System-Level Debugging and Monitoring of FPGA Designs

WP-01170-1.0 White Paper

This white paper describes the latest state-of-the-art methods for debugging and monitoring large FPGA designs both during the simulation phase of development and after device configuration, and details the current practices that Altera has identified across a representative number of customer designs. In addition, the paper presents a platform that enables FPGA designers to easily add runtime visibility into their FPGA systems while ensuring the scalability needed in today's increasingly large designs and compilation times.

Introduction

For the purposes of this paper, today's FPGA designs can be divided into two different categories: those with embedded soft-core processors and those without. This division is useful when examining what debugging and monitoring infrastructure designers use in their FPGA systems. The techniques discussed here are used both to collect data needed for root-cause diagnosis of defects and to monitor the performance of a system under a real-world load.

Debugging Systems Without a Processor

For designs without soft-core processors, designers usually use embedded logic analyzers to gather runtime data from the design. Tools such as Altera's SignalTap™ II embedded logic analyzer (available in Quartus® II design software) are useful for looking at a predetermined set of signals and examining their behavior in time. When the design is suspected of being defective, the designer chooses a new set of signals to investigate, recompiles the design, and then reconfigures the device. This process is then iterated until the defect is found and fixed. Often a designer will use SignalTap II to gather input vectors to use as stimulus in a simulation, in order to avoid the costly recompilation step.

In modular systems, it is usually only necessary to tap the module's interface to identify the root cause of a defect, so in the above scenario, the designer typically ends up looking at module interfaces. However, as the number of modules increases, it becomes exponentially cumbersome to go through these iterations. In response, some designers decide to constantly tap all of their module interfaces. However, if they want access to all of the interfaces after deployment, this method can leave a vast amount of debug logic instantiated during deployment.

Debugging Systems with a Processor

When a design contains a soft-core processor, it is not unusual for the designer to reserve a percentage of its capabilities to run a debug and control loop. This loop usually listens for commands on one of the systems communication channels, such as a UART or a TCP/IP connection. Once a defect is suspected, the designer then connects to the debug and control loop through this channel and using a private command language, is able to poke around the system and observe as much as the command language allows.

In many cases, soft-core processors are added to a system explicitly to provide this type of service. Aside from the required sharing of resources this approach requires, using soft-core processors has two major drawbacks. First, it takes development time to create the debug and control loop and its command language. Second, the nature of the communication channel usually prevents this method from being used during the simulation phase. In addition, systems running the debug and control loop using the mission processor might be unable to reproduce the failure event once the debug loop starts using resources and interacting with the execution of the other processing tasks.

Requirements for Debugging Today's Systems

Based on the above description and customer feedback, a modern system-level debugging tool must be able to do the following:

- Minimize the number of hardware compilation steps needed
- Be consistent and reusable during the simulation, lab test, and deployment phases
- Use the existing system's address space as an efficient means of accessing the design state
- Use the minimum possible amount of resources
- Provide a flexible scripting interface to access the available runtime information
- Cause minimal disruption of the system transaction sequence
- Have the flexibility to use already available communication mechanisms

More importantly, an effective debugging and monitoring tool should be able to help the designer answer more subtle questions such as:

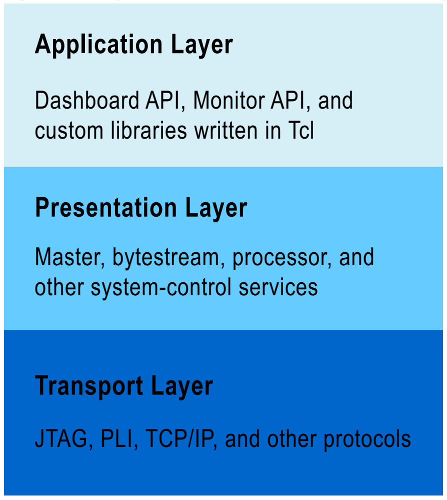
- Why is the system performance not matching the expected performance?
- Where are the bottlenecks?
- How much processing capacity is available given the current traffic?

The debug infrastructure should provide the means to answer these questions and others that may arise perhaps even after deployment, with minimal or no changes to the design.

Altera's Approach

Altera addresses the designer's debugging and monitoring needs using a system-level approach with System Console. It is comprised of communication intellectual property (IP) that is instantiated inside a Qsys design and a software stack that runs on a host computer. The System Console architecture is divided in three layers, as shown in Figure 1.

Figure 1. System Console Architectural Layers


Application Layer
Dashboard API, Monitor API, and custom libraries written in Tcl
Presentation Layer
Master, bytestream, processor, and other system-control services
Transport Layer
JTAG, PLI, TCP/IP, and other protocols

At the bottom of the architecture, closest to the hardware, is the data transport layer. Different hardware IP blocks can provide the service required by this layer, allowing for flexibility in the type of transport technology to use. Currently those choices are JTAG, PLI, and TCP/IP, with more to come.

The second layer, the presentation layer, is comprised of services that provide various application programming interfaces (APIs) to interact with the design. Some examples of APIs in this layer include the commands to issue memory-mapped transactions, interact with embedded processors, and send and receive bytes from a bytestream device. Since this layer depends on the definition of the layer below, all of the services on this layer are agnostic about the underlying transport protocol, allowing for flexibility and reuse as the design moves from simulation, to the lab, and later to deployment.

The third layer is the application layer, with its own set of APIs. These APIs help the designer bridge the world of signals and transactions to the realm of a full-fledged debugging and monitoring application. This layer includes the Dashboard API and the Monitor API. The Dashboard API allows the creation of GUI applications that range from the very simple, such as displaying the current value of an address location, to the truly complex, such as Altera's External Memory Interface (EMIF) Toolkit. The Monitor API provides an efficient mean to create periodic access to a range of memory addresses.

For more information about the EMIF, refer to the Documentation: External Memory Interface page of the Altera® website.

Achieving the Ideal

As described previously, an ideal system-level debugging tool should be able to perform and fulfill various requirements. Altera's System Console meets these requirements.

Minimize the Number of Hardware Compilation Steps Needed

For systems that contain an embedded soft-core processor, System Console can control processors already present in the system, provided that the processor has its debug core enabled. Once it has taken control of the processor, System Console can access any of the memory-mapped slaves that the processor can access. The system designer can also add a communication bridge, such as the JTAG-to-Avalon® interface master, to their design. This small piece of IP also provides System Console with access to the memory-mapped slaves that the master can access. Taking advantage of the existing memory-mapped interconnect, once the system includes a communication bridge, the designer does not need to recompile the hardware to access a new signal. As long as the state is visible through the slave's address map, it is reachable with just one command.

Be Consistent and Reusable During the Simulation, Lab Test, and Deployment Phases

One of the available communication bridges used with System Console is the JTAG-to-Avalon master, which can be used during simulation as well as during deployment. During simulation, the JTAG-to-Avalon master becomes a PLI-to-Avalon master and the very same commands used to interact with its JTAG counterpart can be issued to interact with the simulation. This mode allows troublesome transactions to be injected into the system and then the simulator's high-visibility range can be used to identify the root cause of the problem. In addition, most of the scripts that the designer developed to interact with the design during the simulation phase can be reused during later stages because the commands remain the same in the simulation and hardware phases, increasing the consistency of the development flow.

Use the Existing System's Address Space as an Efficient Means of Accessing the Design State and Use the Minimum Possible Amount of Resources

By taking advantage of the Qsys interconnect, System Console is able to control many different masters and thereby access the different slaves that make up a system and the systems's address space. In addition, it can easily send and receive Avalon transactions using the standard interfaces available in the modules that constitute the system. These transactions lead to a natural manipulation of the design state, reusing the existing actors in the system.

Provide a Flexible Scripting Interface to Access the Available Runtime Information

All of the capabilities of System Console are presented as Tcl procedures, yielding an advanced programmable environment that allows for the development of solutions ranging from simple scripts to sophisticated GUI applications. System Console is architected so that the services available to the designer are independent and agnostic of the transport layer used to reach the device. This yields a Tcl API that remains constantly independent from the communication method, be it JTAG, PLI, TCP/IP, or any other.

Using this programmable environment, debug transcends from analysis of individual signals to a transactional interaction with the system. By building on top of transactions and Tcl, the designer can raise the level of abstraction and start thinking of debugging packet headers as they traverse a router or an image block as it is processed in a video pipeline.

Cause Minimal Disruption of System Transaction Sequence

In addition, System Console is fully integrated with the In-System Sources and Probes editor in the Quartus II software, which provides access to arbitrary signals across the whole system. These probes do not use any of the interconnect resources; instead, they create a low-bandwidth parallel network that provides access at a very reasonable resource cost. In fact, the main cost for using In-System Sources and Probes becomes the recompilation required once new signals are added. This cost must be weighed against the benefits of no disruption to the transaction flow.

For more information about using the In-System Sources and Probes editor, refer to the *System Debugging Tools Overview* chapter in volume 3 of the *Quartus II Handbook*.

Have the Flexibility to Use Already Available Communication Mechanisms

As previously mentioned, System Console can take control of existing communication infrastructure, which provides a convenient way to gain visibility on legacy designs that were not designed with System Console in mind, but already contain communication IP. Typical examples of this flexibility is a Nios® II processor with its debug core enabled or In-System Sources and Probes.

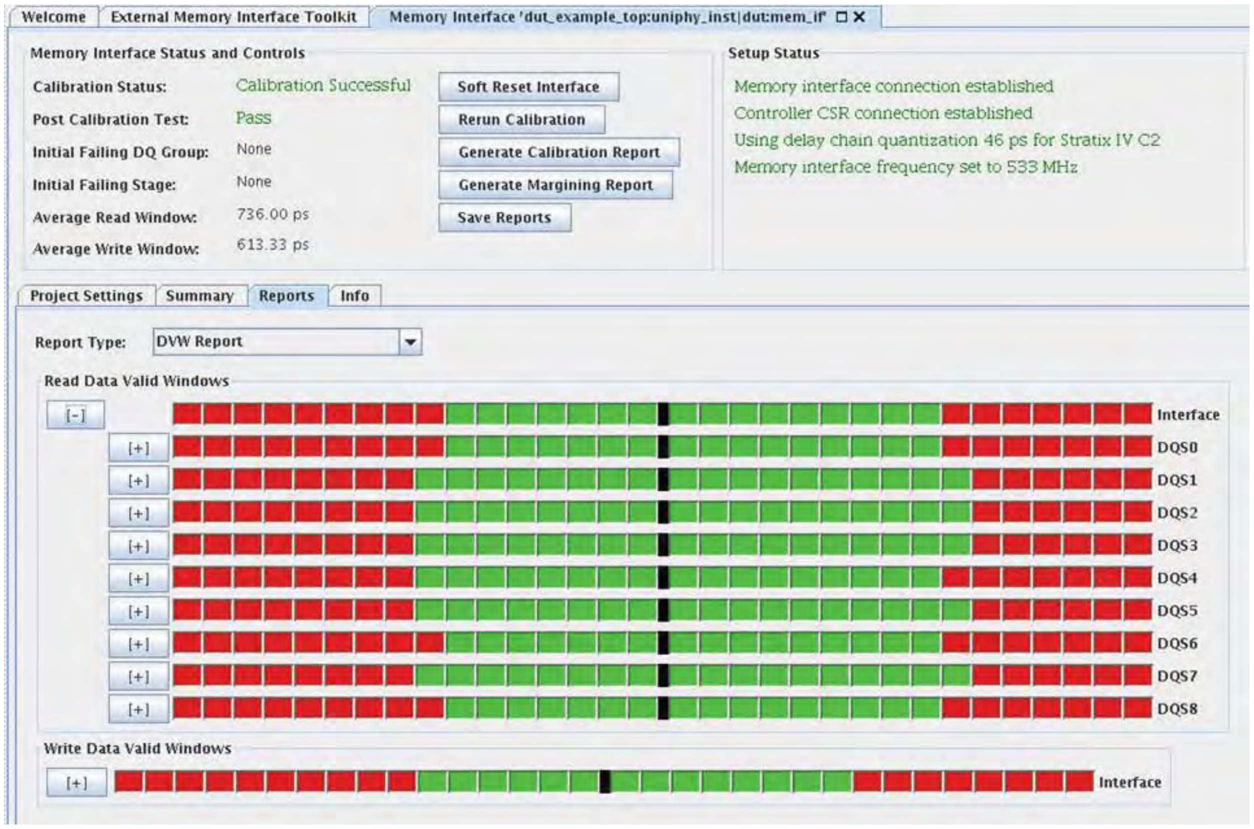
Sharing the Platform

The System Console platform is accessible to any of the Quartus II development software users through the System Console Tcl API. Its capabilities include:

- Controlling available Qsys interconnect masters
- Controlling In-System Sources and Probes
- Verifying the status of the reset and clock networks
- Controlling Virtual JTAG megafunctions
- Creating GUI dashboards to interact with the design

Altera leverages the capabilities of the System Console platform when developing new debugging tools. The External Memory Interface Toolkit (Figure 2) is one example of a tool created using the System Console platform. This tool kit aids and provides detailed information about the calibration process of external memory interfaces.

Figure 2. External Memory Interface Toolkit



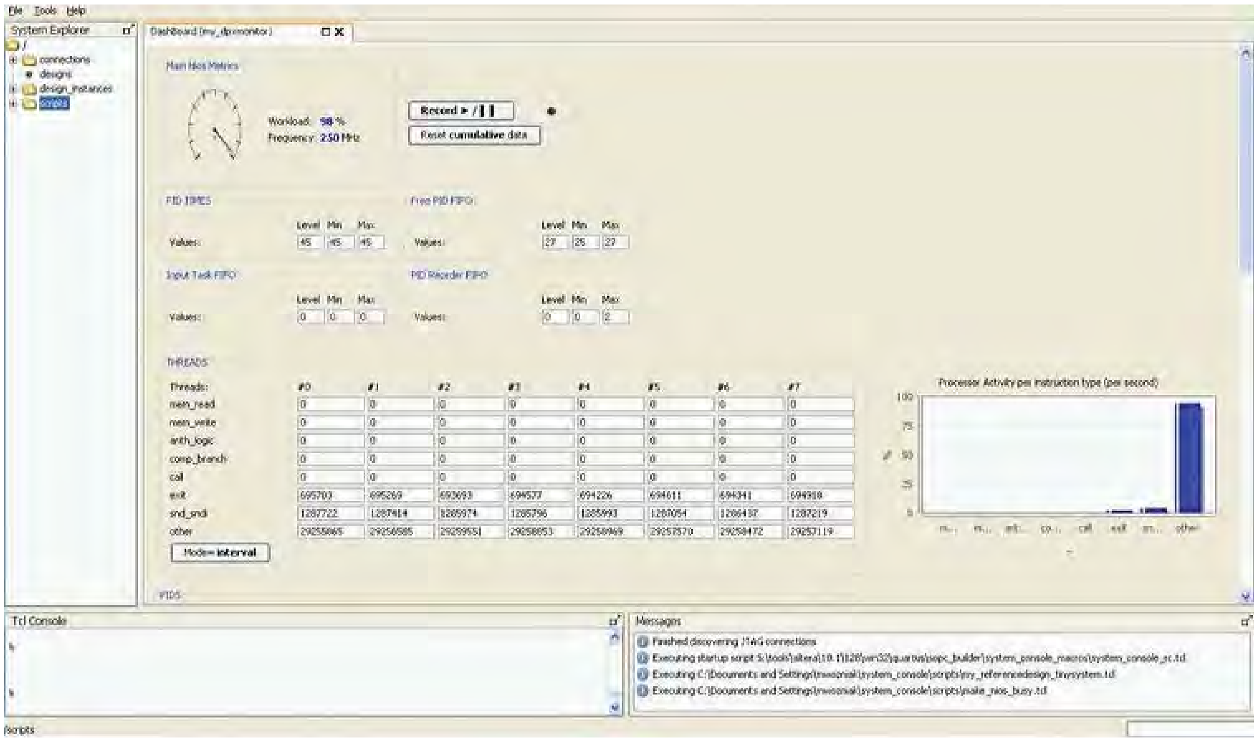
Calibration Status: Calibration Successful
Post Calibration Test: Pass
Initial Failing DQ Group: None
Initial Failing Stage: None
Average Read Window: 736.00 ps
Average Write Window: 613.33 ps

Memory interface connection established
Controller CSR connection established
Using delay chain quantization 46 ps for Stratix IV C2
Memory interface frequency set to 533 MHz

Report Type: DVW Report

Some designers use the System Console Dashboard API to create detailed status dashboards for their designs. Figure 3 shows a status board for a packet processing application, displaying different statistics of interest in real time.

Figure 3. Dashboard for a Packet Processing Application



Workload: 98 %
Frequency: 250 MHz

PID TIMES

	Level	Min	Max
Values:	45	45	45

Free PID FIFO

	Level	Min	Max
Values:	27	26	27

Input Task FIFO

	Level	Min	Max
Values:	0	0	0

PID Reorder FIFO

	Level	Min	Max
Values:	0	0	2

THREADS

Threads:	#0	#1	#2	#3	#4	#5	#6	#7
mem_read	0	0	0	0	0	0	0	0
mem_write	0	0	0	0	0	0	0	0
arith_logic	0	0	0	0	0	0	0	0
comp_branch	0	0	0	0	0	0	0	0
call	0	0	0	0	0	0	0	0
exit	695703	695269	693693	694577	694226	694611	694341	694918
snd_sndi	1287722	1287414	1285974	1285796	1285993	1287054	1286437	1287219
other	29255865	29256585	29259551	29258853	29258969	29257570	29258472	29257119

Mode= interval

PIDS

Messages

```
Finished discovering JTAG connections
Executing startup script S:\tools\altera\10.1\128\win32\quartus\sopc_builder\system_console_macros\system_console_rc.tcl
Executing C:\Documents and Settings\rwozniak\system_console\scripts\my_referencedesign_tinysystem.tcl
Executing C:\Documents and Settings\rwozniak\system_console\scripts\make_nios_busy.tcl
```

Conclusion

This white paper explored the current state-of-the-art methods for debugging and monitoring large FPGA systems and presented a set of requirements needed to be met by the infrastructure tasked to provide system visibility. To meet these requirements, Altera created System Console, a platform that provides users with the flexibility, reusability, and efficiency required to solve today's wide range of system debugging needs.

Further Information

- System Console: Faster Board Bring-Up and On-Chip Debug: www.altera.com/products/software/quartus-ii/subscription-edition/qsys/systems/qts-systems-console.html
- *Analyzing and Debugging Designs with the System Console* chapter of volume 3 of the *Quartus II Handbook*: www.altera.com/literature/hb/qts/qts_qii53028.pdf
- Video: "Faster Board Bring-Up with System Console": www.altera.com/education/demonstrations/qsys/board-bringup/System_Console_Board_bringup_Final.html
- Video: "Building a Custom Verification GUI with System Console": www.altera.com/education/demonstrations/qsys/system-console/System_Console3_final.html

- Documentation: External Memory Interface:
  www.altera.com/literature/lit-external-memory-interface.jsp
- *System Debugging Tools Overview* chapter in volume 3 of the *Quartus II Handbook:*
  www.altera.com/literature/hb/qts/qts_qii53027.pdf

Acknowledgements

- Silvio Brugada, Sr. Software Engineer, System-Level Debug, Altera Corporation

Document Revision History

Table 1 shows the revision history for this document.

Table 1. Document Revision History

Date	Version	Changes
November 2011	1.0	Initial release.