GENERIC PARTIALLY RECONFIGURED PROCESSOR SYSTEMS APPLIED TO SOFTWARE DEFINED RADIO

Stephen Neuendorffer (Xilinx Research Labs, San Jose, CA;
stephen.neuendorffer@xilinx.com)
Chad Epifanio (Xilinx, San Jose, CA;
chad.epifanio@xilinx.com)

ABSTRACT

Partially reconfigured FPGA systems are typically architected for a specific design, such as a Software Defined Radio System, with application-specific interfaces and system design. However, we notice that many of the characteristics of such systems are common across a variety of applications. This paper describes a generic system platform, based around a control processor with an operating system and partial reconfiguration that aims to simplify the design of such systems. The platform is implemented using a standard processor design flow targeting the PowerPC 405 processor embedded in Xilinx Virtex 4 FX devices. The design flow used for targeting the user portion of the platform is very similar to the standard (non-partial reconfiguration based) Xilinx design flow. As a result, it is possible to quickly and easily implement a processor-based design using this flow. An example design based on a 2x2 MIMO OFDM system is also shown.

1. INTRODUCTION

FPGAs are a vital part of a modern software-defined radio. In the past, their use was mostly limited to processing the high-rate data in the physical layer. But as modern FPGAs become ever larger and more cost-effective, they are absorbing more of the radio system. Microprocessor cores can be embedded in the FPGA to implement control-plane tasks and higher-layer protocol processing. In the extreme case, the entire radio can be implemented on a single device [1][2], a technique which can reduce power consumption and eliminate data transfer bottlenecks. In this paper we present a powerful new technique which simplifies the development of FPGA systems utilizing embedded processors, and overcomes some common problems associated with SDR.

One of the benefits of SDR is that the waveform application can be changed at run-time. This is especially important for military radios which need to run a wide range of waveforms. Normally, the entire FPGA is reloaded with a new image when a new waveform is to be run. Unfortunately, this will wipe out state information in any embedded processors and the peripheral interfaces. Implied is the need to reboot any OS running on the processor. On Xilinx FPGAs, partial reconfiguration techniques can be used to eliminate these problems.

There has been much discussion in the literature on the use of Xilinx partial reconfiguration (PR) [3,4,5,6,7]. Mostly it revolves around dynamically swapping one functional unit running in the FPGA fabric with an other functional unit. However, partial reconfiguration can also be used as a form of encapsulation, to isolate some portion of the design from the rest. For instance, the entire processor subsystem can be placed in one PR region, while the rest of the waveform application can be placed in another PR region. This achieves some notable benefits:

- The processor subsystem can be highly optimized to run at maximum possible speed. Design changes to the waveform application outside this region can not disturb the carefully optimized design.
- From the user's point of view, the soft-core processor has characteristics of a stand-alone hard-core processor. The processor can boot up first, and control the rest of the FPGA load process. It also retains state information while the waveform application is changed.
- Similarly, the processor subsystem can be fixed for a given platform, allowing waveform developers to focus on the waveform application instead of embedded processor tools. This has particular benefit for board providers who wish to reduce tooling burden on their end users.
- The waveform application can be loaded via peripheral interfaces not normally supported. The first FPGA load is done via normal methods, say by EEPROM via the Xilinx System Ace controller. Once the processor is running, it can reload waveform PR images delivered by any attached peripheral, such as USB or PCI, from some remote controller in the system.

- Because the processor subsystem is fixed, it is easier to verify for high assurance and mission-critical systems.

It must be noted that the techniques presented in this paper are not limited to SDR applications. They are amenable to a wide array of designs that utilize embedded processors.

1.1. Partial Reconfiguration Basics

Partial Reconfiguration (PR) is the ability to reconfigure part of an FPGA (a *reconfigurable region*), while another part of the FPGA (a *static region*) remains active and operating. Although PR is possible to various extents in several Xilinx FPGA families, this paper focuses on PR in Virtex 4 FX FPGAs. This family of FPGAs is well suited to PR, due to a combination of architectural features. In particular, configuration frames (the minimal addressable unit in the FPGA configuration space) are 16 CLBs tall, enabling reconfigurable regions to be easily tiled horizontally and vertically. Additionally, routing configuration bits can be overwritten to identical values without generating signal glitches. This capability is critical to allowing static routes to cross a reconfigurable region. Lastly, these FPGAs contain PowerPC 405 hard cores embedded in the FPGA fabric, which are well-supported under Linux. A processor system built around this core using Xilinx Embedded Development Kit (EDK) forms the basis for controlling the PR process.

In this paper, we make use of the Early Access (EA) PR flow, based on Xilinx ISE 9.1.1. These tools support a variety of merge-based partial reconfiguration, as described in [1]. In the EA PR flow, the static region is implemented first with constrained placement. The static region is allowed to contain routes through any reconfigurable regions, and the associated routing resources are excluded from being used by modules targeted for reconfigurable regions. In order to generate partial bitstreams that can be loaded into a reconfigurable region, the static routes are merged with the placed and routed design for the reconfigurable regions, ensuring that static routes remain active.

2. FPGA ARCHITECTURE

A representation of the design for the static region is shown in Figure 1. The static design contains a relatively simple system architecture which aims at being simple and relatively small in FPGA area, leaving the bulk of the FPGA area available for application-specific high speed data

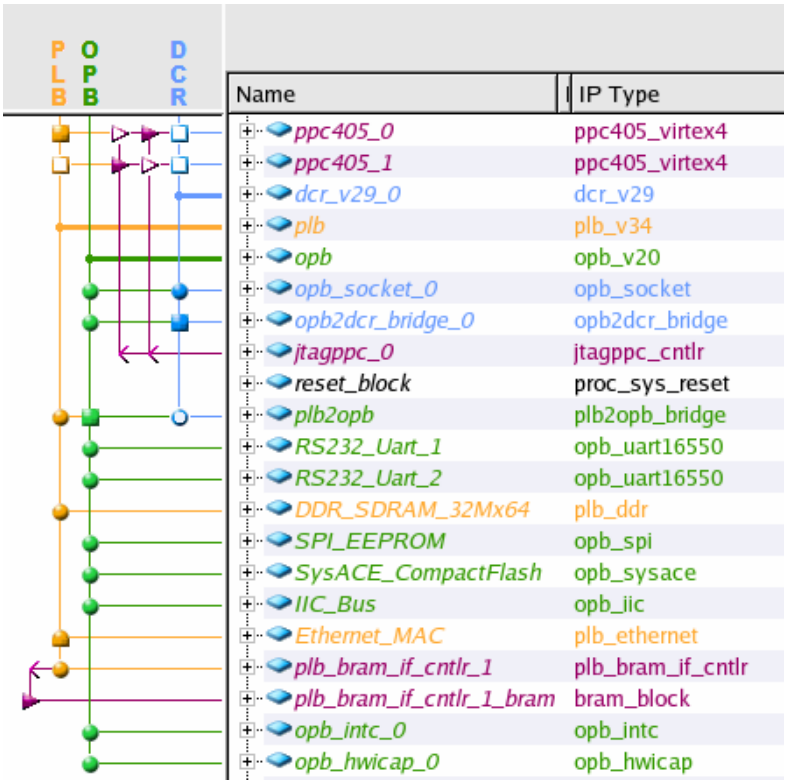

Figure 1: Static System Design

processing. We assume that the main data-processing pipeline is configured, controlled and debugged using the control processor. Although low-data rate communication is possible using a bus slave and interrupt architecture, master access to main memory from the partially reconfigured region (which could provide much higher communication bandwidth) is not supported in this design. We have made this design decision in expectation that high data rate communications to not involve the processor and exist entirely within the FPGA fabric.

The interface from the static region consists of one CoreConnect OPB slave interface, enabling a single memory mapped peripheral to placed in the reconfigurable region. Alternatively, if multiple peripherals are required, a bus bridge can be used to connect additional slaves, or local masters. In total, this design requires less than 8000 LUTs and 7 BRAM blocks. The largest individual component (requiring over half the LUTs) in the design is the 10/100 Ethernet MAC.

3. EDK DESIGN FLOW

The static design is implemented using a combination of EDK for system composition and synthesis and scripted execution of ISE using GNU Make. The reconfigurable region is abstracted at design time using an EDK pcore. This pcore contains a VHDL `component` instantiation of the reconfigurable region, along with the bus macros needed by the EA PR flow to cross the boundary between the static and reconfigurable portions, and a small amount of DCR-based control logic for enabling and disabling the bus macros during reconfiguration. This enable logic is necessary in the design due to avoid glitches during reconfiguration on bus signals being driven from the reconfigurable region. Note that no hand modification of the generated HDL code is necessary. This process is summarized in the left side of Figure 2.

Reconfigurable modules are designed and synthesized separately from the static region in a separate EDK project directory, or as straight VHDL source in a separate directory. The implementation of a reconfigurable module is also scripted using GNU Make, and makes use of the fact that EDK generates independent netlists for each IP core in a design. In particular, only *context logic* (consisting of Clock Managers, Clock buffers, and bus macros, and any HDL hierarchy containing them) needs to be placed and

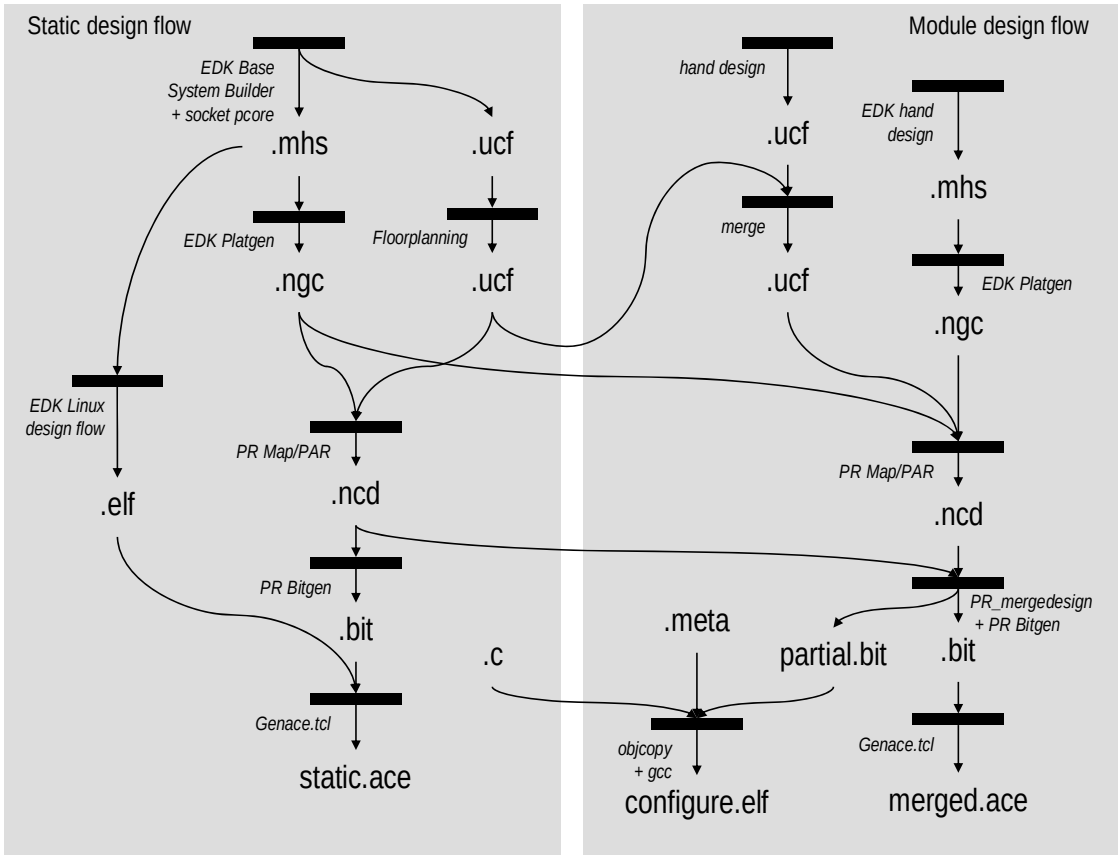

Figure 2: Design Flow

routed along with the reconfigurable region. The netlists containing context logic are copied from the static design into the reconfigurable module design so they can be accessed by the place and route tools. After place and route, the `.ncd` files for the static design and the reconfigurable modules are merged in order to create correct bitstreams. This implementation process can also be executed automatically in Xilinx PlanAhead 9.2. This process is summarized in the right side of Figure 2.

Although it is certainly more complex to create a partial reconfiguration design than a design using the standard implementation flow, the bulk of the additional effort required (such as instantiation of bus macros, floorplanning) is only associated with the static portion of the design. For a *given* static design, the implementation of a PR module is largely the same as the standard implementation flow. When combined with the correct set of OS mechanisms (such as code to interface with the ICAP device driver) and a precompiled operating system kernel, the resulting design process is actually significantly *easier* and *faster* than a similar system built from scratch using EDK. Furthermore, this system can be characterized and verified independently, providing a guaranteed level of performance.

4. SOFTWARE INTEGRATION

Although any program could be run on the processor from the previous section, we have focused on a platform integrated with Linux 2.6. Using an operating system makes it relatively easy to share the processor between any system tasks needed for managing the platform, in addition to user code that might be necessary for a particular application. In addition, the operating system provides a certain level of safety and robustness of the system: application code is not capable of interfering with the operation of the systems infrastructure. The Linux system itself is built leveraging the EDK mechanisms for generating Linux board support packages, ported to the 2.6.22 kernel series. In addition, the kernel provides a small device driver for managing the reconfigured region (described in the next section).

In this context a new application stack, consisting of user-space application code, kernel-space device drivers and FPGA circuits, can be designed and delivered without modification of the predefined platform. In order for this to be possible, design dependencies must be carefully managed so that the application stack and the platform remain consistent. Dependencies between user-space application code are managed using familiar operating system abstractions for managing processes and dynamic linking. Dependencies between code running in kernel space are managed using Linux kernel modules, which can be loaded and unloaded without rebooting the system. At the FPGA level, partial reconfiguration and partial bitstreams are essentially a way to orthogonalize a large system into pieces that can be loaded and unloaded independently.

Figure 3 shows the entire platform (including the FPGA design, plus the operating system components) on the left, and a specific application design on the right, along with the design dependencies between them. Interface dependencies (shown as dotted lines) indicate where one component depends on the interface of another component. For instance, the dotted line from user application to the libraries indicates that the user application depends on defined interface of the library (in this case, the function calls exported by a dynamically linked library as global symbols). Stronger build dependencies (shown as solid lines) indicates when one component depends on a specific implementation of another component. For instance, the solid line from the block labeled ‘user device drivers’ to the block labeled ‘Linux kernel’ indicates that Linux kernel cannot be recompiled without (possibly) needing to rebuild all kernel modules [2]. Note that using the current Xilinx EA partial reconfiguration flow, not only must the static portion of the FPGA design and the reconfigured module agree on an interface, but the reconfigured module must be reimplemented if the static portion is reimplemented.

5. RECONFIGURATION PROCESS

After partial bitstreams have been created they can be loaded into the device through the Internal Configuration Access Port (ICAP). The ICAP is accessible through Linux drivers shipped with EDK. In addition, however, reconfiguration must be coordinated with enabling and disabling bus macros and with notifying the Linux kernel of the appearance or removal of devices. These secondary operations are coordinated through a second device driver interface, accessed through `/dev/xilinx_socket`. This device is typically written with a file containing configuration meta-information. This meta-information is described by a serialized version of the Linux `struct platform_device` combined with a checksum. If the file appears to contain a valid record, then bus-macros are enabled and the structure is de-serialized and passed to the Linux kernel. The `xilinx_socket` device driver also keeps a record of the structure and in response to a new set of meta-information (or an invalid record) will notify the Linux kernel that the previous device is no longer available (in order to maintain consistency), and disable the bus macros.

This process leaves open the possibility for somewhat catastrophic errors. For instance, incorrectly notifying Linux of the presence of a device may result in kernel bus errors and possibly the inability to unload and reload the affected kernel modules. Even worse, reconfiguring the reconfigured region while bus macros are enabled almost inevitably results in a stalled processor. In order to reduce the possibility for errors, the ICAP device and the `xilinx_socket` device are treated as privileged and can only be accessed by processes with root permissions. Our intention is that coordination of this process would be managed by a single daemon process executing in a privileged process. Application processes wishing to perform reconfiguration would interact with the daemon.

Currently, the bitstream and meta-information are linked into a single executable (`configure.elf` in Figure 3) along with the code for interacting with the devices. This process traps `SIGINT`, and in response resets the state of the `xilinx_socket` device. As a result, the lifecycle of the application in the FPGA is tied to the lifecycle of an OS managed process. This approach is similar in spirit, but greatly different in implementation from [5], which performs essentially the same processes using the Linux kernel's ability to implement new executable formats.

Bitstream Validation

Another source of potential catastrophic errors is the possibility of loading a bitstream which is not a partial bitstream consistent with the executing static design. In order to avoid such errors, we prepend each partial bitstream with additional data containing a hash of the .ncd file for the static design. The 32 bit signature is also stored in the IDCODE register of the static design (and in order to maintain consistency, in the register of partial bitstreams as well). Before reconfiguration, the IDCODE register is read (using the ICAP) and the value compared with the value stored at the start of the bitstream. If these signatures fail to match, then the reconfiguration process is halted. Although this technique is sufficient to prevent unintended errors (such as a designer who failed to reimplement PR modules after reimplementing the static design), it is not sufficient to prevent against an attack against the system. For instance, an attacker interested in denial of service for a publicly available computing resource could craft an inconsistent bitstream containing the correct hash. In such cases, larger signatures, combined with cryptographic techniques [8][9] would be necessary.

6. RADIO DESIGN

In order to better understand how the platform might be

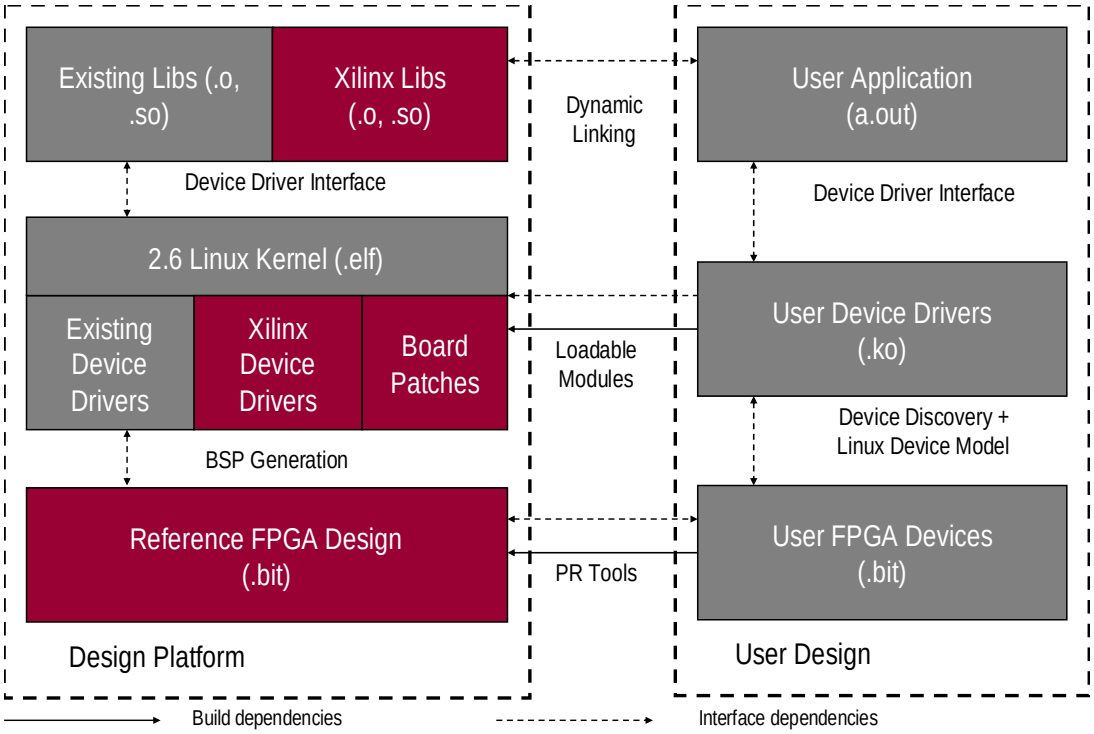

Figure 3: Platform Architecture

used in practice, we retargeted an existing SDR design to the platform. The original design is an open source 2x2 MIMO-OFDM design from Rice University targeting the WARP platform containing a Xilinx Virtex-II Pro 70 FPGA. The original design utilized a control processor running a lightweight operating system to bridge a wired Ethernet interface over the wireless channel. The original design used a PLB interface for packet transport with the FPGA fabric and OPB interface for control information.

The existing design was modified to target a prototype implementation of our reconfigurable platform by removing the bulk of the processor subsystem, and inserting an OPB bus and OPB-OPB bridge to connect multiple OPB core interfaces to the OPB interface exposed by the platform. Additionally, the design was resynthesized to target a Virtex 4 FX FPGA. As an initial design point, the packet transport interface was interfaced the OPB bridge using an OPB-PLB bridge and a minimal PLB bus configuration. Because this configuration results in an additional communication bottleneck between the processor and the wireless system, we use this system primarily for area comparison. We anticipate that future versions of the platform will provide for higher bandwidth busses. The resulting system uses 30611 LUTs, 106 BRAM blocks and 170 DSP48 blocks.

7. CONCLUSION

The processor encapsulation techniques presented in this paper may appear at first glace to be overly complex. However, one must look at this technique in the context of design reuse. The digital systems engineer expends a bit more effort constructing this processor object (alluding to object-oriented software techniques) so that the digital application engineers can limit their scope to just the application (the waveform in this case). The encapsulated processor object will remain stable, regardless of the work of the applications engineers. The use of the EDK tool is primarily limited to the digital systems engineer. The applications engineer may not need to use EDK at all, removing one tool from a seemingly growing list of tools they need to accomplish their job. This may have particular attraction for board vendors, as encapsulation of the processor subsystem makes their product easier to work with. In some sense, the processor subsystem gets absorbed into the board support package, allowing the user to concentrate on the application.

The software framework presented here is the first steps towards exploring the interaction between software and the now-dynamic hardware that partial reconfiguration allows. We describe how new hardware functional units can be dynamically loaded into the FPGA via partial reconfiguration, and any necessary drivers incorporated into the operating system, without requiring a recompilation of the core software stack. Also described are methods to reduce common system errors that can occur during partial reconfiguration, making the entire system more robust. Taken together, the hardware and software encapsulation techniques presented here offer FPGA system-on-a-chip developers a means to create more robust, easier-to-use systems that will reduce development time/money in the long run.

[1] "JTRS SDR Kit." Xilinx. Retrieved 24 Sept. 2007. <http://www.xilinx.com/dsp/defense/jtrs_sdr_lounge.htm>

[2] "Rice University WARP: Wireless Open-Access Research Platform." Rice University. Retrieved 24 Sept. 2007. <http://warp.rice.edu/trac/wiki>

[3] P. Sedcole, B. Blodget, T. Becker, J. Anderson and P. Lysaght, "Modular dynamic reconfiguration in Virtex FPGAs", *IEE Proceedings Computers & Digital Techniques*, Vol. 153, No. 3, pp 157-164, May 2006.

[4] J. Corbett, A. Rubini, and G. Kroah-Hartman, Linux Device Drivers (3rd Ed.), O'Reilly Press, Sebastapol, CA, USA, 2005.

[5] Brandon Blodget, Philip James-Roxby, Eric Keller, Scott McMillan, Prasanna Sundararajaran. A Self-Reconfiguring Platform. 13th International Field Programmable Logic and Applications Conference (FPL). Lisbon, Portugal, September 1-3, 2003. Lecture Notes in Computer Science 2778.

[6] J. Williams and N. Bergmann, Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip, In Proceeedings of the 2004 International MultiConference in Computer Science & Computer Engineering (ERSA) Las Vegas, Nevada, June 21-24, 2004.

[7] Hayden Kwok-Hay So, Robert W. Brodersen, "Improving Usability of FPGA-Based Reconfigurable Computers through Operating System Support" In Proceedings of 16th International Conference on Field Programmable Logic and Applications (FPL '06).

[8] J. Castillo, P. Huerta, V. Lopez, J. Martinez, A secure self-reconfiguring architecture based on open-source hardware International Conference on Reconfigurable Computing and FPGAs (ReConFig), September. 2005.

[9] R. Fong, S. Harper, and P. Athanas, "A Versatile Framework for FPGA Field Updates: An Application of Partial Self-Reconfiguration", Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping, San Diego, CA, Jun 2003.