USING PARTIAL RECONFIGURABILITY TO AID DEBUGGING OF FPGA DESIGNS

Andreas Ehliar
Department of Electrical Engineering
Linköping University
58183 Linköping
email: ehliar@isy.liu.se

Jacob Siverskog
Linköping University
email: jacsi169@student.liu.se

ABSTRACT

This paper discusses the use of partial reconfigurability in Xilinx FPGA designs in order to aid debugging. A debugging framework is proposed where the use of partial reconfigurability can allow for added flexibility by allowing a debugger to decide at run time what debugging module to use. This paper also presents an open source debugging tool which allows a user to read-out the contents of memory blocks in Xilinx designs without needing to use any JTAG adapter. This allows a user to debug an FPGA in situations which would otherwise be difficult, i.e. in the field.

1. INTRODUCTION

The ability to reconfigure parts of an FPGA is a powerful feature that has generated a lot of interest in academia. The typical use case is to dynamically swap in hardware modules as needed in order to be able to run a complex hardware design on a relatively small FPGA.

One good example of this is an encryption application where only one encryption algorithm will be in use at any given moment. By using partial reconfiguration (PR), accelerators optimized for specific encryption algorithms can be loaded as needed [1]. Similar ideas have also been used in for example video coding and software defined radio.

In this paper we will explore how partial reconfigurability can be used to debug FPGA designs in an area efficient manner. We will also discuss how the ICAP (internal configuration access port) on some Xilinx devices can allow a design to be debugged in the field, even when no JTAG adapter hardware is available. We are targeting the Xilinx Virtex-4 device, but the concepts should translate equally well to other FPGAs with support for PR, including the Altera Stratix V FPGA (the first Altera FPGA to support PR).

The rest of this paper is organized as follows; Section 2 discusses related work, Section 3 discusses the requirement which we place on the debug modules described in this paper, Section 4 introduces our proposed debug framework, Section 5 discusses a stand-alone module for blockram readback, Section 7 discusses future work and Section 8 contains our conclusions.

2. RELATED WORK

There are a number of publications that deal with the use of PR in a general sense, but relatively few publications discuss the use of PR in the context of debugging FPGA designs.

In [2], the authors discuss how PR can be used to create a flexible debugging interface for the eMIPS processor. For example, if hardware watchpoints are needed, a module which supports hardware watchpoints can be downloaded into the area in the eMIPS processor that is reserved for processor extensions. And when hardware watchpoints are no longer needed, another module can be downloaded into this area. This allows the user of the eMIPS processor to use a wide variety of powerful debugging options, as long as no more than one of these facilities are enabled simultaneously.

Another use of PR for debugging occurs in the PATIS framework [3]. PATIS is a framework that enables rapid implementation of FPGA designs by dividing the system into several PR modules that can then be independently compiled in parallel. (It will also speed up incremental compilation as only modules that have been modified needs to be recompiled.) PATIS also contains a debugger which allows the contents of flip-flops located at module boundaries to be read-out using the ICAP port.

Finally, while not strictly related to debugging of FPGA designs, partial reconfiguration can be used to great effect when testing the functionality of FPGA chips. In essence, parts of an FPGA can be used as a built in self test (BIST) module for the remaining parts of the FPGA fabric to significantly speed up wafer testing as compared to more traditional solutions [4].

3. REQUIREMENTS ON FPGA DEBUGGING

In this paper we will classify debugging of FPGA designs into two scenarios:
- Debugging which takes place during the development phase.
- Debugging which takes place in the field once a product (or prototype) has been shipped to a customer.

The first category is well catered for. A designer has a number of debugging options available for this scenario. For example, Chipscope [5] on Xilinx and SignalTap [6] on Altera devices can be used to embed an internal logic analyzer into the design. If these logic analyzers are not sufficient, it is even possible for the designer to change the functionality of the design to simplify debugging. For example by including hardware based assertions.

In many cases it is also common that development takes place on a generic prototyping board with an FPGA which is larger than the intended target device. This allows a wide variety of debugging options to be included in the design without worrying about area issues.

However, as soon as a system is shipped to a customer, debugging is significantly more difficult. The customer may not have access to the hardware and software tools required for conventional debugging tools; and even if they have, instructing the customer in their use might be difficult. Assuming that the customer would be able to use for example ChipScope, including it will still incur an area overhead. (Which could be significant if a large number of signals are of interest.)

One possible solution to this problem, is to send a special debug version of a design to the customer when it is apparent that some sort of debugging is necessary. This bitstream could include for example a ChipScope core where the most relevant signals have been included. This is a reasonable approach if the design contains a bug (or bugs) which is easily reproduced.

On the other hand, if the bug is hard to reproduce and manifests itself only rarely, it is much better if the debug support is already present in the original FPGA configuration bitstream which is sent to all customers. This is not a guarantee that the cause of the bug can be found and isolated, but the inclusion of some sort of debug support ensures that at least some sort of post-mortem analysis can be carried on the system.

Ideally, the following constraints should be fulfilled for a designer to include debug support in a shipping system:

- Little impact on system performance
- Small area overhead
- No special hardware requirements (e.g. JTAG adapter)
- No special software requirements
- Ability to debug a live system

The first two constraints are probably the most important. Few designers can afford to include debug support that would significantly reduce the clock frequency or significantly increase the area of a design. However, a designer can probably live with the fact that some special software or hardware might be required to use the debug framework efficiently.

4. DEBUG FRAMEWORK

In this section we will propose a debug framework that can fulfill all requirement listed in the previous section. It has a relatively small area and performance impact on the system performance. It is also, in theory, able to operate over a standard serial port, negating the need for special hardware.

Finally, the interface presented over the serial port is created in such a way that no special software is required to operate it. A standard terminal program, such as HyperTerm in Windows or Minicom in Linux, should be enough to operate the user interface of the framework. (However, it should be noted that the current proof-of-concept version of the tool requires a JTAG adapter to reconfigure the framework, but this requirement could easily be removed in the future by allowing configuration bitstreams to be downloaded over the RS232 interface.)

4.1. Framework architecture

The architecture of the framework is shown in Fig. 1. A debug interface presents an interface across either the JTAG port or a serial-port (or both). When a serial-port is used, the debug interface contains a small 8-bit micro controller which is responsible for presenting a text-based user-interface to the user.

4.2. Requirements on the developer

While it (in theory) would be possible to dynamically select any signal in the FPGA and dynamically reroute it using par-
tial reconfiguration, this is not supported in any official tools from the FPGA manufacturer. This means that the developer will need to manually select the signals that should be visible to the debug framework and route these to bus-macros located adjacent to the PR region.

However, it is highly unlikely that the developer will be able to determine exactly which signals that are required for debugging. (If the developer knew what kind of bugs that would appear, he would probably be able to fix them before the design is released to customers.)

Therefore it is necessary for the designer to determine which signals that are most likely to be needed during the debugging of a wide variety of debugging scenarios. Likely candidates will vary depending on the design, but will probably include for bus signals, the current state vector of finite state machines, and important input/output signals. If a certain module has proven particularly prone to bugs during development it would make sense to add extra visibility into this module by routing a large number of these signals to the debug framework.

4.3. Proof-of-concept implementation of the framework

Using the PR version of ISE 9, we have implemented a proof-of-concept version of the debug interface shown in Fig. 1.

Two tools have been implemented and tested in the framework so far, a logic analyzer block and a design read back module capable of reading back the contents of any blockram in the design. In order to reconfigure the PR region of the FPGA, the JTAG interface was used for simplicity reasons. We have not yet implemented the ability to download a new configuration file through the RS232 interface (although we have verified that our proof-of-concept implementation is capable of using the ICAP interface to reconfigure the FPGA).

Unfortunately, we had some hard to debug problems with the PR version of ISE 9 which made development difficult. Notably, merely changing the name of a module (without making any other design changes) could cause a design to malfunction when downloaded to the FPGA. (These issues are the reason that only two of the proposed modules have actually been integrated into the PR flow, even though more modules have been designed.)

The area for the microcontroller part of the debug framework is roughly 350 LUTs and 200 flip-flops. Note also that each signal that is routed to a bus-macro will occupy some area.

4.4. Logic analyzer

A logic analyzer is one of the most useful tools available for debugging a digital design. It is also one of the two modules which we have actually integrated into our debug framework.

The trigger condition for the logic analyzer is based on pattern matching, that is, once a certain (configurable) pattern appears on the inputs to the logic analyzer module, it will fill the entire blockram with samples and stop once it is full.

A logic analyzer that is implemented using only one blockram is capable of storing up to 18 kbit of data when (under the assumption that a suitable number of signals are captured, i.e. 9, 18, 32, or 72). The area for the logic analyzer module is roughly 400 LUTs and 200 flip-flops (besides the blockram used for storage). (Although it should be noted that this module has not been optimized for size yet.)

If more than 72 signals are of interest, it is either necessary to use more blockrams, lower the sampling frequency, or select between a number of different logic analyzer configurations. For example, one configuration could monitor bus transactions while another configuration might monitor a state machine in conjunction with the input and output signals of said state machine. The ability to partially reconfigure the FPGA means that several such configurations could be supported without any area-overheads, as long as no more than one configuration is needed simultaneously.

4.5. Design read back module

This module is intended to contain support for reading back configuration data for debugging purposes. While this could, in theory, read everything from LUT configuration to routing information, the current version only supports the read back of blockram contents. The area for this part of the design is roughly 300 LUTs and 200 flip-flops and 1 blockram. This module also requires access to the ICAP module to be able to read-back blockrams.

For more information about this module, see section 5, which describes a stand-alone version of this module which doesn’t require the rest of the debug framework.

4.6. Bus monitor and interface

This module is a cross between a logic analyzer and a hardware assertion module. That is, it is able to monitor bus transactions, and store transactions matching a certain pattern to a blockram. It is also able to monitor that the specifications for the bus transactions are followed (e.g. making sure that the request signal is not deasserted before the acknowledgment signal is asserted). In some cases it may also be worthwhile to be able to initiate transactions on the bus.

The complexity of this kind of module is dependent upon which bus protocol that is used, and if features such as burst reads and burst writes should be supported. As a comparison, a wishbone initiator compatible with the debug framework has an area of roughly 100 LUTs and 100 flip-flops. If bus-monitoring is also required, the area will increase to roughly the same as the logic-analyzer described above.
4.7. Event counters

This module should allow the user to see for how many clock cycles a certain signal has been asserted. This can be of use as a sanity check by for example verifying that the same amount of packets that have been received by the system have also been transmitted by the system. This kind of module could also be used for profiling of a system (e.g., profiling the activity of various buses).

The architecture of the event counters module is shown in Fig. 2. To save area it is based on a hybrid approach where short distributed counters are used to store the LSB part of all counters and a central counter module will store both the MSB and LSB part of all counters in a blockram. The central counter module will periodically sample each distributed counter in a bit-serial fashion and update the blockram based counter accordingly.

It is also important to size each distributed counter accordingly. For example, if 128 counters are needed, each counters needs to be at least 11 bits wide in order to avoid overflow when reading out all counters in a bit-serial fashion.

The general architecture of a small counter is shown in Fig. 3. The counter can operate in two modes:

- Counting mode
- Counting and shifting mode

In normal operation, the device will function as a simple counter. To indicate that the device should enter count/shift mode, a one should be clocked into the serial control flip-flop. (The top-most flip-flop in Fig. 3.) When in count/shift mode (that is, a one has been clocked into the serial control flip-flop), the counter will simultaneously shift out the current value. Once a counter has shifted out its current value, Global control signal[0] is pulsed for one clock cycle to indicate that the next distributed counter should enter count/shift mode.

In order to implement this in an area efficient manner, the LUTs used for the adder are reused as a shift register by forcing the counter value to be added to itself, thus left-shifting the value. In order to avoid corrupting the counter when new events arrive during left-shifting, a global control signal (Global control signal[11:1] in the figure) is used to mask out the bits that should be left-shifted. (The expand to 11 bits module in Fig. 3 will either output 11 zeroes or 11 ones, depending on the current value of the serial control flip-flop.) Initially, the entire counter should be left-shifted, in the next clock cycle, the LSB bit should not be left shifted, and so on. This allows continuous operation of the event counters.

This leads to an event-counter architecture which is very area-efficient. The total area of one of the small counters is \(N + 1\) LUTs and \(N + 2\) Flip-Flops. \(N\) of those LUTs are used for the adder, and the final LUT is used for the multiplexer choosing whether to pass through the serial output of another counter instead of the current counter value.

For example, if 128 signals should be monitored, only 1536 LUTs are needed for the counters (excluding the central counting module with around 100 LUTs). A more traditional architecture where each individual counter is 36 bits wide would require 4608 LUTs to monitor 128 signals (excluding readout logic).
4.8. Savings from using partial reconfiguration

It is clear that area can be saved by using partial configuration. The exact savings are dependent upon how the debug module is used. If the only use of the framework is to use a logic analyzer with a fixed configuration there are not going to be any real savings. In fact, due to the fact that bus macros have to be used, the usage of partial reconfiguration will actually increase the area.

On the other hand, if three or four different modules are needed in order to debug different problems, the area savings can be significant. This should allow a designer to include a very flexible debug framework in a real product unless the FPGA is almost completely utilized.

5. BLOCKRAM DEBUG MODULE

5.1. Architecture

The architecture of the blockram debug module is fairly simple. The same 8-bit MCU that is used to control the debug framework described in Section 4.1 is used to control the stand-alone module. This controller is connected to an ICAP interface module and it is also connected to a serial-port which allows the user to communicate with the debug module over RS232. The final area of the debug module is around 670 LUTs and 410 flip-flops (depending upon synthesis options). We have run the module at 50 MHz (which is more than enough), but it can be synthesized to well over 100 MHz in even the slowest Virtex-4.

5.2. Design flow

When using this module, the first step is to simply to include the module in the top-level design and connect it to a serial-port. However, at this point in the design flow it is not possible to know the location of the blockrams that are present in the design (except for designs where all blockrams have been floorplanned).

To solve this problem a non-standard design flow is used where the place and routed design is converted into XDL format (which is an ASCII representation of the place and routed netlist). This allowed us to write a simple tool to find the names and location of all blockrams in the design. The designer is then presented with a list of all blockrams and allowed to choose which blockrams that should be readable in the final design. After the debug module has been updated with this information (located in the program memory of the MCU) it is necessary to convert the design back to the NCD (native circuit description) format in order to use the bitgen tool to generate the FPGA configuration bitstream. As no rerun of the place and route tool is required, this is a fairly fast operation.

5.3. User interface

The user interface is a simple text based menu system. The user is presented with a list of blockram names and can print out the content of the desired blockram. The content is displayed in hexadecimal. It is easy enough to use that the customer discussed in Section 3 should have no problems using it as long as they are familiar with a terminal program. The customer could then email the output of the memories to the developers for further debugging.

5.4. ICAP issues

While conceptually easy to implement, we encountered a few problems that should be of interest to other people interested in using the ICAP interface for similar purposes.

Most importantly, Xilinx’ CAD tools does not take the timing of the ICAP module into account. A lot of time was spent on debugging this issue using various methods, including modifying the design in the FPGA Editor to be able to probe various internal signals using an external logic analyzer. At this point we realized that the design would suddenly start working if a certain signal was routed to a pin on the FPGA. We finally concluded that the hold-time requirements of the registers inside the ICAP module were violated.

This can be solved by either using location constraints or by instantiating dummy LUTs to ensure that the combinational delay is large enough so that no hold-time violations can occur. (Although the LUT based approach is more flexible as it does not require the user to constrain the placement of the logic surrounding the ICAP module.)

Another important issue is that the blockrams have only two ports. This means that it is impossible to read out the contents of a blockram without potentially disturbing the system currently running on the FPGA, at least if both ports of the blockram are used. Unintuitively, reading back the contents of one blockram may also disturb other blockrams as well. This means that it is not possible to read back the contents of a blockram on a live system without potentially disturbing it, regardless of whether the blockram that is read back is accessed or not. One solution to this is to stop the
clock by disabling one or more clocks. (This requires the debug module to be connected to a clock buffer capable of disconnecting all clocks that are connected to a blockram in the design.)

A related problem is that it is not possible to store the ICAP data directly into a blockram (since reading back the contents of a blockram may corrupt the data written to another blockram). Therefore a temporary buffer is used for storing the ICAP results (one frame of 41 words) before writing it to the blockram. For the same reasons, the MCU needs to be stalled during blockram read back operations in order to avoid corrupt instruction fetches from the program memory.

Another issue which we encountered was that the layout of the blockram contents is not documented by Xilinx. However, by telling bitgen to output a .LL file, the mapping of the blockram contents to the configuration bitstream could be figured out. The details of this mapping is out of the scope of this paper, the interested reader is referred to the source code of our blockram debug module.

6. ALTERNATIVES TO PARTIAL RECONFIGURATION

All of the designs discussed in this paper can be implemented without using PR. For example, the debug framework described in Section 4 can certainly be implemented without PR as long as the area overhead is acceptable. (Although this area overhead can be significant if a large number of debug-modules should be included.)

Even the blockram read back tool can be implemented without using the ICAP port by modifying the synthesized netlist and inserting a wrapper around each blockram. However, the area overhead of such an approach will be significant. And even if the area overhead would be acceptable, the performance impact of such a wrapper is likely to be significant as well.

7. FUTURE WORK

As already mentioned, the old version of ISE that we used lacked the stability required to use the debug framework described in Section 4 on production systems. It would be interesting to see whether newer versions of the ISE tools with PR support (which we unfortunately do not have access to) are stable enough to allow this. Once stable PR support is available from the tools, the proof-of-concept nature of the debug framework could be improved upon, for example by enabling support for downloading of designs via the RS232 interface, hence removing the need for a JTAG adapter.

Another opportunity for improvements is to improve the debug read back module to include support for Xilinx FPGAs besides the Virtex-4. We plan to do this for the FPGAs with ICAP support which we have access to.

An interesting future research direction would be to automatically determine which signals that are of particular interest for debugging so that the designer does not need to do that himself. For example by investigating testbench coverage statistics and inserting extra debugging facilities in modules with low coverage.

Finally, in an ideal world, it should be possible to simply enable a checkbox in the synthesis tool - Reserve 1500 LUTs for PR debugging. Then it would be possible to download a wide variety of standardized debugging tools to the FPGA without having to reconfigure the entire FPGA.

8. CONCLUSIONS

Partial reconfiguration is a powerful feature which can be used to create area-efficient and flexible debugging structures. While the tools used in this paper (ISE 9 with PR support) were only stable enough for a proof-of-concept implementation we still believe that the concepts presented in this paper are sound. Hopefully, this paper can serve as an inspiration for debug tool developers to investigate the possibility of using the PR features of modern FPGAs.

However, the blockram debug module is immediately useful as it does not rely on any PR support in the CAD tools. Since it is released as open source we hope that other designers will use it (either for its intended purpose or as a reference on how to use the ICAP module).

9. REFERENCES