THINKING OUTSIDE THE FLOW: CREATING CUSTOMIZED BACKEND TOOLS FOR XILINX BASED DESIGNS

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

Dake Liu
Department of Electrical Engineering
Linköping University
Sweden
email: dake@isy.liu.se

ABSTRACT

This paper is intended to serve as an introduction to how to build a customized backend tool for a Xilinx based design flow. A Python based library called PyXDL is presented which allows a user to manipulate XDL files which contain a placed and routed design. Three different tools are presented which uses this library, ranging from a simple resource utilization viewer to a tool which will insert a logic analyzer into an already routed design, thus avoiding a costly complete rerun of the place and route tool.

1. INTRODUCTION

Traditionally, users are not very interested in the inner workings of the FPGA tool chain they are using. As long as everything is working correctly there is no perceived need to invest time and effort on learning about obscure implementation details. Although most users have probably looked at a routed design in for example Xilinx’ FPGA editor relatively few users have modified such a design.

There are however large opportunities for those who are interested in inspecting and modifying placed and routed designs. For example, a design viewer could be constructed that not only shows the slices of the design, like the floorplanner does, but also figures out the functionality of a slice and shows a symbol for a mux, adder, inverter, and so on. This will allow a user to quickly see if the synthesizer has created reasonable logic without having to load the FPGA editor which usually shows much more detail than necessary.

In terms of modifying a placed and routed design, most users are probably interested in tools that are helpful for debugging a design such as instrumenting a design to improve the visibility of internal signals. The FPGA editor has included functionality to insert probes into a design and route those signals to external pins for a long time and the ChipScope [1] product has improved on this functionality by allowing the user to insert a full logic analyzer into the FPGA.

Finally, when the usage of partial reconfiguration of FPGAs is more widespread it is likely that already placed and routed designs will have to be modified before deployment.

This paper presents a simple way to write useful programs capable of inspecting and modifying placed and routed Xilinx designs. The used method is to use the xdl tool to translate Xilinx proprietary NCD (Native Circuit Description) files into XDL (Xilinx Design Language) text files which can easily be processed by an application. A Python library called PyXDL has been developed to analyze and modify XDL files and three different backend tools written in Python has been written to demonstrate the capabilities of this library. The first tool can take a design and report the resource utilization of individual modules in the design. The second tool is a design viewer capable of showing the type of logic in each LUT as described above. The final tool allows a logic analyzer core to be inserted into an already routed design and present a user interface over RS232.

While it might seem esoteric and cumbersome to write your own backend tool the main parts of the Python library and tools described in this paper were actually written over a period of less than two weeks (except for the logic analyzer core which was already written for another project where it had to be manually instantiated in the RTL source code). It is therefore feasible for even smaller developers to write their own customized tools and we hope that this paper might serve as an inspiration for like-minded developers.

2. RELATED WORK

As previously mentioned, the FPGA editor included in ISE can show a design in more detail than most users care for. It is also possible to change the design although this is probably impractical for larger changes. There is also a command line version of the FPGA editor available called fpga_edline
which is capable of executing scripts created by the FPGA editor.

Unfortunately there is no documented way to control the FPGA editor from a user written program. The included scripting support is just a way to repeat previously defined commands, the script language is not a complete programming language. This makes it unsuitable for an application that needs to read data from a design as opposed to making changes to a design at fixed locations.

A much more interesting alternative is the JBits SDK [2] from Xilinx. This allows Xilinx designs to be manipulated from Java. In fact, it probably contains all the functionality that a user could want in terms of design manipulation. It isn’t publicly available and users have to ask for access to it. The main drawback is that JBits has been discontinued and there is no support at all for newer FPGAs in it (newer than Virtex-II) and there seems to be little interest from Xilinx to add such support. In fact, if JBits was publicly available with support for all new FPGAs from Xilinx, there wouldn’t have been any need to write this paper.

Finally, abits [3] is a tool similar in spirit to JBits which allows Atmel bit streams to be manipulated.

3. THE XDL FORMAT

The XDL file format is an ASCII based translation of Xilinx’ proprietary NCD file format. It will typically contain two types of statements, instances and nets. An instance can be any logic element in the FPGA such as for example a slice, ram block, or DSP block. It may or may not be placed at a certain location. A net statement will describe the name of a certain net and the instances it is connected to. It may also contain routing information. An example of a very simple XDL file is shown in Figure 1.

A drawback of the XDL file format is the scarcity of documentation. Earlier releases of ISE such as 6.3 contained written documentation about the file format [4]. Unfortunately this documentation has been removed in later versions of ISE. Even so, some details of the XDL format wasn’t documented in 6.3 either. Luckily some basic information about the format is included in every XDL output file created by the \texttt{xdl} tool unless the \texttt{-noformat} switch is given.

4. PYXDL - PYTHON BASED XDL MANIPULATION LIBRARY

A Python based library called PyXDL has been developed to simplify development of backend applications. The basic idea behind the library is to convert a placed and routed design into XDL by using the \texttt{xdl} tool included in ISE. This file can be modified as required and converted back into Xilinx native NCD format. This allows small changes to be made to a design without requiring a complete and often time consuming synthesize, placement, and routing iteration. This is accomplished by telling \texttt{par} (the place and routing tool) to only route un-routed nets and only place unplaced instances. (The guide-file feature of par is used for this purpose.) This flow is illustrated in Figure 2.

4.1. Constraints

One problem which occurs when merging two designs, which isn’t immediately obvious when looking at the XDL files, is the constraints files. The timing constraints in these must also be merged if reliable timing estimates is expected.

4.2. Resource analyzer script

The design resource analyzer is a small tool written for a designer who wants to know the resource utilization of a certain module or modules in larger design. One way to figure this out is to synthesize that particular module separately. This method may or may not work depending on the properties of the larger design. For example, if the synthesizer can determine that only relatively few values can appear on a certain input port of a module included in a larger design, the synthesizer could potentially remove large parts of the module.

As hinted at in the previous section it would be better to be able to analyze a large design directly to find the resource usage of individual components. This is exactly what the resource analyzer script does as shown in Figure 3. The script itself is very simple and the most complex part is actually printing the design usage in a hierarchical and cumulative fashion. This kind of XDL parsing, although easy, can still lead to useful results. A regression test incorporating this script could for example warn about a submodule which has
Fig. 2. The typical Xilinx flow augmented with the PyXDL tool to merge a design such as a logic analyzer into a placed and routed design. The new part of the flow is shown in gray.

Fig. 3. Using the resource analyzer script to view the resource utilization of various parts of a design.

grown (or shrunk) by a large factor when compared to the previous run.

4.3. Design viewer

The design viewer is capable of viewing a design and showing the configuration of the slices. It is similar in functionality to the floorplanner. In Figure 4 a part of an OpenRisc based design is analyzed by the design viewer.

4.4. Logic analyzer

Putting a logic analyzers into a chip is not a new idea. Both Xilinx and Altera already offers such products (ChipScope and SignalTap). There are also some logic analyzers written by hobbyists available on the net such as Fpgadb [5].

The main idea behind this section is to show that it is easy for any user to duplicate the main selling point of ChipScope, i.e. the capability to insert a core into an already synthesized and routed design. While it would be easy to create a logic analyzer core which fully mimics ChipScope by connecting to the internal boundary scan primitive we did not intend this tool to be a ChipScope clone. Instead, the intention was that this tool should be useful in systems that might not easily be connected to a PC with a ChipScope client such as remote systems. Therefore the logic analyzer core is operated via a simple serial port interface.

An example of the output of the logic analyzer is shown in Figure 6 and an example of a simple GUI which allows the core to be easily inserted into a design is shown in Figure 7.

4.4.1. Implementation details

The design of the logic analyzer is shown in Figure 5. It consists of a simple 8 bit microcontroller which is responsible for presenting a text based user interface to a serial port. The MCU is connected to a logic analyzer core via a Wishbone bus. This bus also creates an easy way to extend the
functionality of this core with additional modules. The logic analyzer is currently hardcoded for a maximum of 64 signals which is stored to a 2 kilo-word large buffer.

The Python GUI allows the user to load an XDL design and select which nets to monitor. After the user is satisfied with the selection the program will load the synthesized version of the logic analyzer and remove any elements which will make it hard to merge the logic analyzer into the design (e.g. IOBs and BUFGs). The appropriate flip-flops in the logic analyzer is added as an extra destination of the selected nets. The program memory of the MCU is also modified so that net information such as name and width is available to it. Finally, a user selected clock net is connected to all flip-flops in the logic analyzer core.

The curious reader is also referred to Appendix A which contains an example of how PyXDL can be used to merge a small design into a large design.

4.5. Availability of PyXDL

The PyXDL library will be published under the GPL at http://www.da.isy.liu.se/~ehliar/pyxdl/ together with the sample applications described in the previous sections. The RTL code of the logic analyzer core will also be made available under the MIT license so that users can use and distribute merged designs without worrying about the stricter terms of the GPL license.

5. DISCUSSION

The applications presented in this paper shows only a few of the many possibilities that could be tapped by a creative designer. The applications described earlier could of course be improved by improving them. The design viewer could be improved to show more points of interest to a designer such as clock domain crossings, pipeline depths, and perhaps even show some sort of design complexity metrics for different parts of the design (a long pipeline without feedback is far less complicated and probably easier to test and verify than a state machine with many feedback paths).

The logic analyzer could be improved by adding additional modules to it such as counter modules for statistic gathering. Another interesting addition would be to replace the RS232 interface with another interface such as for example Ethernet or USB.

5.1. Other possible applications

There are many other interesting applications which would be possible to develop. One example would be for those interested in very large FPGA designs that must be mapped onto several FPGAs. A tool could be created that automatically partitioned the XDL file into more than one FPGA.

A similar tool could be made that partitioned a design for a large FPGA into different region of such an FPGA. The advantage of such a design would be that the time consuming
Fig. 6. The logic analyzer user interface showing instruction fetches on a Wishbone bus. The analyzer has been set to trigger when STB and ACK are both asserted.

placement and routing of the partitioned design could easily be parallelized on a cluster of computers.

5.2. Remaining issues

There are unfortunately some issues that are hard to solve in a satisfactory fashion. The main problem is that there is very little information available about routing. Whereas placement is relatively straightforward, reliably routing a design requires detailed timing information about the internals of the FPGA, something which Xilinx hasn’t released for modern FPGAs and most likely will not release for the foreseeable future.

Another problem that any tool of this kind will face is that the synthesized design isn’t exactly the same as the RTL source code. The various optimizations employed by the synthesizer will remove and rename many nets, making it harder to find the correct signal/bus to inspect. This could be mitigated if more back-annotation information was available to the tools.

Finally, the PyXDL library has only been tested on Virtex-4 based designs.

6. CONCLUSION

We have shown that it is easy to create powerful backend tools for a Xilinx based design flow such as a logic analyzer inserter. By manipulating the design file directly a time consuming full synthesis/placement/routing iteration is avoided and therefore increasing productivity. It is our intention that this paper will inspire other designers to explore these possibilities as well.

7. REFERENCES

Appendix A. PYXDL EXAMPLE

This appendix contains an example of how to use PyXDL to merge a synthesized design into a larger design. The example consists of a design which will monitor a signal and assert an external signal forever if an internal signal has ever been asserted (e.g., an error signal of some sort). In order to shorten the example, the constraints file is not updated with the timing group from the small design. Some values are also hardcoded instead of dynamically getting the values from the XDL files such as the name of the clock networks.

**PyXDL source code to merge a synthesized design (test.xdl) into a large design (system.xdl):**

```python
from xdl import xdl, xdlnet
from pcf import pcf
from xdlutil import par

with guide
    largedes = xdl("system.xdl")
    largedespcf = pcf("system.pcf")
    # Clock network for the large design
    clocknet = largedes.netsbyname["clk BUFGP"]

tinydes = xdl("test.xdl")
    # Unplace stuff we don’t need
    tinydes.unplace(design())
    tinydes.remove_unused_dcminssts()
    tinydes.remove_inst("clk")
    tinydes.remove_net("clk")
    # Create a unique prefix for the other design so
    # that we don’t have to worry about name clashes
    tinydes.add_prefix("TEST/")
    # Convert flip flop in the IOB to an internal signal
    myiob = tinydes.insts["TEST/testin"]
    testinpin = tinydes.convert_input_to_internal(myiob)

    oldclknets = tinydes.netsbyname["TEST/clk BUFGP"]
    # Remove old clock network
    tinydes.remove_net("TEST/clk BUFGP")
    tinydes.remove_inst("TEST/clk BUFGP/BUFG")

    # Merge designs
    largedes.mergedesign(tinydes)
    # Merge old clock network into new design
    for pin in oldclknets.inpins:
        largedes.add_inpin_to_net(clocknet, pin[0], pin[1])

    # Select signal to monitor
    thenet = largedes.netsbyname["traceit/state FFd1"]
    largedes.add_inpin_to_net(thenet, testinpin[0], testinpin[1])

    # Add the IOB to the PCF constraint file and
    # select where to place it (at pin AC6)
    largedespcf.addiob("TEST/testout","AC6")

    # Place and route the design
    par with guide(largedes, largedespcf, "new.ncd", "tmp")
```

**Verilog source code for a simple monitor application. testout will be asserted if testin has ever been asserted:**

```verilog
module test(
    input clk,
    input wire testin,
    input wire rst,
    output reg testout);

    reg tmp,sample;
    wire fbloop;

    always @ (posedge clk) begin
        sample <= testin;
        tmp <= fbloop;
        testout <= tmp;
    end

    FD monitorfd(.C(clk),.D(fbloop | sample),
        .Q(fbloop));

endmodule // test
```