A Timeshared, Runtime Reconfigurable Hardware Co-processing Architecture

by

Benjamin S. Gelb

S.B., Massachusetts Institute of Technology (2008)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009

Author . . [illegible] . . . .
Department of Electrical Engineering and Computer Science
May 22, 2009

Certified by [illegible] . . . .
Christopher J. Terman
Senior Lecturer
Thesis Supervisor

Accepted by . . . . . . [illegible] . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A Timeshared, Runtime Reconfigurable Hardware Co-processing Architecture

by

Benjamin S. Gelb

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2009, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

The constant desire for increased performance in microprocessor systems has led to the need for speicalized hardware cores to accelerate specific computational tasks. In this thesis, we explore the potential of using FPGA partial reconfiguration to create a platform for customized hardware cores that may be loaded on demand, at runtime, and replaced when not in use. We implement two new software tools, *bitparse* and *bitrender*, to demonstrate the *bitstream relocation* technique. Further, we present a functional microprocessor system coupled with a runtime reprogrammable peripheral synthesized on a Xilinx Virtex-5 FPGA and discuss its performance implications.

Thesis Supervisor: Christopher J. Terman
Title: Senior Lecturer

Acknowledgments

Thanks to my parents, who taught me to work hard and dream big, for their unending support and inspiration in both my career and in life. And thanks to Alan Bosch, a great mentor and dear friend who nurtured my childhood interest in electronics.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The microprocessor is instrumental in nearly all digital systems due to the fact that the same processor may be applied to an enormous range of different types of tasks without modification or redesign. This extreme flexibility is what makes microprocessor platforms so attractive - they can easily and cheaply be applied in almost any situation. Still, there are some classes of problems which are not handled efficiently by a microprocessor architecture due to high bandwidth or restrictive latency requirements or extreme parallelism that must be executed iteratively. These types of applications typically fall broadly into the categories of compression, signal processing, or encryption, all of which are coming to have greater and greater importance in modern computing.

For these sorts of problems, the answer is often to design a custom integrated circuit (ASIC). More recently, Field Programmable Gate Arrays (FPGAs) have provided a more flexible option, though they still tend to be employed for a single specific function within any given system design. What gives the CPU its power is that the application is nearly completely decoupled from the actual hardware - something not generally true even in a programmable logic implementation.

Recent developments in FPGA technology have allowed for the possibility of runtime partial reconfiguration of digital systems. This means that the contents of the logic resources of an FPGA may be manipulated at runtime, while leaving other areas of the chip completely undisturbed. The general motivation is to reduce area and power requirements of a system by time-sharing logic regions across many possible reconfigurations, and further allowing for the realization of arbitrary hardware at runtime.

The aim of this project is to leverage the re-programmability of an FPGA to develop a microprocessor system that supports hardware acceleration of software routines. Such hardware acceleration would allow portions of user software to be replaced by efficient, specialized, hardware cores which are loaded on demand. Further, the architecture must lend itself to fast switching between hardware "tasks," just as a CPU is multiplexed across processes. This would elevate programmable logic to the same degree of flexibility as a common CPU platform - a general purpose resource that can be rapidly applied to a wide range of problems.

1.1 Partial Reconfiguration

An FPGA is a type of programmable logic device. It has a set of logic elements layed out in silicon which are connected together within the chip by programmable signal pathways. By programming the function of these elements and programming the wires linking them together, arbitrary digital logic may be created. FPGAs are rapidly increasing in popularity as size and speed have increased, making many applications possible in an FPGA design that once were limited to an ASIC-based solution.

An FPGA has a volatile configuration memory, and usually must be reprogrammed each time it is powered on. The configuration information, typically called a "bit-stream" is quite large, and so this reconfiguration takes a substantial amount of time, measurable in tens of milliseconds.

Partial reconfiguration allows only a specific part of the FPGA's configuration memory to be reprogrammed while leaving the rest of the configuration memory in its original state. The Xilinx Virtex family supports partial reconfiguration. In fact, it has been designed so that partial reconfiguration may take place while the running FPGA system is still operating, without concern for causing any logic glitches or other problems. The "static" logic, which is the part of an FPGA design that isn't changed by partial reconfiguration, is left alone. In fact, the static logic may actually control the partial reprogramming process.

1.2 Candidate Problems for Acceleration

To motivate this work, its worth glancing at a few of the types of problems commonly performed in microprocessor systems that may be accelerated in hardware implementations.

With the recent explosion in networks and the Internet, encryption has continued to play an expanding role in microprocessor systems. In [4], a design is presented which uses a reprogrammable region in a microprocessor system to implement the various encryption algorithms used in SSH. The region is loaded with one of several supported algorithms at a time, but may be changed on the fly. A significant speedup over software is realized.

Hardware acceleration has a large potential role in multimedia as well. Video compression, in particular, is employed in demanding applications with very high bandwidth and latency requirements. The importance of hardware acceleration here is probably best evidenced by the existence of MPEG-2 encoder and decoder hardware available for standard PCs. Another testament to the level of interest in video compression is evidenced by its frequency in technical literature (a search for "H.264 hardware," for example, on Google Scholar produces a long list of results). Couple this with the rate of evolution of video compression standards and the advantage of reprogrammable hardware acceleration becomes clear; compression algorithms may be swapped on demand, and future algorithms not known at system design time may also be supported. Various acceleration schemes for multimedia appliances have been proposed as well, based on partial reconfiguration [11].

In addition to these application examples, our reconfigurable peripheral can be used to accelerate many garden variety mathematical transforms such as DCT, FFT, etc, that are commonly used in signal processing and other applications. The reach of this reprogrammable peripheral is not limited to these specific examples either, any code section that can benefit from the parallelism offered by a hardware implementation can be sped up.

1.3 Roadmap

The remainder of this work is divided into chapters. First, some fundamentals about FPGAs are presented. In chapter 2, the underlying architecture of a popular FPGA family, the Xilinx Virtex-5, is explained. Chapter 3 reviews the standard FPGA design flow, and discusses the modified tools and flow provided for use in partial reconfiguration designs.

With the fundamentals out of the way, chapter 4 begins to look at our particular question; what is the best way to develop a microprocessor system coupled with a reprogrammable hardware peripheral? Based on several prior works, various design tradeoffs are evaluated, and an approach is proposed.

Chapters 5 and 6 move on to the implementation stage. In chapter 5, two new software tools are presented. These tools, *bitparse* and *bitrender*, allow for deeper analysis of FPGA programming bitstreams. The *bitparse* tool is also used to implement *bitstream relocation*, which allow retargeting compiled logic to different parts of the FPGA. Finally, chapter 6 presents a partial implementation of a microprocessor system coupled with a reprogrammable peripheral. Some initial results are presented, and future improvements are discussed.

Chapter 2

Virtex-5 FPGA Architecture and Programming Procedure

Before delving into the intricacies of design runtime-reprogrammable circuits, it's necessary to visit the fundamentals of FPGA design and practice. In this chapter, we take a look at the structure of the Xilinx Virtex-5 and the associated toolchain and development process. The Virtex-5 is the most recent generation of the Xilinx Virtex line, and so is a good reference point for the current state of FPGA technology in general.

2.1 Basic FPGA Logic Elements

In the abstract, the FPGA can be thought of as a sea of gates which can be composed into any digital logic circuit. Of course, doing this in an efficient manner requires some more specific structure.

2.1.1 Slices and CLBs

The fundamental primitive of the FPGA is a lookup table unit, often augmented with a register for the output, and a special carry chain for building arithmetic structures. The base unit of logic in a Xilinx device is called a "slice." A diagram of a Virtex-5 slice is shown in figure 2-1.

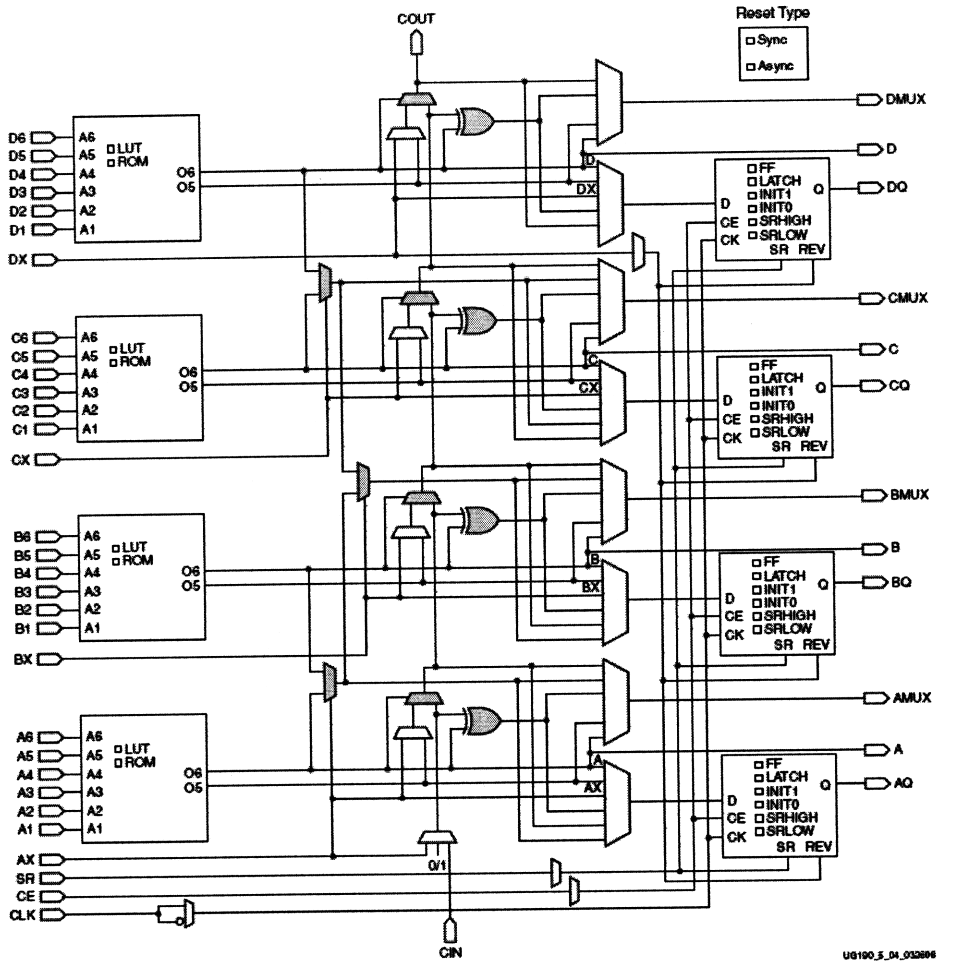


Figure 2-1: A Xilinx Virtex-5 Slice.

Each slice has four lookup-table units, which take six inputs and compute one output, which may optionally be registered. By stringing together a series of these units, arbitrary circuits can be created. The slice, of course, is programmable - the values in the lookup table may be set for any logic function, and the various data path elements may be switched on and off as desired. Slices are then grouped together to fit into a Complex Logic Block (CLB).

In the Virtex-5, two slices comprise a CLB (though this number is different in other Xilinx architectures). The importance of the CLB is that it is a standard unit from a geometric perspective all of the primitive elements on an FPGA are arranged in a grid which has cells of one CLB in size. This grid forms the basis for the interconnect architecture, which can programmed to connect primitives together. Figure 2-2 shows a set of slices (one CLB) and its connection to a switch matrix, which is part of the interconnect infrastructure.

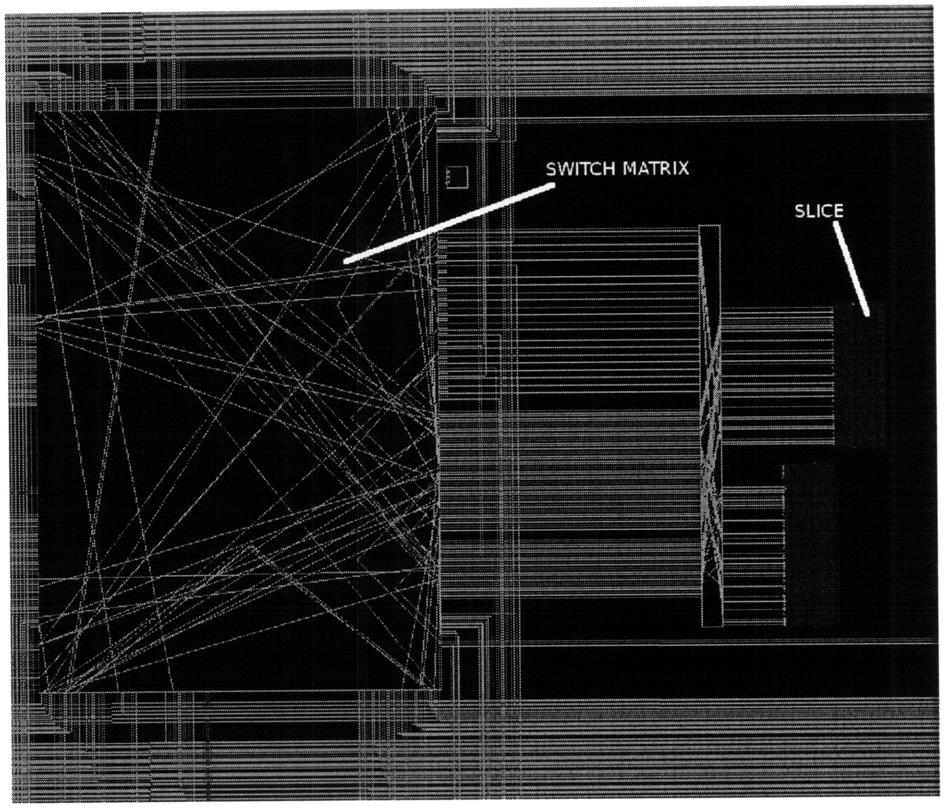


Figure 2-2: A Xilinx Virtex-5 CLB.

Each CLB plus switch matrix unit are arranged in a grid across the FPGA. The connections in the switch matrix are programmable according to a complex set of connection rules. The rules, of course, are identical for each switch matrix. Figure 2-3 illustrates the grid pattern by showing a broader view with several CLBs.

2.1.2 Routing Resources

Each switch matrix maps signal wires connected to its associated CLB to various external wires going in to and out of the matrix. Wires are usually one of two types, either DOUBLE or PENT. Each type of wire has three interconnection points (BEG, MID, END) and so connects to three switch different switch matrices. Both DOUBLE and PENT wires are unidirectional; they must be driven at the BEG end, and can

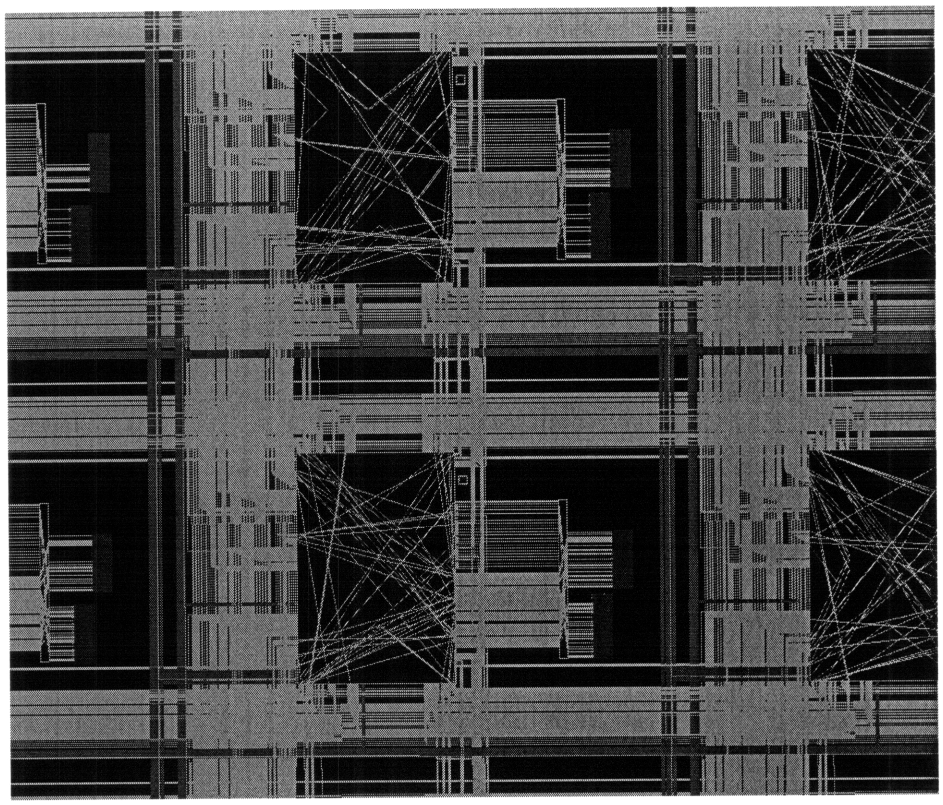


Figure 2-3: A broader view of the Xilinx Virtex-5 fabric.

be sampled at the MID and END connection points.

Where DOUBLE and PENT wires differ is in their length. A DOUBLE wire connects immediately neighboring CLBs together. Its two segments (BEG to MID and MID to END) both have length 1. A PENT line has length 5, with its two segments having length 3 and 2. A new development specific to the Virtex-5 is that the two segments of DOUBLE and PENT wires need not be in the same direction. For example, a PENT wire may travel three CLBs to the right, and then two CLBs upward. Effectively, this allows "diagonal" interconnection of CLBs which leads to a large performance improvement [10].

In addition to the PENT and DOUBLE lines, another type of wire is the LONG line, which is 18 CLBs long and has a connection every 6 CLBs, for a total of 4 connections across 18 CLBs. Unlike the other two types of routes, LONG lines are bidirectional (though they must have a defined direction at compile time - there are no tri-state elements within the Virtex-5).

With this regularized routing structure, graph methods can be applied to estimate optimal placement and routing for a particular circuit, which is indeed exactly what FPGA tools do.

2.2 Clock Distribution

In addition to the logic and signaling primitives, the Virtex-5 must have a clock distribution architecture to bring clock signals to any CLB that may require it. To do this, the FPGA is segmented into clock regions, which are areas that are 20 CLBs tall, and half the width of the chip. Each region has 10 horizontal clock lines which run through the center of the region. Additionally, running vertically through the center of the entire FPGA are 32 global clock lines. Finally, each column of CLBs has 10 global clock lines which connect to every CLB in the column.

These three levels of clock lines have programmable interconnection points between them to allow the construction of a clock distribution tree from the main vertical lines, to the regional horizontal lines, to the single-column vertical lines. Unused branches are kept disconnected in order to save power.

2.3 Special Purpose Blocks

Though the majority of the FPGA is made up by CLBs like the ones described earlier, some columns of the FPGA grid are populated by other hardware blocks. These blocks share the same routing network as the regular CLBs, but have different function. A variety of special-purpose hardware elements may be found on different FPGA varieties, but two which are nearly ubiquitous are block memory (BRAM) elements, as well as hardware multiply-add-accumulate blocks (called XtremeDSP slices on the Virtex-5). Both of these elements are common in hardware designs, but neither of them lends themselves to a particularly efficient implementation in CLBs. Thus, they are implemented as special blocks in hard silicon.

On the Virtex-5, a BRAM block holds 36Kb of data and can be configured with a variety of different data port widths and access modes. Physically, a BRAM occupies one CLB column and spans the height of 5 CLB blocks vertically. On a given FPGA, columns of BRAM elements span the entire height of the chip (no BRAMs are mixed with regular CLBs within a column). A single clock region, therefore, is exactly four BRAM elements tall.

The XtremeDSP blocks are arranged in groups of two, such that two blocks span 5 CLBs vertically. Similar to the BRAMs, a column of DSP blocks replaces an entire column of CLBs.

In addition to these types of blocks, special I/O blocks, consisting of high-powered drivers, are used to interface with the FPGA's external pins. Additionally, various clock management blocks are scattered across the chip to deal with both clock synthesis and distribution.

2.4 Configuration Memory

Configuration of the FPGA is accomplished by a large static RAM which overlays the configurable hardware elements. The bits of this RAM control the configuration of the CLBs, routing elements, I/O blocks, etc.

The configuration memory is divided into configuration frames. Each frame consists of 41 32-bit words, and spans the height of one clock region (20 CLBs, with a row of horizontal clock distribution blocks running through the center). One 32-bit word corresponds to each CLB, and the middle 32-bit word corresponds to the clock distribution block. Each type of block, CLB or otherwise, is multiple frames in width. Table 2.1 gives the widths of the various types of blocks.

Because the number of frames that make up a single column may vary depending on the type of block in a column, all types of blocks in a single column must be the same type, or at least the same number of frames wide.

Block Type	Number of Frames
CLB	36
DSP	28
Block RAM	30
IOB	54
Clock Column	4

Table 2.1: Frame Width of Virtex-5 Block Types

2.5 Bitstream Format and Configuration Procedure

The result of the compilation of an FPGA design is a configuration bitstream file, which contains an image of the FPGA configuration memory. The file itself is not just a bit-per-bit image, however, but rather sequence of programming commands to set up the FPGA programming interface, load the image, and start the FPGA.

The FPGA programming interface exposes a bank of registers used to control the configuration process. The commands in the FPGA bitstream manipulate these registers to effect the desired programming result. Four registers in particular are of interest for the purposes of loading an image into the configuration memory. These are the Frame Address Register (FAR), the Frame Data Input Register (FDRI), the CRC Register (CRC), and the Command Register (CMD).

To load configuration frames into the FPGA, the FAR is initialized to the desired starting location. Then, a Write Configuration Data (WCFG) command is written to the CMD register. Configuration data is loaded one 32-bit word at a time into the FDRI register. Every 41 words, the FAR automatically increments, advancing to the next frame. This allows contiguous frames to be streamed in one word per cycle without stopping to move the FAR pointer. Note that on the top half of the FPGA, frames are loaded from bottom to top, and on the bottom half of the FPGA, from top to bottom.

Additionally, the programming procedure calls for a CRC verification of the configuration image before the FPGA will accept the programming and start up. Prior to loading the image, an RCRC command is written to the CMD register, which zeros the CRC. The CRC then is computed on each subsequent word of data written into a configuration register. Finally at the end of the programming sequence, the expected CRC value is written into the CRC register. If a match occurs, the programming is completed and the FPGA starts up.

Chapter 3

Xilinx Early-Access Partial Reconfiguration Design Flow

Typically an FPGA design is run through a three step process to be converted from an HDL description into a hardware implementation. First, the design is translated to a netlist of gate-level primitives. In the map step, a primitive netlist is mapped to particular hardware blocks available on a specific FPGA. In the place and route step, the blocks selected in the map step are arranged on the FPGA and connected to complete the circuit. In the standard Xilinx ISE tool flow, four tools perform these three tasks - the *XST* synthesis tool and *ngdbuild* tool, together, the *map* tool, and the *par* tool. At the very end, the routed design is converted to a programming bitstream. Xilinx provides a tool called *bitgen* which performs this task.

3.1 Partial Reconfiguration Design Structure

A partial reconfiguration design is one that is intended to be programmed in pieces. A static portion of logic will always be present on the FPGA, but some parts of the design are are meant to be loaded separately (and presumably exchanged with one another). In a partial reconfiguration design, the high-level RTL description must be constructed in a fairly particular way in order to allow the creation of partial bitstreams. The top-level module should contain little more than global clock generation logic, I/O buffers, and blackbox instantiations of any static or partially reprogrammable logic. The top-level design should also contain bus macros which connect the static and partially reprogrammable regions. Bus macros are manually-routed bits of logic designed to cross the boundary between the partially reprogrammable and static logic regions.

The reason behind this strict structure is so that the static and partially reprogrammable parts of the system may be synthesized, placed, and routed separately. This is an obvious necessity for creating partial bitfiles.

3.1.1 Bus Macros

Bus macros are hand-routed bits of logic that provide a bridge between static and reprogrammable regions in a partial reconfiguration design. The requirement to statically route and place these macros falls out from the fact that the place and route tools must be able to generate a configuration file for the static part of the design in absence of the reconfigurable part and visa versa. If the entry and exit points for the boundary-crossing signals were not defined, the routing and placement problem would be underconstrained.

In different generations of devices, bus macros have been constructed differently. In Virtex-II devices, macros were based on TBUF (tri-state buffer) primitives. The buffers were placed at the module boundaries and used to provide an interconnect that could be enabled at will, and disabled during reprogramming. After the Virtex-II series, however, TBUF primitives were no longer available as part of the FPGA fabric. Instead, slice-based macros have emerged as the prevalent technique.

Typically, a slice-based macro consists of two slices in adjacent columns. One slice is placed inside the reprogrammable region, and one is placed inside of the static region. The outputs of one of the slices is then connected to the corresponding inputs of the adjacent slice. The direction is chosen based on whether the desired route is to take a signal into the reprogrammable region, or out of the reprogrammable region. Bus macros are often referred to as left-to-right, or right-to-left, top-to-bottom, etc, depending on the direction. Xilinx provides a library of macros that fall into each of these categories. In practice, custom macros may be created that combine different signal directions in a single macro.

Another variation emerged with the release of the Virtex-5, which is the single-slice bus macro. This is basically nothing more than a regular slice, which is specially designated as a bus macro and manually placed in the re-programmable region. The major difference with this approach is that the routes that cross the static/re-programmable boundary are no longer statically routed. The place and route program is free to choose any routes to cross the boundary so long as they terminate on the specially designated bus macro slice. This means that a dependence is created between the routing of the static and re-programmable region, namely that the reprogrammable region must be explicitly excluded from using the routes that cross the boundary. Because these routes are determined by the place and route program (not explicitly by the designer), the reprogrammable region must be synthesized with full knowledge of the routing used in the static region. Further, if the static region is re-synthesized for any reason, the interface at the module boundary could change as well, requiring that the reprogrammable modules be re-synthesized also.

In the design paradigm supported by Xilinx, those might not be very big concerns. The Xilinx tools are geared toward single, unified designs that may have some reprogrammable regions, but that all of the possible states of these regions are known at design time. It may well be that this is reasonable for many types of designs, and that therefore the area savings of single-slice macros is advantageous. In our case, however, we're interested in creating a generic hardware peripheral where the exact nature of the configuration of this peripheral is most decidedly not known at design time. Therefore, this extra coupling is problematic. In our case, custom bus macros using two slices were created for all static/reprogrammable region boundary crossings.

3.2 Modified Partial-Reconfiguration Toolchain

The Xilinx Early-Access flow requires the use of modified versions of the *map*, *par* and *ngdbuild* tools. The modifications are provided as a set of patches against Xilinx ISE 9.2 SP4.

3.2.1 Special Layout Constraints

The modified tool chain adds some extra options to the AREA_GROUP constraint, which is typically used to place logic in logical groups for floor planning. AREA_GROUP is used to define the boundaries of the reconfigurable region.

The *ngdbuild* program is a part of the standard synthesis flow. It takes synthesized netlists and repackages them in a different netlist type format which mostly exists to deal with various design entry styles. The reconfigurable region is implemented as a blackbox, which means that the *ngdbuild* step would expect a netlist for the reconfigurable region. Because we want to synthesize the static netlist only, however, the reconfigurable logic is desired to be unexpanded. The normal *ngdbuild* tool would throw an error here. The modified *ngdbuild* looks for a special UCF constraint, however, which is added to an AREA_GROUP constraint that defines the reconfigurable region. AREA_GROUP MODE=RECONFIG specifies that the *ngdbuild* program is allowed to leave this region unexpanded.

Another important AREA_GROUP option added in the patched tool chain is the AREA_GROUP ROUTING=CLOSED option. This specifies to the *par* tool that no routes may cross the region boundary. Without this option, routes from the static region may enter the reprogrammable region, again creating an undesirable close coupling between the static and reprogrammable regions placement and routing. With the ROUTING=CLOSED option, only bus macros may cross the region boundaries.

3.2.2 Partially Routed Nets

The modified tool chain also relaxes the design rules checks that are performed at several steps in the design. Typically, nets which are not fully routed at the end of place and route result in an error. In a partial reconfiguration design, however, nets going in and out of the reconfigurable region are not fully routed, since the static and reconfigurable regions are synthesized separately.

3.2.3 Tracking Routing Resources Across Regions

The place and route tool also produces a file called *arcs.exclude*, which lists all of the routing resources used in the static part of the design. This file is then renamed *static.used*, and given as input to the place and route tool while routing the reprogrammable regions. By using two-slice macros and the ROUTING=CLOSED constraint, this is effectively unnecessary, though the par tool still requires the file. This extra information is what makes single-slice macros, and static routes in the reprogrammable region possible.

3.2.4 Final Merge Step

The EAPR flow adds a final merge step into the process as well. The routed static and partially reconfigurable regions must be passed through a program called *PR_mergedesign*. This program produces as output a full bitstream, which contains the static design merged with a partial reconfiguration design. It also contains the partial reconfiguration bitstream. This tool is run once for each possible partial reconfiguration bitstream. This tool is a bit opaque in that exactly what manipulations it performs is not well documented. It might seem at first glance that simply taking the routed output of the reconfigurable region and running it through bitgen would be sufficient to create a partial bitstream. Running through *PR_mergedesign*, however, is required.

A close look at the inputs and outputs of the *PR_mergedesign* tool in the Xilinx FPGA Editor gives some suggestion at what this tool actually does. For starters, the routed partially reconfigurable module always includes some wires outside of the PR boundary. Clocking and reset logic, for example, is usually routed completely by the *par* tool. So, this extra wiring must be removed from the partial bitstream (since it is already present in the static logic), the *PR_mergedesign* pass takes care of this.

Chapter 4

Proposed Architecture

The purpose of the work done here is to propose a possible architecture for reconfigurable hardware co-processing. The idea is to provide a mechanism to allow integration between a microprocessor system and a reconfigurable hardware system. The goal is to allow certain portions of software processing that lend themselves to efficient hardware implementations to be exported on to a reconfigurable hardware peripheral, and, in doing so, free up the microprocessor to do other processing. The approach taken here combines two key ideas. First, to virtualize the FPGA resources allocated to the re-programmable peripheral by multiplexing across time, much in the way that a CPU is timeshared across processes. Second, to allow the re-programmable area to be used flexibly by allowing dynamic placement of hardware cores within the re-programmable region.

4.1 Partial Reprogramming Limitations

Accomplishing these goals requires some recognition of the facts of life of FPGA programming. Perhaps the most straightforward constraint is the fact that FPGA configuration bitstreams are large, and loading them onto a chip requires a finite and significant amount of time. On the Virtex-5 series of FPGAs the internal programming interface (called ICAP) accepts 32-bit data words at a maximum clock speed of 100MHz [19]. A full FPGA configuration bitstream might consist of 10 or more megabits of information. This means that the time requirements for reconfiguring even only part of a chip are in the 100s of microseconds to even a few milliseconds range, depending on the size of the reprogrammable region.

The order of the reprogramming time leads to some important observations as far as developing a practical system. One of the first is that a reprogrammable hardware co-processing peripheral will likely be of greatest utility on large blocks or streams of data. If reprogramming time dominates the operation of the reprogrammable hardware, then a software implementation could end up being faster, regardless of the efficiency of the hardware implementation.

By the same token, the overhead of using a reprogrammable peripheral can be reduced if the programming time itself can be reduced. While in a strict sense, this isn't really possible (the maximum speed of the programming interface is a hard limit) the effective programming time can be reduced by allowing a future hardware configuration to be programmed while the current one is still active [9, 5]. This can be accomplished most simply by having more than one peripheral area. One is used while the other is reprogrammed, and visa-versa, essentially allowing a "pre-fetch" of the next hardware core to be utilized. There is an undesirable aspect of this solution, however, which is that each of these reprogrammable areas must be sized to accommodate the largest expected core, and therefore much of the area allocated to the reprogrammable peripheral is wasted [8].

We choose a slight variation on this theme which creates a reprogrammable region which is designed to be larger than most cores that will be loaded into it. Cores are allowed to be placed at many different locations within this larger region, allowing simultaneous operation and reprogramming. As many modules as will fit may be packed into the region at a time, and one very large module may also be allowed to consume the entire area. This allows more flexibility in area use, while still allowing for "pre-fetching" in the average case. Cores may be located along any 5-CLB boundary, and may span *multiple* 5-CLB blocks. The point is to reduce the amount of space that is wasted by increasing the granularity, to allow for better packings.

4.2 Dynamic Core Placement Strategies

Our chosen strategy requires that the absolute placement of reprogrammable cores not be locked into place at design time. Instead, our design needs to be able to determine the placement of the core within the reprogrammable region at the time the core is loaded.

4.2.1 Bitstream Relocation

One possible strategy is simply to generate all possible permutations of cores that might be loaded into the system at one time. For a small number of reprogrammable components, such as in [14], this could be a viable strategy, and is the one advanced by the typical Xilinx partial reprogramming framework. In our case, however, we wish to support arbitrary loadable cores that were not known at design time. This means a different solution is necessary.

Fortunately, it turns out that FPGAs have a highly regular structure. As mentioned earlier, the entire device is divided into a grid of logic cells (CLBs and special purpose blocks) which are controlled by the programming bits in a configuration memory. This memory can simply be thought of as a 2-D overlay of bits on top of the grid of logic cells, and may be repositioned somewhat arbitrarily along an axis of symmetry. This technique, called *bitstream relocation*, may be performed by directly manipulating the programming bitstream sent to the FPGA. Relocation may be accomplished in linear time [2] and with minimal overhead. This is the strategy employed in this design.

Unfortunately, bitstream relocation is not a prefect solution, either, because even in spite of the highly-symmetric nature of the FPGA, this symmetry is not perfect. Designs may be easily translated in the vertical direction, since each column of logic in the FPGA has the same type of unit in each cell. In the horizontal direction, however, it is another story. Even though a majority of the columns contain CLBs, some contain XtremeDSP blocks, I/O blocks, block memory, and clock generation hardware. Worse, the FPGA targeted in this project (a Virtex-5 XC5VLX50) is not even symmetric across its center. This means that simple relocation is limited to vertical movements only.

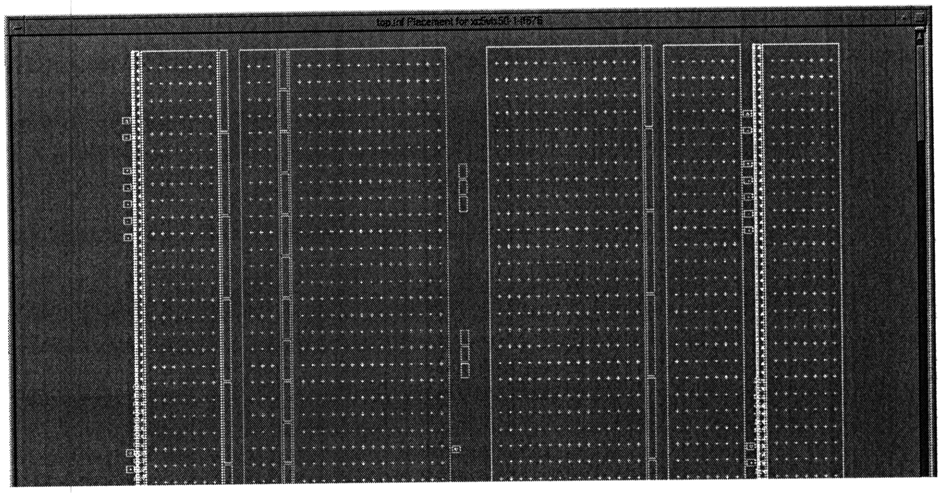


Figure 4-1: Image of the XC5VLX50 showing the lack of symmetry in the horizontal direction.

4.2.2 Reprogrammable Core Interconnect

Another issue that is created by the use of partial bitstream relocation is the problem of connecting the core to the rest (static portion) of the microprocessor system. As mentioned earlier, cross-boundary interconnect is accomplished through the use of bus macros. These macros, however, must be statically placed, and cannot simply be relocated along with the rest of the core, since the routing in the static portion of the design cannot be changed at runtime.

In [8] it is noted that allowing truly arbitrary placement "requires a very complex online communication synthesis between the modules because their location is not known until runtime." While the logical routing information for Xilinx FPGAs is available in a parseable form [15], the runtime of routing algorithms is not one that lends itself to on-the-fly manipulations of the FPGA contents with any sort of practical speed. Running the Xilinx *par* program typically takes several minutes for even a modest design.

In [7] a "block based" online routing strategy is proposed. Alongside each reprogrammable area is a vertical "routing channel," which occupies one (or more)

CLB columns and spans the total height of the reprogrammable region. Each CLB in the routing channel is filled with one of two types of programming - a "Type-I" macro, which simply passes signals through vertically, or a "Type-II" macro, which connects to a core in the reprogrammable area. Type-I macros fill the entire column, and a Type-II macro is placed at the correct point to provide a connection from the reprogrammable module into the routing channel.

This strategy abstracts the low-level routing into much simpler routing "blocks" that are easy to manipulate on the fly, which makes the design tenable. This simplicity comes at a cost, however. The main issues with this strategy are that only a small number of signals may be present in a CLB-wide routing channel. Each CLB has only 8 independent LUT outputs, so only 8 signals may traverse a single set of CLB LUTs. The other and perhaps more serious problem is that this strategy provides little guarantee of timing closure. The Xilinx Virtex-5 documentation advertises a propagation delay of 0.9ns through a single CLB [20]. A long chain of these delays will quickly preclude any modestly aggressive clock speed.

In this design we choose a different strategy. Recalling from section 2.3 that the unit of vertical symmetry is 5 CLBs tall (rather than one, as the problem was posed in [7]), much of the gain of "block based" online routing is brought into question. While it allows single-CLB granularity in the selection of a communications endpoint, the structure of the Virtex-5 dictates otherwise. Rather than implementing this complex scheme, a simple alternative is selected instead. Each 5-CLB-high rectangle of the reprogrammable region is given a set of statically-placed bus macros. In the static region, selection logic simply activates the appropriate set(s) based on the active contents of the re-programmable region. In addition to simplicity, this method alleviates the timing concerns of the "block based" routing method and does not take up any more area in the reprogrammable region - only a single column of CLBs is used. Another added advantage is that no logic reprogramming is necessary to switch between two cores that are already both resident in the reprogrammable region. Instead, the (static) selection logic simply selects the appropriate set of bus macros for the desired core, instead of having to reprogram the routing channel.

For some small number of bus macros (such as in the example implementation in chapter 6), macros may be directly connected to the memory-mapped bus architecture. For a larger number of macros, a more efficient reduction logic, such as a tree, may be desirable to meet area and timing targets.

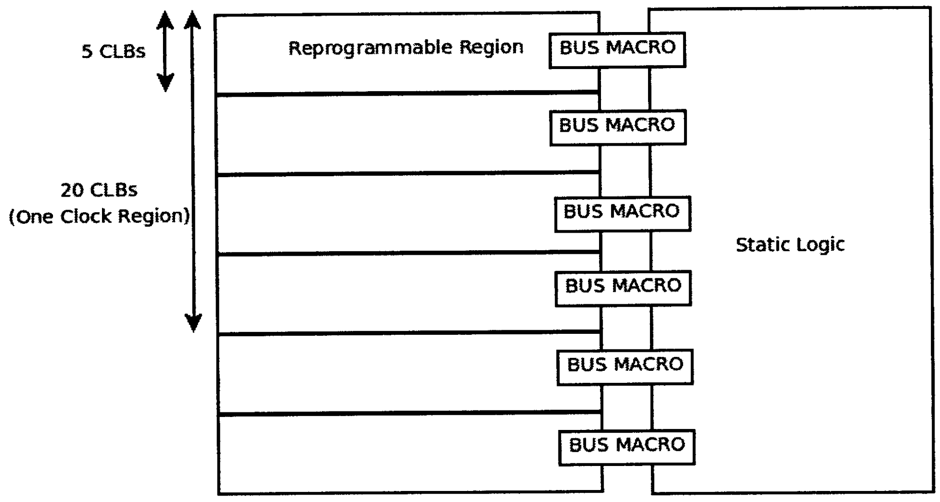


Figure 4-2: Proposed structure of reprogrammable region.

4.3 CPU to Reprogrammable Peripheral Communication

The system proposed here is larger than just the reprogrammable peripheral itself, but also includes a CPU and memory system which controls the programming of the peripheral, and additionally must be able to have a data path to and from the particular reprogrammable core loaded into the reconfigurable peripheral. Many examples of coupled CPU/FPGA architectures exist in the literature.

In [21], partial reconfiguration is used to time-multiplex the area dedicated to the arithmetic operations in the integer pipeline of an open-source LEON3 processor. A multiplier, divider, adder, etc., are loaded on demand into the same physical FPGA area when needed.

Reprogrammable hardware may also be coupled to a CPU as a co-processor. In [6], a reprogrammable hardware peripheral is coupled to a processor with a MIPS

ISA, and uses the co-processor instructions of the MIPS ISA to facilitate communication back and forth between the CPU and the reprogrammable peripheral. In [12], a similar model is used, employing the Xilinx Microblaze processor's *Fast Simplex Links* to do the communication, which function very much like the co-processor instructions on the MIPS. In both cases, the authors recognize a particular class of problem that their respective architectures are appropriate for. This style of tightly CPU-coupled interface essentially allows the creation of custom instructions such as floating-point and discrete transformation algorithms (DCT, FFT, etc). More generally, any frequent software loops that can be unrolled to execute in a very small number of compute cycles in custom hardware.

Finally, a third, more weakly-coupled approach is for the reprogrammable peripheral to simply be memory mapped into the CPU address space [16, 3], much like any traditional peripheral. While the memory-mapped bus used in a processor subsystem may be slower or have increased latency compared to a direct CPU coupling, this approach has its own advantages. For one, the CPU may be used unmodified, which provides for a much cleaner programming interface.

The major advantages become clear when we consider our contention from section 4.1 - namely that the class of problems that are likely to see the greatest benefit from hardware acceleration are those that require operation on large blocks or streams of data. It is precisely this class of problem where the overhead of loading a hardware core into the reprogrammable peripheral will be outweighed by the processing speedup. In this class of problem, data must come from an outside source - either some external interface, or from main memory. In either case, it will have to pass through the main system bus, which nullifies any performance gain seen from the tight coupling to the CPU.

Another issue with tight CPU coupling is that it necessitates that all data destined for the reprogrammable peripheral must be passed through the CPU. Since we've decided we're most interested in high-data-throughput situations, this means that the CPU will be tied up doing nothing more than shuttling data around for a considerable amount of time. By instead using the weakly-coupled, memory-mapped model, along with allowing the peripheral the ability to do DMA transfers, the CPU can instead work on another task while the reprogrammable peripheral is busy processing. It is this final interface model that is chosen for our system.

Chapter 5

Software Tools for Bitstream Manipulation

In order to further understand the Xilinx bitstream format, as well as to verify the correct functionality of partial bitstream generation, and finally to implement bitstream relocation, two new software tools were created. These tools, called *bitrender* and *bitparse* create a graphical representation of a Xilinx bitstream, and allow packet-level disassembly and manipulation of the bitstream file, respectively.

5.1 More Bitstream Particulars

The format of the Xilinx programming bitstream was discussed in some depth in section 2.5, but we'll take a moment here to address a few more details. Most of these details are available in Xilinx documents [19], though some sleuthing was required to make it all work.

5.1.1 Packet Formats

A Xilinx configuration bitstream is a collection of 32-bit words which make up a sequence of *packets*. Each packet has a one-word packet header, followed by some number of data words. There are two types of packets, descriptively named "Type

Packet Type	Opcode	Register Address	Reserved	Word Count
[31:29]	[28:27]	[26:13]	[12:11]	[10:0]
3'b001	2'bxx	14'bRRRRRRRRRxxxxx	2'bRR	11'bxxxxxxxxxxx

Table 5.1: Type I Packet Header. 'R' Indicates the bit is reserved.

Packet Type	Opcode	Word Count
[31:29]	[28:27]	[26:0]
3'b001	2'bxx	27'bxxxxxxxxxxxxxxxxxxxxxxxxxxx

Table 5.2: Type II Packet Header

I" and "Type 2" packets. Both types follow the same structure, but have different information in their headers. These differences are illustrated in table 5.1 and 5.2.

The most major difference is that a Type I word includes a destination register address, whereas a Type II word does not. A Type II word, however, has a much larger *Word Count* field, however, so Type II packets may be much longer than Type I packets. Because Type II packets do not have a destination register, all Type II packets must be preceded by a Type I packet with zero word count. Type I and Type II packets contain an *opcode* field, which specifies whether a transaction is a read, write or NOP operation.

5.1.2 Programming Registers

Also mentioned in section 2.5, there is a large collection of registers accessible on the programming interface, however a specific few of them are most relevant in understanding the basics of programming operation. Table 5.3 gives a listing of these key registers.

5.1.3 Frame Address Composition

Xilinx uses an interesting scheme for uniquely addressing configuration frames. The FPGA is divided into rows and columns. Columns refer to a single CLB or other logic cell. Rows, on the other hand, refer to a region that is the height of a clock

Register Name	Description
CMD	*Command Register.* Sets the current programming mode. An RCRC command may be written into this register to clear the current CRC value. A WCFG command may be written into this register to specify that the data written into the FDRI register should program the FPGA.
CRC	*CRC Register.* Contains the current CRC value.
FAR	*Frame Address Register.* Contains the address of the current frame. Increments automatically as data is loaded into FDRI.
FDRI	*Frame Data Input Register.* Configuration frames get written into this register.

Table 5.3: Important Programming Registers

Block Type	Top	Row	Major (Column)	Minor (Frame)
[23:21]	[20]	[19:15]	[14:7]	[6:0]

Table 5.4: Frame Address Composition

region (a stack of 20 CLBs with clock distribution in the middle). The XC5VLX50 FPGA used in this project has six such rows. Rows are numbered beginning with 0 from the center of the chip - the rows just above and below the equator line are both row 0. A frame address also contains a *top* bit, which indicates if the row in question is above or below the center line.

The frame address also contains a *major address*, which specifies the CLB-sized column that is being addressed. The frame address also contains a *minor address* which specifies the particular frame being addressed. Finally, the frame address contains a *block type* field, which specifies that the addressed block is either a standard CLB/logic configuration block, or a block RAM contents type, which is the initial contents of the on-chip block memory. This type field gives block memory a completely seperate addresses space from the rest of the configuration memory. This makes it easy to reconfigure logic and block memory contents seperately. Table 5.4

shows the specification of a FAR word.

5.2 Bitparse Tool

The *bitparse* tool is a C program that reads a Xilinx configuration bitstream from a file, and constructs an abstract representation of it in memory as a linked list of packet data structures. It also contains code to print a human-readable summary of a bitstream file. Once the abstract representation of the bitstream has been created, the bitstream may be easily manipulated. Finally, the program can recalculate the bitstream CRC word, and write the manipulated bitstream out to a new bitstream file for configuring an FPGA.

5.2.1 Input File Parsing

A bitstream file, which is the result of running the Xilinx *bitgen* tool contains the complete programming bitstream, but also a proprietary header with undocumented contents. Fortunately, the bitstream begins with a known synchronization word, 0xAA995566. The parser searches for this 32-bit sequence, and then parses each 32-bit word that follows allocating packet data structures as it goes along. The definition of this data structure is given in figure 5-1. The parser builds a linked-list of packet structures that comprise the entire bitstream, and stops when it reaches the end of the file.

```
struct packet {
        uint32_t type;
        uint32_t opcode;
        uint32_t word_count;
        uint32_t reg_addr;
        uint32_t *data;
};
```

Figure 5-1: Packet Data Structure

5.2.2 Human-Readable Summary Generation

Once the stream has been constructed using the internal representation, a textual summary of the programming commands can be generated. A partial summary output of a typical bitstream is given in figure 5-2.

```
 1   Type 1  WRITE   Word Count 1    Reg Addr 0x10 (WBSTAR)
 2   Type 1  WRITE   Word Count 1    Reg Addr 0x04 (CMD)
 3           Wrote NULL to command register
 4   Type 1  WRITE   Word Count 1    Reg Addr 0x04 (CMD)
 5           Wrote RCRC to command register
 6   ...

17   Type 1  WRITE   Word Count 1    Reg Addr 0x01 (FAR)
18           btype: 0x00     top: 0  row: 0  col: 0  minor: 0
19   Type 1  WRITE   Word Count 1    Reg Addr 0x04 (CMD)
20           Wrote WCFG to command register
21   Type 1  WRITE   Word Count 0    Reg Addr 0x02 (FDRI)
22   Type 2  WRITE   Word Count 392124
23   Type 1  WRITE   Word Count 1    Reg Addr 0x00 (CRC)
24           CRC word: 0xf1bf8517
25   ...
```

Figure 5-2: Bitparse Human-Readable Summary

This output gives a very nice example of the typical programming procedure. First, we see on line 4 that an RCRC command is written to the CMD register, clearing the current CRC value. Next, line 17 sets the FAR register to start at address 0, and line 19 writes a WCFG command to the CMD register, preparing the FPGA for configuration bits. Next, a zero-length Type I write to the FDRI register, followed by a Type II write of length 392,124 words writes the configuration data into the FPGA. Finally, the CRC value from the bitstream file is written into the FPGA, for comparison against the CRC value calculated inside of the FPGA. If they match, the programming bits will be accepted, and the FPGA can start up.

Another interesting (and comforting) observation comes from the number of configuration words programmed, 392,124. Recall from section 2.5 that each configuration frame contains 41 32-bit words. When we do the division, $\frac{392{,}124}{41} = 9,564$, which is exactly the number of configuration frames on our XC5VLX50 FPGA, according to page 113 in [19].

5.2.3 CRC Calculation

Xilinx Virtex-5 bitstreams are protected by a 32-bit CRC checksum to ensure that no data corruption takes place during programming, which might result in undesired operation. When we use the *bitparse* tool to manipulate a programming bitstream, the original CRC no longer is correct, and we must calculate a new one. Doing this, of course, requires some knowledge about how to calculate the CRC.

The CRC algorithm used is widely cited [13] to be the CRC32C or *Castagnoli* algorithm, which is a popular and well-documented CRC algorithm. Unfortunately, its implementation on the Virtex-4 and Virtex-5 series FPGAs is not publically documented by Xilinx. So, we are left knowing the CRC algorithm used, but not how to construct the bitstream that it operates on.

Fortunately, Xilinx did release implementation details for the CRC calculation on a much earlier FPGA, the Virtex series [17]. The key insight from this document is captured in figure 5-3. The item to note is that the CRC is not computed on the bitstream verbatim, but rather only on the bits that are written to FPGA registers, concatenated with the destination register address. On the Virtex-5, these registers have 5 bits of address instead of 4, so the sequence of bits that must be run through the CRC calculator are groups of 37 bits, one for each 32-bit word written to a programming register. The figure also provides the direction of operation (least significant bit first). The 16-bit CRC algorithm used in figure 5-3 was replaced with CRC32C, and the CRC calculation was easily implemented in C.

5.2.4 Bitstream Relocation

Simple bitstream relocation was demonstrated using the *bitparse* tool by defining two target reprogrammable regions each of which consisted of a single clock region. A partial bitstream (which only programs a small region of the FPGA, rather than the

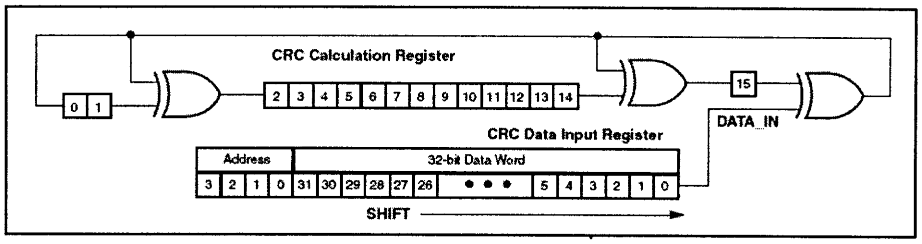


Figure 5-3: CRC Calculation Logic for a Xilinx Virtex FPGA.

whole thing) was created targetting one of the regions. A simple *bitparse* program was created to translate the FAR register writes from the old to the new target location. Finally, a new CRC value was calculated, and the result output to a new bitstream file.

Operating on areas smaller than a clock region is possible within this framework as well, though with some additional complexity. Since a single frame spans the height of a clock region, the parts of a frame that are not slated to be overwritten by the relocated partial bitstream must be written back as well. This means they must either be fetched from a copy of the existing FPGA configuration memory image, or read back from the configuration memory directly prior to reprogramming.

5.3 Bitrender Tool

Bitrender is a software program implemented in C that converts a Xilinx programming bitstream into a viewable PNG file that provides a visual mapping of an FPGA configuration image. This capability is useful for several purposes - chief among them verifying (perhaps to some degree de-mystifying) the rather opaque operation of the *PR_mergedesign* tool in the Xilinx EAPR tool flow which generates partial bitstreams. Additionally, the tool is useful for verifying correct bitstream relocation, as well as assisting in a general understanding of the Xilinx programming format and procedure.

5.3.1 Program Operation

Bitrender uses the same parser as the *bitparse* tool to parse an input bitstream file. Rather than create an internal representation using the data structure give in 5-1, a PNG image is created using the *GD* graphics library. Each configuration bit (excluding BRAM contents) on the FPGA is mapped to a single pixel on the PNG image. A pixel is colored dark blue if it is programmed a '1', gray if it is programmed a '0', and is left white if it is not programmed at all. White lines seperate each logic cell in the grid that covers the FPGA.

Because the FAR register is only initialized a few times in a programming bitstream, the *bitrender* program must keep track of the current value of the FAR internally, by automatically incrementing it with each frame read out from the programming bitstream. By using the FAR format given in 5.4 and the minor number address counts given in 2.1, as well as knowledge about the number of rows, and the type of logic cell in each major column for the particular FPGA, it is fairly straightforward to increment the FAR properly. Then a mapping from FAR to pixel position is used to set appropriate pixel values.

5.3.2 Full Bitstream Results

For our first test, we take a look at the results of running *bitrender* on a typical programming bitstream. We compare the output of the bitrender program to the graphical rendering of the design in the Xilinx FPGA Editor. It is important to note that the two graphical tools are not really performing the same function. The Xilinx FPGA Editor operates on a Xilinx native circuit description, or NCD, file. This file is the direct output of the *par* tool. The *bitrender* tool operates on a Xilinx bitstream file containing an actual programming bitstream. This is the output of running the *bitgen* tool. The *bitgen* tool reads an NCD file to produce a programming bitstream. While the format of a programming bitstream is known, an NCD file is proprietary and its details unavailable.

Comparing figures 5-4a and 5-4b convinces us of the correctness of the *bitrender*

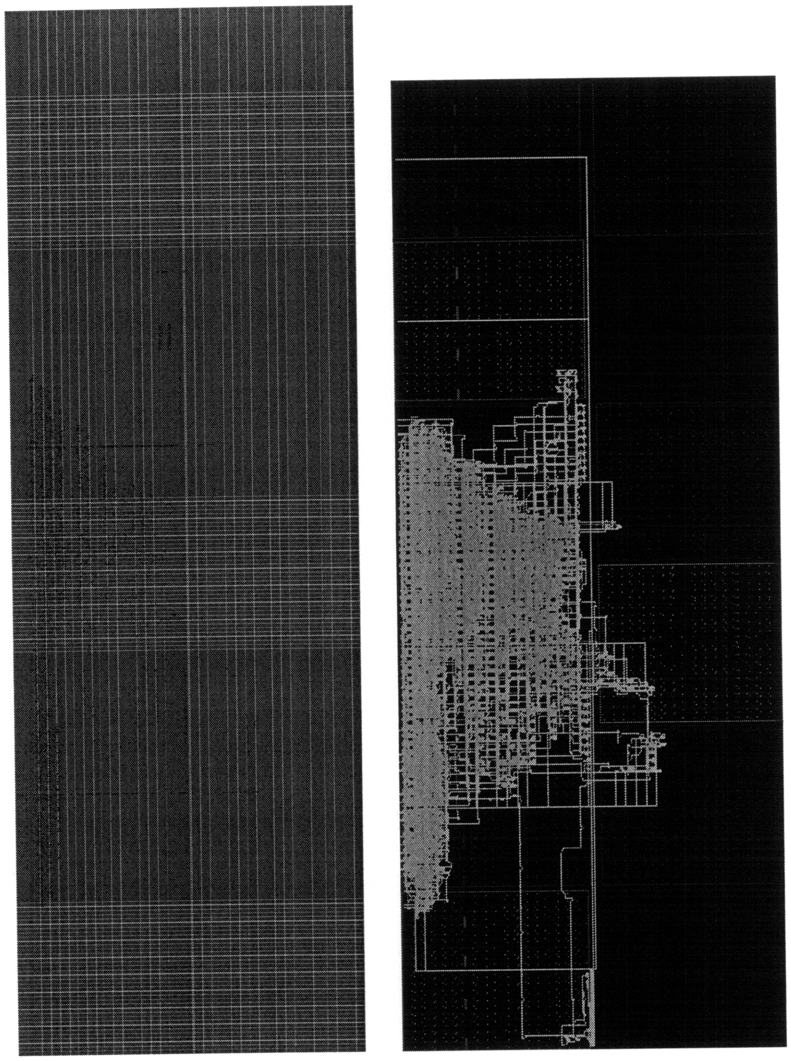

(a) Bitrender output (b) Xilinx FPGA Editor output

Figure 5-4: Comparing the FPGA Editor view and *bitrender* output.

tool. While the outputs are different in some respects (the FPGA Editor attempts to render wiring, instead of just programming bits), it is clear that there is a high correlation between the pixels in figure 5-4a which represent programming bits, and the graphical rendering of FPGA pieces in figure 5-4b.

Figure 5-5 shows a zoomed-in view of the output of bitrender. Individual logic cells are visible, seperated by white lines. The grid row across the center is the middle of a clock region. These cells correspond to the 10 horizontal clock lines that run across each clock region. Two rows of CLBs are above and below this center row. The CLB columns which contain clocked logic each has a bit set which enables the clock driver

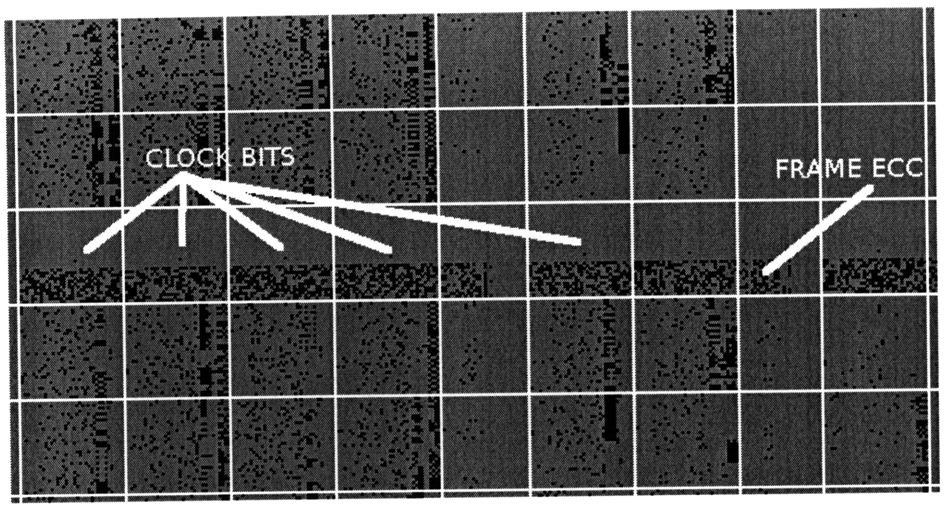


Figure 5-5: Closeup view of *bitrender* output.

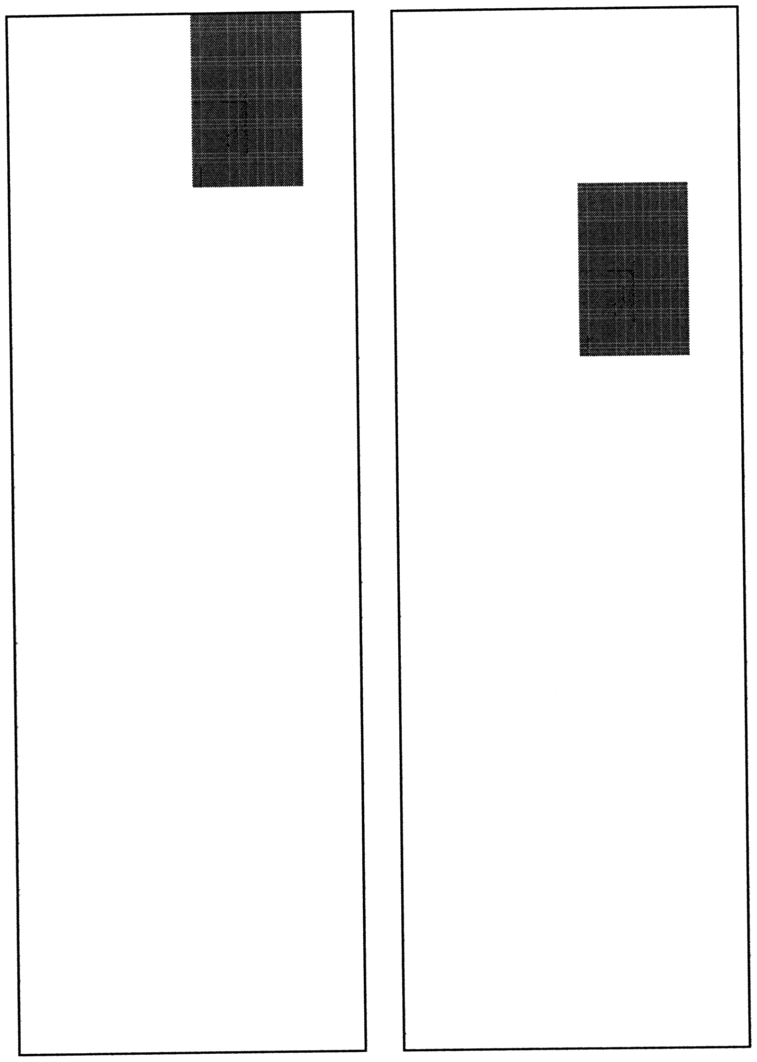
for that column.

Another interesting item of note is the curious noisy stripe which goes horizontally through these clock cells, just below the clock bits. These bits are 12 bits of hamming error correction, designed to allow the detection single-bit errors in a single programming frames. In normal use, these are largely ignored, however they are often employed for configuration verification in space applications, where high levels of radiation are present [18].

5.3.3 Partial Bitstream Results

While some interesting properties are brought out nicely by *bitrender* when run on standard bitstreams, the case where it is most useful is in analyzing partial bitstreams. One reason for this is that bitstream relocation, discussed at length in 4.2.1, is performed directly on the bitstream that is generated by *bitgen*. There is no way to view such manipulations in the FPGA Editor.

Figure 5-6 shows a *bitrender* view of a small partial bitstream. Most of the area of the FPGA is displayed as white, because the bitstream does not program it at all. The bitstream shown in figure 5-6a was relocated vertically by one clock region through the use of the *bitparse*. The result is shown in figure 5-6b, verifying that the relocation was completed successfully.



(a) Partial Bitstream (b) Relocated Partial Bitstream

Figure 5-6: Relocation of a partial bitstream.

5.3.4 Examination of ***PR_mergedesign*** Behavior

As mentioned in section 3.2.4, the *PR_mergedesign* tool is part of the Xilinx EAPR design flow which is run prior to the conversion of NCD files into bitstream files with the *bitgen* tool. It takes as input a static design NCD file (the part that doesn't change) as well as an NCD file for a reprogrammable region. It merges these two files into a single unified NCD file, reconciling global logic, such as clocks, that is used by both the static and reprogrammable regions. It then generates two new NCD files, one corresponding to the static design, and another corresponding to the reprogrammable part of the design, to be run through *bitgen*.

An odd observation, however, is that both of the new NCD files look identical in the FPGA Editor. They both appear to contain the fully-merged design with both static and dynamic regions. Yet running *bitrender* on the resulting bitstream files reveals that, despite appearances in the FPGA Editor, the resulting bitstreams only contain the respective static or dynamic logic.

This suggests that the modified EAPR tools somehow "tag" logic elements that are to be converted into the final bitstream, within the proprietary NCD file format. While this is not documented explicitly, the output of the *PR_mergedesign* tool does reference "tagging" in its textual output. The *bitrender* tool allows us to easily gain access to the result of this tagging, something which is not at all visible within the FPGA Editor view.

Chapter 6

Partial System Implementation

This chapter describes a partial realization of the system described thus far. The implemented system consists of a small CPU implementing the MIPS ISA, a memory-mapped bus architecture, 1MB of off-chip static memory and an RS232 UART. In addition to this static logic, the system also contains a reprogrammable peripheral which is mapped into the shared address space. Finally, a programming controller, also mapped into the address space, allows hardware configuration bitstreams to be dynamically loaded into the reprogrammable peripheral.

Most of the implementation was done in Verilog, with the exception of the open-source MIPS core, which is primarily in VHDL. All parts were synthesized with Xilinx ISE 9.2.04 with EAPR patchset 11.

Finally, this chapter reviews the performance of this partial implementation and discusses key improvements that must be made in order for it to be practical.

6.1 Static Logic

The static logic portion of the design consists of a basic microprocessor system. The system contains a CPU, memory, and some peripherals, all joined together with a memory-mapped bus architecture. The CPU is an open-source MIPS implementation, called PlasmaCPU. To interconnect all of the pieces, the Avalon Bus Architecture, developed by Altera, was used.

6.1.1 Avalon Bus Architecture

The Avalon Bus Architecture is a system for connected memory-mapped devices together in a scalable and systematic manner. It was developed by Altera for use with their microprocessor system, the NIOS, but is perfectly usable in other applications, such as ours, with no Altera-specific IP involved (in fact, we use it here on a Xilinx FPGA, Altera's major competitor). While in the abstract it seems appropriate to call the Avalon a "bus," this is not completely accurate. Tri-states do not exist within our FPGA, so the "bus" is actually a "switch fabric," a group of combinational logic that routes data from the correct source to the correct destination.

There are two types of Avalon interfaces - master and slave. Avalon masters may initiate transactions on the bus. Slaves respond to requests initiated by masters. When appropriate, a single peripheral may have both master and slave interfaces. Figure 6-1 shows a sample Avalon slave interface.

```
// required signals
input [Mem_data_width-1:0] avs_writedata,
output [Mem_data_width-1:0] avs_readdata,
input avs_write,
input avs_read,

// optional signals
input [Mem_byte_count-1:0] avs_byteenable,
input [Mem_addr_width-1:0] avs_address,
output avs_readdatavalid,
output avs_waitrequest
```

Figure 6-1: Sample Avalon Slave Interface Verilog Port List

When a peripheral is selected to perform a read, avs_read is asserted and the avs_address is driven. Some time later, the read data is driven onto the avs_readdata lines and the avs_readdatavalid signal is asserted. A write takes place in a similar manner - avs_write is asserted along with avs_address, (optionally) avs_byteenable, and avs_writedata. The avs_address line is an address relative to the base address of the peripheral, and is a *word* address (i.e. if the data lines are 32-bits wide, each increment of avs_address corresponds to 4 bytes).

If either a read or write is requested and the slave peripheral is not ready to accept the request, the peripheral may apply backpressure to the switch fabric by asserting the avs_waitrequest signal. This essentially freezes the transaction until avs_waitrequest is deasserted, and then the transaction continues. If a peripheral will always be ready to process a request, the avs_waitrequest signal may be omitted.

If read latency on slave devices is fixed at design time, avs_readdatavalid may be omitted and the fixed latency specified during the generation of the switch fabric. The avs_readdatavalid signals exists when the pipeline latency (the number of cycles between accepting a read request, and when the read data is returned) of a peripheral cannot be specified at design time. This is the case in our reprogrammable peripheral because we wish to allow core designers the flexibility to adjust this number for their core.

The Avalon master interface is similar, as seen in figure 6-2. Most of the control lines simply go in the opposite direction, but work in basically the same manner. One key difference is that the avm_address is a byte address regardless of the width of the data path, and that it is an absolute address, since the master must obviously be able to access the entire address space.

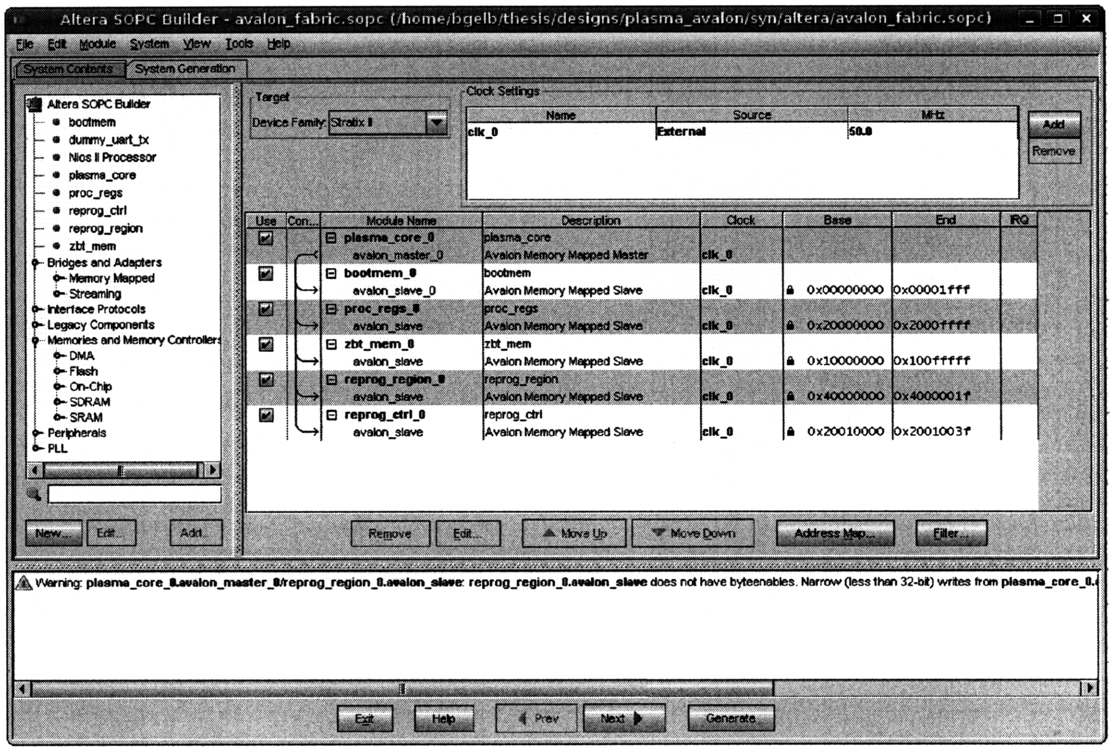
```
output [31:0] avm_address,
input avm_waitrequest,
output avm_read,
input [31:0] avm_readdata,
output avm_write,
output [31:0] avm_writedata,
output [3:0] avm_byteenable
```

Figure 6-2: Sample Avalon Master Interface Verilog Port List

Once the peripherals have been defined, the Altera SOPC Builder application is used to create the switch fabric. Each peripheral is imported into the application, various options are set for the desired signal types and widths, read latency may be specified, etc.. The main view of the SOPC builder GUI is shown in figure 6-3. Each Avalon peripheral is connected to a bus, and assigned address ranges. Once the Avalon fabric has been configured, pressing the *generate* button causes a Verilog file containing the switch fabric to be generated, with ports for connecting to each of the modules.

The generated module takes care of all arbitration (in the case of multiple masters), as well as dealing with the appropriate pipeline delays from a particular peripheral, and doing any byte-rerouting that is necessary in the case where a master and slave peripheral have different sized data ports.

In our configuration, a single clock is used for all of the peripherals and generated routing and arbitration logic (an additional required signal, not listed in figures 6-1 and 6-2).



Target
Device Family: Stratix II

Clock Settings

Name	Source	MHz
clk_0	External	50.0

Use	Con...	Module Name	Description	Clock	Base	End	IRQ
✓		plasma_core_0	plasma_core				
		avalon_master_0	Avalon Memory Mapped Master	clk_0			
✓		bootmem_0	bootmem				
		avalon_slave_0	Avalon Memory Mapped Slave	clk_0	0x00000000	0x00001fff	
✓		proc_regs_0	proc_regs				
		avalon_slave	Avalon Memory Mapped Slave	clk_0	0x20000000	0x2000ffff	
✓		zbt_mem_0	zbt_mem				
		avalon_slave	Avalon Memory Mapped Slave	clk_0	0x10000000	0x100fffff	
✓		reprog_region_0	reprog_region				
		avalon_slave	Avalon Memory Mapped Slave	clk_0	0x40000000	0x4000001f	
✓		reprog_ctrl_0	reprog_ctrl				
		avalon_slave	Avalon Memory Mapped Slave	clk_0	0x20010000	0x2001003f	

```
Warning: plasma_core_0.avalon_master_0/reprog_region_0.avalon_slave: reprog_region_0.avalon_slave does not have byteenables. Narrow (less than 32-bit) writes from plasma_core_0.
```

Figure 6-3: SOPC Builder GUI

The Avalon Specification also supports a variety of advanced features, such as pipelined and burst transfers, though they are not used in our simple system. Full details may be found in [1].

6.1.2 PlasmaCPU MIPS Core

The PlasmaCPU package is an open source implementation of a 3-stage pipelined MIPS processor. The package also includes some implementations of other hardware peripherals (DRAM controller, Ethernet controller, etc), though these were not used for this project. The basic MIPS CPU was used, and modified to provide an Avalon master interface, as described earlier.

The processor contains an interrupt status and interrupt mask register, which are used to monitor and control processor interrupts. Interrupt conditions are generated by the UART (ready to TX, or new RX data available), as well as a system timer. Additionally, the processor contains a counter register which is incremented every clock cycle. All of these registers are connected to a common Avalon slave interface, so they are mapped into the processor's address space. A 8KB of block memory is utilized to store a boot loader. This memory is placed at the very beginning of the address space.

The unmodified MIPS core was able to readily function at 50MHz clock speed, so this speed was used for the project.

6.1.3 RS232 UART

A simple RS232 UART functioning at a fixed speed of 115,200 baud was created to provide an interface to a computer. Data could be sent to the MIPS core over this link, and the MIPS core could return output data over this link as well. The UART provides two interrupt signals (mentioned earlier), one to indicate when the TX UART is ready to accept data for transmission, and another to indicate that the RX UART has ready received which is ready to be read by the processor.

The UART provides an Avalon slave interface, and is memory-mapped onto the system bus as a single 8-bit register, which is used for both TX and RX data.

6.1.4 Main Memory

An off-chip ISSI IS61NLP25636A SRAM was used to provide 1MB of main memory. This memory device is a zero bus turnaround (ZBT) memory, which means that it can accept a new memory transaction on every cycle. The timing ensures that interleaved reads and writes will never contend for the data bus during the same cycle.

A simple adapter module was created that interfaces the ZBT memory module to the Avalon bus. The design is pipelined, so it may accept a new transaction every cycle. Total pipeline latency for a read operation is 4 cycles.

6.2 Reprogrammable Peripheral

The reprogrammable peripheral is made up of a rectangular area of the FPGA which is not used for the static system. Using bus macros, a data and control path in and out of this area is provided. Arbitrary logic may then be placed in this area, and, providing it uses this data interface correctly, may communicate with the microprocessor system.

6.2.1 Peripheral Interface

In creating the peripheral interface, concern must be given not only to the makeup of the signals that comprise the interface, but also to the construction of the required bus macros to create a bridge between the static and reprogrammable regions. The interface must be mapped into a set of bus macros that will be placed at the region boundary.

As discussed in section 4.2.2, a bus macro in our setting is comprised of two sets of logic slices in adjacent columns, with one half of the set on each side of a dynamic/static boundary. Hand-routed nets cross the boundary and join the two slices together, and the slices are configured to simply to pass through any incoming signals to the other side of the boundary. These macros are locked into place, and the dynamic and static logic is placed and routed accordingly.

One key limitation in this type of bus macro is the number of signals that a single

Signal Type	Bit Width
Output Data Path	8 bits
Input Data Path	8 bits
Address	5 bits
Read Request	1 bit
Read Data Valid	1 bit
Write Request	1 bit
Total	24 bits

Table 6.1: Reprogrammable Peripheral Interconnect Interface

macro may carry across the boundary. Each slice has four independent outputs. With two slices per CLB, this means that a maximum of 8 signals may pass through a single bus macro. Keeping this in mind, it seems important that boundary crossing signals are kept to a minimum, so that an excessive amount of chip real estate is not consumed by the bus macros. As such, we choose to limit the width of the data path in our example system to 8 bits. Table 6.1 gives a full listing of the signals. This set of 24 wires packs nicely into a set of bus macros that is 3 CLBs tall. Note that no explicit reset signal is provided, however, a peripheral may be reset by writing a value to a specific register, for example. The exact discipline of any reset logic is left up to the designer of a reprogrammable core.

6.2.2 Bus Macro Creation

The bus macros used in the peripheral interface were created by hand in the Xilinx FPGA editor. Creating bus macros essentially consists of the following steps:

1. Identifying the slices to be used in the macro, and setting one of them as the reference component used for reference during placement.

2. Configuring the LUTs to route signal straight through from some input to each of the outputs in the slices.

3. Assigning each pin to a net, and assigning the inputs and outputs of the bus macro to external pins.

4. Routing the boundary crossing nets.

Once this is done, the bus macro is finished, and is ready to be used. Because bus macros are do not have any accompanying HDL code, as they are directly implemented as routed logic, a “blackbox” description of the module must be created for the purpose of instantiation within an HDL design. This basically consists of a Verilog module with a port list matching that of the bus macro, but with no logic inside. Figure 6-4 gives the Verilog blackbox description for one of the bus macros used in the peripheral interface. Note that the logic given in lines 10-14 is used *only* for simulation, where the hard-routed macro is not used, as indicated to the synthesis tools by the translate_off directive surrounding it. In synthesis, these assignments are omitted, and are instead provided by the hard-routed bus macro.

One thing to note is that the set of bus macros for our reprogrammable region do not include a clock signal, even though they are interfaced to the clock-synchronous Avalon architecture. Clocks are distributed by the clock tree described in section 2.2, which is not broken across the static/reprogrammable boundary. This means that clocks need not be put through a bus macro to get across the boundary, but rather will be naturally available in the reprogrammable region.

In order to make the bus macros work properly with the EAPR toolchain and the ROUTING=CLOSED constraint discussed in section 3.2.1, one special final step is needed, which is not prominently documented by Xilinx. The routed bus macro must be run through the *PR_mergedesign* tool, which applies special “tags” to the boundary-crossing nets, exempting them from the ROUTING=CLOSED setting. A sample output of this final step is given in figure 6-6.

6.2.3 Peripheral Location

Section 4.2 discusses some of the concerns which must be taken into account when choosing a region for partial reconfiguration. Because we are interested in supporting dynamic placement by way of bitstream relocation, it is important to choose an area that not only has the desired types of hardware resources, but also one that allow for

```
module avs_control
(
  input [4:0] avs_addr_ext,
  output [4:0] avs_addr_int,
  input avs_read_ext,
  output avs_read_int,
  input avs_write_ext,
  output avs_write_int
);

// synthesis translate_off
assign avs_addr_int = avs_addr_ext;
assign avs_read_int = avs_read_ext;
assign avs_write_int = avs_write_ext;
// synthesis translate_on

endmodule
```

Figure 6-4: Blackbox Description of Bus Macro

easy relocation. This means that there must be an axis that a core may be translated along while maintaining the same types of hardware resources.

Additionally, it is advantageous (though not required) for a partially reprogrammable area to have its boundaries located along clock region boundaries. This is because clock regions represent the minimum unit of atomically reprogrammable area in the vertical direction. Not falling on this boundary requires reprogramming of bits that actually lie in the static region, which means that a significant portion of reprogramming time is necessarily wasted, regardless of placement of reprogrammable cores.

With these concerns in mind, the area selected for the reprogrammable region consists of the uppermost two clock regions on the left half of the FPGA. The left side was chosen over the right because, on the XC5VLX50, it has the greatest diversity of available logic resources. The left side has a column of block memory, as well as a column of XtremeDSP units. The right side has two columns of block memory, but no XtremeDSP units. Figure 6-7 shows the location of the reprogrammable region.

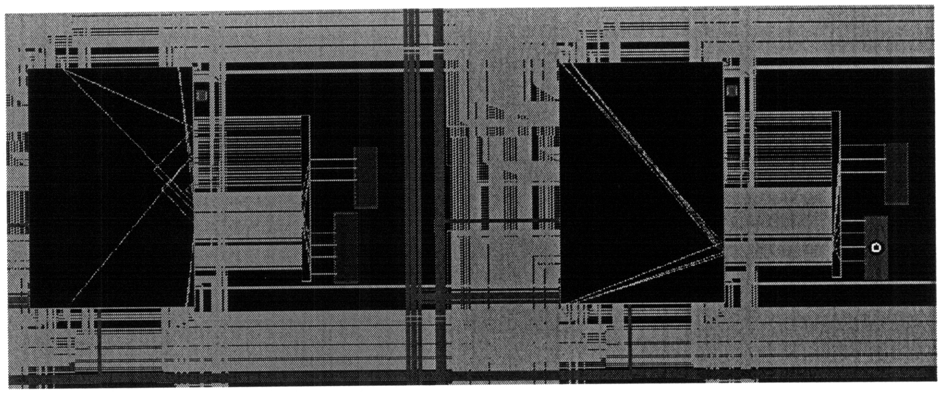


Figure 6-5: One of Three Bus Macros Used for the Reprogrammable Peripheral Interface.

6.2.4 Programming Controller

In order to enable the reprogrammable peripheral to be of any practical use, it must be possible to reprogram it from within a program running on the microprocessor. This means that there must be an additional hardware peripheral in the processor system which connects to the programming interface of the FPGA. The Xilinx Virtex FPGAs provide a mechanism for this called the Internal Configuration Access Port (ICAP), which is located inside of the FPGA fabric and may be instantiated as a primitive logic element within an HDL design.

The ICAP port has an interface similar to an external interface called SelectMAP. SelectMAP can be configured as a 8, 16 or 32-bits wide bi-directional data port, along with several control signals that determine the direction of the data port, as well the the programming mode of the FPGA. As the port is bi-directional, it may be used to both configure the FPGA, as well as readback its current configuration.

The ICAP port is very similar to the SelectMAP interface, except that it has two unidirectional data ports, one for read data, and one for write data. This makes sense, because tri-state signals are not supported within the Virtex-5 fabric. The ICAP port may also be either 8, 16, or 32 bits wide. We choose 32-bits, in order to achieve maximum throughput.

The programming controller instantiates the Virtex-5 ICAP port, and interfaces it with the Avalon bus as a slave peripheral. Two complications in the ICAP specifi-

```
=======================================================
== Convert Macro to Bus Macro for PR Flow.          ==
=======================================================
The process assumes that all routed nets in the design are networks that will be
used to cross the partial reconfiguration area boundaries.  It also assumes that
all logic in the macro definition is bus macro logic.  If either of these
assumptions are not true of this macro then the behavior of the tools will be
unknown with respect to this macro interface.

Analyzing Networks.
   Tagging avs_read_inter.
   Tagging avs_write_inter.
   Tagging avs_addr_inter<4>.
   Tagging avs_addr_inter<0>.
   Tagging avs_addr_inter<1>.
   Tagging avs_addr_inter<2>.
   Tagging avs_addr_inter<3>.
Done with Networks.

Analyzing Logic.
   Tagging ext1 with IS_BUS_MACRO.
   Tagging ext0 with IS_BUS_MACRO.
   Tagging int1 with IS_BUS_MACRO.
   Tagging int0 with IS_BUS_MACRO.
Done with Logic.
```

Figure 6-6: Output from *PR_mergedesign* run on bus macro.

cations make this slightly less than trivial.

The ICAP includes a chip enable signal (CE), as well as a signal to select either a read or write transaction (WRITE). The ICAP port ignores input if the CE line is deasserted. The issue is that the level of the WRITE line may not be changed with CE asserted. Doing this violates the ICAP specification, and results in a programming abort on the FPGA. Because of this a small FSM is employed to facilitate switching from read to write mode or the opposite. It deasserts CE on one cycle, changes the value of WRITE on the following cycle, and then reasserts CE as appropriate to perform the desired data transfers.

The other small issue is that, while programming words are given big-endian as output from the Xilinx *bitgen* tool, the SelectMAP and ICAP interfaces expect data

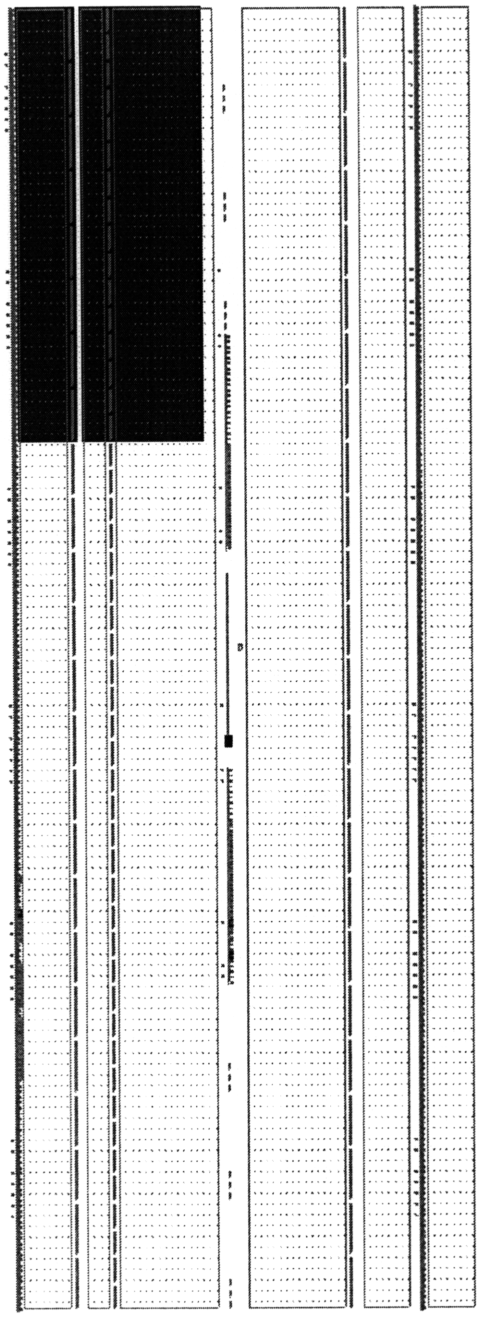


Figure 6-7: Floorplan Showing Location of Reprogrammable Region

to be "bit-swapped" [19], which essentially amounts to little-endian ordering. This means that for each byte within a 32-bit programming word, the ordering of the bits is reversed.

With these two small issues addressed, our programming controller is interfaced to the Avalon bus, and bitstreams may now be loaded from the microprocessor.

6.3 Software Framework

A simple software framework was developed in order to facilitate basic testing of the microprocessor and reprogrammable peripheral system. A simple bootloader was created to start the system, and accept some basic commands over the serial port to allow loading and starting larger programs. Because the system uses a MIPS core, a standard *gcc* toolchain is used to compile and link program binaries.

6.3.1 Bootloader

In order to allow for the evaluation of the implemented system, a mechanism was needed to load and run software programs. Because the system does not really use any permanent storage beyond a few KB of internal block memory, an interface to the outside world was needed. Accordingly, a simple bootloader program (stored in block memory) was developed to allow the loading of larger programs onto the processor system through a serial link.

At startup, the bootloader prints the following message:

```
Hello its bootloader...

READY>
```

The READY output indicates that the bootloader is running and ready to process commands. The bootloader accepts three types of commands, given in tables 6.2, 6.3, and 6.4. The first two commands together allow the loading of a binary image into memory, and then jumping into it. When the program terminates, the thread of control returns to the bootloader and so another program may be run. The print command allows direct address to main memory and all memory-mapped peripherals, which is helpful for debugging.

1 bytes	4 bytes	4 bytes	n bytes
0x01 (load)	load address	load length	data

Table 6.2: Bootloader Load Command

1 bytes	4 bytes
0x02 (jump)	jump address

Table 6.3: Bootloader Jump Command

1 bytes	4 bytes
0x03 (print)	data address

Table 6.4: Bootloader Print Command

6.3.2 Host ELF Loader

A utility was implemented for the host computer was well, in order to facilitate the easy loading of binaries onto the microprocessor system. It takes a MIPS ELF image as input, and generates the necessary bootloader commands to load it into memory and jump into it. This program parses the ELF file format to determine the correct link address and entry point - which may shift around from compile to compile - which greatly simplifies running binaries on the test system.

6.4 Loadable Peripheral Design Considerations

A loadable peripheral core must use the interface specified earlier in this chapter in order to work properly in the system. It provides 32 one-byte memory-mapped locations for use by the peripheral core. How these are used are ultimately up to the designer, but some factors to consider when designing are listed here.

First, if the peripheral core is to be swapped in and out of physical hardware, the state may need to be preserved across these swaps. If so, a mechanism should be provided to easily load and unload state. One possibility is to assemble state elements into a shift-chain, which can be read serially out of or into one of the register locations.

Further, a designer may want to make some status information available in one of the registers which indicates whether or not it is safe to swap out the current active piece of hardware. If an operation takes many cycles, it may be advantageous to allow it to complete before swapping out the hardware.

Additionally, if reset functionality is needed, it should be provided by one of the registers, as no system-level reset reaches the reprogrammable region.

6.5 Preliminary Results

Some simple tests have been performed in order to verify the correct functionality of this partial hardware implementation, as well as to understand its performance limitations.

6.5.1 Sample Core

A 32-location 8-bit register file was implemented as a loadable core, and coupled to the interface described in section 6.2.1. This core was synthesized using the Xilinx EAPR flow. The Xilinx *bitgen* tool produces a partial configuration bitstream, located in a binary file.

Next, the C program given in figure 6-8 was written and compiled into a binary. The constants bitstream_size and bitstream referenced in the C code are provided by the binary image from *bitgen*, which is linked directly into the final ELF binary. The COUNTER_REG register is the 32-bit cycle counter in the processor. This allows the time elapsed during programming to be measured. Finally, ICAP_BASE is the memory address of the reprogramming controller.

6.5.2 Verification of Correctness

In order to verify the correctness of the reprogrammable peripheral, a series of different values were written to each register in the register file. Then, these values were read back and compared against expected results. The test was also performed without

```
int main() {
  unsigned long i;
  unsigned long before, after;

  puts("0x"); print_hex(bitstream_size); puts(" bytes\r\n");
  puts("0x"); print_hex(bitstream_size/4); puts(" words\r\n");

  before = MemoryRead(COUNTER_REG);
  for(i=0;i<(bitstream_size/4);i++) {
    MemoryWrite(ICAP_BASE, bitstream[i]);
  }
  after = MemoryRead(COUNTER_REG);
  puts("before: 0x"); print_hex(before); puts(" cycles\r\n");
  puts("after: 0x"); print_hex(after); puts(" cycles\r\n");

  return 0;
}
```

Figure 6-8: C Program to Load Partial Bitstream.

Implementation	Words	Time (cycles)	cyc per word	@50 MHz	@100 MHz
Implemented	7,464	268,763	36	5.38 ms	2.69 ms
Ideal	7,464	7,464	1	149.28 μs	74.64 μs

Table 6.5: Reprogramming Time

the reprogrammable core resident in hardware. This resulted in readback values of 0xFF for each register, regardless of what value was written. The result of this test was that the reprogrammable peripheral, as well as the reprogramming controller, both function correctly.

6.5.3 Reprogramming Performance Analysis

Table 6.5 shows the result of running this CPU-based reprogramming routine as compared to the theoretical maximum throughput of the ICAP port. These results show that the simple software implementation leaves much room for improvement.

The programming loop compiles to 5 assembly instructions. Because the MIPS core is not equipped with any instruction or data caches, each instruction (and one branch delay slot) must be fetched directly from memory, at a latency of 5 cycles each. The programming bitstream word must also be fetched from memory, which takes another 5 cycles. Finally, the write to the ICAP interface takes one cycle, for a total of 36 cycles to write a single word to the ICAP interface. This accounting matches the observed result.

Clearly, much of the overhead is simply due to the lack of instruction caching. Adding an instruction cache to the CPU should speed things up by more than 3x, as each instruction fetch would take only a single cycle. Still, the number of cycles per ICAP write would be significant, due to the number of instructions in the loop, as well as the data memory read.

In order to make this system practically useful, a performance much closer to the ideal given in table 6.5 must be achieved.

6.6 Future Work

While the given implementation is successful in demonstrating the base functionality of a reprogrammable hardware peripheral, significant performance improvements are needed to allow practical use.

6.6.1 Direct Memory Access

One of the most important improvements needed is the use of Direct Memory Access (DMA) within both the reprogramming controller, as well as the reprogrammable peripheral itself. DMA is implemented by way of special purpose hardware (a DMA "Engine") whose purpose is to copy blocks of data from one part of the address space to another.

A modified reprogramming controller with DMA logic would implement both an Avalon master and slave interface. A software program running on the microprocessor would program the location and size of a bitstream over the slave interface. Then, the DMA logic in the reprogramming controller would access the memory directly via the Avalon master interface. By using burst transfers, a throughput of nearly one word per cycle could be maintained, bringing throughput much closer to the ideal listed in table 6.5.

The second gain comes from the fact that the CPU is no longer tied up by the whole memory transfer. Once the DMA logic begins a transfer, the microprocessor is free to do other processing.

DMA would have similar advantages as far as the actual use of the reprogrammable core. It would help maximize the throughput of the reprogrammable peripheral, and additionally allow the CPU to do other processing while the reprogrammable core is busy.

6.6.2 Bitstream Relocation in Hardware

The implementation of bitstream relocation presented in this work takes place by way of the *bitparse* program. This method has been shown to properly relocate a bitstream, but will encounter a performance penalty, since it necessitates that the microprocessor is directly involved in programming.

Because the relocation procedure largely consists of simply translating frame addresses and reading from some computed offset in memory, it is well suited to a hardware implementation. It should be able to translate a bitstream "on-the-fly" without ever revisiting prior bits.

Chapter 7

Conclusion

This thesis explored the design of a reprogrammable peripheral architecture to enable custom, hardware-accelerated cores that could be loaded on demand. This was achieved through application of the partial reconfiguration capabilities of the Xilinx Virtex-5 FPGA, as well as an architecture to allow efficient runtime reuse of logic resources. Two software tools, along with a partial hardware implementation were presented which demonstate the techniques proposed.

Some future work will be necessary to increase the performance of the peripheral so that it may run near its ideal design speed. The addition of DMA functionality into both the reprogramming logic and the reprogrammable cores themselves will greatly boost performance. Additionally, some of the functionality implemented here in software can be translated into efficient hardware implementations.

Bibliography

[1] Altera, Inc. *Avalon Interface Specifications*, April 2009.

[2] J. Carver, N. Pittman, A. Forin, and N. Pittman. Relocation and Automatic Floor-planning of FPGA Partial Configuration Bit-Streams. Technical report, Microsoft Research, Tech. Rep. MSR-TR-2008-111, 2008.

[3] C. Claus, J. Zeppenfeld, F. Muller, and W. Stechele. Using partial-run-time reconfigurable hardware to accelerate video processing in driver assistance system. In *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*, pages 1–6, 2007.

[4] I. Gonzalez, F. Gomez-Arribas, and S. Lopez-Buedo. Hardware-accelerated SSH on self-reconfigurable systems. In *2005 IEEE International Conference on Field-Programmable Technology, 2005. Proceedings*, pages 289–290, 2005.

[5] S. Hauck. Configuration prefetch for single context reconfigurable coprocessors. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 65–74. ACM New York, NY, USA, 1998.

[6] J. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *IEEE Symposium on FPGAs for Custom Computing Machines*, volume 33. Los Alamitos: IEEE Computer Society Press, 1997.

[7] M. Hubner, C. Schuck, and J. Becker. Elementary block based 2-dimensional dynamic and partial reconfiguration for Virtex-II FPGAs. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 8, 2006.

[8] H. Kalte, M. Porrmann, and U. Ruckert. System-on-programmable-chip approach enabling online fine-grained 1D-placement. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004.

[9] Z. Li and S. Hauck. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 187–195. ACM New York, NY, USA, 2002.

[10] C. Maxfield. Xilinx unveil revolutionary 65nm FPGA architecture: the Virtex-5 family. `http://www.pldesignline.com/products/187203173`, May 2006.

[11] J. Mignolet, S. Vernalde, D. Verkest, and R. Lauwereins. Enabling hardware-software multitasking on a reconfigurable computing platform for networked portable multimedia appliances. In *Proceedings of the International Conference on Engineering Reconfigurable Systems and Architecture 2002*, pages 116–122, 2002.

[12] A. Mitra, Z. Guo, A. Banerjee, and W. Najjar. Dynamic Co-Processor Architecture for Software Acceleration on CSoCs. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 127–133, 2007.

[13] M. Novati. Polaris: 2D Relocation for Self Dynamical Run-time Reconfiguration. Master's thesis, Politecnico Di Milano, 2007.

[14] R. Scholz. Adapting and Automating XILINX's Partial Reconfiguration Flow for Multiple Module Implementations. *LECTURE NOTES IN COMPUTER SCIENCE*, 4419:122, 2007.

[15] N. Steiner. A standalone wire database for routing and tracing in Xilinx Virtex, Virtex-E, and Virtex-II FPGAs. Master's thesis, Virginia Polytechnic Institute and State University, 2002.

[16] H. Tan and R. DeMara. A multilayer framework supporting autonomous run-time partial reconfiguration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(5):504–516, 2008.

[17] Xilinx, Inc. *Virtex FPGA Series Configuration and Readback (XAPP138)*, February 2000.

[18] Xilinx, Inc. *Correcting Single-Event Upsets in Virtex-4 Platform FPGA Configuration Memory*, March 2008.

[19] Xilinx, Inc. *Virtex-5 FPGA Configuration User Guide (UG191)*, February 2009.

[20] Xilinx, Inc. *Virtex-5 FPGA Data Sheet: DC and Switching Characteristics (DS202)*, April 2009.

[21] I. Zaidi, A. Nabina, C. Canagarajah, and J. Nunez-Yanez. Evaluating dynamic partial reconfiguration in the integer pipeline of a FPGA-based opensource processor. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 547–550, 2008.