @ (select)
begin
casex (sel) //pragma parallel_case
 1'b0: io_data_out_ = data_out[3:0]; data_in[3:0] = io_data_in;
 1'b1: io_data_out_ = data_out[7:4]; data_in[7:4] = io_data_in;
endcase
end
        modport fpga_bus ( input io_data_in, data_out, select,
                           output io_data_out, data_in);
endinterface
```

```
module MEMORY ( adder_bus.addr_port addr, //access interfaces
                ctrl_bus.mem_ctrl 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):];
BYTE latch;
Assign data.data_out = latch;
...
...

module top_design (io_bus.fpga_bus io, input clk, input reset, input wren,
                input ren, raddr, waddr, io_data_in, io_data_out, select);

parameter p1 = 7;
parameter p2 = 7;

input [p1:0] raddr, waddr;
input [3:0] io_data_in;
output [3:0] io_data_out;
logic[p2:0] wdata;
logic[p2:0] rdata;

addr_bus #(.p1(p1))          addr();   \\Instantiate interfaces
data_bus #(.p2(p2))          data();
ctrl_bus                     ctrl();
fpga_bus io();
MEMORY #(.p1(.p1)), .p2(p1))          mem(.*);

assign ctrl.clk         = clk;
assign ctrl.reset       = reset;
assign ctrl.wren        = wren;
assign ctrl.ren         = ren;
assign addr.rd_addr     = raddr;
assign addr.wr_addr     = waddr;
endmodule
```

Conclusion

With increasing competitive pressures and shorter product life cycles, designers have less time to develop high performance and complex designs. At the same time, the development cost of an ASIC is increasing rapidly, making it less feasible to use ASIC devices for many cost-sensitive applications without extensive testing and simulation. To overcome these problems, FPGA prototyping is being adopted to provide a timely and cost-effective design methodology that enables systems to be verified before committing to a much more costly ASIC design.

However, FPGA prototyping is not without its difficulties. One major obstacle has been connecting all the logic blocks both within an FPGA and across multiple FPGA devices. With the adoption of SystemVerilog many of the obstacles in connectivity can be addressed in a more simplistic and efficient manner than with other HDL languages.

Roger Do *is a member of the FPGA Solutions Marketing Team in the Design Creation and Synthesis Group at* Mentor Graphics *In this role, Roger is responsible for outbound marketing and business development for FPGA synthesis products.*

Roger joined Mentor Graphics in 1999 and brings over 14 years of experience in the semiconductor industry where he served in a variety of applications, marketing, and field sales roles. Prior to joining Mentor Graphics, Roger held various positions at Lattice Semiconductor, Lucent Technologies, and Texas Instruments. He holds a bachelor's degree in Electrical Engineering from Texas A&M University.