WP276 (v1.0) December 17, 2007

Programmable Development and Test

We all know the benefits of using Field Programmable Gate Arrays (FPGAs): no NRE, no minimum order quantities, and faster time-to-market. In an ideal world, designs would never need to be changed because of design errors, but we all know that sometimes this is necessary.

However, a great number of people do not exploit the fact that the devices can be configured with test applications during the development and production test stage.

This white paper explores efficient options to help in product development and accelerate testing on the production line.

Introduction

With even the lowest cost Spartan™ FPGAs providing hundreds of thousands of gates and embedded memory blocks, nearly every design is some kind of system rather than just individual functions or glue logic. With so many functions integrated inside a single device, how can you be sure that each is performing correctly? Many communications to FPGAs occur at a very high speed and cannot be treated as simple digital Low and High levels. Continuity checks are not enough to verify circuit boards, backplanes, and cables when devices are operating in excess of 100 MHz, let alone at several giga-Hertz. Finally, how can you be sure that the other components connected to your Xilinx device are also functioning correctly?

Typical Design

Creating a design consisting of hundreds of thousands of gates has never been easier. HDL synthesis tools together with IP cores can soon fill the embedded memory blocks and logic resources of a large Spartan device. In many cases, the whole system can be synthesized to fit a single device, allowing you to avoid all the old issues of multi-chip partitioning. Pass it through the place-and-route tools to provide a finished design, and all you need is the PCB.

Of course, life isn't that simple. Regardless of design capabilities, it is very rare that a design, especially a large one, will work the first time. (Your design may have been 100% right, but the specification may have been lacking.) Engineering is about making things work, and this is where we need all the help we can get.

Development of Modules

When we develop a system, we break it into smaller sections and implement each one in turn. It often makes sense to simulate each of these modules to ensure that it works as expected. However, there are times when that simulation is difficult and time consuming.

It is not that surprising that certain amazing DSP algorithms can result in very complex modules requiring extensive simulations. Consider a 1024-point FFT algorithm: 20,480 multiplications must be performed. With a single multiplier, this will take at least 20,480 clock cycles to complete. This takes time to simulate, especially when timing is included.

I have also found that simulating a simple function, such as a UART, can be very time consuming. I was trying to prove that a 100 MHz clock transmitted at 9600 baud; a simulation of 104,116 clock cycles was required just to transmit one character, as shown in Figure 1. I don't mind doing it once, but I have a limited amount of time.

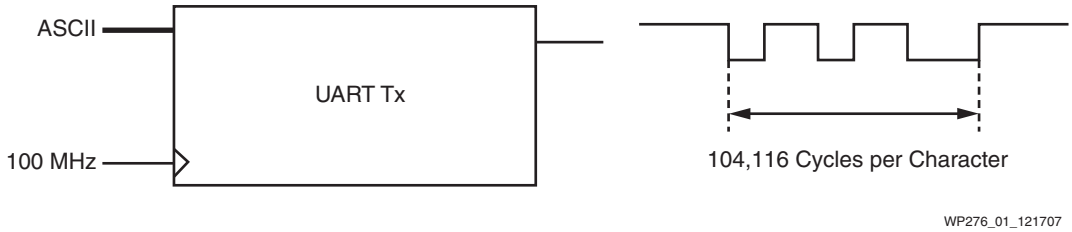

Figure 1: **UART Simulation**

This is where it is useful to convert the design to hardware and use the programmable silicon to actually try out the design at real speed and avoid simulation completely. It is also helpful to have a board with a Xilinx FPGA designed purely to help with development—although you will often find that one of your previous product boards can be adopted for this task.

The basic prototyping requirement is to be able to easily connect an oscilloscope or logic analyzer to some I/O and configure the device from the development environment of your PC or workstation, as shown in Figure 2. The best arrangement will also enable you to make data communications to and from the target device so that test vectors and results can still be handled at the keyboard and monitor without getting tangled up in wires. This is where products from companies like Nallatech (www.nallatech.com), which uses a FUSE programming environment, are helpful. For a complete list of development boards for Xilinx FPGAs, see the Boards and Kits page on xilinx.com.

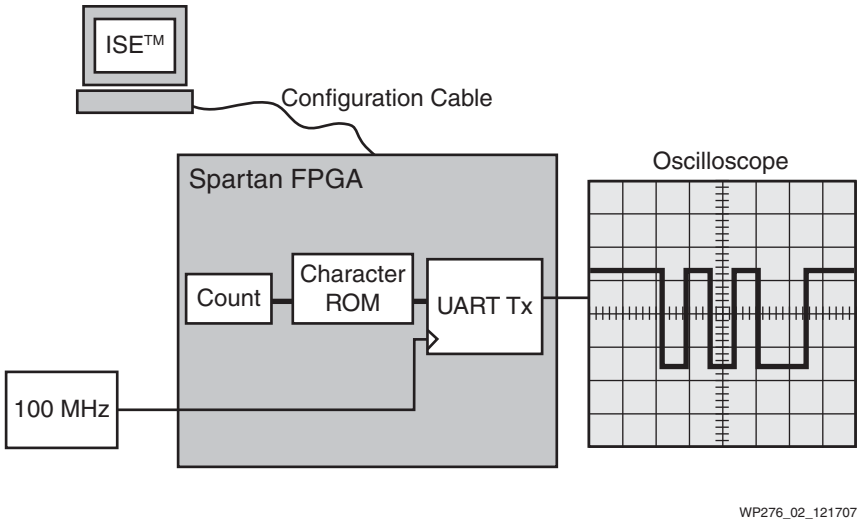

Figure 2: **Basic Debug Setup**

Start with the primitive setup shown in Figure 2. This setup makes it very easy to prove the operation of the UART transmitter. Initially, only have the UART transmitting the same character repeatedly to enable a clear image to be displayed on the oscilloscope for baud rate measurements. Later, expand the hardware test bench to include a character ROM holding all the characters and a counter. This allows the UART to be connected back to the PC serial port for some real-life testing.

Keep the Unknown to a Minimum

I will be the first to admit that testing in hardware can also be difficult, and obviously that is why there are so many tools for simulation. It is not easy to probe every signal needed. For this reason, it is important to keep the amount of unknowns to a minimum, and then perform simple “go” and “no go” experiments. Any failure sends you straight back to the drawing board and possibly some simple simulations. This should give you some clues as to where to start looking.

To help avoid the silly things like specifying the wrong I/O pins, it is a good idea to create a hardware wrapper that becomes a known, good piece of logic connecting to a

"virtual socket" inside the target device where you put your module under test, as shown in Figure 3.

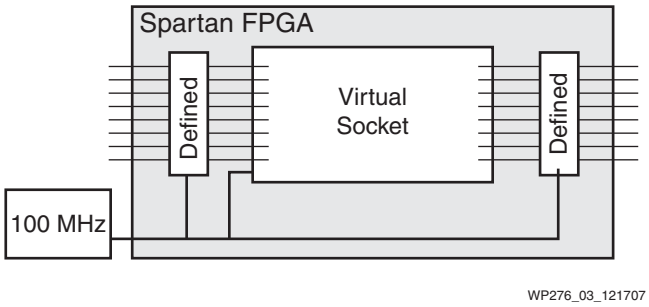

Figure 3: **Hardware Wrapper**

In some cases, the wrapper will just provide simple wire connections; however, I would recommend that it at least contain some flip-flops to make the overall test design synchronous. This will also be useful when testing high-performance macros, because you can find that dropping a small macro connected to I/O into a relatively large device becomes a placement battle leading to timing issues that wouldn't exist in the final application. We are trying to save time, not spend time performing place-and-route and trying to meet impossible time specifications that are irrelevant.

In other situations, it may be desirable to have a more complex hardware wrapper. When testing my UART macro, I also wanted to exercise the FIFO buffer by attempting to overfill it and then to check that it would empty and stop the UART transmitting. There was a danger that my test circuit was becoming more complex than the actual macro being tested, and that was asking for trouble.

This was the point at which I realized the PicoBlaze™ processor had another use. It was obvious that the easiest way to test a UART was to connect it to a processor, but now it was clear that a pre-defined embedded processor was an excellent way to help test anything. PicoBlaze and MicroBlaze™ processors are "known" and tested pieces of hardware.

The hardware test bench was now able to be "software," allowing for relatively complex tests to be generated without the fear of introducing hardware errors. A block diagram of this test bench is shown in Figure 4.

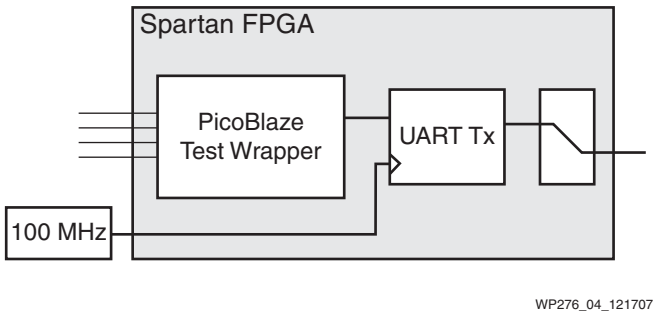

Figure 4: **Test Bench Using PicoBlaze Processor**

Software Helps Find the Design Faults

The introduction of simple software into the test wrapper requires you to think in a slightly different way and to use a different methodology. I have found that using two different methodologies together during the test phase has been an excellent way to discover the faults in my designs.

When things don't work as hoped, I have to look at both my software (assembler) code for the PicoBlaze processor in my test wrapper and the VHDL for the hardware module under test. Sometimes I find that the problems are in my software code, sometimes in the VHDL module, and very often in both. This sounds desperately negative, but I actually find that it provides a very stimulating way to make progress through the debugging phase with each methodology helping to expose the faults of the other. In contrast, I can find myself looking at the same piece of VHDL and a simulator screen all day unable to expose a bug that is right under my nose. (A case of seeing what I want to see rather than what is actually written!)

It may be useful to make a complete software model of the hardware module under test and use the wrapper to provide a simple comparison of results. It is highly likely that the hardware module will be many times faster than the software module, but this type of test would be much more about functional verification and still provide acceleration over traditional simulation.

In most cases, I have found it adequate to use the processor as a form of stimulus or result analyzer only. The advantage of mixed hardware and software sections is still present, because you are thinking of their interaction and are again led to investigating the cause of any failures or differences.

Production Test and Built-in Test Equipment

All Xilinx devices support JTAG for boundary scan. This enables the I/O of devices to be driven independently of design application and makes it easier for automatic test equipment (ATE) to force each track of the PCB to test for short and open circuits, etc. However, this does not allow devices and connections to be tested at full speed or provide a practical way to test every function of a component such as a comprehensive RAM test.

This is one reason for a system to be able to perform some level of self-test. Some tests may be simple microprocessor code to exercise parts of a board, such as a routine to make each LED flash on and off in turn. More complex systems may emulate the transmission of known data packets or perform memory tests.

The addition of built-in test and production test facilities increases the complexity and cost of the design (Figure 5). Even simple test logic can impact performance, as it often interferes with the data path. By exploiting the reconfigurable nature of the FPGA, a

completely separate design can be created to provide the test functions, leaving the whole device available for the real design.

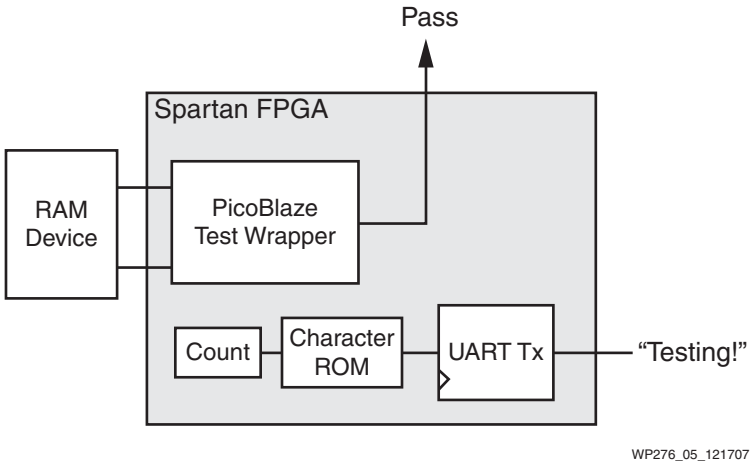

Figure 5: **Built-In Test Equipment**

If you have been performing module tests using hardware, many of the test circuits will be ideal for adding to this test configuration. Others can be made quickly with little regard to efficiency, as there are often excessive amounts of resources to use without the real design being present. In the example in Figure 5, the UART transmitter is now used to generate a fixed message to the ATE or some other part of the product, and a PicoBlaze processor is used to perform a comprehensive RAM test and simply report a pass or fail status.

If the test configuration will only be used in production testing, the configuration bitstream can be part of the ATE package and will most probably be downloaded via JTAG. If the same test configuration, or an alternative test configuration, is required after deployment, a larger configuration PROM or other storage space is virtually the only overhead.

Conclusion

I am not suggesting that you suddenly become an expert in reconfigurable computing, partial reconfiguration, or reconfiguration "on the fly." I am merely suggesting some simple ways in which you can use the programmable nature of the devices to really help in product development, and quite possibly lower the cost of built-in test equipment and accelerate testing on the production line.

As on-chip systems become more common and design complexity increases, simulation becomes difficult or even impossible in practical terms—why simulate what you can do for real?

Revision History

The following table shows the revision history for this document:

Date	Version	Description of Revisions
12/17/07	1.0	Initial Xilinx release.

Notice of Disclaimer