Dynamic Reconfiguration and Incremental Firmware Development in the Xilinx Virtex 5

J. Jones [a], M. Stettler [a]

[a] Princeton University, Princeton, NJ, USA
[b] Los Alamos National Laboratory, NM, USA

john.jones@cern.ch

Abstract

The size and complexity of the latest generations of FPGAs has increased dramatically. This in turn means that the time taken to develop and build even small firmware projects is increasing exponentially. Pre-constrained logic placement and routing are becoming critically important for the use of specialized components in the FPGA such as serial link interfaces. This necessitates significant changes from 'normal' firmware tool flows in order to effectively develop systems based on these devices.

In this paper we discuss several methods for improving turnaround speed and design safety, including: pre-placed and pre-routed hard macros / Relationally Placed Macros (RPMs), and pre-synthesised black-box netlists. Possible methods of dynamic partial reconfiguration are also discussed in this context.

I. Introduction

The current CMS trigger and DAQ electronics are largely based on FPGA and processor technology several generations behind the currently commercially-available devices. Most systems in CMS are based on either the Xilinx Virtex-II or the Xilinx Virtex-II Pro (and similar Altera-made FPGAs). Since then Xilinx has developed the Virtex-4 and Virtex-5, scaling in feature size from 250nm to 65nm and incorporating many new hardware features such as Digital Signal Processing (DSP) cores, tri-mode Gigabit Ethernet (GbE) MACs and PCI express (PCIe) endpoints amongst others. Furthermore, total logic capacity has increased approximately ten-fold with a corresponding increase in maximum clock speed.

However this creates a new problem: FPGA firmware synthesis and routing, being an NP-hard problem, is an extremely computationally-intensive process. Given the arrangement of logic and routing in the devices and the scaling of feature size in deep submicron technology, the complexity of logic placement and routing is increasing at a greater-than-geometric rate. Without the development of new firmware development techniques to manage this increase in complexity, firmware build times can increase from minutes to hours and even days.

Compounding these issues are limitations in the peformance of the available development tools. In particular, when using certain advanced features of the device, the behaviour of the design and performance may change significantly. In the worst case a new revision of a previously working design can cease to function after minor changes are made to the design [1].

II. FPGA Tool Flows

A. *Standard Xilinx FPGA Tool Flow*

A typical Xilinx tool flow might proceed as follows:

- ***Synthesise*** a design from either VHDL or Verilog (or both). This may be achieved using either vendor-specific tools such as Xilinx's XST or Mentor Graphics' Precision Synthesis. The end product is a Look-Up Table (LUT)-level netlist. At this stage propietary IP cores may also be introduced.
- ***Translate*** a LUT-level netlist into a device-specific format and combine it with user-defined constraints. These can also be included as device-specific attributes in original VHDL or Verilog. At this point any black-box components must also be included, such that the design is complete once translate has finished.
- ***Map*** the logic component of a design into place in the device. This process uses a coarse timing definition to place the components.
- ***Place and route*** the design using a precise timing model for the device, including temperature dependency.
- Generate a device programming file that can be downloaded into the FPGA.

This tool flow will always provide the best implementation, although it may take several iterations as part of the process involves a pseudo-random search. Ultimately it is impossible for the tools to test every permutation of design that will fit in the FPGA, so this is a necessary step, but it also implies that an undefinable amount of time will be taken to achieve timing closure. In large, complex designs, the turnaround time is growing rapidly, from minutes to hours and even days.

In order to reduce this turnaround time, there are various techniques that can be adopted. The key requirement is to avoid repeatedly processing unchanging components of a design every time a minor revision is made. There are several ways to achieve this.

B. *Pre-Synthesised Netlists*

The simplest (and most generally-applicable) approach to speeding up a firmware build is to use the same method used for pre-built IP core generation. This involves synthesising a core component of the design that doesn't frequently change, and then integrating it into the full design at a later stage. This approach has few limitations, but can only speed up the first stage of the build process.

The only required change compared to normal synthesis is to turn off pad instantiation. This prevents the tools from inferring the top-level ports as FPGA pins. The output is an ngc/edf file that can be re-imported during a normal build. The file name and ports must also match those in the component definition of the main design.

C. *Post-Place/Route Macro*

A more advanced technique involves taking a sub-component of the design, then synthesising, placing and possibly routing it. The placement of the component must also be defined as a constraint. The end product is known as a Relationally-Placed Macro (RPM), a logic and routing template that must have a fixed location in the device.

Unlike the creation of a pre-synthesised netlist, the creation of an RPM is not a completely automated process. Firstly, the connections must be defined between the macro and the outside world, and timing constraints must also be carefully set. When the design is created it must be ensured that all timing requirements can be satisfied in the final design as well, because once the macro is built it *cannot* be changed by the tools. Secondly, part of the process involves using FPGA Editor to directly convert the design, strip out unused wiring and logic and define the macro ports.

RPMs can either be designed directly at the hardware level using FPGA Editor, or converted from a placed (and possibly routed) design. The first of these processes is completely manual. The second one involves taking an automatically-generated design and removing unwanted nets. This can be partly automated using a script but the initial macro generation must be done by hand. It is also necessary to prevent the removal of logic from the design that is optimised away by the tools. This involves turning off logic trimming during map.

Unfortunately, Xilinx RPMs have several limitations: firstly, a placed macro cannot use any FPGA blocks that connect to modules that are also externally connected on the device (the tools currently do not 'understand' what this means). An example of this includes the gigabit transceivers.

Routing can also be included in a macro, however unfortunately, creating a routed RPM can crash the Xilinx tools and is therefore not recommended. Unless the design has a very specific layout that's required for optimal behaviour, it is better to not use RPMs but instead use guided place and route, where the tools will use a previous build as a reference for the new one.

D. *A Bus Macro Example*

As an example, one can consider a simple pass-through bus macro. Bus macros are commonly used to pass signals between static and reconfigurable firmware modules in a design (see section III). In order to do this one must create a pass-through LUT which acts as a buffer connected to another on the other side of the design. One can then divide the design where the routing crosses a logic boundary to create a boundary interface where the logic never changes.

Figure 1 shows a LUT-level view of the macro in a Virtex 5. Only one of the inputs is used per 6-LUT as it is simply buffering the input signal to the output. This macro also includes a register on the output (although this is optional).

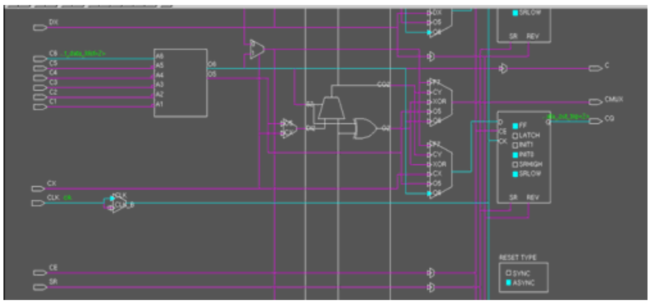

Figure 1: An example of a pass-through bus macro at the LUT level.

Figure 2 shows a higher-level view of the design, including 8 bits of bus between the top and bottom CLBs. As one can see, all of the wires are pre-routed apart from the clock tree. A clock system connection must be handled separately as it connects to a resource shared with the full design.

The origin marker on the bottom-left CLB in the design is a reference point for the LOC (location) constraint used to place the macro in the design.

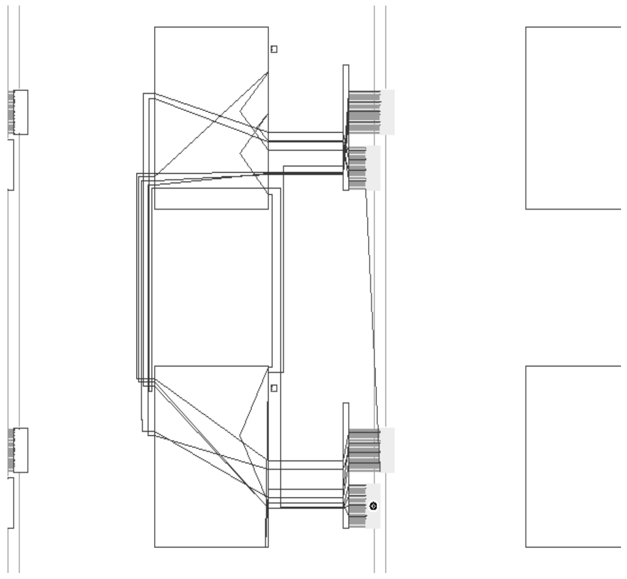

Figure 2: An example of a pass-through bus macro at the CLB level. The lines left of the CLBs going through the routing matrix are pre-routed, whereas the lines passing directly between the CLBs are still unrouted clock nets.

E. *Constraint-Based Placement and Routing*

In order to avoid the limitations of hard macros, one can use the Xilinx User Constraints File (UCF) to force the tools to use particular CLBs and routing connections for a given component of a synthesised design. This does however require a non-standard step and involve large constraint files. One generates the constraints in much the same way as an automatically-generated hard macro, with one significant difference: instead of creating a macro design and re-inserting it during the translation phase of the firmware build, one exports the placement and routing constraints to a UCF. This can then be re-imported during standard firmware build.

This does however require an exact name match between the net and logic names in the new build and the one from which the constraints were originally generated. This is in fact rather difficult as the tools hierarchically-optimise the firmware, and so it's possible for changes in different components of the design to stop this from working. The solution is to pre-build the component corresponding to the constraints as a LUT-level netlist. The tool flow will then be forced to keep the names that were present in the original design.

In spite of the complexity of this process, it ultimately allows one to pre-constrain a portion of a design precisely. This leads to the possibility of modifying one component of a design while keep others completely static, allowing for dynamic reconfiguration.

III. Dynamic Partial Reconfiguration

Dynamic partial reconfiguration is the process by which one can pre-place and route part of a design while changing another component depending on the demands of the system. It has many uses, in particular being used to reduce the required device size by configuring for two mutually-exclusive modes of operation (e.g. an encryption core where one chooses between DES/AES). It is also useful when there's no access method to modify the device's configuration apart from the device itself. Xilinx FPGAs contain an Internal Configuration Access Port (ICAP) which has all of the capabilites of the external configuration interfaces.

Xilinx FPGA programming can be broken down into 'frames', which represent the smallest configuration unit of the device. These frames can also be changed without powering down the device. In Virtex FPGAs, it is possible to modify a portion of a frame without disrupting the configuration of the rest of the frame. The only complication this creates when considering partial reconfiguration is that the FPGA's master reset does not occur for the logic's initial state, so every component in the firmware must contain an explicit reset circuit.

If one reads Xilinx documentation on partial reconfiguration, it will describe two processes: difference-based reconfiguration and module-based reconfiguration. These will first be briefly discussed, followed by an alternative method that uses contraint-based placement and routing.

A. *Module / Difference-Based Reconfiguration*

Difference-based partial reconfiguration is the simplest technique available. It involves editing the design in Xilinx FPGA Editor and simply re-saving the design. One can then use the Xilinx configuration file generator bigen to create a file containing the difference between the two firmwares (using the '-r' flag). As such it is useful when the design changes are very small but it is not generally useful.

Module-based partial reconfiguration uses area constraints to segregate a static portion of the design from the dynamic component (see figure 3). As such it splits the design into regions, where a bus macro is used to cross the boundaries between the different parts of the design. Figure 4 shows an example of this where the static component of the design is on the left and the dynamic component of the design is on the right. This process unfortunately requires the splitting of the bitstream into components, thereby requiring custom build steps, some of which are not supported in the latest tools or require special patches.

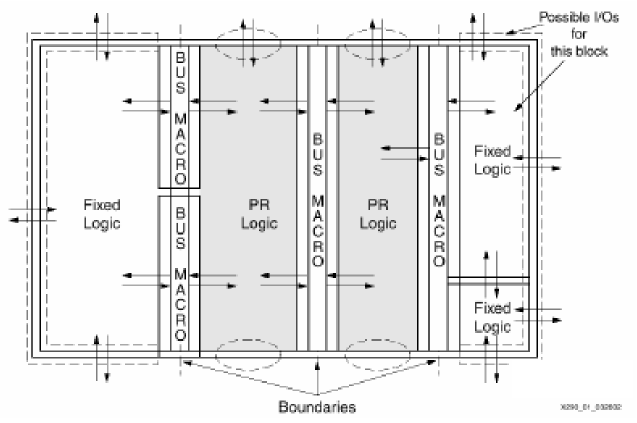

Figure 3: An example from the Xilinx online documentation of module-based design. Note the use of bus macros to traverse the boundaries between configuration regions.

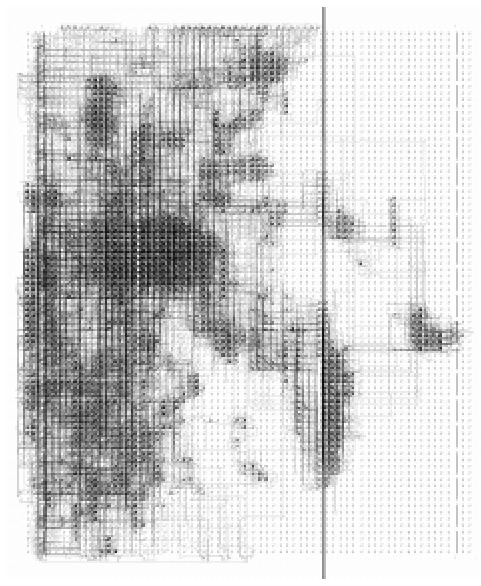

Figure 4: An example of module-based reconfiguration [4]. The left component of the design is a static microprocessor core. The right component is a reconfigurable AES core. Note the boundary between the devices is marked by a bus macro.

B. *Constrained Reconfiguration*

Using constraint-based firmware builds presents an alternative approach to the generation of partial reconfiguration firmwares. In this approach one uses loose area constraints to place a static firmware on one part of the design. This component contains a bridge to the ICAP inside the FPGA, and a secondary interface for interfacing to the reconfigurable firmware. In addition there is a bus macro used to create an interface port in another portion of the design. By exporting the routing and logic of this design as a constraints file, new firmwares can be generated by simply feeding the constraints into the new design. The only other requirement is that the static component of the firwmare is not changed, otherwise every related design must be rebuilt. The partial configuration bitstream can then be generated using the same method used for difference-based reconfiguration. The major advantages of this technique are that there are no arbitrary region constraints and that bus macros are not explicitly needed. Static and dynamic firmwares can even be interleaved.

C. *A Virtex-5 Floating Point Multiplier*

As an example, consider the following. Figure 5 shows an initial configuration providing a GbE connection via a Xilinx MGT and a bus bridge in the lower part of the FPGA. However nothing is connected to the bus in this firmware.

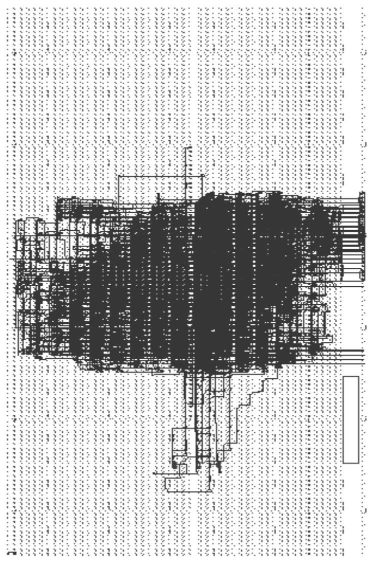

Figure 5: A simple static configuration firmware. Note the bus macro in the middle of the lower portion of the design.

Figures 6 and 7 show more developed firmwares with multiple DSP cores (the first is a double-precision multiplier, the second a double-precision square). The interesting feature of these designs is that the firmware interleaves with the original static firmware, allowing better use of FPGA resources than in module-based reconfiguration.

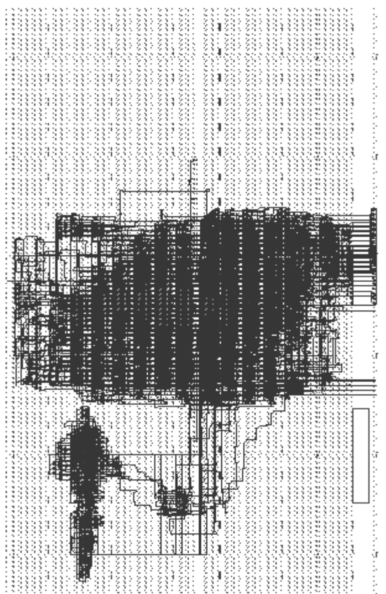

Figure 6: A more complex firmware with a double-precision multiplier connected to the bus in the lower part of the FPGA.

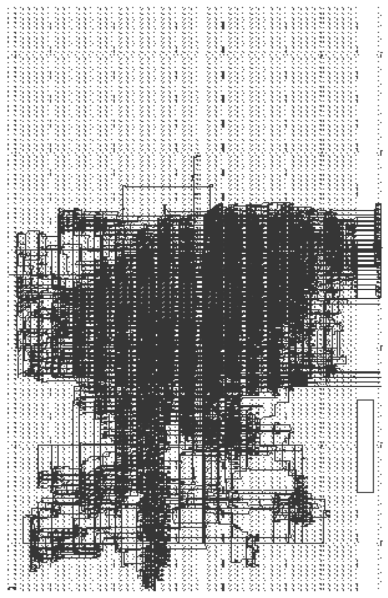

Figure 7: A two-multiplier firmware interleaved with the original static firmware.

IV. Conclusions

There are many ways in which one can improve turn-around time and reliability of firmware builds. Pre-built netlists offer a safe and reliable way of accelerating turnaround, while hard macros have some additional benefits. Unfortunately there are reliability issues with these processes in the current Xilinx tools. Contraint-based design can be used as a work-around until the tools implement hard macros in a more reliable manner.

Partial reconfiguration is an extremely useful tool in some applications, and can be largely automated. There are many caveats but for a certain class of applications the benefits of dynamic firmware modification outweight the difficulty in its implementation.

V. References

[1] G. Iles et al., “Performance and Lessons of the CMS Global Calorimeter Trigger”, TWEPP 2008, Naxos, Greece.

[2] S. Lopez-Buedo et al., “How to Implement Self-Reconfigurable Coprocessors on Spartan-3”, presentation, Universidad Autonoma de Madrid.

[3] C. Conger, “Partial Reconfiguration”, presentation, CHREC Center, University of Florida.