Successfully Debugging Complex FPGA Designs

Abstract

FPGAs are now one of the most common devices used to implement complex electronics systems. With their ever increasing capacity and the inclusion of complex on-chip functional blocks, FPGAs have been very successful at replacing ASICs and ASSPs that previously made up the bulk of complex system designs. These increased capabilities have also made development much more complex however, and not just during the design stage. Once the FPGA has been designed it also needs to be debugged. Careful design and simulation can eliminate many errors, but even the most careful development process will miss subtle definition errors, unexpected sub-system interaction issues or even board level timing or noise problems that require significant detective work to find and fix. Also, it might not be worthwhile to exhaustively simulate an FPGA-based design since the downside of initial bugs is not nearly as catastrophic as with an ASIC. Typically these types of issues require a hardware ‘test bed’ to detect and resolve errors. Several hardware oriented techniques have been implemented by FPGA manufacturers and third parties to simplify hardware-based debugging. A careful understanding of each of these key techniques, including the various advantages and disadvantages, is useful when considering which technique or combination of techniques is suitable for a particular design. This white paper will describe the most common hardware debugging approaches, describe their strengths and weaknesses and illustrate how the new DN_Readbacker tool from the Dini Group improves, but does not replace existing approaches.

Introduction

With flexible external interfaces, hard wired processors, memories, and MACs modern FPGAs have the resources to create entire complex sub-systems in a single device. Additionally, the customization possible with FPGAs means that specifications are likely to change during the design process. This is called feature creep. The ability of FPGAs to be reconfigured enables in-field updates, even after a product has been released to market. The implication: quick design changes and architecture options are a typical occurrence. The ability to rapidly debug design changes and the invariably accompanying design errors is critical. Without an efficient debug capability much of the advantage of FPGA design flexibility will be sacrificed for the additional debugging time required.

Common Error Sources in FPGA Designs

FPGAs make it easy to implement very large complex systems. But invariably errors will find a way into a design. Functional definition errors are probably the most common source of errors in FPGA designs and can be very difficult to find since the designer has misunderstood a particular requirement. This means the error will be overlooked and perhaps not even tested for, if the same engineer is doing the design and the test bench.
An example of a common functional definition error might be if a state machine doesn’t transition to the right state when an infrequent combination of inputs is present, perhaps during a fault detection check.

Errors can also often show up in system interfaces where the interaction or hand-shaking protocol isn’t fully understood or appreciated. Interface latency, for example, might be incorrectly specified, with an unexpected buffer overflow or underflow might result during an uncommon ‘corner case’. In a worst case scenario, the error might show up as corrupted data, making it very difficult to track down the source of the error.

System level timing issues are another very common source of design errors. Asynchronous events, in particular, are a common source of errors when synchronization across different timing domains is not handled correctly. When operating at speed these types of errors can be very problematic and may show up very infrequently, perhaps only when specific data patterns manifest themselves, or when asynchronous clocks align in opportunistic windows. Many common timing violations fall into this category and are very difficult, if not impossible, to simulate.

Often design issues must ripple through the design until an error actually manifests itself. Tracing the starting error back to the root cause is a frustrating and time consuming task. This type of detective work can be simplified with the right debug tools, but it is important to understand the advantages and disadvantages of the tools you plan to use. The next section describes some of the common approaches to on-chip debugging and sets the stage for showing how the DN Readbacker can make your on-chip debugging effort much faster and easier when used in combination with the other approaches.

**Existing On-Chip Debug Solutions**

There are several techniques for on-chip debugging offered by FPGA vendors and third parties. The basic approach involves adding extra debug logic to the design for accessing and controlling signals and storage elements of interest during the debug or testing process. The simplest example of on-chip debug logic routes signals to IOs for observation by a scope, logic analyzer, or other piece of external hardware. You can only select a small number of signals using this approach and if the signals you selected are not the right ones, you need to select new signals and recompile your design. This can be a time consuming process. It is invasive and it is time consuming since steering internal nets to IO pins impacts the routing and therefore the timing of the DUT (design under test).

More complex approaches add significantly more debug logic on-chip, in essence an embedded logic analyzer that can trigger on a specific signal or combination of signals and then capture signals of interest and store them in an on-chip block memory. After the desired data has been captured it can be sent, usually via a JTAG port, to a debug control program for observation. Observation can be done using a waveform display, to show
individual signals, or the captured data can be compared against an expected result to quickly identify unexpected variations. In general, most of the functions a traditional logic analyzer supports are also supported by the on-chip versions.

The embedded logic analyzer approach requires significant FPGA resources and perhaps several large block memories. In many designs, few block memories and logic cells are left over so it may not be possible to fit extra debug logic on the device. Even if it is possible to fit extra logic, it can significantly change on-chip timing since it ‘steals’ resources and routing tracks from the base level design. This may make it difficult to close timing on the now larger design. Even worse, you may end up debugging errors that wouldn’t exist if the debug logic wasn’t added. Even if the design does fit and passes timing, you are still faced with the issue of which signals to select for debugging and will need to recompile to change the signals under observation - a time consuming task that significantly slows debugging.

Side-Bar: Short description of existing solutions.

**Common On-chip Debugging Approaches**

**Xilinx Integrated Logic Analyzer (ILA):**

The new name for ChipScope, this is a logic analyzer IP core that includes many advanced features of modern logic analyzers, including Boolean trigger equations and edge transition triggers using block RAM to store captured signals - synchronously and in real time with the design under test. JTAG is used to control the debug process.

**Identify™ from Synopsys:**

An RTL debugger with live running hardware instrumentation and debugging at the RTL level. Observation results are annotated on the design RTL source code for improved traceability. Supports multiple FPGA families.

**ProtoLink™ from Synopsys:**

ProtoLink™ is a multi-FPGA debug solution that provides simulator-like visibility for FPGA prototype boards that support the custom ProtoLink interface for data capture. This approach requires quite a bit of forward planning since a proprietary connector and 40 IO signals are required. FPGA resources are used to move data from the prototype board to the ProtoLink interface board which can store up to 2 seconds of probe data. Complex triggering and waveform and RTL annotated views are supported. The depth of the acquisition memory is very helpful for debugging embedded processor code.
Certus™ from Mentor:

Certus is an on-chip logic analyzer and debug interface that works at the RTL level through all major FPGA tool flows. Compression is used to reduce on-chip storage requirements. Automated signal and clock selection features make it easy to select from thousands of signals quickly and easily and to capture signals in the correct clock domain. Uses JTAG port.

<table>
<thead>
<tr>
<th>Product</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx ILA</td>
<td>Advanced trigger features-boolean, state machine, storage qualification</td>
<td>Requires significant on-chip resources</td>
</tr>
<tr>
<td></td>
<td>Configure and run within Xilinx tool flow</td>
<td>Requires recompile to change signal selection</td>
</tr>
<tr>
<td></td>
<td>Real time operation</td>
<td>Timing changed from base design</td>
</tr>
<tr>
<td>Synopsys Identify™</td>
<td>Real time results annotated to RTL code</td>
<td>Requires significant on-chip resources</td>
</tr>
<tr>
<td></td>
<td>Synthesis and placement bypass to speed iteration</td>
<td>Requires bitfile generation and reload</td>
</tr>
<tr>
<td></td>
<td>Real time operation</td>
<td>Timing changed from base design</td>
</tr>
<tr>
<td>Synopsys ProtoLink™</td>
<td>Real time results</td>
<td>Uses some FPGA resources for data transport</td>
</tr>
<tr>
<td></td>
<td>Stores up to 2 seconds of probe data</td>
<td>Requires customer interface to prototype board</td>
</tr>
<tr>
<td></td>
<td>Waveform or annotated RTL views of captured data</td>
<td>Requires custom interface board for data capture</td>
</tr>
<tr>
<td></td>
<td>Supports complex triggering</td>
<td>Uses Synopsys design flow</td>
</tr>
<tr>
<td>Mentor Certus™</td>
<td>Lossless compression minimizes storage requirements</td>
<td>Requires significant on-chip resources</td>
</tr>
<tr>
<td></td>
<td>Automatic clock discovery</td>
<td>Requires recompile to change signal selection</td>
</tr>
<tr>
<td></td>
<td>Waveform viewer</td>
<td>Timing changed from base design</td>
</tr>
<tr>
<td></td>
<td>Multiple FPGAs supported</td>
<td></td>
</tr>
</tbody>
</table>

++++++++++++++++++++++++++++++++++++++

Using Existing On-chip Resources

Existing on-chip debug approaches use the configurability of FPGAs to add extra on-chip logic, in the form of a simple logic analyzer, to capture data and control the debug process. Unfortunately this technique ‘perturbs’ the system under test and in the worst case can even introduce additional errors that adds another layer of debugging to an
already complex design. Additionally, a recompile and reconfiguration process is needed to change the signals of interest, which adds significant time to the debug cycle.

It would be best to use an existing on-chip resource in order to eliminate the introduction of additional errors to a design. If the existing resource also made available all the key on-chip signals, no recompile would be needed to select new signals of interest. Luckily Xilinx FPGAs already have an existing dedicated capability on-chip that can be used for these purposes.

The Xilinx Configuration Readback Function

Let’s start with an example FPGA board. Our **DNV7F4A** (figure 1) is one of the largest FPGAs boards in the Virtex-7 generation. When packed with logic, debugging the 50+ million gates of logic is a challenge. The block diagram of the **DNV7F4A** is shown in figure 1. Figure 2 shows an individual SLICE of the FPGA. The difference between SLICEL and SLICEM is not relevant. It is the outputs of the orange flip-flops in Figure 2 that are shifted out. This figure depicts the dataflow of the flip-flop state to the database that represents the state of the design.

Xilinx FPGAs use an internal shift register chain to bring in configuration data to load the various elements of the FPGA fabric, the flip-flops in the SLICEs and to initialize the dedicated logic blocks that allow the FPGA to be customized by the user. This shift register chain is also used during the FPGA configuration readback verify function to check that the configuration bit stream was loaded correctly. Readback verify transfers the internal configuration data directly from the FPGA fabric and dedicated logic blocks into the shift register and then shifts it out so it can be accessed for verification or testing. In order to extend the capabilities of the readback function, a readback capture mode is provided whereby the registers, within the FPGA fabric configurable logic blocks, are
sampled along with all the normal configuration data. This provides access to all the state information within a design and can be very useful for testing and debugging. Figure 3 below shows a simplified illustration of how the FPGA fabric register data (the blue squares) is captured in the configuration shift register (the grey boxes) that links all of the registers in the entire FPGA via a single scan chain. Note that even registers in the IO logic cells are sampled in this same way. The diagram only shows the FPGA fabric registers for simplicity. The readback data is transferred via an 8-bit bus to a housekeeping FPGA, which transfers it via PCIe to the main memory of the host computer, typically a PC. The resulting database is matched with the signal names and a GTKWave compatible file is created for viewing.

![Figure 3: Xilinx Configuration Readback Logic - Simplified Version](image)

**DN Readbacker Overview**

The DN Readbacker tool takes advantage of the dedicated on-chip readback capture capability in Xilinx FPGAs to take a ‘snapshot’ of the state of all the registers on the chip. This snapshot can then be shifted out and the results are used by the DN Readbacker function to show the value of any register of interest in a design. Note that because it can take a significant amount of time to shift out all the data from the FPGA, captures can only be made periodically, typically one capture every second or so. With that constraint in mind, it’s possible to display results using a standard waveform viewer, either statically or with the one second capture period, as illustrated in Figure 4 below.
A common debugging technique that can illustrate the capabilities of DN Readbacker is to use it to trace back from a known error to the root cause of the error. By inspecting the state of the registers that drive the signals into the logic that is causing the error, you can see where the inputs are not acting as expected. A simple example can show this approach in detail.

An Example Using DN Readbacker to Find Bugs in Your Code

Let’s say you have a logic function that checks for a timeout condition, used in a watchdog timer. An alarm register is set if the watchdog timer has overflowed. The watchdog error signal is just one of many inputs into the error control state machine, as seen in Figure 3, below. Let’s further assume that the design was working fine until the clock frequency was increased from 100MHz to 200MHz, but now the error control state machine isn’t reacting correctly in some error conditions. A quick check on timing shows that there is still lots of margin in the design. You appear to have a bug in your code.
You can quickly and easily use DN Readbacker to look at the various inputs to the error control state machine, which all come from the outputs of various registers, without needing to recompile your code and reload your design. After seeing that most of the inputs are fine you notice that the watchdog alarm signal is incorrect - it isn’t creating an alarm when it should. This causes other errors in the design, downstream. You zero in on the code fragment of the watchdog counter shown at the bottom of Figure 3. When watch_counter reaches the CLKS_TO_WATCH limit the watch_error register should be set, but this isn’t happening. Knowing that the error only just showed up, you realize that the watch_counter register width wasn’t changed when the frequency was increased, but the width of CLKS_TO_WATCH is being set by the CLK_FREQUENCY parameter, which was changed to increase the operating frequency. CLKS_TO_WATCH has more bits than watch_counter so the comparison is never false. The error signal is never activated. Bug found!

Figure 3: WatchDog Timer Block Diagram and Code Fragment

if(watch_counter < CLKS_TO_WATCH)
    watch_counter <= watch_counter + 1;
else
    watch_error <= 1'b1;

Sidebar- DN Readbacker Key Features
- Noninvasive real time readback of FPGA register state
● 100% coverage of FPGA registers
● 1 complete readback/second (depending on FPGA size)
● Running clock or single step
● Works on all DINI Group Xilinx-based FPGA boards:
  ● Xilinx UltraScale, Virtex-7/Kintex-7, Virtex-6
● Output to standard .vcd file and displayed in GTKWave
● No RTL support required
● Noninvasive observation of all FPGA registers
● Not necessary to redo synthesis or place/route

+++++++++++++++++++++++++++++++++++

DN Readbacker- Key Features

DN Readbacker differs from embedded logic analyzers in several key respects. These differences allow DN Readbacker to be used very quickly and efficiently to observe the register state of a design as a ‘snap-shot’ at a selected time. Here are some of the key features, constraints and ‘care abouts’ when using DN Readbacker:

● DN Readbacker does not require adding ANY logic to the design and does not require re-running the synthesis and or implement tools to use. Its use is completely noninvasive in both FPGA resources and debugging time.

● DN Readbacker requires that the bitfile is generated with PERSIST on the configuration pins, and that an "ll" file is generated. If this wasn't done initially, then only the "generate bitfile" step must be rerun to enable the DN_Readbacker functionality. So the time for set-up is, worst case, just a couple of minutes.

● With DN Readbacker there is no built-in logic trigger to determine when a capture takes place. A capture is triggered completely asynchronously by the user via the host software. This is a key difference when compared to embedded debugging techniques that support on-chip triggering options. For some complex debugging cases on-chip triggering may be a necessity and the use of an invasive approach may be preferred. (Note that on-chip triggering can be used in conjunction with DN Readbacker in situations where it is a necessity).

● EVERY register in the design is read (except for the shift register latch and the Select RAM since these are not really registers, and the memory, select IO and clocking resources since these are destructive reads) so you need not specify a small subset of registers ‘of interest’ ahead of time, as required in embedded logic analyzers.
• DN Readbacker only reads a single snapshot at a time, it does not buffer up a window of many captures the way most embedded logic analyzers can. This limits the frequency of captures to the frequency at which the captured data can be shifted out of the FPGA. Since a large FPGA may have over 100K frames, and a frame can take around 20us to shift out, it can take a one second between captures.

• DN Readbacker is free for Dini Group customers and runs from the Emu software.

Additional DN Readbacker Use Examples
Here are some additional examples of typical debug use cases where DN Readbacker can be used to quickly and easily locate and eliminate bugs in a design.

Power on and Nothing Happens
How often has this happened to you? Many times the very first bug in a design keeps the design from initializing and beginning execution. Is the clock functioning correctly? Does the device reset function correctly? Has an error condition been computed incorrectly that stops further processing? Has the memory training state machine locked-up? So many things could be wrong. To use an embedded debugger you would need to instrument the design, select a set of ‘likely suspects’ and iterate until you find the source of the error, rerunning place and route each time you need to change the list of ‘likely suspects’. With DN Readbacker you can immediately observe the state of resets, clock domain operation, the state of the reset state machine and all the other possible error sources without a single ‘recompile’.

Embedded Processor Hangs
Now that FPGAs have included ‘hardened’ embedded processors on-chip debugging has become even more complex. The potential list of ‘likely suspects’ expands dramatically and even just the selection of processor related debug signals and the definition of complex triggering configuration can take a significant amount of time. Even with a simple bug like a processor ‘hang’ it can be difficult to zero in on the culprit without a few iterations. Even more insidious, perhaps the debug tool itself freezes. What do you do then? With DN Readbacker you can immediately scan out all the state registers and quickly identify what, if anything, the processor is doing and why it’s stuck.

Data Transmission Errors
Once the design seems to be working it is not unusual for more difficult bugs to appear. For example, perhaps a peripheral interface is accepting data but once in a while it stops accepting data. With an embedded debugger you will need to identify the signals of
interest and rerun place and route. If there are multiple data channels you may need to instrument a large number of signals. With DN Readbacker you can simply track the data path to see where data flow stops. You can even look at FIFO registers and see if any are full and not accepting data. Once you identify the likely FIFO you can zero in on the potential source of the error and then look at the relevant code to find the bug. Using DN Readbacker to point you at the section of your code with the bug can be a big time saver.

More complicated Debug

Users, of course, are free to add logic to help trigger the readback. And all of our products have single step clock capability. This means you can get a clock by clock waveform of the entire design for a large number of clocks, even when spread across multiple FPGAs. The only limitation is the size of the hard drive and the amount of time you are willing to wait.

Using DN Readbacker with Other Debug Tools

Additionally, DN Readbacker can be used in conjunction with other debug tools as a guide to more detailed investigations that may require design changes or more debug set-up time.

Future Improvements

Future improvements are planned for DN Readbacker to increase performance and efficiency. For example, by implementing more intelligent read ordering (currently reads are just executed sequentially based on frame number) read latency can be further reduced increasing the scanning frequency.

Conclusion

DN Readbacker can provide dramatic time savings when debugging your FPGA design. Because it’s non-invasive in FPGA resources and doesn’t require rerunning place and route on the design for re-instrumentation it delivers very fast debug cycle times. DN Readbacker can often quickly identify the area of code that is causing the error so you can find and fix your bugs in record time.