Methods and Tools for Bring-Up and Debug of an FPGA-Based ASIC Prototype

A mix of good methodology and automation helps to locate and resolve prototype flaws and design bugs

Introduction

Software simulation of RTL is no longer capable of providing all of the verification required for today’s complex ASIC designs. (For the purposes of this paper, the term ASIC will be understood to encompass ASSP and SoC devices.) Modern ASICs are a complex mixture of hardware and software, and it is necessary to verify the design within the context of the complete system, running the full range of software at speeds that approach real-time. Thus, successfully validating an ASIC design on an FPGA-based prototype before committing to silicon is now a key project milestone for most design teams.

As part of this, the “Prototyper” specialist has emerged on the verification team. The Prototyper provides a unique knowledge mix of good verification practices, FPGA design savvy, and design-for-prototyping methodologies that help bring the benefits of the FPGA-based prototype to the design project. The Prototyper faces a daunting challenge with regard to creating a system that has satisfactory performance, is portable for field test, and is sufficiently inexpensive that multiple copies can be deployed to the hardware and software engineering teams concurrently.

During the “bring-up phase” of prototype design, multi-board systems are often required to provide physical interfaces and enough capacity to host the entire ASIC RTL. Creating a reliable hardware platform will inevitably require troubleshooting the circuit board connectors, cabling and partitioning logic. Following the bring-up phase, hardware debug can ensue. The ability to interface to real world stimulus and to achieve clock speeds fast enough to execute system software can dramatically increase test coverage. If something goes wrong, however, the Prototyper must employ some clever methods and tools to reveal logic defects buried within one or more FPGAs.

This paper examines some of the best practices for both successful bring-up and logic debug of FPGA-based ASIC prototypes.
An Example ASIC Design

In order to provide a point of reference for the following discussion, consider the block diagram for a generic ASIC design as illustrated in Figure 1.

![Block diagram of generic chip.](image)

The design shown in Figure 1 is obviously a simplification. In reality, the CPU may be a multi-core device and/or there may be multiple CPU and/or DSP cores. There may also be multiple on-chip memory subsystems and busses, and a wide variety of memory, data, and communications interfaces.

A modern ASIC design is rarely created in its entirety from the ground up. Instead, it is composed of a mixture of newly-developed custom functions, such as hardware accelerators, along with existing Intellectual Property (IP) blocks. In turn, the existing IP blocks may be internal in origin (from previous projects) or acquired from a third-party supplier. In some cases the IP blocks will be well-known and trusted; in other cases they will be “unknown quantities” to the design and verification teams. And even in the case of well-known IP blocks in which the teams have a high degree of confidence, the new design may expose these functions to previously untested “corner conditions” and/or operating modes that may not behave as expected in the context of the current system.

In addition to the underlying hardware portion of the design, today’s ASICs may employ hundreds of thousands of lines of firmware and tens of millions of lines of software application code. In many designs, the software design and verification takes more time and resources than the hardware design, making software the bottleneck in the chip development process. Furthermore, re-spinning a chip is so expensive and time consuming — and missing the market window is so catastrophic — that the software has to be completed and tested to a high level of confidence prior to final silicon becoming available. This means that software developers need some form of prototype that can boot the operating system and run their firmware and application software at close to real-time speeds.

Alternative Verification Technologies

There are a variety of verification technologies available. Early in the process, software simulation of the RTL is almost invariably used to debug and verify portions of the design at the block level. The advantage of software simulation is total visibility into the design; the disadvantage is low speed. Even though the performance and capacity of software simulators have increased dramatically in recent years, the extreme size and complexity of today’s ASICs means that it is possible to simulate the entire design for only a few tens of clock cycles. This is simply not sufficient to exercise the design under real-world operating conditions and data-processing loads.

Hardware emulators may be used to accelerate the verification process. The advantages of hardware emulators are that they are orders of magnitude faster than their software simulator cousins while still providing a reasonably high level of visibility into the design. The main disadvantages are that hardware
emulators are physically large (heavy and bulky), they can be complex to setup and use, and they are extremely expensive, which makes them unsuitable for deployment to multiple, geographically-dispersed hardware and software development teams.

In order to address the issues discussed above, FPGA-based prototyping is becoming the technology of choice for the design and verification of even the most complex ASICs. The relatively low cost of an FPGA-based prototype means that multiple systems can be provided to the hardware and software development teams. Also, the close-to-real-time speeds of FPGA-based prototypes means that they can be interfaced to the rest of the real-world system without requiring the creation of specialized subsystems that are the bane of hardware emulation environments.

When the FPGA-based prototype is being used and a problem is detected, it is critical to be confident that this is a real design bug and not an error associated with a faulty prototyping board or an issue with the prototyping methodology. Generally speaking there will be fewer problems with a commercial board, which will already have undergone a full production test and signoff by the vendor. Having said this, problems may still arise, and making the assumption that the prototype is OK when it isn’t can cause huge delays. It is important to have a high degree of confidence that the verification environment is functioning correctly in order for the verification team to concentrate on debugging the design itself.

All of this means that there are two main steps in the verification process. The first is to bring up and verify the prototyping system and associated environment. It is only then that the verification team can detect, debug, and correct design bugs and fully validate the design. Once all this is done, the prototype will be in a good state to be deployed for use by the software development teams.

**Troubleshooting Faults During System Bring-up**

For the purpose of this portion of the discussion, assume the use of a HAPS® FPGA-based prototyping solution from Synopsys. In the case of another third-party system or a custom-developed system, users must ensure that equivalent capabilities, tools, and utilities are provided and/or create such items themselves.

The first step is to make sure that the main FPGA-based prototyping board is functional at the most basic level. The types of tests that may be used at this most fundamental level vary a great deal in their sophistication and scope. For example, in the case of a HAPS system, each board is supplied with a special termination block that can be used to check the external connectors for open faults, short circuits, and signal continuity. Next, any available built-in self-tests, including scan techniques, should be run, as should any external host-based routines.

After the main board is known to be functional, the next task is to check for problems in any daughter cards that are going to be used for this project, especially the connections between the various boards through connectors and cables. It is also necessary to test for issues related to external sources such as clock and reset signals.

Once the correct operation of the base level functionality of the main board and its associated daughter cards has been established, the next step is to run some test designs. By running a familiar small design that has well known results, the correct setup of the system can be quickly established. Quite apart from anything else, running test designs makes sure that the users become comfortable with the FPGA-based prototyping flow, including partitioning a design across multiple FPGAs, running synthesis and place-and-route, generating bit files, programming the FPGAs with the bit files, and practicing basic debugging techniques.

In the case of HAPS systems, these are supplied with appropriate test designs, but users are encouraged to develop their own library of tests that best address their unique design scenarios and that can be re-used for different prototypes in the future. Some typical test designs and procedures that may be run to ensure correct configuration and operation of the system are as follows:
Sign-of-life tests are very simple. Writing values into registers and reading these values back from an external port can confirm that the FPGA is configuring correctly.

Counter tests can be used to test that the clocks and resets are connected and working correctly. Write a simple counter test-design with the clock and reset as inputs. Connect the outputs to LEDs and/or to FPGA pins that connect to test points on the board and/or read the counter register using an external host program.

Daughter card reference designs provided by the vendor, or equivalent test designs created by the user, can be used to check the functionality of and add-on daughter cards. Such test designs should verify the interfaces between the main system and the daughter cards, the proper functionality of the daughter cards, and links to external ports, such as network analyzers.

High-speed I/O tests should be used if the prototyping project makes use of advanced inter-FPGA connectivity features like pin multiplexing or high-speed time division multiplexing (HSTDM). Owing to the environmental dependency of low voltage differential signaling LVDS and other high-speed serial media, the test designs should employ the same physical connections in order to properly replicate the final paths to be used in the FPGA-based prototype.

Multi-FPGA test designs should be used if the real design is to be partitioned across multiple FPGAs. In such a case, it is advisable to verify the prototyping system and environment with a simple test design that can be easily partitioned across all of the FPGAs. An example test design of this type could be a FIR filter with each tap partitioned into a different FPGA.

Once the base-level functionality of the entire prototyping system has been confirmed, the next step is to introduce the ASIC design to the prototyping system. It’s important to note that the user is not yet at the stage of debugging and verifying the ASIC design itself — all that is being done at this stage is to make sure that it is in a fit state for the debugging and verification process to begin.

Prior to bringing up the FPGA-based prototyping system, the RTL should have been verified to an acceptable degree using software simulation and be signed off by the RTL verification team as being ready for prototyping. The team will have created some form of “Hello World” test that has been run on the RTL, and it is very useful if this same test harness can be reused on the FPGA-ready version of the design.

Despite all best efforts, initial FPGA implementations can differ from the intended implementation. The most common issues with initial FPGA implementations:

- **Timing violations:** The timing analysis built into the synthesis and place-and-route tools operates on each FPGA in isolation. Timing violations across multiple FPGAs — on paths routed through FPGAs or between clock domains in different FPGAs, for example — would not be highlighted during normal synthesis and place-and-route.

- **Unintended logic removal:** It may not be obvious at first, but modules and Input/Output (I/O) sometimes seem to “disappear” from the resulting FPGA implementation due to minimization during synthesis. The common cause is improper connectivity or improper modules and core instantiations that result in undriven logic, which is subsequently minimized. Early detection of accidental logic removal can save valuable FPGA implementation time and bench debug time.

- **Improper inter-FPGA connectivity:** Despite all efforts, due to improper pin location constraints, the place-and-route process will assign I/O to unintended pins, thereby resulting in unintended and incorrect inter-FPGA connectivity. This often occurs the first time the design is implemented onto a custom prototype PCB and also when the design is drastically modified. EDA tools, like Synopsys Certify® multi-FPGA partitioning and implementation software (designed to target the on-board FPGA, memory, and connector resources of a prototype PCB) provide a “board file” feature written in Verilog syntax that includes the FPGA I/O interconnect for HAPS-family motherboards or custom prototypes. Pin constraints are automatically generated based on the I/O plan and board file connectivity to avoid misroutes or unexpected floating assignments.
All of these issues, along with tips on how to handle them, are presented in detail in the *FPGA-Based Prototyping Methodology Manual* (FPMM) by Doug Amos and René Richter from Synopsys and Austin Lesea from Xilinx (ISBN: 978-1-61730-004-2).

### Debugging Design Flaws

Rather than transferring the ASIC design from the RTL/software simulation domain into the FPGA-based prototyping system in its entirety, it may be preferable to incrementally transition the design block by block. In this scenario, the designer first simulates the entire design for a few clock cycles in the software simulator, then moves a portion of the design into the FPGA-based prototype and — using the original testbench — co-simulates the bulk of the design in the software domain with the portion of the design in the FPGA-based prototype. If a discrepancy occurs, the errant block is easy to identify, and it can quickly be debugged using the techniques discussed below.

Of course, co-simulation of this type requires a cycle-accurate bi-directional link between a high-performance software simulator and an FPGA-based prototyping system. One such co-simulation link is HDL Bridge from Synopsys. HDL Bridge provides a bi-directional interface between Synopsys’ high-capacity, high-performance RTL simulator, VCS, and Synopsys’ high-capacity, high-performance HAPS family of FPGA-based prototyping solutions.

Once the entire ASIC design has been transitioned over into the FPGA-based prototyping system, in-depth verification and debug can begin. It is important to note that every design is different, and there is no “one size fits all” approach to debugging, but it is possible to offer some guidance on ways in which to gain visibility into the design.

Assuming that the design was well verified before its release to the FPGA-based verification team, users should be looking for new faults to manifest themselves because the design is now being exposed to new — more in-depth — stimulus that was not previously provided by the testbench during software simulation. The cause of such faults can be categorized as *logic bugs* (undiscovered RTL errors in the ASIC design), *interface bugs* (unforeseen issues with the external interfaces), and *software bugs* (first seen during software-hardware integration).

In order to isolate, identify, and correct these bugs visibility into the design is needed, this can be obtained by extracting internal signals in real-time and also by collecting samples for later extraction and analysis:

- **Real-time signal probing:** In this simplest method of probing the design’s internal nodes, directly modify the design in order to bring internal nodes to FPGA pins for real-time probing using bench instruments such as logic analyzers and oscilloscopes. This is a common debugging practice and offers the benefit that, in addition to viewing signal states, it is also easier to link signal behavior with other real-time events in the system.

- **Embedded trace extraction:** This approach generally requires special tools support to add instrumentation logic and on-chip RAM in order to sample internal nodes and store them temporarily for later extraction and analysis. This method consumes only a small amount of logic resources, very little routing resources, and only those pins that are used to probe the signals. Advances in trace extraction techniques allow for off-chip sample data storage either to external logic analyzers or dedicated memory ICs of the prototyping system. These deep resources expand signal visibility by supporting more probe points as well as capturing more of the system’s operation over time.

Two other techniques of expanding debug capability that are of special interest with regard to debugging software are as follows:

- **Bus-based instrumentation:** Some teams instantiate instrumentation elements into the design in order to read back values from certain key areas of the design, to read or load memory contents, or even to drive test stimulus and read back results.
Custom debuggers: In this case, portions of the design are “observed” by extra elements that are added into the design expressly for that purpose. For example, a soft CPU core may be connected onto an internal bus in the design in order to detect certain combinations of data or error conditions. This technique is almost always user-generated and very application-specific.

Debugging can easily become an iterative process. In order to achieve faster turn-around on small changes it is possible to use incremental techniques as follows:

- **Incremental synthesis:** When changes are made to the design, the synthesis tool compares parts of the new design to the same parts from the previous run. If they are identical the tools will simply reuse the previous results rather than recreate them. In Synopsys FPGA implementation tools, incremental synthesis is supported by the compile point synthesis flow, which does not require any changes to the RTL; instead, it is controlled by small changes to the project and constraint files only, and can even be driven from a GUI.

- **Incremental place-and-route:** Different environments support incremental place-and-route in different ways. In the case of Xilinx FPGAs, incremental place-and-route can be achieved using the design preservation flow, in which the design is broken into blocks referred to as partitions. Partitions create boundaries around hierarchical module instances so that they are isolated from other parts of the design. A partition can either be implemented (mapped, placed, and routed) or its previous preserved implementation can be retained. Another fast turnaround method which bypasses synthesis and placement phases of the FPGA implementation is to take advantage of incremental reroute of embedded debug logic. This is typically driven by an FPGA debug environment, like Synopsys Identify® RTL debugger, where the user indicates the probe and trigger signals to be substituted then the debug system generates a script of route ECO commands that the FPGA place-and-route program applies directly to the native FPGA design database.

Many of these techniques are introduced in detail in the FPGA-Based Prototyping Methodology Manual (FPMM) by Doug Amos, Austin Lesea, and René Richter (ISBN: 978-1-61730-004-2).

Synopsys’ Identify software provides simulator-like visibility into FPGA hardware operation. The Identify RTL debugger allows the user to instrument RTL HDL and then, still at the RT-Level, debug the implemented FPGA on live, running hardware. The Identify debug software verifies a design in hardware, similar to simulation — only much faster and with in-system stimuli. The Identify debugger allows the user to designate sample triggers, navigate the design graphically, and mark signals in the RTL that are to serve as probes. After synthesis, the results are viewed and annotated onto the RTL source code, the HDL Analyst® RTL View, or third party waveform viewer. This ensures RTL-to-implementation equivalence and correct operation of the FPGA design.

Recognizing the need for more automation to ease both the initial bring-up and subsequent RTL debugging tasks of FPGA-based prototypes, Synopsys is adding more capabilities to the Identify debugger specifically for FPGA-based prototypes like the HAPS systems. The Identify debugger works seamlessly with incremental synthesis, incremental place-and-route, and the HAPS family of FPGA-based prototyping solutions. Some new bring-up troubleshooting and debug capabilities that have been recently added to the Synopsys FPGA-based prototyping solutions are as follows:

- **Debugger mux groups:** This allows up to eight groups of signals to be selectively channeled through a single Identify in-circuit emulator, thereby providing 8X the signal visibility without the need to run synthesis or place-and-route.

- **HAPS Deep Trace Debug:** This supports the off-FPGA storage of sample data, thereby providing 100X signal visibility, dramatically increasing sample capacity and history (thousands of signals for millions of cycles), and saving on-chip FPGA resources. Identify debugger sample memory can now be targeted to either internal FPGA memory or a high-capacity HAPS SRAM daughter card. To ease initial assembly of the board with a HAPS system, a self-check design ensures correct location and configuration of the system prior to a debug session.
Incremental instrumentation: This provides very fast turnaround time to change Identify debugger probe and trigger points by using the reroute features of the Xilinx FPGA place-and-route system.

HAPS Real Time Debug (RTD): RTD makes internal signals visible to the outside world by routing FPGA signals to the Mictor Logic Analyzer Interface for HAPS systems. From there, these signals can be tapped by external logic analyzer equipment. Now Identify debugger's flexible triggering expressions can be applied to isolate design flaws and use the high-capacity storage of the logic analyzer.

Summary
Software simulation of RTL is no longer capable of providing all of the verification required for today's complex ASIC designs. Successfully validating an ASIC design on an FPGA-based prototype before committing to silicon is now a key project milestone for most design teams.

During the “bring-up phase” of prototype design, multi-board systems are often required to provide physical interfaces and enough capacity to host the entire ASIC RTL. Creating a reliable hardware platform will inevitably require troubleshooting the circuit board connectors, cabling, and partitioning logic. Following the bring-up phase, hardware debug can ensue. The ability to interface to real world stimulus and to achieve clock speeds fast enough to execute system software can dramatically increase test coverage. If something goes wrong, however, the users must employ some clever methods and tools to reveal logic defects buried within one or more FPGAs.

In addition to state-of-the-art ASIC and FPGA synthesis technology, Synopsys' Certify software is the leading tool to partition an ASIC design for use in an FPGA-based prototype. Also, Synopsys' Identify RTL debugging software provides simulator-like visibility into FPGA hardware operation. The combination of Synopsys' synthesis, Certify, Identify, and the HAPS family of FPGA-based prototyping solutions supports the verification and debug of even the most sophisticated ASIC designs.

References


