5-2010

Accelerated Frame Data Relocation on Xilinx Field Programmable Gate Array

Ramachandra Kallam
Utah State University

ACCELERATED FRAME DATA RELOCATION ON XILINX FIELD PROGRAMMABLE GATE ARRAY

by

Ramachandra Kallam

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Engineering

Approved:

Dr. Aravind Dasu Major Professor	Paul Israelsen Committee Member
Dr. Chris Winstead Committee Member	Dr. Byron R. Burnham Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2010

Abstract

Accelerated Frame Data Relocation on Xilinx Field Programmable Gate Array

by

Ramachandra Kallam, Master of Science

Utah State University, 2010

Major Professor: Dr. Aravind Dasu
Department: Electrical and Computer Engineering

Emerging reconfiguration techniques that include partial dynamic reconfiguration and partial bitstream relocation have been addressed in the past in order to expose the flexibility of field programmable gate array at runtime. Partial bitstream relocation is a technique used to target a partial bitstream of a partial reconfigurable region (PRR) onto other identical reconfigurable regions inside an FPGA, while partial dynamic reconfiguration is used to target a single reconfigurable region. Prior works in this domain aim to minimize "relocation time" with the help of on-chip or on-line processing. In this thesis, a novel PRR-PRR relocation algorithm is proposed and implemented both in software and hardware. Dedicated hardware architecture, called the accelerated relocation circuit (ARC), is designed and presented for fast relocation. An analytical model is also proposed to evaluate the performance of the PRR-PRR relocation algorithm and highlight the speed-up obtained by the proposed hardware implementation. ARC has been tested on two categories of designs: dynamically scalable systolic array designs and fault tolerant designs. It has been compared against the software implementation of the algorithm, BiRF, hardware architecture for bitstream relocation, and a software solution for bitstream relocation. An average speed-up of 153x for ARC over BiRF is observed, with the additional advantage of not storing any bitstreams, thus saving invaluable block random access memory (BRAMs). Accuracy of proposed analytical model was found to be more than 95% for all the test cases.

(57 pages)

To Mom.

Acknowledgments

I would like to thank my advisor, Dr. Aravind Dasu, for giving me an opportunity to work with him. I also thank him for the continuous guidance and motivation provided throughout my master's program. I thank my committee members, Dr. Chris Winstead and Paul Israelsen, for extending their support.

I thank my friend, Parthasarathi Valluri, who has motivated me to step into research. I thank my sister, Anusha Natarajan, without whom this would not have been possible. I thank Jeff Carver, who has initially helped me in my learning curve and also been there for regular research discussions throughout my program. I thank Jonathan Philips and Rob Barnes for their help. I thank Hari Samala, Varun Voddi, and Shant Chandrakar for their moral support to finish my thesis. I thank Arvind Sudarsanam for his technical guidance and reviewing my work. I thank Anitha Vemuri for all the support. I thank all my friends who have made my stay in Logan memorable.

I thank my family for their support. I thank my mom for the love and patience she showed during my studies.

Ramachandra Kallam

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this introductory chapter, the motivation for developing a faster relocation methodology for Field Programmable Gate Arrays (FPGAs) is discussed. The key contributions of this thesis are presented next followed by an overview of the content presented in this thesis.

1.1 Motivation

FPGA-based partial reconfiguration and relocation techniques are being increasingly used to enhance dynamic scalability of systolic arrays and fault tolerance of space-based processors. Partial bitstream relocation (PBR) has been pursued by the FPGA community for the past five years for a variety of applications. Sreeramareddy et al. [1] show that fast partial bitstream relocation techniques are helpful in supporting certain fault-tolerant techniques, where time to replace a faulty circuit with the correct circuit (using relocation) and restart the computation is critical to the performance. Sudarsanam et al. [2] have explored the applicability of PBR-based partial dynamic reconfiguration (PDR) to allow for rapid rescaling of kernels for navigation and image processing in satellites. Another forward looking application that motivated me to investigate fast circuit relocation is the potential applicability in a 3D FPGA stack to mitigate hot-spot formation. This is potentially equivalent to or more promising that code migration proposed by the many-core community to mitigate hot spots.

1.2 Key Contributions

The key contributions of this thesis are:

1. A novel PRR-PRR relocation algorithm on FPGAs;

2. A dedicated hardware architecture for relocation, “Accelerated Relocation Circuit;” and
3. A performance model to analyze the algorithm.

1.3 Thesis Overview

Following the introduction section, the basics of FPGAs, an introduction to PDR and early access partial reconfiguration (EAPR), a partial reconfiguration methodology from Xilinx, is discussed in Chapter 2. It also gives an introduction on PBR and a literature review on PDR and bitstream relocation. Chapter 3 presents the PRR-PRR algorithm and also the performance model to analyze the algorithm. In Chapter 4, the software implementation of the algorithm is given followed by a detailed description of the hardware architecture, called the accelerated relocation circuit (ARC). An analysis of the software and hardware implementation is also presented in this section. Results of the proposed algorithm in comparison with previous works is presented in Chapter 5. Chapter 6 gives the conclusions and the future work.

Chapter 2
Background

2.1 Field Programmable Gate Arrays

A field programmable gate array (FPGA) is a semiconductor device which is made of logic blocks that are connected with electrically programmable interconnections as shown in fig. 2.1. These logic blocks can be re-programmed to perform different functions, which gives them an advantage over application specific integrated circuits (ASIC) whose functionality is fixed. This feature of re-programmability has led the FPGA to be used for prototyping circuits. FPGAs can be programmed using hardware description languages (HDL) like Verilog and VHDL. FPGAs were invented by Xilinx, and they are also the market leaders [3]. FPGAs can be used in a wide range of applications, few of them being telecommunication base stations, televisions, in-car infotainment systems. A Xilinx FPGA is made up of various blocks, digital clock manager (DCM), digital signal processing (DSP48) units, block RAMs (BRAM), programmable interconnect points (PIP) and configurable logic blocks (CLB), etc.

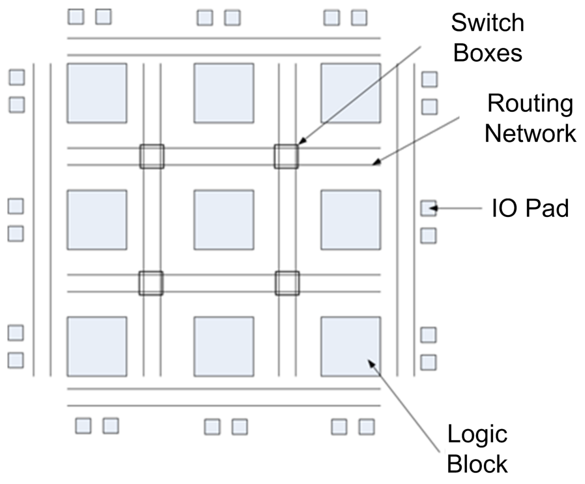
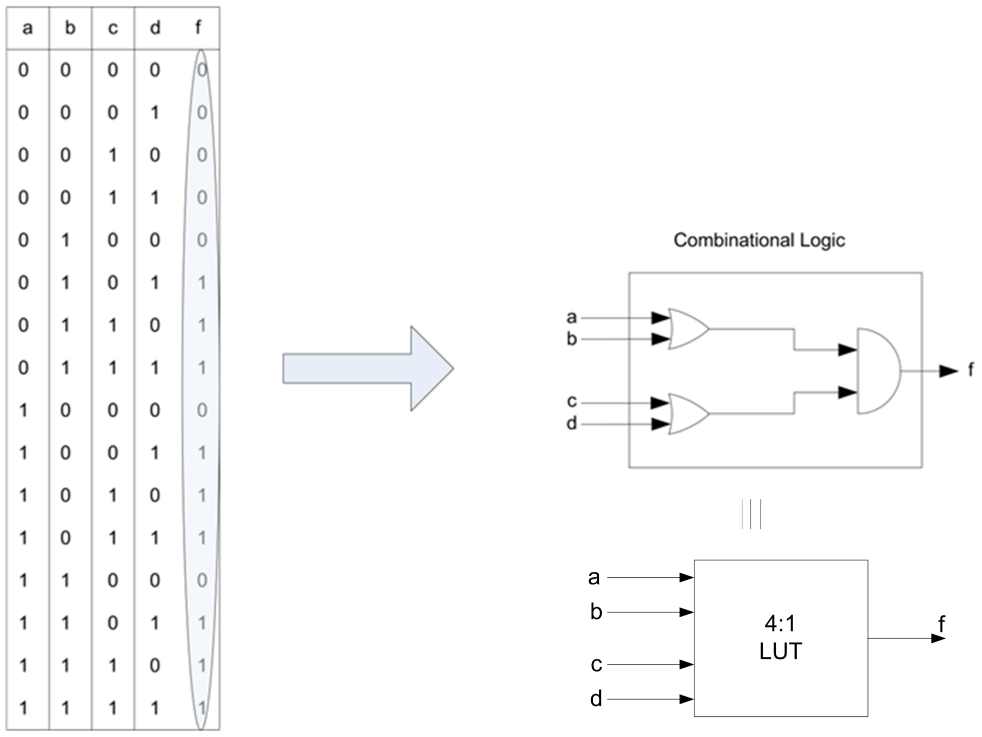


Fig. 2.1: General structure of an FPGA.

CLBs can be used for implementing sequential as well as combinatorial logic. Each CLB consists of four interconnected slices. Slices consist of look-up tables (LUT), flip-flops (FF), multiplexers, and a few gates. There are two different types of slices in a CLB, namely SliceL and SliceM. The basic difference being, the LUTs in SliceL can be used only to implement logic, where as SliceM LUTs can be additionally configured as 16 $\times$ 1-bit synchronous distributed RAM and shift registers. LUTs are implemented as multiple inputs single output blocks, with input varying depending on the FPGA family. For example, a Virtex-4 FPGA has a 4-input 1-output LUT and any logic is broken down into 4-input Boolean function. Logic with large number of inputs is implemented across multiple LUTs and can be combined using multiplexers. Figure 2.2 shows the mapping of logic into a LUT. The values in the f column are stored in the LUT and they are given as output by the LUT depending on the inputs a, b, c, and d, which act as address lines.



a	b	c	d	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Fig. 2.2: LUT mapping.

The storage elements in a slice can be configured to function as either edge triggered D flip-flops (D-FF) or level sensitive latches. The input of the D-FF can be directly driven by the output of the LUT or can be driven from outside the CLB. In addition to the distributed RAM, there are 18Kb BRAMs which offer minimal timing penalty. The number of BRAMs depends on the type of the FPGA. Each BRAM can be configured for different aspect ratios, ranging from 16K $\times$ 1, 8K $\times$ 2, to 512 $\times$ 36, meaning that the output port can have a width ranging from 1 bit to 36 bits and the depth varying from 16K to 512K, respectively.

FPGAs have their share of advantages and disadvantages compared to alternate solutions such as an ASIC. FPGAs have a faster time-to-market, as there are no layouts, masks, or other manufacturing costs involved for system designers. There are no non-recurring engineering costs (NRE) which are typically associated with ASICs. It has a simple design cycle compared to ASICs, more predictable project cycle and the advantage of re-programmability [4]. The disadvantages of an FPGA compared to an ASIC are higher power consumption, increased chip area, and lower clock rates.

A “bitstream” is a file used to configure an FPGA. The design flow from HDL description to a bitstream is shown in fig. 2.3. The Xilinx synthesis tool takes the HDL design description and generates the physical representation for the targeted FPGA. A netlist is created at the end of this process. “Translate” merges this netlist and any constraint information such as placement, I/O constraints, and outputs of a Xilinx native generic database (NGD) file. After translation, “map” process takes the NGD file and runs a design rule check and then maps the logic design to the specified FPGA. This outputs a native circuit description (NCD) file, which is used in the next process. Xilinx’s place and route (PAR) tool reads the NCD file and performs optimal placement and routing on the mapped device primitives.

2.2 The Addressing Layout of Virtex-4 FPGA

Xilinx FPGAs are made up of small units called frames, which are addressable. A frame is the smallest unit of granularity in the FPGA. In other words, the smallest unit which can be read from or written to a FPGA is called one frame. All frames in the Virtex-

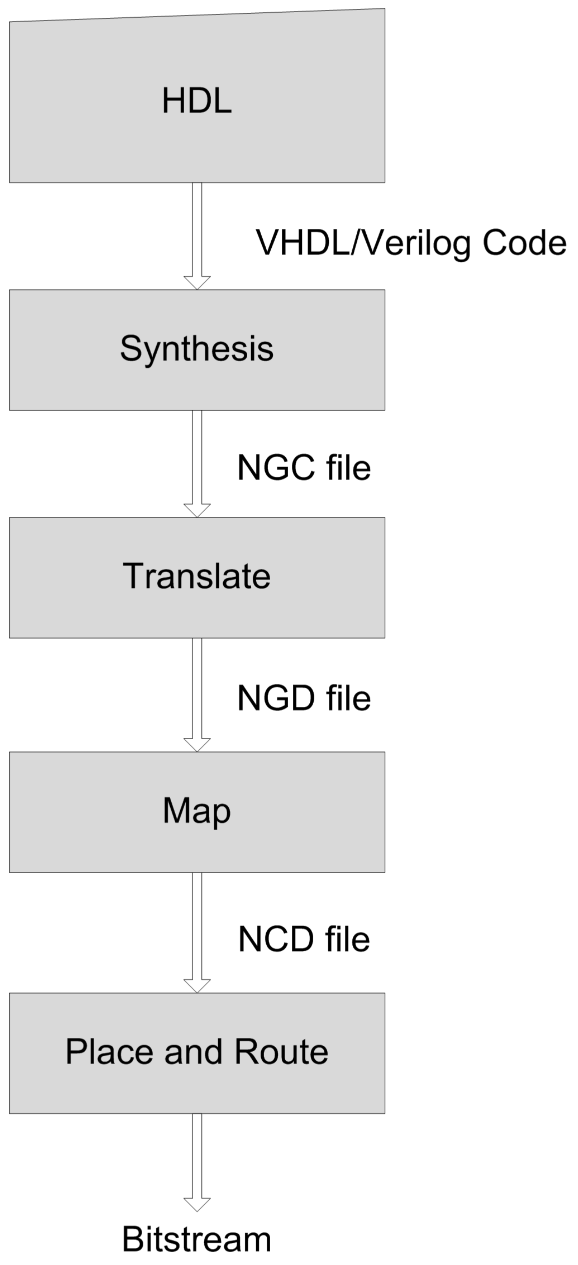


Fig. 2.3: Design cycle for generating a bitstream.

4 have a fixed identical length of 1312 bits or 41 32-bit words. The address of a frame is composed of five components and is stored in the frame address register (FAR), which is a 32-bit register. The five components that constitute a frame address are shown in fig. 2.4 [3].

The layout of a Virtex-4 SX35 chip is shown in fig. 2.5. The five components that constitute a frame address are explained below with respect to this device, but are similar to other Virtex family devices from Xilinx.

Top/Bottom: The FPGA is divided into two halves vertically called the top and bottom wherein the frames in the top half of the FPGA are mirror images of the frames in the bottom part of the chip.

Block Type: The FPGA is made of components like CLBs, DSPs, and BRAMs, etc. All the components are categorized into block types for addressing purposes. BRAM interconnect and BRAM content are categorized into different block types. Each block type has a 3-digit code which is part of the FAR. The categorization is shown in fig. 2.4.

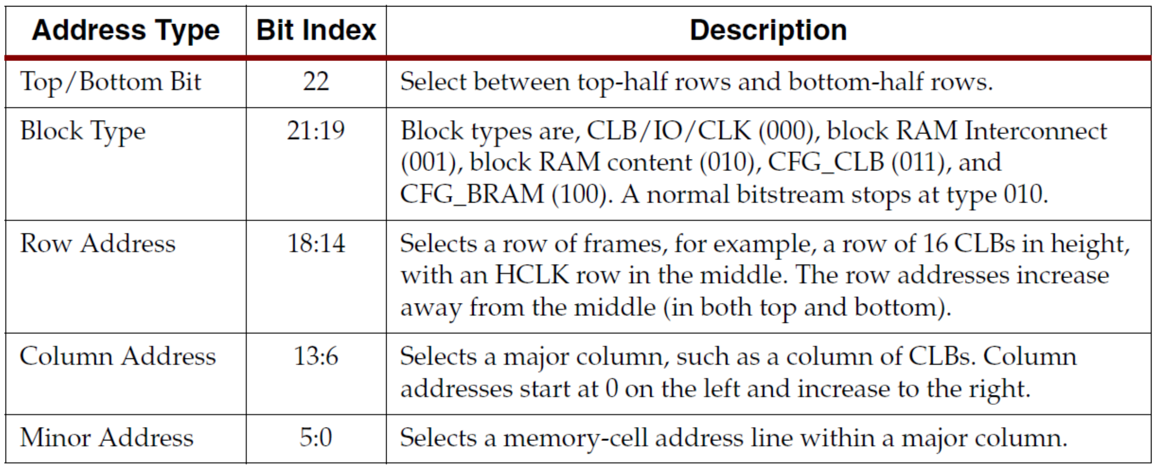
Row Address: FPGA is divided vertically again into horizontal clock regions (HCLK). These regions are numbered from 0, starting at the middle of the chip and incrementing upwards. The bottom half of the chip also has the same addressing, starting with 0 at the middle of the chip and increments downwards. However, there is the top/bottom bit which differentiates between the top and bottom of the chip. The height of one frame is one HCLK row.

Column Address: As shown in fig. 2.5, each block type has its own addressing starting from 0 on the left and increases to the right. This address is called the "column address" or "major address" or "major column."

Minor Address: Each CLB/DSP/BRAM column is composed of a certain number of minor addresses, also called as frames. This number varies with the type of resource. Table 2.1 shows the number of frames per each FPGA resource.

2.3 Partial Dynamic Reconfiguration

One of the key features of FPGAs is re-configurability. It is the ability to change the



Address Type	Bit Index	Description
Top/Bottom Bit	22	Select between top-half rows and bottom-half rows.
Block Type	21:19	Block types are, CLB/IO/CLK (000), block RAM Interconnect (001), block RAM content (010), CFG_CLB (011), and CFG_BRAM (100). A normal bitstream stops at type 010.
Row Address	18:14	Selects a row of frames, for example, a row of 16 CLBs in height, with an HCLK row in the middle. The row addresses increase away from the middle (in both top and bottom).
Column Address	13:6	Selects a major column, such as a column of CLBs. Column addresses start at 0 on the left and increase to the right.
Minor Address	5:0	Selects a memory-cell address line within a major column.

Fig. 2.4: Frame address register.

hardware functionality of the underlying circuit by altering the bitstream. Spatially reconfiguration can be done either in full or partial modes. Full reconfiguration is the process of changing the entire design on the FPGA to obtain a different functionality on the chip. Virtex-II/ Virtex-II Pro, Virtex-4, and Virtex-5 have the ability to be reconfigured partially, which is the process of changing only a part on the FPGA, leaving the remaining parts unaffected. Temporally, partial reconfiguration is further classified into two types.
1. Static partial reconfiguration: Reconfiguring a portion of the chip when it is inactive without affecting other areas of the chip.
2. Partial dynamic reconfiguration (PDR): Reconfiguring portions of the chip on the fly, i.e., when the device is active without disturbing the operation of the rest of the chip. This is also termed as just partial reconfiguration as static partial reconfiguration is a deprecated feature.

Some of the terminology used with PDR is given below.

Partial Reconfigurable Region (PRR): A portion of the FPGA with predefined boundary which is set to be reconfigurable.

Partial Reconfigurable Module (PRM): A module which resides in the PRR. A PRR can have multiple PRMs, which can be interchanged during runtime through reconfiguration.

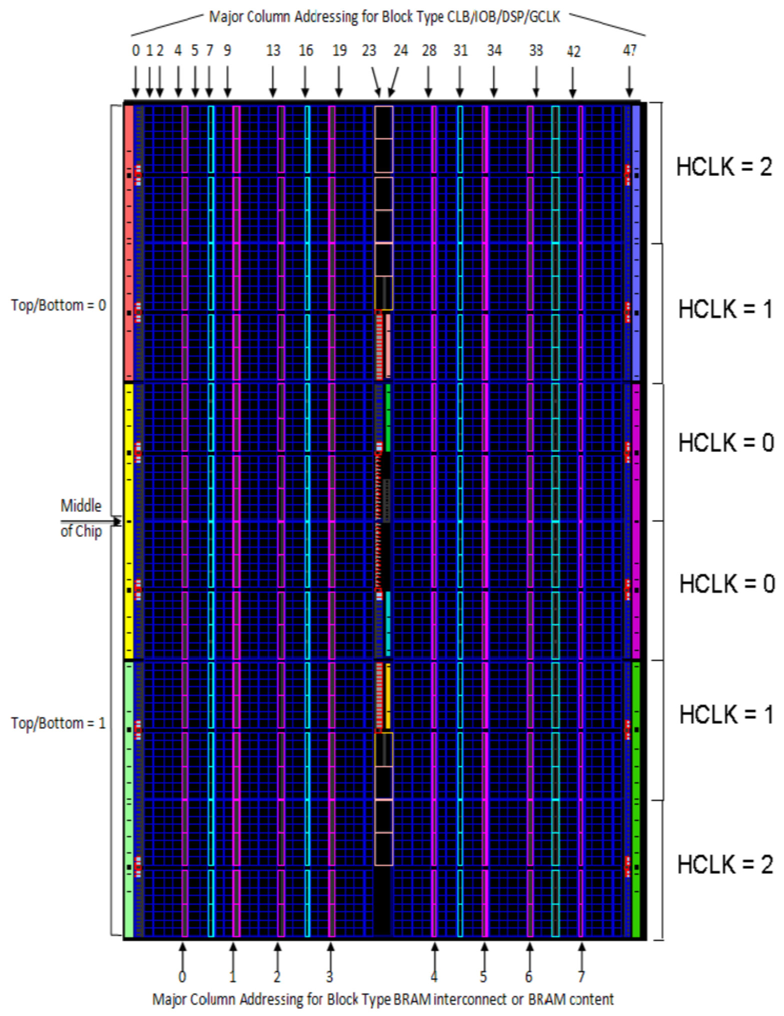


Fig. 2.5: Virtex-4 SX 35 FPGA layout.

Static Logic: All the designs/modules which are not assigned to any PRR belong to static logic.

Bus Macros (BM): Hard macros that must be instantiated in RTL for communication between static logic and PRRs. Any communication between static logic and PRRs has to go through the bus macros, with the exception of clock. This is necessary as the bus macros has the capability of locking the routing between PRRs and the static logic, making the PRRs pin compatible with the static design. All bus macros provide a data bandwidth of 8 bits. Bus macros should be placed in such a way that they straddle across the PRR

Table 2.1: Number of frames per column.

Column Type	Frames per Column
CLB	22
IO	30
DSP	21
CLK	2
BRAM Interconnect	20
BRAM Content	64

boundary. However, we should make sure that they do not straddle across a DSP or a BRAM column. There are different kinds of bus macros provided by Xilinx based on different factors [4].

1. Signal direction (left-to-right, right-to-left, top-to-bottom, and bottom-to-top):

The physical placement of the bus macros on the PRR region and the logical signal direction, i.e., whether a signal is input or output determines which type of bus macro to choose from. For example, inputs that pass through the left side of a PR region require left-to-right (l2r) bus macro, while inputs that pass through the right side of a PR region require a right-to-left (r2l) bus macro. In the same way, outputs from a PRR passing through a bus macro on the left side of the region require a r2l bus macro and passing through right side will require a l2r bus macro. Figure 2.6 shows a PRR with two inputs and two outputs.

2. Physical width (wide and narrow):

Xilinx provides narrow and wide bus macros. Wide and narrow refer to the physical width of the bus macro, but not the bandwidth. Narrow bus macros are two CLBs wide and wide bus macros are four CLBs wide.

3. Whether signals passing through the bus macros are registered or not (synchronous and asynchronous):

Synchronous and asynchronous bus macros are also available from Xilinx. Synchronous bus macros provide superior timing performance.

Outputs from a PRR can toggle unpredictably during active partial reconfiguration; therefore, an enable signal is provided for the bus macro. When the enable signal is de-asserted (enable = 0), the bus macro drives a constant 0 to the static design. So, a good

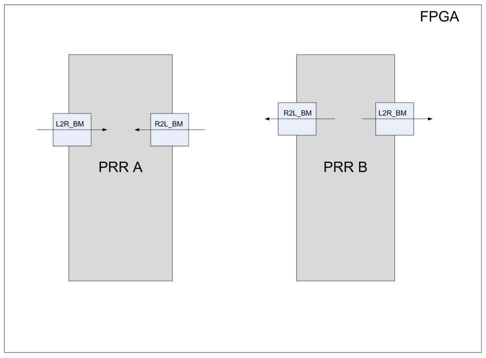
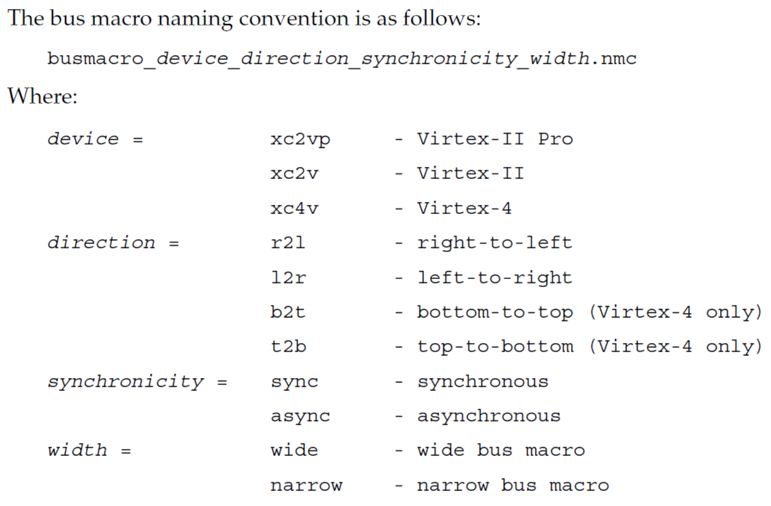


Fig. 2.6: Signal direction of bus macros.

practice is to disable the bus macro during partial reconfiguration and enable it again after reconfiguration. Bus macro naming convention is given in fig. 2.7. The naming convention is obtained taking into account the different types of bus macros available.

Partial Bitstream: Bitstream representing a PRM configuration.

Internal Configuration Access Port (ICAP): The ICAP primitive has four input ports (CLK, CE, WRITE, I) and two output ports (O and BUSY) as shown in fig. 2.8. CE (clock enable) and WRITE are active low. When the ICAP is to be set to write mode, both CE and WRITE needs to be set low and the data needs to be sent into the I port. The BUSY signal goes low indicating the ICAP is busy. While setting the control ports, one should make sure that CE is high while setting WRITE. So, in this case, while setting it to write



The bus macro naming convention is as follows:

busmacro_*device*_*direction*_*synchronicity*_*width*.nmc

Where:

device =	xc2vp	- Virtex-II Pro
	xc2v	- Virtex-II
	xc4v	- Virtex-4
direction =	r2l	- right-to-left
	l2r	- left-to-right
	b2t	- bottom-to-top (Virtex-4 only)
	t2b	- top-to-bottom (Virtex-4 only)
synchronicity =	sync	- synchronous
	async	- asynchronous
width =	wide	- wide bus macro
	narrow	- narrow bus macro

Fig. 2.7: Naming convention of a bus macro.

mode, WRITE should be set low before setting the CE low. The timing diagram of the ICAP, when setting it in write mode is given in fig. 2.9. When the ICAP needs to be set to read mode, WRITE should be left high and CE needs to be set low. If the BUSY signal goes low during readback, it indicates that the ICAP is reading valid data. If both CE and WRITE are driven in the same clock cycles, ICAP goes into abort stage and it does not work. The timing diagram for the ICAP for read mode is as shown in fig. 2.10.

The ICAP primitive can be instantiated in any design and one can write their own drivers to communicate to the ICAP. Xilinx also provides an IP core, called the HWICAP, which can be added into the system and can be used with Xilinx provided software functions. HWICAP is controlled using Microblaze [5], a soft processor core from Xilinx.

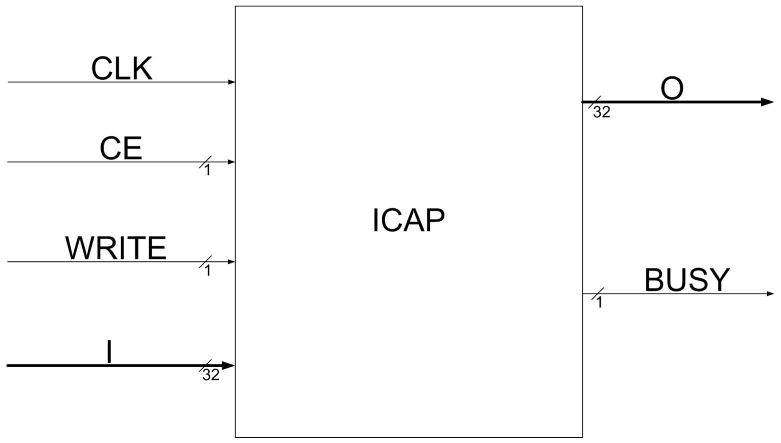


Fig. 2.8: Internal configuration access port.

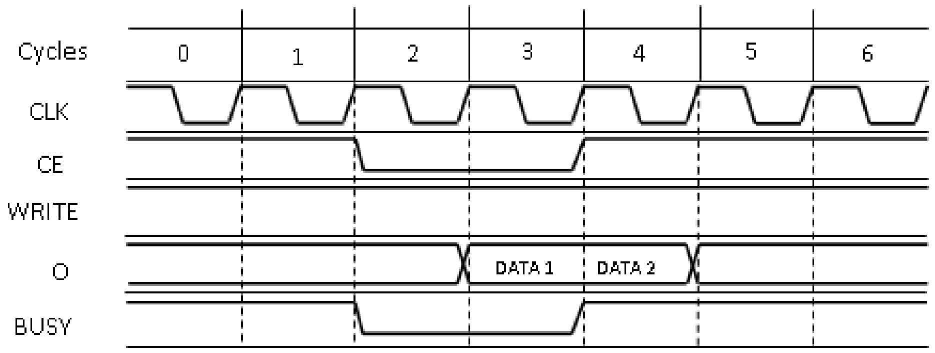


Fig. 2.9: ICAP in write mode.

As an example, an FPGA with two partial reconfigurable regions is shown in fig. 2.11. Each PRR has three PRMs assigned to it. The user should make sure that the device resource utilization of each PRM is less than that of the PRR, i.e., each PRR should be big enough to accommodate the largest PRM.

Functional density on the FPGA can be improved using partial reconfiguration [6]. The advantages of partial reconfiguration are discussed in Kao [7]. It is one of the most efficient ways of running different applications on a single device, resulting in reduced resource utilization per design, reduced power consumption and overall lower costs, updating the hardware with a different configuration even remotely.

The time to configure the FPGA is directly proportional to the bitstream used to configure the FPGA. With partial reconfiguration, one can build partial bitstream meant for each PRM. So, by modifying only portions of the design using partial bitstreams instead

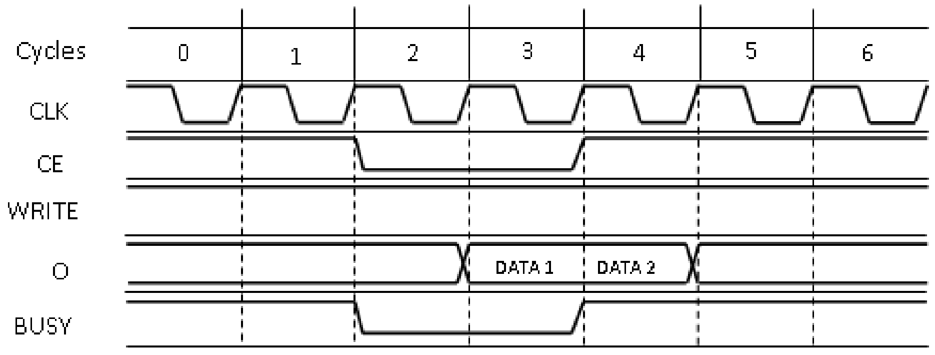


Fig. 2.10: ICAP in read mode.

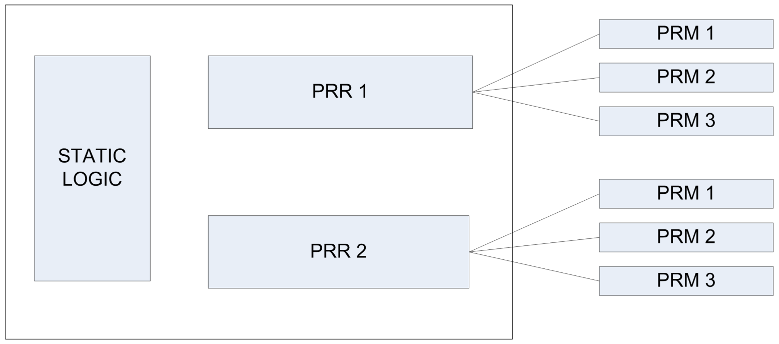


Fig. 2.11: Partial dynamic reconfiguration.

of reconfiguring the entire FPGA with a full bitstream, one can achieve shorter reconfiguration times.

Several different tools have been designed to handle partial reconfiguration. However, all these tools are cumbersome and not reliable. Also, the bus macros used in these methods are designed manually which can be tedious and error prone. To alleviate some of these problems, Xilinx introduced a partial reconfiguration flow, called the "early access partial reconfiguration (EAPR)" in 2007 [8]. EAPR is introduced mainly to provide an easy user-friendly methodology to perform PDR. With EAPR, Xilinx provides the netlists of the bus macros which can be directly used in the design process after instantiating them in the HDL code. EAPR also provides PR tools which integrates a tool called PlanAhead [9] and ISE with which PDR becomes easy as all PRRs and PRMs can be defined in PlanAhead using a graphical interface and then call ISE to generate the necessary bitstreams. The general flow of this methodology is provided in fig. 2.12 and the flow is explained below.

HDL (Design Description): In this phase, the design is divided into static part and reconfigurable parts depending on which parts to be reconfigured. Decisions like, number of PRRs, number of PRMs for each PRR, which functionality (PRM) is to be implemented in each region (PRR), are taken. All the designs are designed and synthesized in Xilinx ISE. One needs to disable I/O buffers for the design before synthesizing as the peripheral in which this design will be instantiated will communicate to the outside world via OPB bus. Also, the reconfigurable modules implemented in each region should be pin compatible and should be having the same port names. This is one of the drawbacks of EAPR as it is difficult to maintain the same number of input/output ports for all the reconfigurable modules.

Xilinx Platform Studio (EDK, System Level Design): The peripherals are created in embedded development kit (EDK). The system level design is done in this tool, where all the required peripherals such as a soft processor (used to control the system), ICAP (used to perform reconfiguration), etc., are added and all necessary interconnections established. Then, the design is synthesized and a netlist is created.

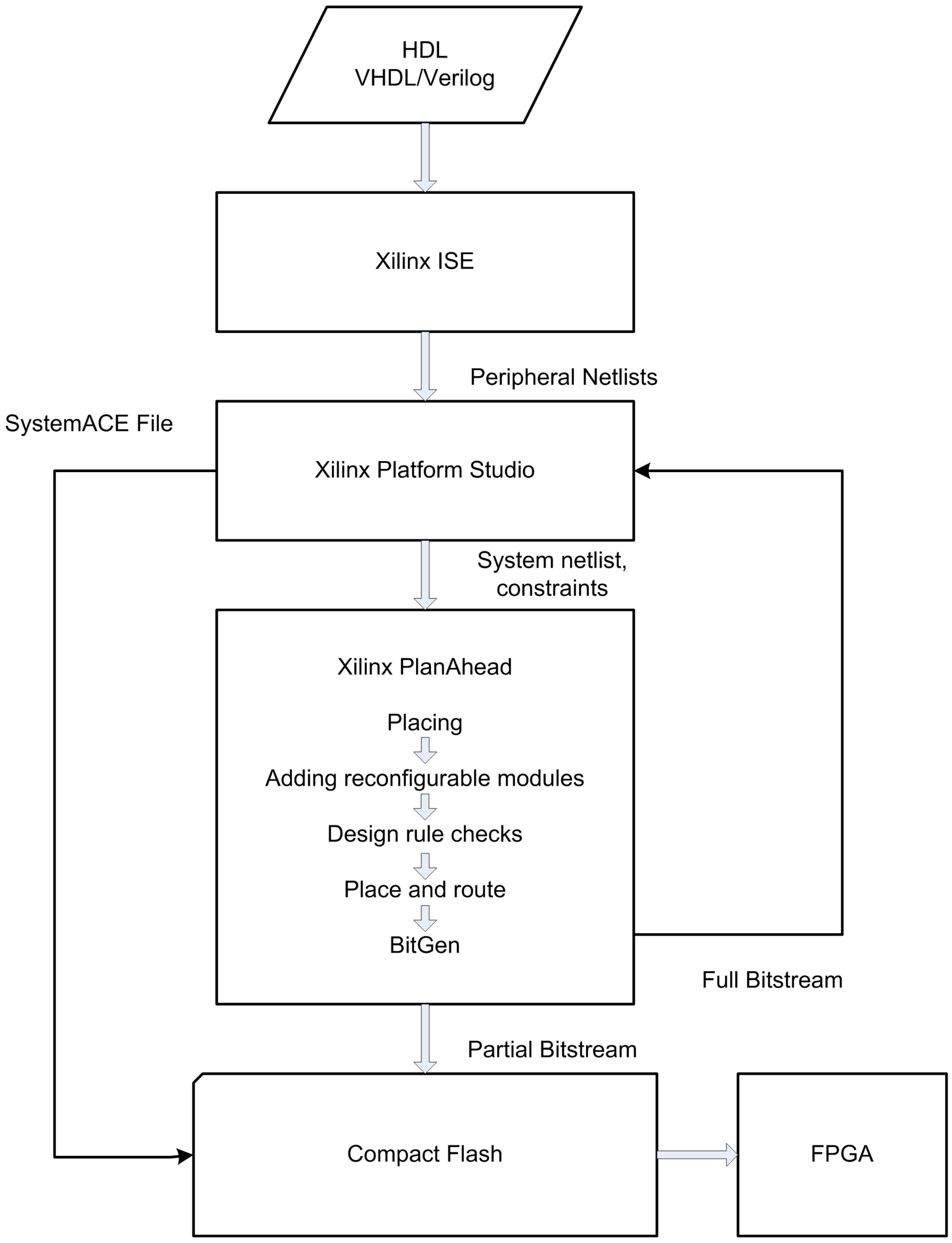


Fig. 2.12: Early access partial reconfiguration.

PlanAhead: The design netlist created in EDK is taken as input to a tool called PlanAhead. Then, all the peripherals are floor planned maintaining the resource utilization. The reconfigurable regions are defined and all the reconfigurable modules for each region are added. Then, design rule checks are performed to make sure that the floor planning complies with all the rules. The design is then placed and routed using Xilinx ISE place and route tools which will be invoked through PlanAhead. After successful place and route, the bitgen tool is run to create full and partial bit streams.

The partial bit streams are then copied to the compact flash card. The full bitstream is exported to EDK. The software that will be run in the soft processor is created and compiled. A system ace file is created which combines the full bit stream and the software which is then exported to the compact flash card. The card is then plugged into the FPGA board so that the system ace file will program the FPGA once the board is powered on.

2.4 Partial Bitstream Relocation

The bitstream created for each PRM in the PRR is bound to that particular PRR with respect to physical boundaries, which means the bitstream cannot be used directly for configuring a PRR at a different location on the chip. For example, if there are two identical PR regions with the same logic configured in each of the PRRs, the bitstream of one PRR cannot be used to configure the second PRR as the physical location of the PRRs is different and the bitstream contains the information of the PRRs boundaries. So, in this case, both the bitstreams have to be stored either off chip, say in a flash card which the FPGA reads when required or on-chip in the Block RAMs. By doing so, memory is wasted by storing two different bitstreams with same logic.

An alternative approach is to store only one bitstream and modify the bitstream in such a way that it can be used for other PRRs (with identical underlying resources). To do this, one needs to understand the contents of a bitstream. A partial bitstream, generated from bitgen [10], the Xilinx tool used to generate bitstreams, associated with a PRR can be described as a combination of two components: frame data (FD) and commands to

- synchronize/desynchronize with the ICAP,
- write a frame, and
- cyclic redundancy check (CRC) processing.

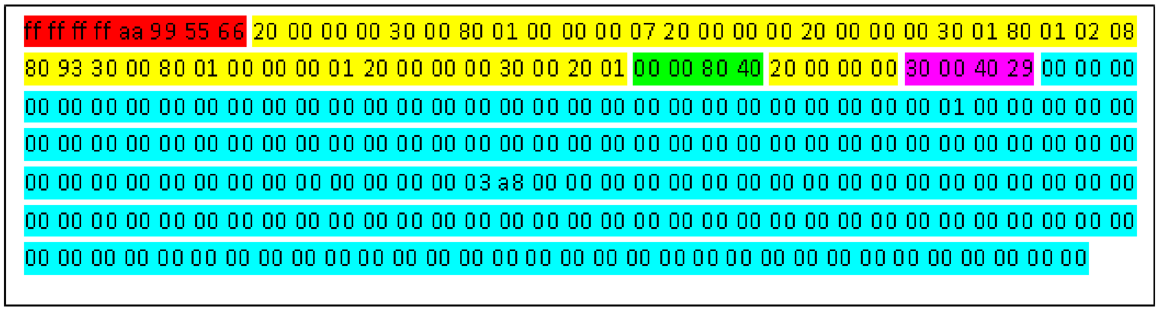
The frame data follows a certain set of commands, usually to set up the ICAP into write mode and provide the frame address to which the frame data will be written. A chunk of a partial bitstream is taken as an example and is shown in fig. 2.13.

The bitstream starts with a DUMMY (FF FF FF FF) and a SYNC (AA 99 55 66) word, shown in red. This is followed by commands necessary to set up the ICAP into write mode, highlighted in yellow. The word highlighted in green is the frame address and the word following the frame address indicates the number of words to be written, highlighted in pink. This is followed by the frame data, which is used to configure the frame. To understand the commands in the bitstream, an in-depth understanding of how these commands are formed is necessary.

The Virtex-4 configuration logic consists of a packet processor, a set of configuration registers, and global signals that are controlled by the configuration registers. The packet processor controls the flow of data from the configuration interface (ICAP) to the appropriate register. The registers control all other aspects of configuration.

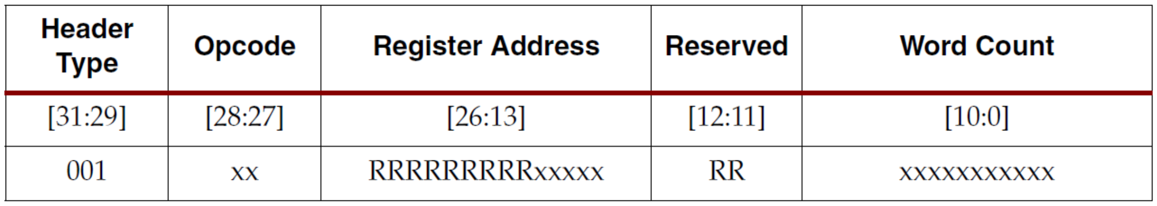
The FPGA bitstream consists of two packet types: Type-1 and Type-2. Type-1 packet is used for register reads and writes. The Type-1 packet header is followed by the Type-1 data section, which is the number of words indicated in the header. The Type-1 packet header format is given in fig. 2.14 and the opcodes for different functions is shown in fig. 2.15 [11].

The Type 2 packet, which must follow a Type 1 packet, is used to write long blocks. No address is presented here because it uses the previous Type 1 packet address. The header section is always a 32-bit word. Following the Type 2 packet header, shown in fig. 2.16, is the Type 2 data section, which contains the number of 32-bit words specified by the word count portion of the header.



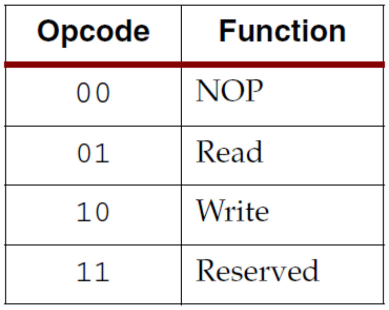
```
ff ff ff ff aa 99 55 66 20 00 00 00 30 00 80 01 00 00 00 07 20 00 00 00 20 00 00 00 30 01 80 01 02 08
80 93 30 00 80 01 00 00 00 01 20 00 00 00 30 00 20 01 00 00 80 40 20 00 00 00 30 00 40 29 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 03 a8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Fig. 2.13: Chunk of partial bitstream.



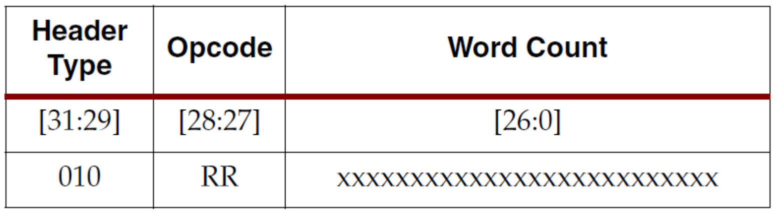
Header Type	Opcode	Register Address	Reserved	Word Count
[31:29]	[28:27]	[26:13]	[12:11]	[10:0]
001	xx	RRRRRRRRRxxxxx	RR	xxxxxxxxxxx

Fig. 2.14: Type-1 packet header format.



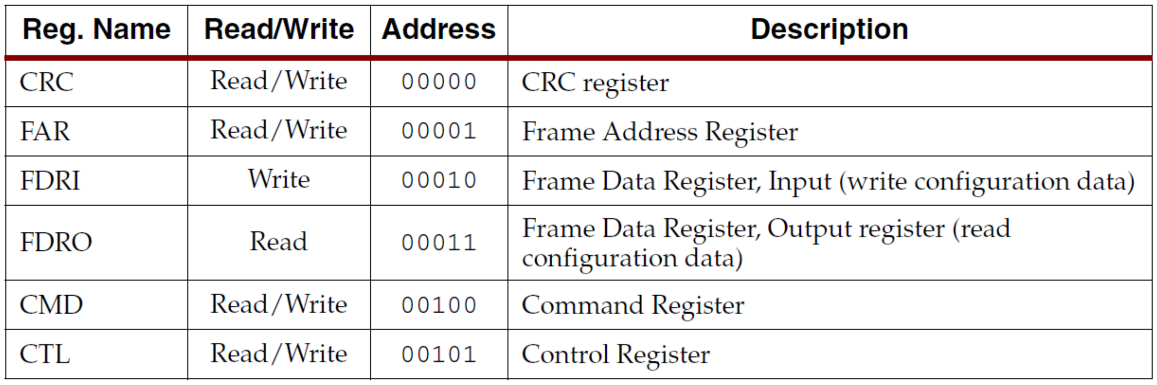
Opcode	Function
00	NOP
01	Read
10	Write
11	Reserved

Fig. 2.15: Opcode format.



Header Type	Opcode	Word Count
[31:29]	[28:27]	[26:0]
010	RR	xxxxxxxxxxxxxxxxxxxxxxxxxxx

Fig. 2.16: Type-2 packet header format.



Reg. Name	Read/Write	Address	Description
CRC	Read/Write	00000	CRC register
FAR	Read/Write	00001	Frame Address Register
FDRI	Write	00010	Frame Data Register, Input (write configuration data)
FDRO	Read	00011	Frame Data Register, Output register (read configuration data)
CMD	Read/Write	00100	Command Register
CTL	Read/Write	00101	Control Register

Fig. 2.17: Configuration registers.

All bitstream commands are executed by reading or writing to the configuration registers. Some of the commonly used registers are listed in fig. 2.17. One can refer to the Virtex-4 configuration guide [11] to get a complete list of registers and detail explanation of each register. One of the registers, FAR is discussed in sec. 2.2.

The partial bitstream, shown in fig. 2.13, is analyzed with respect to packets and the FAR register. "00 00 80 40" is the frame address in the partial bitstream. Let us convert this into binary and then partition it into separate chunks based on FAR as shown in fig. 2.4.

Frame address in binary: 0000 0000 0000 0000 1000 0000 0100 0000

FAR format: 000000000 0 000 00010 00000001 000000

Looking at the 22nd bit (0), we can say that the frame is located in the top half of the chip. Going from left to right, the next three bits (000) gives the block type. So, the frame can belong to either a CLB or IO or clock or DSP units. The next five bits (00010) give the row address. This frame resides in HCLK 2. The next eight bits (00000001) gives the major column address and the frame is located in major column 1, which is a CLB column. The last six bits (000000) is the minor address and the addressing shows that this is the first frame of the major column.

Now, let us take the word following the frame address, "30 00 40 29." This word will be partitioned into format shown in fig. 2.14.

Binary: 0011 0000 0000 0000 0100 0000 0010 1001

Type-1 format: 001 10 00000000000010 00 00000101001

The last three bits (001) is the header type and it indicates that this is a Type-1 packet. The next two bits (10) represent the opcode. According to fig. 2.15, the opcode is "write." The next 11 bits show which register it is writing to, and according to fig. 2.17, it is writing to FDRI. The next two bits are reserved and the last 11 bits gives the number of words it is writing to the register, which is 41, the number of words in one frame.

This bitstream can be modified, i.e., the frame addresses can be identified and can be changed to a different address representing the physical location of another PRR and can be used to configure the second identical PRR. This process is called partial bitstream relocation (PBR) or just bitstream relocation. Using bitstream relocation, one can reduce memory usage on-chip or off-chip as only one bitstream needs to be stored instead of multiple bitstreams for identical PRRs. For the above example, by storing only one bitstream instead of two, there will be a 50% memory savings.

2.5 Literature Review

2.5.1 Partial Dynamic Reconfiguration

An overview of the trends in the field of partial dynamic reconfiguration is provided in Mesquita [12]. Several different tools have been developed to provide a design interface for partial reconfiguration. Partial reconfiguration in Xilinx FPGAs is done using bus-macros which provides an interface to the dynamic sections on the board. Manual designs using the bus macros by hand can be tedious and error-prone. PaDReH [13] is one example of design tool, taking SystemC [14] as an input specification and performing functional validation, partitioning and scheduling, and reconfigurable infrastructure generation steps to derive a reconfigurable bit-stream, ready to be loaded on an FPGA. Caronte [15] takes a somewhat different tool chain approach by utilizing Xilinx EDK (embedded development kit). JPG [16] is a tool which generates partial bitstreams in the partial reconfiguration flow. JPG provides a java interface in order to aid in the partial bitstream generation. FIGARO [17] provides a partial reconfiguration tool for Atmel FPGAs. In Sedcole et al. [18], arbitrary-sized reconfigurable modules are supported through a technique called bit-stream merging, in which regions within a frame can be reconfigured by reading the frame's current bit-stream, merging it with the new bit-stream, then writing the merged bit-stream back to the frame. This allows for partial reconfiguration regions to share bit frames which is not possible with the Xilinx partial reconfiguration flow. In Diessel and Milne [19], they use Circal process algebra as input to the system to derive and generate the reconfigurable modules to be used in the system. Other partial reconfiguration tools perform real-time monitoring and decision making in reconfiguring the blocks. A tool has been developed which monitors current FPGA status and determines how modules should be swapped in and out [20]. It also performs defragmentation by rearranging unused LUTs into contiguous blocks to open up larger areas to be able to be used by the incoming modules. In Tan and Demara [21] and Singhal and Bozorgzadeh [22], they reconfigure only the non-common parts and not reconfigure the common parts such as adders, multiplexers, or register banks. This reduces reconfiguration time, but can introduce extra delay if additional routing is required between components. The current limitation of partial dynamic reconfiguration is the time to reconfigure the dynamic section with the new functionality and co-processor that controls the reconfiguration.

2.5.2 Partial Bitstream Relocation

Techniques for PBR can be classified based on the following five criteria: (a) Location of processor that manipulates the bitstream: on-chip or off-chip; (b) Type of on-chip processor: hardware or software; (c) Bitstream storage for on-chip processing: on-chip BRAMs or off-chip Flash memory; (d) Type of wrapper used to communicate with internal communication access port (ICAP): Xilinx provided hardware ICAP (HWICAP) or a custom wrapper; (e) Type of relocation supported: relocation to identical or non-identical PRRs. Existing works on PBR are analyzed based on these criteria in the next section. PARBIT [23] is one of the earliest tools developed to support PBR. This tool runs on an off-chip processor. It extracts a partial bitstream from a bitstream file and transforms it to be relocated to a new PRR. pBITPOS [24] is one of the earliest tools that can relocate BRAMs and 18×18 multipliers. This tool is similar to PARBIT and targets Virtex II and Virtex II Pro family of FPGAs. REPLICA [25] is a dedicated hardware relocation filter that transforms the bitstream when it is being downloaded from off-chip memory. This approach targets Virtex-E devices, can relocate to identical PRRs, and has no support for relocating PRRs containing BRAMs or 18×18 multipliers. The next version, REPLICA2Pro [26], is similar to REPLICA, but has support for relocating PRRs containing BRAMs and 18×18 multipliers, and targets Virtex II and Virtex II Pro family of FPGAs. While REPLICA is implemented using an additional FPGA device, REPLICA2Pro is implemented on the same FPGA as the one containing source and destination PRRs. REPLICA [25] and REPLICA2Pro [26] use a custom wrapper to communicate with ICAP. BiRF [27] is yet another hardware-based relocation filter that communicates to the ICAP via a custom wrapper. In addition to Virtex II Pro FPGAs, this approach can target Virtex 4 and 5 series of FPGAs. Montminy et al. [28] proposed a software-based approach to perform relocation and use an embedded processor (Microblaze) to modify the partial bitstream. Communication to ICAP is provided via a Xilinx IP core called the hardware internal configuration access port (HWICAP) wrapper. Carver et al. [29] also transform the relocatable bitstream on an embedded Microblaze processor. However, they rely on on-chip BRAM to store a copy of the bitstream. They target Virtex 4 series of FPGAs. Becker et al. [30] proposed a software driver for the HWICAP core, which parses a stored bitstream, identifies and modifies the frame addresses, and relocates it to a destination PRR. This driver is executed on an embedded soft-core processor (Microblaze) of a Xilinx FPGA. Further speed-up in the process of relocation can be obtained with the help of custom circuits in hardware that can communicate directly with the ICAP and perform relocation. This method is novel compared to all of the above techniques as it has the ability to relocate to non-identical regions on a device. They read a bitstream from off-chip flash memory and relocate using software running on an embedded Microblaze processor talking to the HWICAP wrapper. Bitstream relocation techniques to date have focused on reading inactive bitstreams stored in memory, on-chip or off-chip, whose contents are generated for a specific PRR and modified on demand for configuration into a PRR at a different location. All of the above techniques rely on reading a copy of a bitstream residing in memory. Memory requirements are satisfied in two ways: (i) Using on-chip BRAMs, which are limited and expensive; and (ii) Using off-chip memories, which are slow. All the approaches also recalculate the cyclic redundancy check (CRC) value, while performing relocation. While on-chip software-based [30] and hardware-based [27] PBR techniques have been realized, the speed of such techniques is affected by the need to access partial bitstreams from off-chip memory.

Chapter 3
PRR-PRR Dynamic Relocation

The general block diagram of the existing bitstream relocation techniques is shown in fig. 3.1, where a soft processor or a dedicated hardware reads the bitstream from a flash card off-chip or from BRAM on the chip, and modifies the frame addresses in the bitstream to target a new location and writes it to the ICAP, thus configuring the destination PRR.

In this thesis, I present a novel PRR-PRR relocation technique that works directly on frame data bits, and not on partial bitstream like all preceding techniques, thus eliminating the need to store partial bitstreams.

PRR-PRR relocation technique generates source and destination addresses of each frame in the PRRs, reads the frame data from an active PRR (source) in a nonintrusive manner, and writes it to destination PRR, hence eliminating the need to store any partial bitstreams. A discussion of the top-level algorithm for the proposed PRR-PRR relocation technique is followed by a detailed description of an analytical model that is proposed to evaluate the performance of the proposed relocation technique.

3.1 PRR-PRR Relocation Algorithm

When a design residing in a PRR (source) needs to be relocated into another PRR (destination), source and destination addresses of these PRRs are given as inputs to the algorithm; shown in fig. 3.2. The algorithm then analyzes the addresses and generates the first frame address for source and destination regions. These frame addresses are provided to the function, “relocate,” which reads the frame data from the source region stores it in memory and then copies it back to the destination region. This process is repeated until all the frames in the source PRR are read and written to the destination PRR. If the source or the destination PRR is on the other side of the chip (source being in the top half and

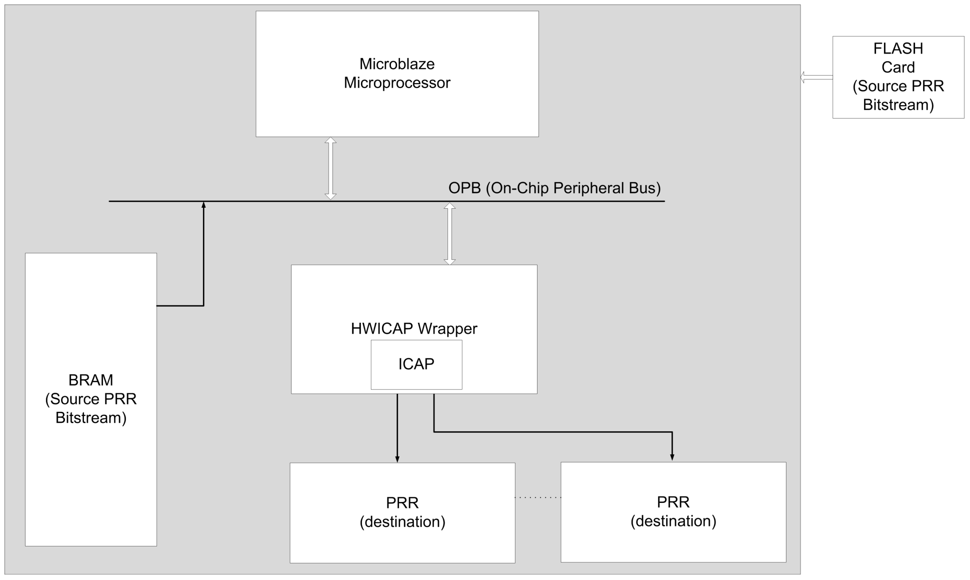


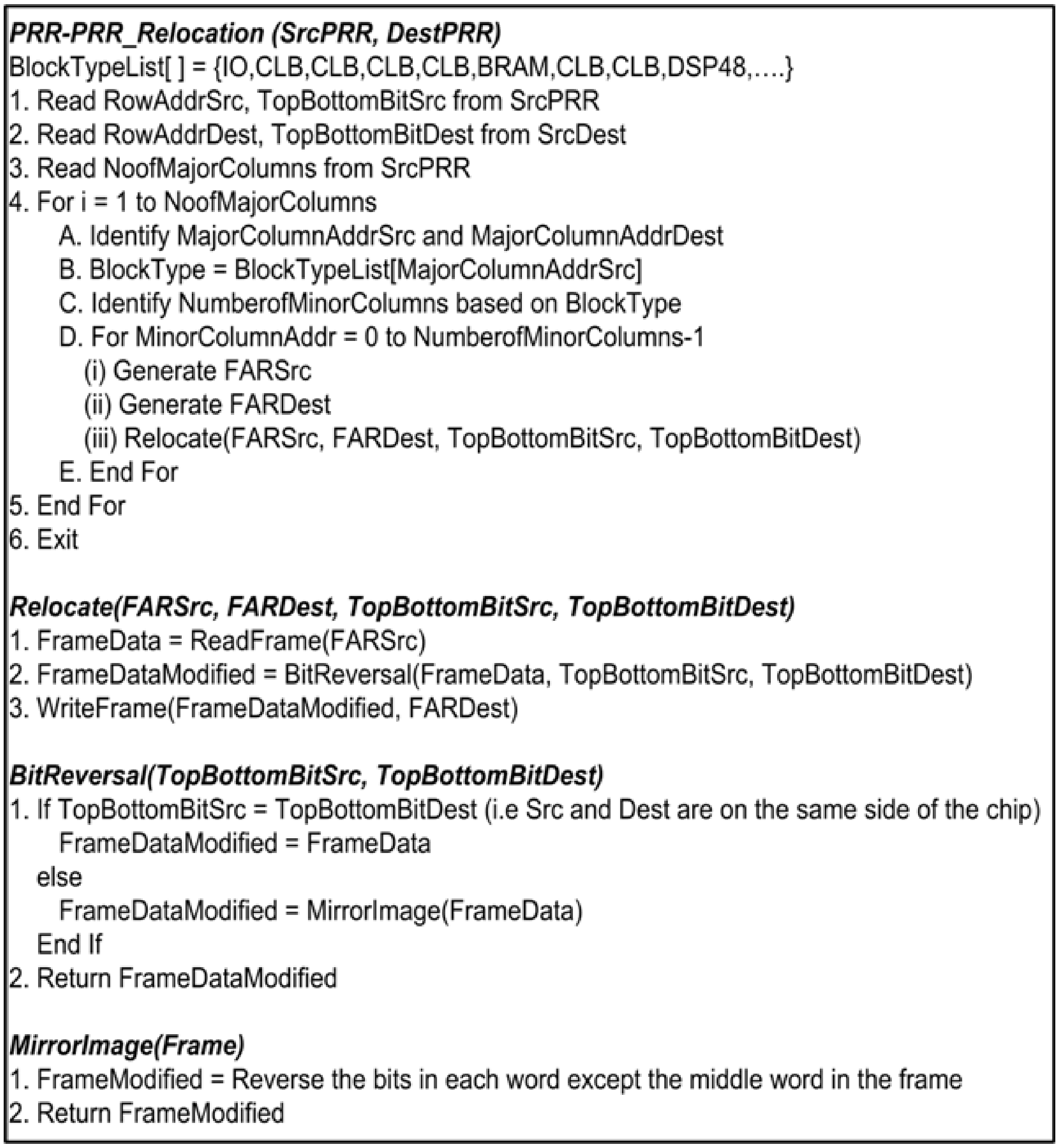
Fig. 3.1: Bitsream relocation technique.

destination being in the bottom half or vice versa), BitReversal will reverse all the bits in the frame to make a mirror image of the frame before storing it in the memory.

3.2 Performance Model

To understand and appreciate the salient features of the novel algorithm, an analytical model is designed that can be used to estimate and analyze its performance for a given partial reconfigurable design on a Virtex 4 SX35 FPGA. In this discussion, time is measured in terms of number of clock cycles and a word represents 32 bits. From fig. 3.2, we observe that the proposed relocation algorithm operates on multiple frames (one frame at a time). Number of frames (nFrames) depends on two factors: (1) Design size, and (2) Generation of PRR using EAPR tool flow from Xilinx. Time to relocate each frame is composed of top three variables listed in Table 3.1. Overall time taken to relocate all the frames in the source PRR is calculated as shown in eq. (3.1).

$$\begin{aligned} T_{Overall} = nFrames \times (T_{readFD} + T_{writeFD} \\ + isOppHalf \times T_{bitReversal}) \end{aligned} \tag{3.1}$$



```
PRR-PRR_Relocation (SrcPRR, DestPRR)
BlockTypeList[ ] = {IO,CLB,CLB,CLB,CLB,BRAM,CLB,CLB,DSP48,….}
1. Read RowAddrSrc, TopBottomBitSrc from SrcPRR
2. Read RowAddrDest, TopBottomBitDest from SrcDest
3. Read NoofMajorColumns from SrcPRR
4. For i = 1 to NoofMajorColumns
     A. Identify MajorColumnAddrSrc and MajorColumnAddrDest
     B. BlockType = BlockTypeList[MajorColumnAddrSrc]
     C. Identify NumberofMinorColumns based on BlockType
     D. For MinorColumnAddr = 0 to NumberofMinorColumns-1
        (i) Generate FARSrc
        (ii) Generate FARDest
        (iii) Relocate(FARSrc, FARDest, TopBottomBitSrc, TopBottomBitDest)
     E. End For
5. End For
6. Exit

Relocate(FARSrc, FARDest, TopBottomBitSrc, TopBottomBitDest)
1. FrameData = ReadFrame(FARSrc)
2. FrameDataModified = BitReversal(FrameData, TopBottomBitSrc, TopBottomBitDest)
3. WriteFrame(FrameDataModified, FARDest)

BitReversal(TopBottomBitSrc, TopBottomBitDest)
1. If TopBottomBitSrc = TopBottomBitDest (i.e Src and Dest are on the same side of the chip)
     FrameDataModified = FrameData
   else
     FrameDataModified = MirrorImage(FrameData)
   End If
2. Return FrameDataModified

MirrorImage(Frame)
1. FrameModified = Reverse the bits in each word except the middle word in the frame
2. Return FrameModified
```

Fig. 3.2: Top-level algorithm of proposed PRR-PRR relocation technique.

Table 3.1: Variables used in the proposed performance model.

Name	Description
T_{readFD}	Time taken to read FD from ICAP
$T_{bitReversal}$	Time taken to reverse bits in case the frame is relocated to the opposite half of the FPGA
$T_{writeFD}$	Time taken to write FD to ICAP
$T_{gen}^{syncRdCmds}$	Time taken to generate set-up commands, and store them in a buffer
$T_{writeICAP}^{syncRdCmds}$	Time taken to write set-up commands to ICAP
$T_{readICAP}^{FD}$	Time taken to read FD from ICAP
$T_{gen}^{desyncCmds}$	Time taken to generate de-synchronization commands, and store them in a buffer
$T_{writeICAP}^{desyncCmds}$	Time taken to write de-synchronization commands to ICAP

Reading FD from ICAP is a three step process. The first step comprises of a sequence of set-up commands to synchronize the ICAP and setting it in "read" mode. The second step is the process of reading the FD from ICAP and storing it in a FIFO buffer. The third step comprises of a sequence of de-synchronization commands to terminate the reading process. Writing data to ICAP follows a similar process, with the only difference being the sequence of set-up commands sent to the ICAP. Time taken to read FD is computed as the sum of the last five variables listed in Table 3.1. Similarly, time taken to write FD can also be computed. It can now be observed that there are three fundamental components of the proposed performance model: T_{gen}^{α}, $T_{writeICAP}^{\beta}$, and $T_{readICAP}^{\gamma}$. Each of these fundamental components (ex., $T_{writeICAP}^{\beta}$) depend on the number of words in the data begin processed (β) and are computed as sum of $T_{overheadW}$ and $T_{write}(x)$. Here, $T_{overheadW}$ is the time taken to write "zero" words to ICAP. In other words, it is the time taken to start writing to the ICAP. $T_{write}(x)$ is the time taken to write x words to the ICAP, where x is the number of words in the data being written to ICAP (β). Both $T_{overheadW}$ and $T_{write}(x)$ depend on the type of implementation and the type of interface used to communicate with ICAP. Similar formulas to compute T_{gen}^{α} and $T_{readICAP}^{\gamma}$ are also derived.

Chapter 4

Implementation

In this thesis, two ways are presented to use active frame data bits as sources of the relocation process. In sec. 4.1, a software-based approach is presented and in sec. 4.2, a hardware-based approach is presented.

4.1 PRR-PRR Software

The algorithm described in the sec. 3.1 was implemented on a soft processor, Xilinx Microblaze, that talks to the ICAP using the Xilinx HWICAP IP core via the OPB, as shown in fig. 4.1. Low-level device drivers are provided by Xilinx to communicate with HWICAP and these drivers are used to read all the frames from the source PRR and write it to an identical destination PRR.

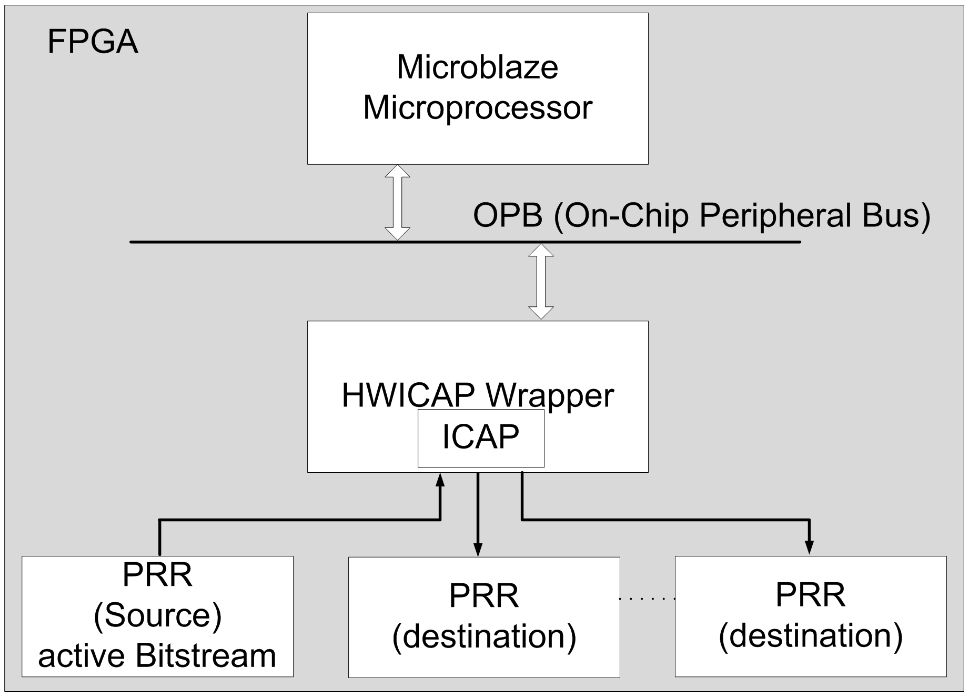

FPGA
Microblaze Microprocessor
OPB (On-Chip Peripheral Bus)
HWICAP Wrapper
ICAP
PRR (Source) active Bitstream
PRR (destination)
PRR (destination)

Fig. 4.1: Software implementation of PRR-PRR algorithm.

The low-level functions provided by Xilinx to read and write one frame is shown in fig. 4.2. For reading one frame, the user needs to specify the frame address by giving all the five components (top/bottom, block type, row address, MajorFrame, and MinorFrame) of the frame address to form the FAR.

These functions are used in a loop to read one frame from the source PRR and write it to the destination PRR until all the frames are relocated. HWICAP has an internal buffer where this frame will be stored from where the write function will read and write it to the ICAP to configure the frame in the destination region.

4.2 PRR-PRR Hardware - Accelerated Relocation Circuit (ARC)

A hardware architecture is designed to realize the PRR-PRR methodology, which is termed, "Accelerated Relocation Circuit (ARC)" after testing the new algorithm in software. The general system architecture is shown in fig. 4.3.

ARC consists of three main components: (1) FAR Generator, (2) Relocator, and (3) ICAP Wrapper. Physical location of the source and destination PRRs on the FPGA are represented using two 16-bit words (SrcPRR and DestPRR). Figure 4.4 shows the formation of the 16-bit address.

The 16-bits are divided into four sections: top/bottom bit (1 bit), row address (5 bits), starting major column (5 bits), and ending major column (5 bits). It is to be noted here

```
XStatus XHwIcap_DeviceReadFrameV4(XHwIcap *InstancePtr, Xint32 Top,
                      Xint32 Block, Xint32 HClkRow,
                      Xint32 MajorFrame, Xint32 MinorFrame)

XStatus XHwIcap_DeviceWriteFrameV4(XHwIcap *InstancePtr, Xint32 Top,
                      Xint32 Block, Xint32 HClkRow,
                       Xint32 MajorFrame, Xint32 MinorFrame)
```

Fig. 4.2: Functions to read and write one frame.

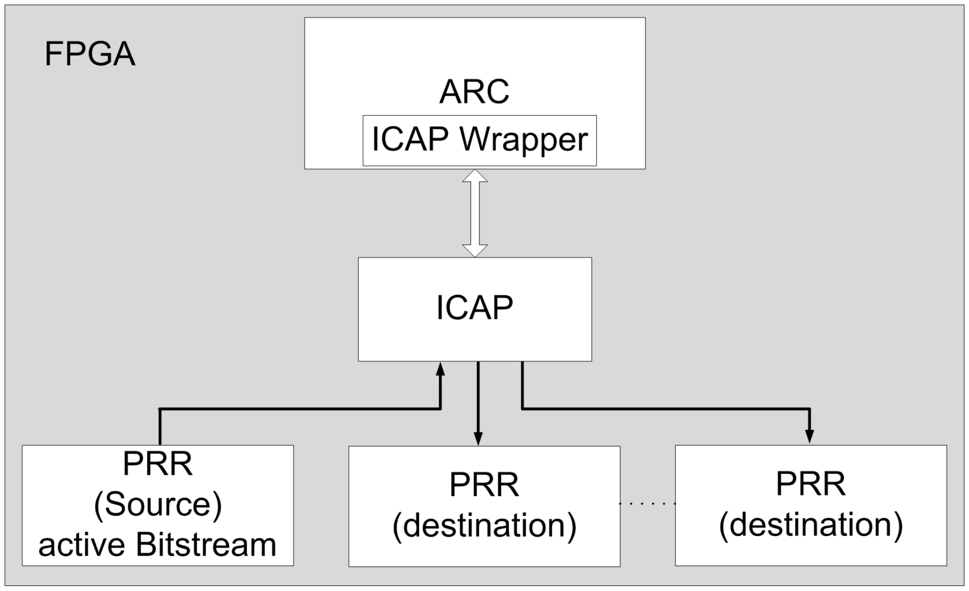


Fig. 4.3: Hardware implementation of PRR-PRR algorithm.

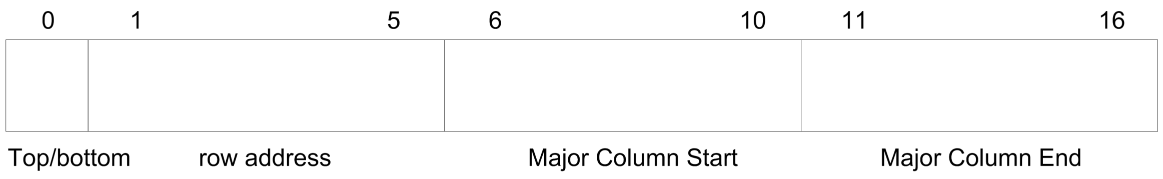


Fig. 4.4: 16-bit address of a PRR on the FPGA.

that the row address does not have a start and end address because ARC can handle only PR regions that fit in one HCLK row.

The top-level block diagram of ARC is shown in fig. 4.5. SrcPRR, DestPRR, and the control signals (reset and go) are received from the Microblaze, or any on-chip soft processor in a Xilinx Virtex 4 FPGA. A subtle advantage of ARC is that the top-level controller logic is fairly simple and can also be realized using a simple state machine instead of software code running on a Microblaze processor. Remainder of this section discusses the sub-modules of ARC.

4.2.1 FAR Generator

The functionality of the FAR Generator is shown in fig. 4.6. It is responsible for decoding SrcPRR and DestPRR and uses the decoded information to generate the complete sequence of frame addresses for the source and destination PRRs. FAR generator executes two instances of the GenerateFAR module to decode SrcPRR and DestPRR and generate FARSrc and FARDest. Upon generation of both FARSrc and FARDest, a control signal ($Relocator_{go}$) is sent to the relocator. Proposed FAR generator is capable of autonomous generation of the complete sequence of FARs for relocating an entire PRR. Information about the type of block (BlockType DSP48,CLB,BRAM) corresponding to a major column address is required for generating an FAR and the sequence of BlockTypes (BlockTypeList) can be derived for any given Virtex 4 FPGA. After generating a single FAR, each instance of the GenerateFAR module waits for the $Relocator_{done}$ signal before generating the next FAR.

4.2.2 Relocator

Proposed architecture for the Relocator module is governed by a state machine and the corresponding state transition diagram is shown in fig. 4.7. Based on the values of FARSrc and FARDest, the Relocator module reads one frame from the source PRR and writes the frame to the destination PRR. Functionality of the relocator module is split into two phases: (i) Readback phase (Read_Done = 0), and (ii) Write phase (Read_Done = 1). During the

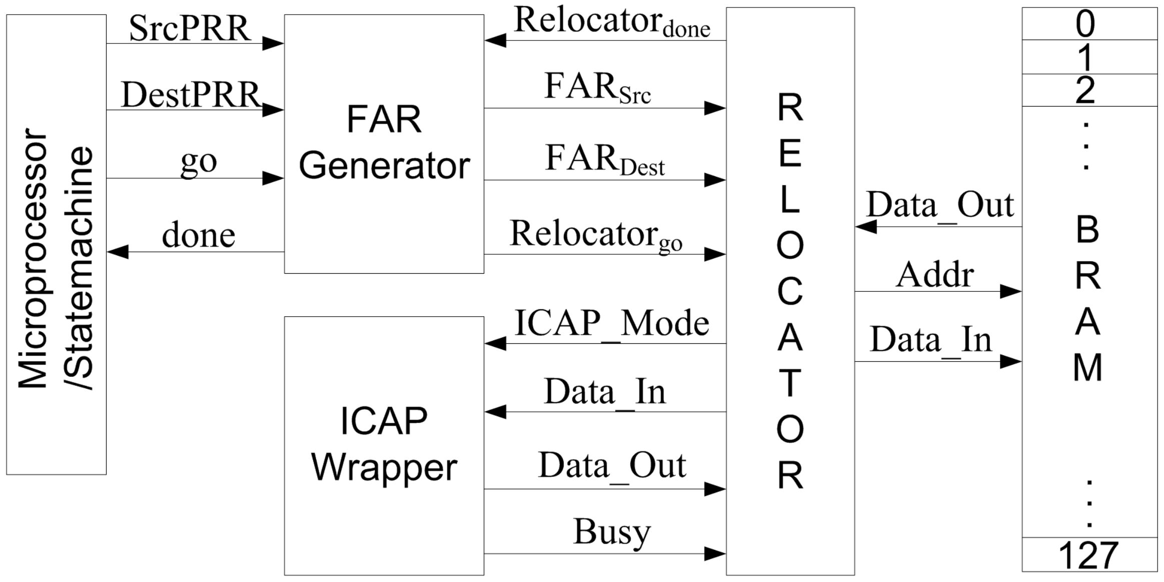


Fig. 4.5: Block diagram of ARC.

readback phase, the relocator module sets the mode of ICAP operation (ICAP_MODE) to "write" and then sends the readback command sequence (RCS) to ICAP. RCS consists of the following: (a) commands to synchronize with the ICAP, (b) command to set the command register (CMD) to read configuration, (c) FARSrc, and (d) number of words to read from ICAP. The RCS to read one frame is shown in fig. 4.8. After sending RCS, the relocator sets the ICAP into "read" mode to read one frame. To read one frame from ICAP, it is required to read a combination of 83 words that includes one dummy word, one pad frame (41 words), and one data frame (41 words) [11]. This combination of 83 words is termed as frame data. A BRAM module is used to temporarily store the FD. The size of this BRAM module is 128 $\times$ 32-bit. After the FD is read, the relocator sets the ICAP_MODE to "write" and sends the de-sync commands to ICAP. The de-sync command sequence is shown in fig. 4.9.

With de-syncing with the ICAP, readback phase is completed and the writing phase begins. In this phase, a write command sequence (WCS), which contains FARDest, is written to the ICAP. The write command sequence is shown in fig. 4.10. FD is fetched from BRAM and data frame and pad frame are written to ICAP. The data frame has to be

```
GenerateFAR (PRR)
BlockTypeList[ ] = {IO,CLB,CLB,CLB,CLB,BRAM,CLB,CLB,DSP48,….}
1. Read RowAddr, TopBottomBit from PRR
2. Read $MajorColumnAddr_{Start}$, $MajorColumnAddr_{End}$ from PRR
3. For MajorColumnAddr = $MajorColumnAddr_{Start}$ to $MajorColumnAddr_{End}$
      A. BlockType = BlockTypeList[MajorColumnAddr]
      B. Identify NumberofMinorColumns based on BlockType
      C. For MinorColumnAddr = 0 to NumberofMinorColumns-1
            (i) Generate FAR using TopBottomBit, BlockType,
                  RowAddr, MajorColumnAddr, and MinorColumnAddr
            (ii) Send FAR to Relocator
            (iii) Wait for $Relocator_{done}$ signal
      D. End For
4. End For
```

Fig. 4.6: FAR generator.

written first and then the pad frame follows. This is the order to be written while writing a frame, hence writing 82 words.

De-sync commands are sent to ICAP, with which the writing phase will be done, after which a $Relocator_{done}$ signal is sent to FAR generator which generates the next pair of FARs. This process is repeated until all frames in source PRR are relocated to destination PRR, after which the FAR generator sends a "done" signal to the Microblaze. Additional processing is required to relocate the design if the source and destination PRRs are located on opposite halves of the chip. Data coming out of ICAP needs to be bit reversed and stored in BRAM as a mirror image to the source frame. In the proposed architecture, this processing is performed on the fly, thereby removing any possible timing overhead at the cost of minimal area overhead (for bit reversal).

4.2.3 ICAP Wrapper

ICAP wrapper instantiates the ICAP primitive which gets its input control signals and input data from the relocator module. The output of this wrapper is sent to the memory (BRAM) to store the information coming out of the ICAP.

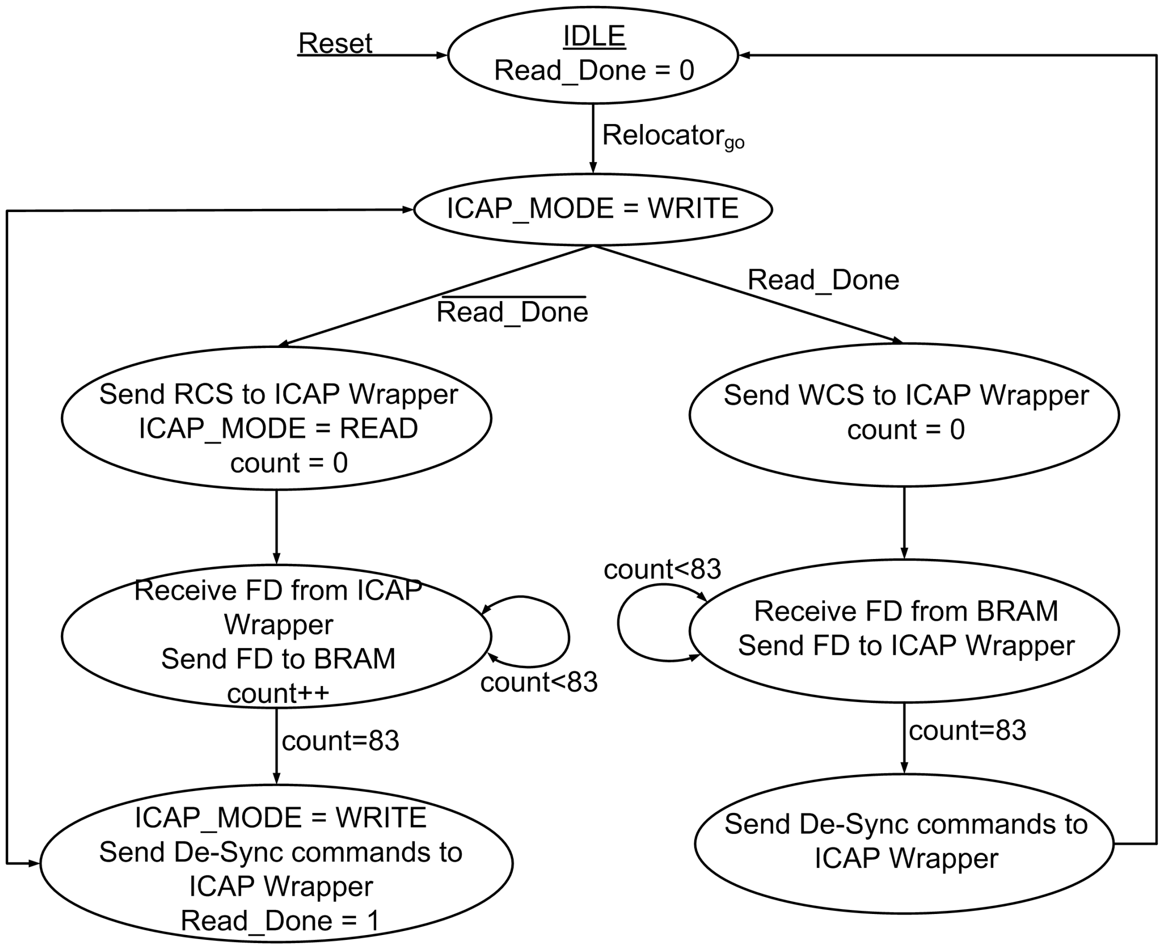


Fig. 4.7: State diagram of the relocator module.

```
FFFFFFFF
AA995566
20000000
20000000
30008001
00000007
20000000
20000000
30008001
00000004
20000000
20000000
30002001
FAR_Src    – provide the source frame address
28006053 – reading 83 words
20000000
20000000
20000000
20000000
20000000
20000000
20000000
```

Fig. 4.8: Read command sequence.

```
30008001
0000000D
20000000
20000000
```

Fig. 4.9: De-Sync command sequence.

```
FFFFFFFF
AA995566
20000000
20000000
30008001
00000007
20000000
20000000
30012001
10042FDD
30018001
02088093
30002001
FAR_Dest   – provide the source frame address
30008001
00000001
20000000
30004052  – writing 82 words
```

Fig. 4.10: Write command sequence.

4.3 Performance Analysis

A comparative performance analysis of the hardware and software implementations of PRR-PRR relocation algorithm is provided here. Performance is estimated using the proposed analytical model for relocating a single frame. Table 4.1 shows a comparative listing (software vs ARC) of the various timing estimates for the variables defined in the proposed model.

At different stages in the relocation process, a sequence of commands is generated. In the software implementation, the commands are generated in sequence and written to a buffer before writing it to ICAP. In hardware, the commands are hardcoded and written directly to ICAP. T_{gen}^{α} values for software implementation are much higher (for different α's). For the software implementation, we observe that there is considerable overhead associated with the process of communicating with ICAP ($T_{overheadW}$ and $T_{overheadR}$). Corresponding numbers for the hardware implementation are much smaller. Once the ICAP is ready, time taken to write (or read) x words is x clock cycles (in case of ARC) and is some function

Table 4.1: Performance analysis of ARC versus software implementation.

Variable name	Number of words	ARC	Software
T^{α}_{gen}	x	1	$f_1(x)$
$T_{overheadW}$	n/a	4	81
$T_{overheadR}$	n/a	4	81
$T_{write}(x)$	x	x	$f_2(x)$
$T_{read}(x)$	x	x	$f_3(x)$
$T^{syncRdCmds}_{gen}$	18	1	691
$T^{syncRdCmds}_{writeICAP}$	18	22	100
$T^{FD}_{readICAP}$	83	87	154
$T^{desyncCmds}_{gen}$	4	1	149
$T^{desyncCmds}_{writeICAP}$	4	8	81
T_{readFD}	n/a	119	1175
$T^{syncWrCmds}_{gen}$	24	1	1024
$T^{syncRdCmds}_{writeICAP}$	18	22	100
$T^{FD}_{writeICAP}$	82	86	260
$T^{desyncCmds}_{gen}$	4	1	149
$T^{desyncCmds}_{writeICAP}$	4	8	81
$T_{writeFD}$	n/a	118	1614
$T_{bitReversal}$	82	0	13310
$T_{overall}$	n/a	237	16099

of x (in case of software). Table 4.1 lists the values for other variables in the performance model and also lists the overall time. In this table, some values are represented as fi(x), which indicates that the value is a function of the number of words (x) and is much larger than x. In case of relocation to opposite half of FPGA, bit-reversal needs to be performed. This is a time consuming process in software as it involves reading the sequence of bits from the frame buffer into a temporary buffer, reversing the bits, and then storing it back into the original buffer. This process involves a large number of sequential memory transactions (in a software implementation) and takes 13310 clock cycles. In hardware, bit-reversal is performed on the fly, and does not require any additional clock cycles. Overall time taken for software is estimated to be 68 times larger than that of ARC.

Chapter 5

Results

The proposed software and hardware (ARC) approaches were implemented and tested to run at 100 MHz on a Virtex 4 SX35 FPGA based ML402 evaluation board [31]. Xilinx ISE tool flow is used to synthesize, map, place, and route the design. Each project is implemented using the EAPR tool flow from Xilinx [8]. The approaches are compared against estimates of performance of BiRF [27] and the relocation approach proposed by Carver et al. [29], based on information available in the respective publications. Test cases used to evaluate the different approaches are of two types, as listed below.

- Dynamically scalable systolic array designs [2]: Number of processing elements (PE) can be increased during runtime, thus requiring the relocation of a single PE design to an empty PRR. In Table 5.1, test cases 1-6 belong to this type.
- Fault tolerant designs [1]: Relocation is required to replace a faulty circuit. In Table 5.1, test case 7 belongs to this type.

Before testing the above test cases, a simple project is implemented to show the relocation performed by ARC. Two designs are taken, an up counter and a down counter. These designs are manually placed on the chip in PlanAhead as shown in fig. 5.1.

The up counter is assigned to PRR A, one of the down counters to PRR B, and the other down counter to PRR C. PRR C is specifically taken to show relocation to the other side of the meridian in the chip. As shown in fig. 5.1, PRR A is located in the top half of the chip in HCLK (row address) 2. The PRR starts at major column 1 and ends at major column 2. Both the major columns in this PRR are CLB columns. So, the 16-bit address for this PRR is obtained as shown in fig. 5.2.

The 16-bit address obtained for PRR A is 0x0822. The addresses for other two PRR regions are obtained in the same manner. First, the source PRR is taken as PRR A, which

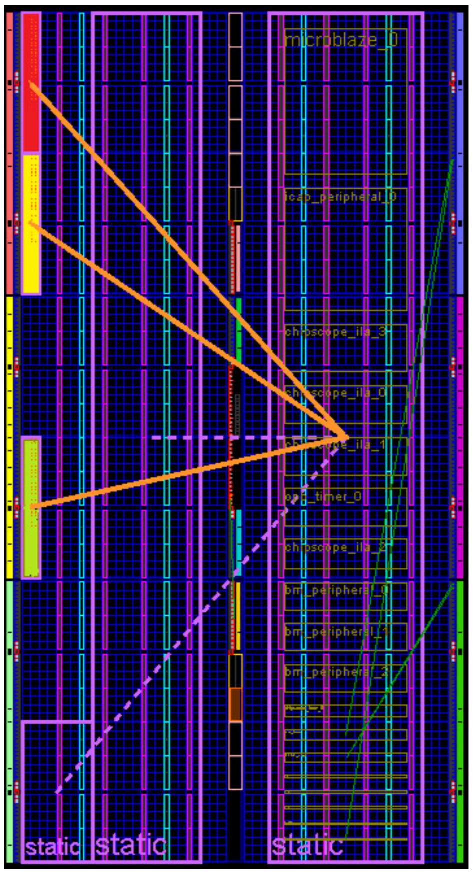


Fig. 5.1: PRRs placed for ARC implementation.

0	1				5	6				10	11				16
0	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0
Top/bottom	row address					Major Column Start					Major Column End				

Fig. 5.2: Address of PRR A.

is the up counter and the destination PRR is taken as PRR B which is the down counter in the top half of the chip. So, the 32-bit address that has to be supplied to ARC will be 0x08220422, 0x0422 being the address of PRR B.

Figure 5.3 shows the waveforms, showing the output of the up counter and down counter. These wave form are obtained using a Xilinx tool called, ChipScope Pro [32], which can be used to tap signals and observe them during run-time, i.e., when operating on the FPGA. The wave form of down counter can be seen distored after a few clock cycles, indicating the relocation ARC is performing by reading frames from PRR A and copying them to PRR B. While relocation, PRR B outputs garbage value as the frames are being changed one by one. Figure 5.4 shows the starting seven clock cycles from which the operating of up counter and down counter can be seen clearly. After the relocation, a reset signal has to be sent to the down counter to reset all the flip flops in the design. After the reset, the down counter in PRR B, which was counting down, starts counting up as shown in fig. 5.5.

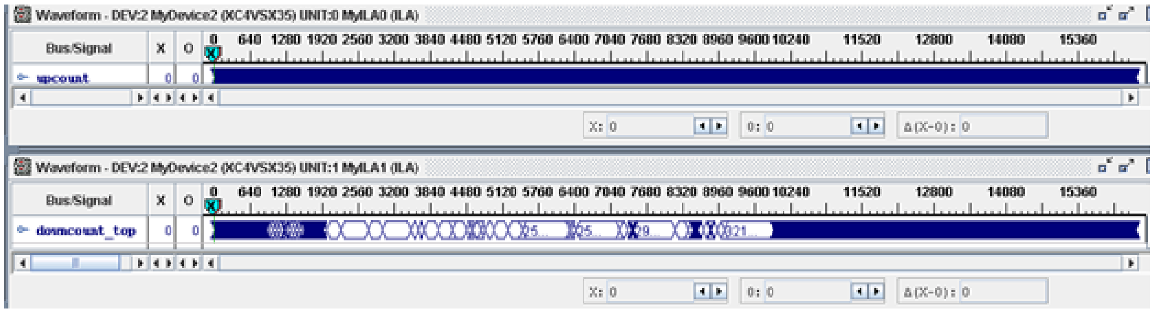

Waveform - DEV:2 MyDevice2 (XC4VSX35) UNIT:0 MyILA0 (ILA)
Bus/Signal
X
O
upcount
Waveform - DEV:2 MyDevice2 (XC4VSX35) UNIT:1 MyILA1 (ILA)
downcount_top
X: 0
O: 0
Δ(X-O): 0

Fig. 5.3: Designs before relocation.

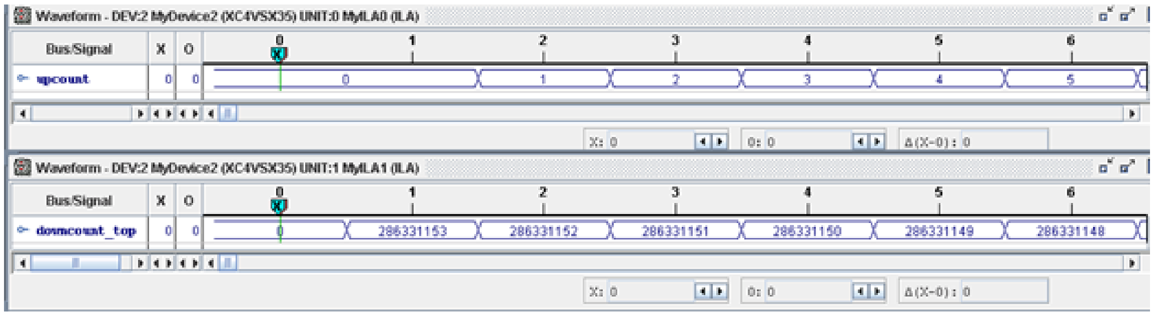

Waveform - DEV:2 MyDevice2 (XC4VSX35) UNIT:0 MyILA0 (ILA)
Bus/Signal
X
O
upcount
0
1
2
3
4
5
Waveform - DEV:2 MyDevice2 (XC4VSX35) UNIT:1 MyILA1 (ILA)
downcount_top
0
286331153
286331152
286331151
286331150
286331149
286331148
X: 0
O: 0
Δ(X-O): 0

Fig. 5.4: Functionality of designs.

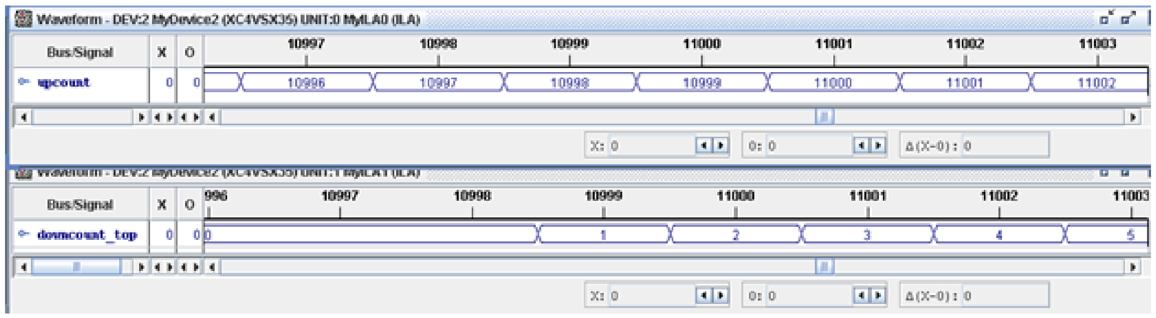


Fig. 5.5: Designs after relocation.

The Resource estimates of ARC and the comparisons are shown in Table 5.2. The footprint shown is on a Virtex 4 SX35 chip. To estimate the number of BRAMs, the microblaze is instantiated with 64KB of memory.

Table 5.1 shows the relocation times for the various test cases. It is observed that the proposed software implementation provides slower relocation times for all test cases when compared with the software implementation proposed by Carver et al. [29].

For relocating to the same half of FPGA, performance of our software implementation is found to be 1.7 times slower (2.53 for relocation to opposite half). But, a major disadvantage of Carver et al. [29] is additional BRAM requirement to store the partial bitstream, as listed in Table 5.2.

Proposed ARC is compared with BiRF [27]. When the source and destination PRRs are on the same half of the FPGA, an average speed up of 153× is observed. This speed-up can be attributed to the fact that BiRF requires off-chip communication to read the bitstream that needs to be relocated. There is insufficient information about relocation to the other half of the FPGA using BiRF, and hence performance cannot be reasonably estimated. We also observed that the difference between estimated performance results (using proposed model) and actual results (using implementation on FPGA) is less than 5% for all test cases.

Proposed relocation algorithm is applicable to any Virtex 4 FPGA as long as source and destination PRRs are floor planned to have identical set of device primitives and routing resources. Accelerating relocation can have a major impact on performance, under

Table 5.1: Time taken (in ms) for relocation using proposed approaches and related approaches. DWT stands for Discrete Wavelet Transform.

Index	Test case	# frames	Bitstream size (bytes)	# BRAMs	PRR-PRR ARC	PRR-PRR software		BiRF [27]	Carver *et al.* [29]	
					Same/Opp half	Same half	Opp half	Same half	Same half	Opp half
Systolic Array PE (not using DSP48)										
1	Faddeev [1]	195	31159	14	0.48	5.77	22.24	84.7	3.38	8.86
2	DWT [1]	195	30693	14	0.48	5.77	22.24	83.4	3.33	8.73
3	Matrix multiplier [1]	432	68469	30	1.07	12.79	49.26	186.1	7.42	19.47
Systolic Array PE (using DSP48)										
4	Faddeev [1]	195	32349	15	0.48	5.77	22.24	87.9	3.5	9.2
5	DWT [1]	195	33045	15	0.48	5.77	22.24	89.8	3.58	9.39
6	Matrix multiplier [1]	432	65261	29	1.07	12.79	49.26	177.3	7.07	18.56
Non Systolic Array (using DSP48)										
7	DWT [2]	303	47897	21	0.75	8.97	34.55	130.2	5.19	13.62

Table 5.2: Resource requirements of ARC and BiRF (Microblaze is instantiated with 64 KB of memory).

Relocation Architecture	LUTs	FFs	BRAMs
ARC (with Microblaze)	2787	1928	33
ARC (with state machine)	1072	686	1
BiRF	2047	1574	32

two conditions: (i) Relocation time is comparable to actual execution time, and (ii) Fast relocation is required to respond to a particular event.

Chapter 6
Conclusions and Future Work

In this thesis, relocation in FPGAs is discussed and the disadvantages with the relocation techniques to date have been observed. This work proposes a novel PRR-PRR relocation algorithm to read frame data directly from an active PRR and relocate it to a destination PRR on the fly, thus accelerating the relocation and removing the need to store any temporary copies of bitstreams. This algorithm is implemented in two ways: (i) a software solution, and (ii) dedicated hardware architecture called the accelerated relocation circuit (ARC). An analytical model is also proposed to evaluate the performance of both software and hardware solutions. Proposed technique is tested on a Virtex 4 SX35 FPGA and performance is compared against two state of the art techniques. An average speed-up of 153$\times$ for ARC over BiRF is observed, with the additional advantage of not storing any bitstreams, thus saving invaluable BRAMs. Accuracy of proposed analytical model was found to be more than 95% for the seven test cases.

The main challenge in this thesis is to understand how the ICAP primitive works. ICAP was used to read and write frames to perform relocation and to find the right sequence of commands to read and write was challenging. Also, ICAP cannot be simulated like other device primitives. This made the debugging the designs very tough until a Xilinx tool, ChipScope Pro, was used to do real-time debugging. This work also required to understand the bitstream through some reverse engineering in order invent this novel PRR-PRR algorithm.

Some of the features that can be added to ARC are given below.

1. ARC can be extended to different Virtex-4 devices. It can also be extended to other families of FPGAs.
2. ARC can be improved to handle PRR's which span more than one HCLK row.

3. Relocation to non-identical regions can be added as a new feature to ARC.

4. The time to read a frame can be optimized by reading only the data frame, and not the pad frame.

5. PDR capability can be added to ARC, which can be used to configure a PRR using a bitstream when the configuration data is not available on the chip.

6. Another solution when the configuration data is not available on the chip, is to read the configuration data off a PRR before wiping it out to configure with a new functionality, store it in BRAM temporarily and configure it back to a PRR when required.

References

[1] A. Sreeramareddy, J. Josiah, A. Akoglu, and A. Stoica, "Scars: Scalable self-configurable architecture for reusable space systems," *Adaptive Hardware and Systems, NASA/ESA Conference*, pp. 204–210, June 2008.

[2] A. Sudarsanam, R. Barnes, J. Carver, R. Kallam, and A. Dasu, "Dynamically reconfigurable systolic array accelerators: A case study with extended kalman filter and discrete wavelet transform algorithms," *Computers Digital Techniques, Institution of Engineering and Technology*, vol. 4, no. 2, pp. 126–142, Mar. 2010.

[3] Xilinx, "Xilinx website," `[http://www.xilinx.com/]`, 2010.

[4] Xilinx, "Fpga vs. asic," `[http://www.xilinx.com/company/gettingstarted/fpgavsasic.htm]`, 2010.

[5] Xilinx, "Microblaze soft processor core," `[http://www.xilinx.com/tools/microblaze.htm]`, 2010.

[6] M. Wirthlin and B. Hutchings, "Improving functional density using run-time circuit reconfiguration [fpgas]," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions*, vol. 6, no. 2, pp. 247–256, June 1998.

[7] C. Kao, "Benefits of partial reconfiguration," *Xcell Journal*, pp. 65–76, 2005.

[8] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Invited paper: Enhanced architectures, design methodologies and cad tools for dynamic reconfiguration of xilinx fpgas," *Field Programmable Logic and Applications, International Conference*, pp. 1–6, Aug. 2006.

[9] Xilinx, "Planahead tutorial," `[http://www.xilinx.com/support/documentation/sw_manuals/PlanAhead10-1_Tutorial.pdf]`, 2008.

[10] Xilinx, "Bitgen," `[http://www.xilinx.com/itp/xilinx6/books/data/docs/dev/dev0120_18.html]`, 2006.

[11] Xilinx, "Virtex-4 configuration guide," `[http://www.xilinx.com/support/documentation/user_guides/ug071.pdf]`, 2009.

[12] D. Mesquita, F. Moraes, J. Palma, L. Moller, and N. Calazans, "Remote and partial reconfiguration of fpgas: tools and trends," *Proceedings of Parallel and Distributed Processing Symposium*, Apr. 2003.

[13] E. Carvalho, N. Calazans, E. Briao, and F. Moraes, "Padreh - a framework for the design and implementation of dynamically and partially reconfigurable systems," *Integrated Circuits and Systems Design, SBCCI, 17th Symposium*, pp. 10–15, Sept. 2004.

[14] T. Grotker, *System Design with SystemC*. Norwell, Massachusetts: Kluwer Academic Publishers, 2002.

[15] F. Ferrandi, M. Santambrogio, and D. Sciuto, “A design methodology for dynamic reconfiguration: the caronte architecture,” *Proceedings of Parallel and Distributed Processing, 19th IEEE International Symposium*, p. 4, Apr. 2005.

[16] A. Raghavan and P. Sutton, “Jpg - a partial bitstream generation tool to support partial reconfiguration in virtex fpgas,” *Proceedings of Parallel and Distributed Processing Symposium, Abstracts and CD-ROM*, pp. 155–160, Apr. 2002.

[17] K. Nasi, T. Karouhalis, M. Danek, and Z. Pohl, “Figaro - an automatic tool flow for designs with dynamic reconfiguration,” *Field Programmable Logic and Applications, International Conference*, pp. 590–593, Aug. 2005.

[18] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght, “Modular dynamic reconfiguration in virtex fpgas,” *Computers and Digital Techniques, IEE Proceedings*, vol. 153, no. 3, pp. 157–164, May 2006.

[19] O. Diessel and G. Milne, “Hardware compiler realising concurrent processes in reconfigurable logic,” *Computers and Digital Techniques, IEE Proceedings*, vol. 148, no. 45, pp. 152–162, July/Sept. 2001.

[20] M. Gericota, G. Alves, M. Silva, and J. Ferreira, “Run-time management of logic resources on reconfigurable systems,” *Design, Automation and Test in Europe Conference and Exhibition*, pp. 974–979, 2003.

[21] H. Tan and R. F. DeMara, “A physical resource management approach to minimizing fpga partial reconfiguration overhead,” *Reconfigurable Computing and FPGA's, IEEE International Conference*, pp. 1–5, Sept. 2006.

[22] L. Singhal and E. Bozorgzadeh, “Physically-aware exploitation of component reuse in a partially reconfigurable architecture,” *Parallel and Distributed Processing, 20th International Symposium*, p. 8, Apr. 2006.

[23] E. Horta, J. Lockwood, D. Taylor, and D. Parlour, “Dynamic hardware plugins in an fpga with partial run-time reconfiguration,” *Proceedings of Design Automation Conference*, pp. 343–348, 2002.

[24] Y. Krasteva, E. de la Torre, T. Riesgo, and D. Joly, “Virtex ii fpga bitstream manipulation: Application to reconfiguration control systems,” *Field Programmable Logic and Applications, International Conference*, pp. 1–4, Aug. 2006.

[25] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert, “Replica: A bitstream manipulation filter for module relocation in partial reconfigurable systems,” *Proceedings of Parallel and Distributed Processing, 19th IEEE International Symposium*, pp. 151b–151b, Apr. 2005.

[26] H. Kalte and M. Porrmann, “Replica2pro: task relocation by bitstream manipulation in virtex-ii/pro fpgas,” *Proceedings of the 3rd Conference on Computing Frontiers*, pp. 403–412, 2006.

[27] S. Corbetta, M. Morandi, M. Novati, M. Santambrogio, D. Sciuto, and P. Spoletini, "Internal and external bitstream relocation for partial dynamic reconfiguration," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions*, vol. 17, no. 11, pp. 1650–1654, Nov. 2009.

[28] D. Montminy, R. Baldwin, P. Williams, and B. Mullins, "Using relocatable bitstreams for fault tolerance," *Adaptive Hardware and Systems, Second NASA/ESA Conference*, pp. 701–708, Aug. 2007.

[29] J. Carver, R. Pittman, and A. Forin, "Relocation and automatic floor-planning of fpga partial reconfiguration bitstreams," *Microsoft Research, WA, Technical Report no. MSR-TR-2008-111*, Aug. 2008.

[30] T. Becker, W. Luk, and P. Cheung, "Enhancing relocatability of partial bitstreams for run-time reconfiguration," *Field Programmable Custom Computing Machines, 15th Annual IEEE Symposium*, pp. 35–44, Apr. 2007.

[31] Xilinx, "Ml40x getting started tutorial," `[http://www.xilinx.com/support/documentation/boards_and_kits/ug083.pdf]`, 2006.

[32] Xilinx, "Chipscope pro software and cores user guide," `[http://www.xilinx.com/support/documentation/sw_manuals/chipscope_pro_sw_cores_9_1i_ug029.pdf]`, 2007.