A SYSTEMATIC APPROACH FOR REGISTER FILE DESIGN IN FPGAs

by

Hasan Erdem Yantır

B.S., Computer Engineering, Yeditepe University, 2011

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering
Boğaziçi University
2014

A SYSTEMATIC APPROACH FOR REGISTER FILE DESIGN IN FPGAs

APPROVED BY:

Assoc. Prof. Arda Yurdakul ....
(Thesis Supervisor)

Assoc. Prof. Alper Şen ....

Prof. Smail Niar ....

DATE OF APPROVAL: 16.01.2014

ACKNOWLEDGEMENTS

Firstly, I would like to thank my supervisor Assoc Prof. Arda Yurdakul for his support, encouraging comments and guidance during the past two years it has taken me to finalize this thesis. I still have to learn about writing and researching effectively.

I am so grateful to Prof. Smail Niar for his contributions on thesis content. His advices were also very valuable for my future academic and social life. I would also like to thank Assoc. Prof. Alper Şen who answered my questions and gave me essential bits of information as I learnt in his courses.

I would want to thank TÜBİTAK (Türkiye Bilimsel ve Teknolojik Araştırma Kurumu) for their financial support (BİDEB-2211 Fellowship) during my graduate education.

I would like to thank Salih Bayar for his help, his support and spending his time to answer my questions. I would also like to thank Gökçehan Kara, Görker Alp Malazgirt and other CASLAB team members for their help, support, friendship and motivation for this thesis.

And finally to my parents and newly born nephews. They didn't make any technical contribution even so they were always there and in my mind when I need moral contribution.

ABSTRACT

A SYSTEMATIC APPROACH FOR REGISTER FILE DESIGN IN FPGAs

For the future of computing, wide usage of heterogeneous and parallel architectures is indispensable since advances in technology scaling cannot satisfy the expected increase in performance of computational platforms anymore. FPGA is a promising platform for such computing systems due to its configurable structure. Each part of an FPGA can be configured to perform a different task that it is best suited for. Multi-port and fast register files are very essential for this type of data intensive computational systems. Otherwise, available computational power cannot be utilized properly. When the characteristics of processing elements are different, such a system needs a heterogeneous register file (RF) that can serve different parts of the FPGA with different characteristics in terms of running frequency, data consumption/production rate, required number of ports, data widths, address spaces and endianness. In this dissertation, we firstly propose a new multi-port RF design which exploits the banking and replication of BRAMs with efficient shift register based multi-pumping (SR-MPu) approach. We also model this register file for the use of HLS tools. Finally, we propose a heterogeneous register file (HRF) architecture for FPGA-based heterogeneous systems. In this RF, world length and address spaces of the processing elements are adjustable. For the power and area reduction, the design takes advantage of frequency differences between processing elements by an efficient multi-pumping system. According to the literature, this is the first study on FPGA-based heterogeneous RFs. Experimental results show that both RF architectures outperform conventional RFs.

ÖZET

YAZMAÇ DOSYALARI İÇİN SİSTEMATİK TASARIM YAKLAŞIMI

Günümüz teknolojisindeki gelişmeler ölçeklenebilirlik alanındaki beklenen verim artışını karşılayamadığından dolayı geleceğin hesaplama sistemleri için heterojen ve eş zamanlı mimarilerin yaygın kullanımı kaçınılmazdır. Yeniden betimlenebilir yapısı sayesinde alanda programlanabilir kapı dizileri (APKD=FPGA) bu tür mimariler için gelecek vaat eden bir altyapıdır. APKD'nin her bir birimi kendisine uygun olan farklı bir görevi yapmak üzere betimlenebilir. Bu tür veri bağımlı hesaplama mimarileri için çoklu giriş-çıkışlı ve hızlı yazmaç dosyası olmadıugı takdirde mevcut hesaplama gücü israf edilmiş olur. Her bir işlem biriminin nitelikleri farklı olduğunda ise, bu mimari çalışma hızı, veri üretim/tüketim oranı, ihtiyaç duyulan port sayısı, veri yolu genişliği, adres aralığı, bitlerin dizilimi gibi farklı nitelikler açısından APKD'nin her bir ünitesine hizmet verebilecek bir heterojen yazmaç dosyasına ihtiyaç duyulmaktadır. Bu tezde, çoklu giriş-çıkışlı yazmaç dosyası tasarımı önerilmiştir. Bu yazmaç dosyası, APKD içinde bulunan rastlantısal erişim hafızasının kümeleme ve kopyalama yöntemlerinin verimli kaydırma yazmacı tabanlı çoklu pompalanma yöntemiyle tasarlanmıştır. Sayısal tasarıma ek, bu yazmaç dosyası yüksek seviyeli sentez araçları için modellenmiştir. Bu çalışma daha da ilerletilerek, APKD tabanlı heterojen sistemler için heterojen yazmaç dosyası tasarlanmıştır. Bu yazmaç dosyasında, işlem ünitelerinin veri genişiliği ve adres aralıkları ayarlanabilmektedir. Harcanan güç ve alan miktarlarının azaltılması için, çoklu pompalama yönteminden faydalanılmıştır. Bu yöntem işlemci ile yazmaç dosyası arasndaki hız farklarını kullanmaktadır. Bu çalışma APKD tabanlı heterojen yazmaç dosyaları alanındaki ilk çalışmadır. Deney sonuçlarına göre her iki yazmaç dosyası mimarisi de geleneksel mimarilere göre daha yüksek başarımlıdır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS/ABBREVIATIONS

ALM	Adaptive Logic Module
ASIP	Application Specific Instruction Set Processor
ASIC	Application Specific Integrated Circuit
BRAM	Block RAM
CLB	Configurable Logic Block
CMOS	Complementary MetalOxideSemiconductor
CPU	Central Processing Unit
DCM	Digital Clock Manager
DSP	Digital Signal Processing
EDA	Electronic Design Automation
ERF	Emulated Register File
ESB	Embedded System Block
FF	Flip Flop
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HLS	High Level Synthesis
HRF	Heterogeneous Register File
IC	Integrated Circuit
IOB	Input Output Block
LAB	Logic Array Blocks
LSB	Least Significant Bit
LUT	Look Up Table
MPo-RF	Multi-port Register File
MPoMPu-RF	Multi-port Multi-pump Register File
MPoSPu-RF	Multi-port Single-pump Register File
MPu-RF	Multi-pumped Register File
MSB	Most Significant Bit
MUX	Multiplexer
PE	Processing Element

PISO	Parallel Input Single Output
PLL	Phase Locked Loop
RAM	Random Access Memory
RF	Register File
RH(+)	Hardware Software Co-design on Reconfigurable Hardware
RTL	Register Transfer Level
SIPO	Single Input Parallel Output
SPo-RF	Single-port Register File
SPoMPu-RF	Single-port Multi-pump Register File
SPoSPu-RF	Single-port Single-pump Register File
SR	Shift Register
TCL	Tool Command Language
UCF	User Constraint File
VHDL	VHSIC Hardware Description Language
VLIW	Very Long Instruction Word
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integrated Circuit
XST	Xilinx Synthesis Technology

1. INTRODUCTION

1.1. Motivation

In the past, performance of computational platforms could be increased with technology scaling without an architectural change. However the technology advance is not sufficient anymore and wide usage of heterogeneous and parallel architectures is envisaged for the future computing systems. Technology trend shifts towards heterogeneous and parallel platforms on both embedded systems and personal computers. Field-programmable gate arrays (FPGA) are great candidates of heterogeneous and parallel platforms for the implementation of such complex applications because of their fully reconfigurable architectures. Each part of an FPGA can be configured to perform a different task that it is best suited for. FPGAs also have built-in dedicated cores such as hardware multipliers, memory blocks, digital signal processing (DSP) blocks, digital clock managers (DCMs) and even dual-core processors [2] to achieve more speed-up in data intensive applications. Since FPGAs are almost fully reconfigurable by their nature, different types of digital circuits can be designed and combined at any time. Nowadays, most embedded applications require partitioning the functionality between processing elements with different characteristics and each of the processing elements may process more data per unit time to be more efficient. Each processing element works on a different set of data and sometimes they require processing the same data set. In order to exploit a greater amount of instruction-level parallelism (ILP) where processing element (PE) can execute multiple instruction simultaneously, PEs have to get the required data in a parallel manner. As FPGAs facilitate more parallel processing, necessity for higher data processing rate takes on greater importance.

One of the obstacles in the way of obtaining high performance in computing is the memory-wall [3]. If the processing elements cannot get the data from register file (RF) at the processing rate, this causes a bottleneck that adversely affects the overall performance. In order to meet the requirement of proper data usage between the computational units, such a computation system needs a register file that can meet the requirements of different computing units on the FPGA. The demand to process more data per unit time requires multiple read and write operations at a time, which can be achieved by the usage of multi-port register files (MPo-RFs) instead of conventional single-port RFs (SPo-RF). For example, a four-issue very long instruction word (VLIW) processor implementation inside an FPGA requires at least 8 read and 4 write ports if an instruction consists of 2 read and 1 write operations as it is in [4]. As the issue-width of a VLIW processor increases, the number of read and write ports on the RF have to increase accordingly to meet the data rate requirements.

Although increasing port number is an efficient method for homogeneous processing systems where features of all processing elements are in coherence, this is not enough for heterogeneous computing systems exclusively hence requirements of processing elements can vary depending on the PE. In order to meet the requirement of proper data usage between the heterogeneous computational units, such a heterogeneous system needs a heterogeneous register file (HRF) that can meet the requirements of different computing units on the FPGA. These requirements can be specified in terms of execution frequency, data consumption/production rate, data width, address space, number of ports, bandwidth and endianness (order of bytes/bits in memory).

1.2. Research Goals

This dissertation focuses on the systematic design approach for efficient and functional register file implementations in FPGA-based computation systems. In fact, the term *memory* could be used in place of *register file* during the dissertation. However we prefer the register file term since implementation is inside the FPGA. We accomplish this systematic design through the following successive goals:

- To propose a new multi-pumping approach for the design of multi-pumped multi-port RFs files in FPGAs. This approach makes the operation frequency of the register file nearly independent from MPuF.
- To combine non-multi-pumped multi-port and the new shift register based multi-pumping approach to create an “emulated multi-port”(ERF) register file that can

operate at higher frequencies and occupy fewer resources.

- To develop an emulated register file model for high level synthesis (HLS) tools.
- To propose a HRF for heterogeneous systems on a single FPGA by using emulated register file designed at first.
- To form an register file synthesis framework for HLS tools that can generate register files for wide-range FPGA designs such as single/multi core soft processors, application specific instruction set processors (ASIP), VLIWs processors, custom processors and even heterogeneous computing systems.

1.3. Organization

This dissertation is organized as follows: the following chapter reviews FPGA structure, multi-pumping methodology and conventional multi-port register file implementations in FPGAs. Register file designs are also classified in Chapter 2. Chapter 3 overviews the existing studies on multi-port register file designs in the literature. Chapter 4 gives methods of designing MPo-RFs by using simple-port block RAMs (BRAM) in FPGAs and explains how shift register based multi-pumping can be used with multi-ported register files so as to obtain smaller register files with increased number of ports. This chapter also compares the performance of different RF implementations. Chapter 5 exploits the modelling of register files designed in Chapter 4 for HLS tools. Chapter 6 explains the heterogeneous register file architecture in an orderly fashion. This section details the usage of multi-pumping on processing elements belonging to different clock domains. The ultimate architecture of HRF also becomes clear in this section. Chapter 7 shows the methods in automatic register file generation by Tool Command Language (TCL). Chapter 8 summarizes the dissertation.

2. BACKGROUND

2.1. FPGA

Reconfigurable technology is represented by FPGAs. FPGAs are programmable chips that can be configured by the end-user to implement any digital circuit that an application specific integrated circuit (ASIC) could perform. The FPGA configuration is generally specified using an hardware description language (HDL). There are two mainly used HDLs which are very high speed integrated circuit HDL (VHDL) and Verilog. With an FPGA and HDL, high performance processors and digital systems can be designed with little investment using a software-like methodology.

Although a central processing unit (CPU) can run at much higher frequency than an FPGA runs, the mass fine-grain parallelism inside an FPGA can be utilized on an application specific basis to yield sometimes orders of magnitude greater performance than additional high performance processors hence processors execute a set of instructions in a sequential manner. Therefore it is possible to build high performance multi-core processors using FPGAs, which avoids non-recurring expenses associated with chip fabrication. As specified in Figure 2.1, FPGAs consist of a two-dimensional configurable logic block (CLB) arrays, programmable interconnection networks, storage structures and hard blocks that implement particular digital, analog or mixed functions directly in complementary metal-oxide-semiconductor (CMOS) logic like digital clock managers (DCM), phase-locked loops (PLL), DSP cores, processors etc. In Altera, Logic Array Blocks (LABs) and Adaptive Logic Modules (ALMs) are correspondences of CLBs and slices respectively in Xilinx. The following subsections points out the main parts of an FPGA architecture that are relevant of our study.

2.1.1. Configurable Logic Blocks (CLBs)

Each CLB includes different number of slices depending on the FPGA vendor and model. Slices are the smallest configurable part that can implement a sequential or

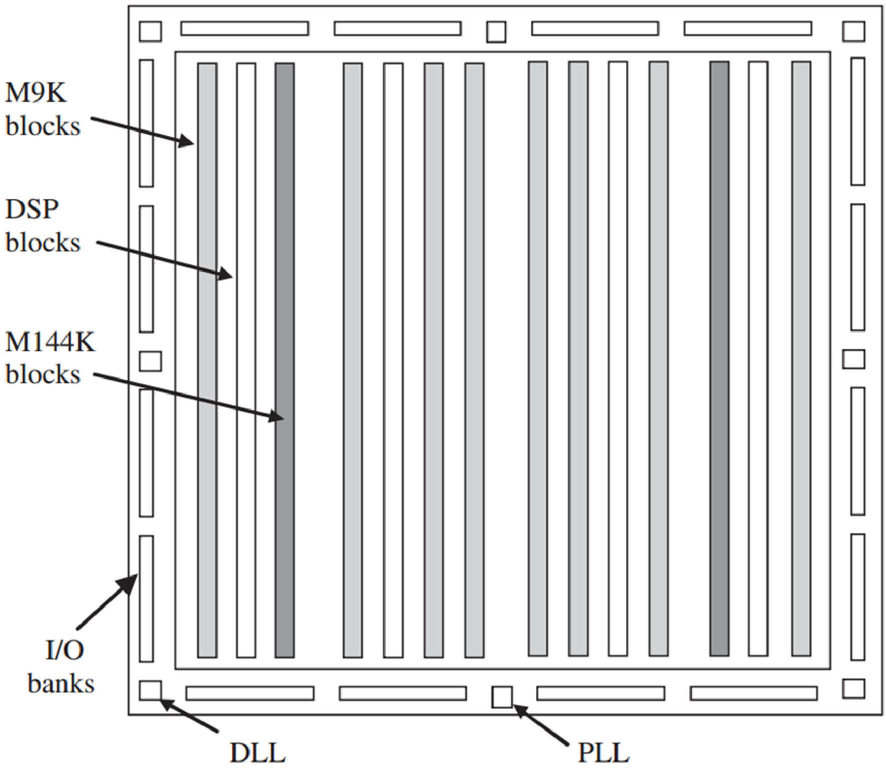


Figure 2.1. Floorplan of Altera Stratix III FPGA [1].

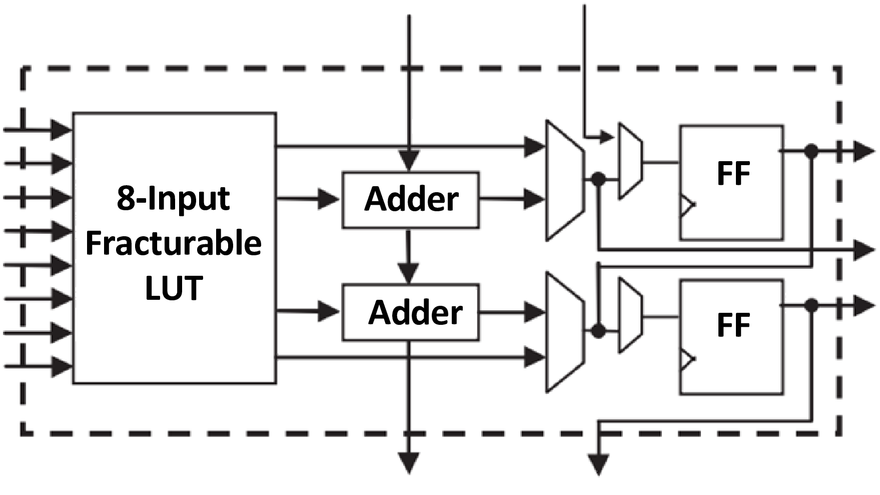


Figure 2.2. Block diagram of Altera Stratix III ALM [1].

combinational function. Another name of the slices is logic cell. One logic cell consists of look-up tables (LUTs), multiplexers (MUXs) and flip flops (FFs). All slices/logic cells are entitled with their locations in the form of XY coordinates e.g. name of the first slice at the leftmost bottom corner is X0Y0. There are different types of slices and corresponding functionality of each slices varies. For example, while SLICEM can be used as memory, shift registers and logical, SLICEL can be used only for logical operations. In an FPGA, slices can be used for storage. Inside the slices, look up tables are used to hold the data and multiplexers are implemented to function as steer logic (address decoding). The maximum size of data that can be stored in one SLICEM may be changed depending on FPGA family.

2.1.2. Interconnection

In an FPGA, each logic cell can do a little e.g. up to 6-input logical operations in Virtex-5, however when many of the logic cells are connected together, complex logic functions and memory structures can be created. The connections between different CLBs and slices are handled by the programmable switching matrixes. Switching matrixes include wires/muxes placed around the logic cells. Switching matrix can be programmed by writing sequence of 1s or 0s to their memory locations and depending on these values switching matrix opens or closes a path.

2.1.3. Memory Components

In an FPGA, a register file or memory can be implemented by using two components: Block RAMs or Distributed Slices.

2.1.3.1. Block RAMs. BRAMs are implemented directly in CMOS logic and have only two ports that can each function either as a read or a write port depending on the configuration. Because of directly CMOS implementations, BRAMs can run at a higher frequencies than ones created by slices. Altera FPGAs has two type of BRAMs which are M9K and M144K. As understood from the naming conventions, M9K can hold 9 kb data and M144K can hold 144 kb in various widths and depths. In Altera, there is another storage structure between BRAMs and ALM implementations called Memory Logic Array Block (MLAB). They are widely distributed like ALMs and implements memory in a denser manner like BRAMs. MLABs are useful for small memories, shift registers, FIFO buffers. In Virtex-5, a BRAM can operate at up to 550 MHz at the fastest speed grade (-3) [5]. Almost in all FPGA families, BRAMs are dual-ported likely due to low hardware cost. One of the exceptional case is Altera Mercury device family which includes a simple quad-port capable embedded system block(ESB) RAM [6]. This architecture has built for high bandwidth in data processing operations.

2.1.3.2. Distributed Slices. A register file can be implemented by using FFs, LUTs and multiplexers inside an slice. Many slices can be combined to form a register file with specified features. This methodology is called as distributed implementation since slices are distributed through the FPGA.

2.1.4. Clock Management

In contemporary FPGAs, there exist dedicated devices to manage the clocks. They provide very flexible, high performance clocking to different processing elements inside the FPGA. For a heterogeneous computing system consisting of different processing elements running at varying frequencies, clock management is crucial. In general, there are two devices to manage clocks inside an FPGA: DCM and PLL.

2.1.4.1. Digital Clock Manager (DCM). An FPGA-based system requires phase aligned clocks distributed through the FPGA. However there may be phase differences between the clocks of processing elements because of different length routings. DCMs are used to eliminate clock distribution delays by clock deskew. DCM can also be used for frequency division, multiplication, phase shifting. In Chapters 4 and 6, we will show the utilization of DCMs to multiply clocks and eliminate phase differences.

2.1.4.2. Phase Locked Loop (PLL). The main purpose of PLLs is to serve as a frequency synthesizer for a wider range of frequencies than DCM suppors. It is more advanced than DCM. It is also used for jitter eliminator for external or internal clocks.

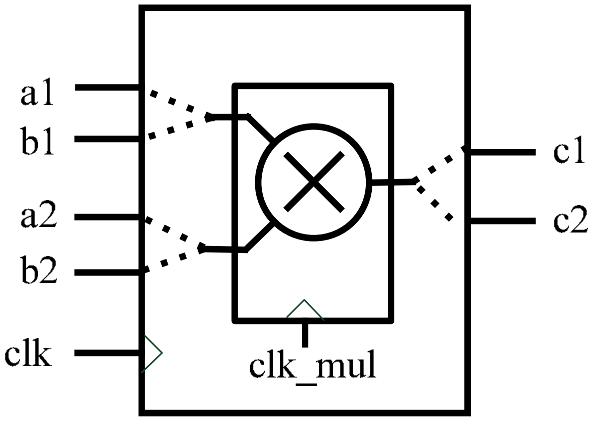


Figure 2.3. A basic multi-pumped multiplication circuit.

2.2. Multi-Pumping

Multi-pumping (MPu) is a method used in digital circuits. This method means illustrating one resource as if there exist multiple replications of this resource. In a multi-pumped design, circuits behave as if they have multiple replicas of themselves, while the internal multi-pumped circuit frequency is by the times of external operating system frequency. Thus, a multi-pumped circuit is able to make multiple operations in one clock cycle time of the outer circuit. DDR2 and DDR3 synchronous dynamic random access memories also exploit the MPu to increase the data rate [7]. In DDR2, the external bus is clocked at double speed of the internal memory cells, so four words of data can be transferred per memory cell cycle. This ratio is four in DDR3 memories.

Figure 2.3 illustrates the most fundamental working principle of MPu approach. The inner circuit in the figure is a multiplier driven by clock *clk_mul*. Outer circuit is a computation unit that takes two pair of arguments and gives multiplication result of each pair. Outer circuit uses inner multiplication circuit for multiplication operation and *clk* is the clock of the outer circuit. The inner multiplication unit works two times faster than outer circuit that uses this inner multiplication unit and computes the data in order. At the first clock cycle $a1 \times b1$ operation is done and in the second cycle $a2 \times b2$ operation is done and overall time that takes to make this computations lasts one clock cycle for outer circuit. For this reason, the outer circuit that uses this multiplication unit supposes there are two units although there exist only one multiplication unit. In other words, the operating frequency is halved and functionality is doubled. In order to provide input from outer circuit to inner circuit, two clocks should be synchronized.

Shorty, in MPu approach internal ports run at faster frequencies than the external ports and the ratio of these two frequencies gives MPuF. The formula of MPuF is given in Equation 2.1. In the figure, the MPuF of the design is two (i.e. $f_{clk\_mul} = 2 * f_{clk}$)

$$MPuF = \frac{Processor\ Period}{RF\ Period} \tag{2.1}$$

2.3. Conventional Multi-Port Register File Implementations

2.3.1. Distributed

The most straightforward method to implement a multi-ported register file is using only the basic logic elements of an FPGA as illustrated in Figure 2.4. The figure shows a D-location register file with n-read and m-write ports. RF data width can be variable. In the implementation, flip flops inside the slices are used to store the data. This structure requires $D \times m$-to-one multiplexers at the inputs and $n \times D$-to-one multiplexers at the outputs to steer the input and output data to the locations and proper output ports respectively. This design is very flexible i.e. implementable by an HDL and tools can synthesize this HDL file. However, this design is not scalable at all because the combinational and wiring delays of logic elements dominate as the number of ports increases severely. As the number of ports and register file size increases, this method becomes area inefficient and synthesis process takes a long time.

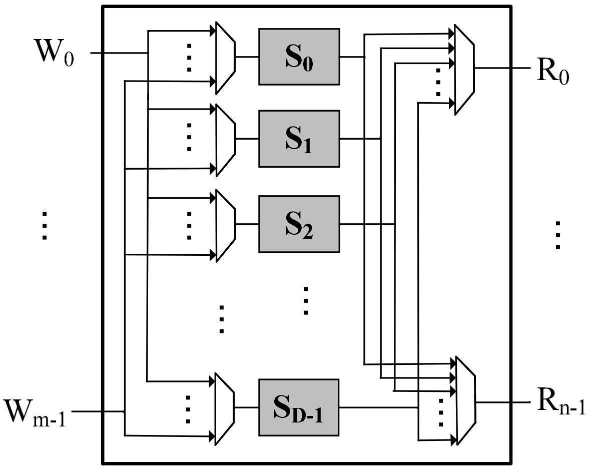


Figure 2.4. Register file implemented as distributed.

2.3.2. Replication

In replication, dedicated storage units (i.e. BRAMs) are used to implement a multi-port register file. In this method, number of read ports is increased by replicating data across the BRAMs. Each BRAM holds a replicated data for each additional port. Xilinx Synthesis Technology (XST) [8] can synthesize this register file automatically, if corresponding HDL file is described properly i.e. no more than one write port. Figure 2.5 shows an register file with n-read ports. However, this technique cannot provide more than one write port since only one write port exists in BRAMs and this write port is used to keep all BRAMs up-to-date.

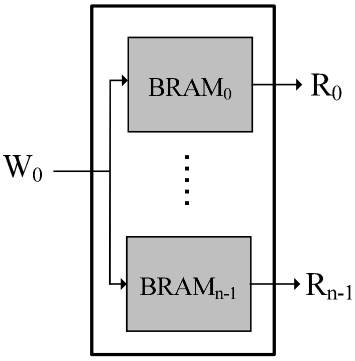


Figure 2.5. Register file implemented by replication.

2.3.3. Banking

If the common write port is not routed to each block RAM in replication, each BRAM can have its own write port. This method is called as banking. Banking method divides the memory locations among multiple BRAMs and each BRAM provides an additional read and write port. However address space is partitioned between the BRAMs and each read and write port can reach its corresponding address space. Figure 2.6 shows an example register file with n-read and n-write ports implemented by banking.

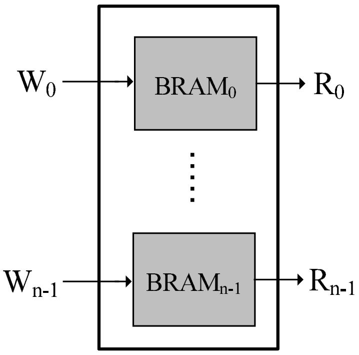


Figure 2.6. Register file implemented by banking.

2.3.4. Multi-Pumping

Multi-pumping is setting the register file internal operating frequency by a multiple of the system's operating frequency. So, multiple read and write operations in register file can be carried out in a time-shared manner. The ratio between RF internal operating frequency and system external operating frequency is defined as multi-pumping factor (MPuF). For example, if a register file with one-read and one-write is multi-pumped by MPuF=2, then this register file gives the illusion of being a two-read and two-write register file. If this register file (2R&2W) had been implemented by banking, the address space had to be partitioned in two parts however the register file implemented by using multi-pumping does not affected from the address space fragmentation. Figure 2.7 shows an illustration of multi-pumping. In the figure, a one-read, one-write register file (BRAM block) is multi-pumped by MPuF=n then n-read and n-write register file is obtained.

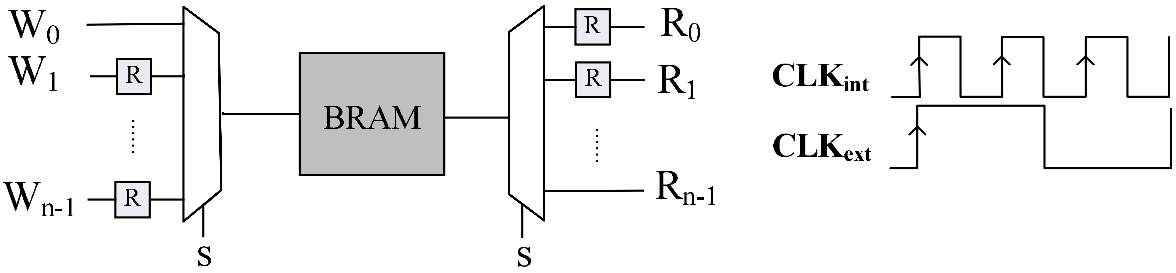


Figure 2.7. Register file implemented by multi-pumping.

Figure 2.8 shows the comparison of two 2R&2W RF implementations which are a multi-pumped register file (MPu-RF) and a pure register file without multi-pumping. The architecture of the pure register file will be elaborated on Chapter 4. As inferred from the figure, MPu-RF occupies one BRAM while non MPu-RF occupies two BRAMs. For this reason, exploiting multi-pumping provides significant area reduction.

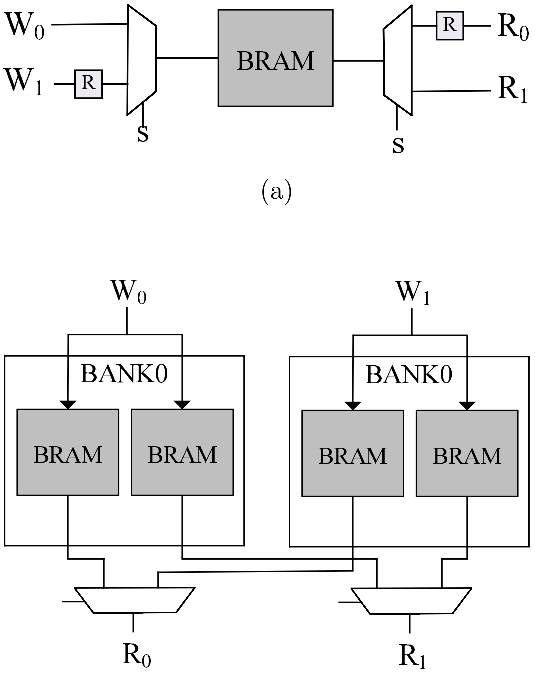


Figure 2.8. Comparison of MPu-RF with non-MPu-RF.

2.4. Register File Taxonomy

Multi-pumping can be applied not only to single-port register files (SPo-RFs) but also to multi-port register files (MPo-RFs). In here, single-port means a simple dual-port register file like Xilinx's BRAM which has single read and single write port. When no multi-pumping performed on SPo-RF and MPo-RF, then they can be named as single-port single-pumped register file (SPoSPu-RF) and multi-port single-pumped register file (MPoSPu-RF) respectively. SPoSPu-RF is a type of simple dual-port RFs. In our RF designs, we have used this primitive to build more complex RFs. MPoSPu-RF is a type of true multi-port RFs (not emulated) hence it has already multi ports without multi-pumping. When it is applied to SPo-RF and MPo-RF, the resulting designs are called single-port multi-pumped register file (SPoMPu-RF) and multi-port multi-pumped register file (MPoMPu-RF) accordingly. Multi-pumping approach provides emulated multi-port register file that has more ports than it has physically. To make it more comprehensible, we may classify RFs with different port sizes and MPuFs by constructing an RF Taxonomy (see Figure 2.9) like Flynns' [9], which is dedicated for the classification of computer architectures. In our taxonomy, RFs are classified regarding to port number and MPuF. Here, note that after applying multi-pumping to SPoSPu-RF (named as SPoMPu-RF), it becomes a multi-ported RF. So, according to our taxonomy, all three MPoSPu, SPoMPu and MPoMPu are multi-ported while SPoSPu remains as a single-ported RF.

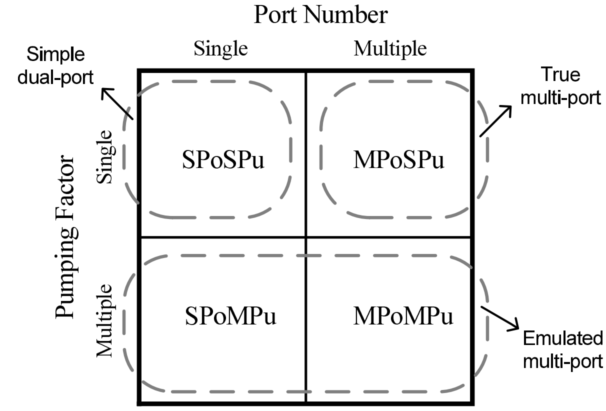


Figure 2.9. Register file taxonomy.

3. RELATED WORK

3.1. Multi-port Register Files for FPGAs

A parallel or heterogeneous processing system firstly feels the need for a multi-port register file (MPo-RF) since it is used commonly by all processing elements. The MPo memory implementation in FPGA is one of the most resource consuming part of a design. There are different ways in designing of MPo-RFs in FPGAs. The first method is utilizing memory capabilities of slices that are available on an FPGA. The study [10] gives an example of a slice-based multi-port memory with one write and three read ports. In [11], a register file for Application Specific Instruction-set Processors (ASIPs) is designed by using distributed resources. In this register file, the size and number of read write ports are determined at design time and the size of each register can change during run-time. Jones *et al.* [12] implements a multi-ported memory by using FPGA's distributed slices to exploit ILP efficitively in their VLIW processor. However, all these designs are not scalable at all because the combinational and wiring delays of slices dominate as the number of ports increases [13]. The second method relies on using block RAMs that are available in traditional FPGAs [14, 15]. However, they have only two ports and are not sufficient for a heterogeneous and parallel architecture that usually needs many ports and shows different characteristics. The third method is combining on-chip block RAMs (BRAM) in conventional methods to form an MPo-RF. There are various studies based on this approach: [16–21].

The usage of BRAMs inside the FPGAs is a well-known methodology to implement RF. In MPo-RF design with BRAMs, two common methods are replication and banking. Most soft processors without multithreading and ILP use replication to provide 2R&1W RF required to support a typical three-operand Instruction Set Architecture (ISA). For example, Nios II, MicroBlaze and Picoblaze soft-core processors use BRAMs to implement an RF with 2R&1W by replication. In addition, Xilinx synthesis tool (XST) provides automatic replacement for multi-read and one-write RFs in this manner [8]. When more ports are required, generally the MPo-RF is implemented by using FPGA slices. Sagmir *et al.* [17] suggest a method to build MPo-RFs by using BRAMs that supports not only multi-read but also multi-write. In this approach, BRAMs are grouped by replication and form a bank. In a bank, all of the BRAMs contain the same data, i.e., the same data are written to all BRAMs inside a bank. Number of read ports can be increased by adding extra BRAMs to each bank. Number of write ports can be increased by adding extra banks. When compared with pure slice implementations, extravagantly usage of BRAM resources is more favorable. However in this method, an RF is partitioned between register banks. Multiple-read can be done from the same bank, but two or more write operations cannot be done to the same bank. In order to handle this problem, a set of methods were proposed by Anjam *et al.* [18]. In these methods, registers trying to access the same bank at the same time are renamed and directed to different banks. These operations are handled by the compiler and the assembler. In [19], the same problem solved by a live value table (LVT) that holds the ID of the most recently updated bank as shown in Figure 3.1. During a read operation, the most recent value is selected by LVT outputs, and it is directed to the output of the MPo-RF. However, LVT, which also has the same number of RW ports with the RF, is implemented by utilizing slices. Hence, this method comes with additional resource usage and propagation delay due to LVT. Moreover, as the total number of banks and the size of the RF increases, the speed of the design decreases and its resource usage increases. In [16], XOR operations are used to recover the most recently-written value. However these methodologies come with extra resource usage, decrease in operating frequency and increase in power consumption.

In [22] and [23], an ESL synthesis framework for FPGA-based heterogeneous systems has been proposed. In this framework, an application written in C is converted to synthesizable version of a heterogeneous multi-processor system-on-chip architecture. In this system, processors are connected to a shared memory by using Xilinx LMBs (local memory buses) so only one processor can access the shared memory at a time. In [19] and [16], write-after-read strategy is adopted in multi-pumped memory locations if simultaneous read and write to the same location occur. Even though this strategy might be beneficial in multi-processor architectures that use the same memory for communication, this is not true for VLIW processors where compilers determine

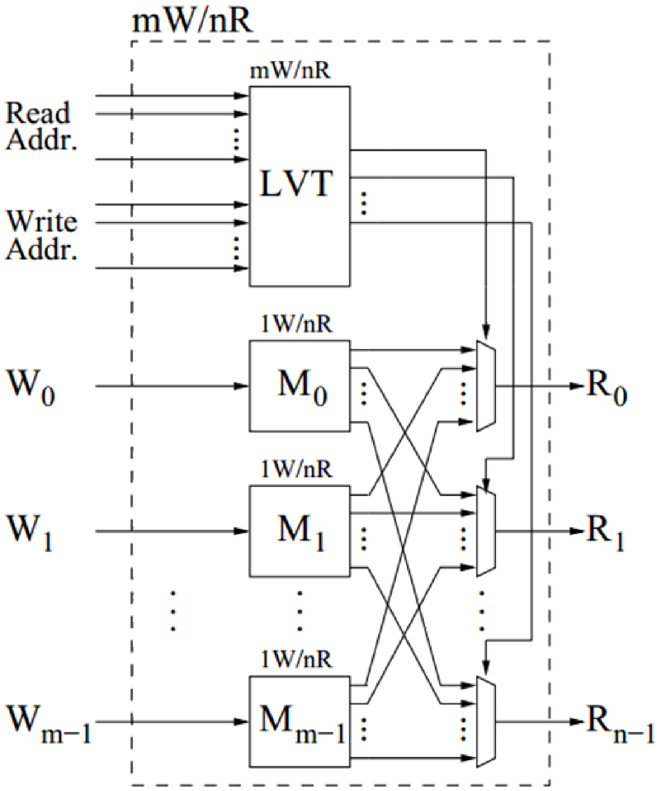


Figure 3.1. Register file implemented by using a live value table (LVT).

the access order of data-path operations to the RF. Even the basic compilers handle all kinds of data dependencies so carefully that a read and write to the same location never occur in the same clock cycle [4]. Therefore in our multi-pumped register files, we did not consider to take any precautions to cope with this phenomenon. A similar approach has been taken in [18] for MPo-RFs.

Multi-pumping (MPu) might also be used for designing MPo-RFs. The conventional multi-pumping approach is based on the utilizing multiplexers and demultiplexers at the inputs and outputs of an RF as described in [19] and [16]. Yet, this approach is also not scalable because of the increased combinational delay in multiplexers and demultiplexers when the number of read- and write ports increases. In [24], a method is suggested by Xilinx for doubling total port number of an RF by multi-pumping. In this design, the ports of a true dual-port memory are multiplexed by two. Hence, there are two write and two read ports. Dedicated registers are used for reading each read port of the multi-ported memory. Having a separate register for each read is not a feasible design choice for higher MPuF due to long wiring delays. Manjikian [25] proposes a MPuMPo-RF for his superscalar processor. This RF uses a different form of multi-pumping, exploits both rising and falling edges of clock. This design requires FFs that are sensitive to both edges of a clock. However, contemporary FPGAs do not have FFs that can trigger on both edges of a clock. In [19] multiplexers are connected to write ports and demultiplexers are connected to read ports. However the size and the propagation delay of the MPo-RF increases as MPuF increases. Hence, this design approach is also not scalable. A recent study by Canis *et al.* [26] exploits the multi-pumping for resource sharing in high-level synthesis. However, they have used multi-pumping for computational units with at most MPuF two and not conducted on aggressive multi-pumping. In their design, while multiplexers are used to feed the multi-pumped circuit at the input, shift registers are used to get the outputs.

3.2. Multi-port Register File Design with CMOS Technology

In a computing system designed as a custom chip, RFs can be implemented as hard-IPs [27, 28]. However, on FPGAs, this is an infeasible solution, because vendors do not want to adversely affect the flexibility offered by the FPGA: Firstly, the desired features may easily change from one design to another. Furthermore, using a general purpose multi-port register file that contains more ports than required is both a waste of energy and an under-utilization of rare resources on the chip. Therefore, MPo-RFs on FPGAs have to be implemented with on-chip resources of the FPGA. Since register files are digital circuits, a CMOS based design can be inspired for an FPGA-based register file design.

Cruz *et al.* [29] propose a multiple-banked RF architecture that uses another second level register file to reduce the number of ports per bank. Barasubramonian's study [30] is also an example of multi-banked RFs. However, this RF is designed by taking into account a processor's datapath and lacks of generality. A RF designed for FPGAs should provide enough flexibility hence environment is reconfigurable and can house different types of PEs.

There exist some nonspecific RF architectures implemented in a heterogeneous system. As an example, the work in [31] proposes a register file design that supports multiple accesses from two processors. One of these processors is a high performance core, and the other one is a low power core. In this design, each processing element can use 2R&1W ports and there are some additional mechanisms to provide data consistency between storage units. It uses multiplexer-based port-sharing which has been proven to be inefficient for FPGAs [28]. In the study [32], each processor uses its local memory and a unified external memory is shared between the processors. This system uses a non-uniform memory access (NUMA) layout and there is no shared memory that can be used concurrently between the processors. In [33], each processing element uses a dedicated memory with a shared external memory. A heterogeneous register file should service different processing elements that show different characteristics in terms of operating frequency, number of ports, data width, address space and endianness. To the best of our knowledge, none of these studies and studies for FPGAs has touched structurally heterogeneous register files for heterogeneous computing platforms on FPGAs as it has been addressed in this thesis.

4. EMULATED MULTI-PORT REGISTER FILE DESIGN IN FPGAs

4.1. Multi-Port RF Designs in FPGA

In traditional FPGA architectures, there exist dedicated BRAMs for storage [14, 15]. As an example, FPGA used for this research (Xilinx Virtex-5 XC5VLX110T) includes total 148 BRAMs. In Virtex-5, each BRAM can store up to 36 Kbits of data and can be configured as either two 18 Kb RAMs or one 36 Kb RAM. However these BRAMs have only two ports which are used either as write or read port depending on the BRAM mode (true or simple dual port) as stated in Chapter 2. In true dual port mode, the BRAMs have two ports. Each port can change its behavior (read or write) during run time i.e. 2R or 1R&1W or 2W. In this configuration each register can store maximum 32-bit wide data with four bits parity and 1024 locations. In simple dual port mode, a BRAM has exactly one dedicated port for reading and another dedicated port for writing. Moreover, this configuration cannot be changed during run time. In this mode, each register can store 64-bit wide data with eight bits parity and 512 locations.

In MPo-RF design with BRAMs, two common methods are replication and banking. XST supports generation of multi-read and one-write RFs by using only replication [8]. To increase the number of write ports, both replication and banking can be exploited as Sagmir *et al.* suggested [17]. In this approach, BRAMs are grouped by replication and form a bank. In a bank, all of the BRAMs contain the same data, i.e., the same data are written to all BRAMs inside a bank. Number of read ports can be increased by adding extra BRAMs to each bank. Number of write ports can be increased by adding extra banks. Figure 4.1 illustrates how six simple dual-port BRAMs can be used to implement 512×64-bit RF with 2R&3W ports. In the example, there are three write ports so there should be three banks. The first bank holds the data between 0 and 127 and the second bank holds 128-255 and the third bank 256-512.

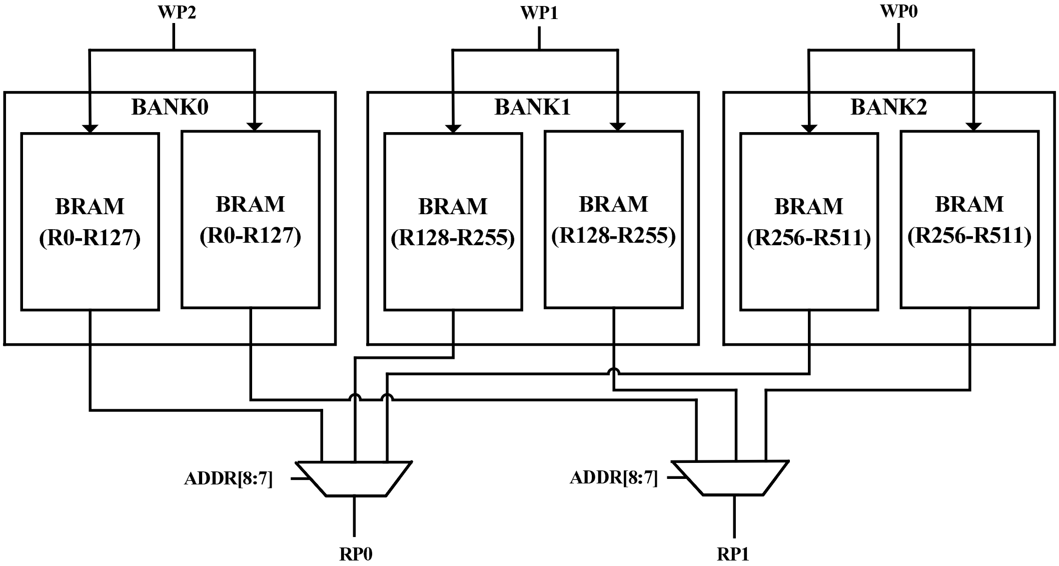


Figure 4.1. MPo-RF with three write and two read ports (2R&3W).

However, though each BRAM has 512 locations, only the first 128 locations are used in the first and second banks, and 256 locations are used in the third bank. For the third bank, the range between 0-255 correspond to the range between 256-512 in the designed MPo-RF. If RF had two read and two write ports, the design would have total four BRAMs with two banks. First bank would hold the data between 0 and 255 and second one would hold 256 and 512. In the design, each register is associated with a separate write port so a multiple write port architecture can be obtained. During a write operation, the value is written to the corresponding register bank. However it is not possible to realize more than one write operations to the same bank at the same time. All write operations should be mutually exclusive with respect to the associated bank. This problem can be handled by software or hardware mechanisms as explained in Chapter 3. For example, usage of LVT table solves this problem with the expense of area, power and speed. This solution does not require partitioning the RF among banks. All banks represent the complete RF. During a read operation, output of the LVT table selects output multiplexers so as to the proper register bank gives the most recent result. The mentioned problem can be solved by software mechanisms easily. For this reason, we have chosen a pure MPoSPu-RF without LVT or any addition as our base design for MPu-RF. We assume that this issue is resolved by the compiler.

4.2. Mult-port Multi-pump RF Designs in FPGAs

In our MPoMPu-RF design, MPu circuit behaves as an interface between outer circuit (e.g. a processor or a set of computational units) and MPo-RF as mentioned in Section 4.1. The term processors will be used to refer the outer circuit from now on. Here multi-pumped means driving the register file with a clock speed which is multiple of the processor speed. In the operation, the processor sends all of the read and write requests at the same time and waits for its one clock period. At the same time memory takes the requests and process them in order at a speed of its internal frequency. After the processor period ends, all of the operations inside the memory are handled, they are ready at the output and all changes are made in the RF. This process takes multiple cycles.

Here it is worth to mention about the reason why we utilized MPu in FPGAs. Almost all FPGAs include some coarse-grain configurable units implemented as ASIC such as BRAMs, hardware multipliers, DSP blocks, etc. It is a rule that these parts of FPGA are faster than the fine-grain reconfigurable parts which are used for implementing functionality [34]. Reconfigurable area is for general purpose and includes combinational logics, connection paths and routing matrices. However ASICs are designed for application specific purposes and are not suitable for generic usage. In other words, ASIC outperforms the FPGA because of its custom design for the dedicated purpose. Most likely BRAMs should run faster than any implemented design on reconfigurable area. For this reason, we aimed to gain advantage of speed difference between BRAMs and logic by utilizing an efficient MPu to MPoSPu-RF. The resulting design is MPoMPu-RF with much more ports than MPoSPu-RF.

4.2.1. Multiplexer based Multi-pumping

MPo-RF implementation with multi-pumping is done with multiplexers to direct the signals of RF to BRAM and demultiplexers to take the outputs as shown in Figure 4.2. In the figure, base MPo-RF consists of k read and m write ports. To increase the number of ports, a multiplexer of n ports is connected to each of

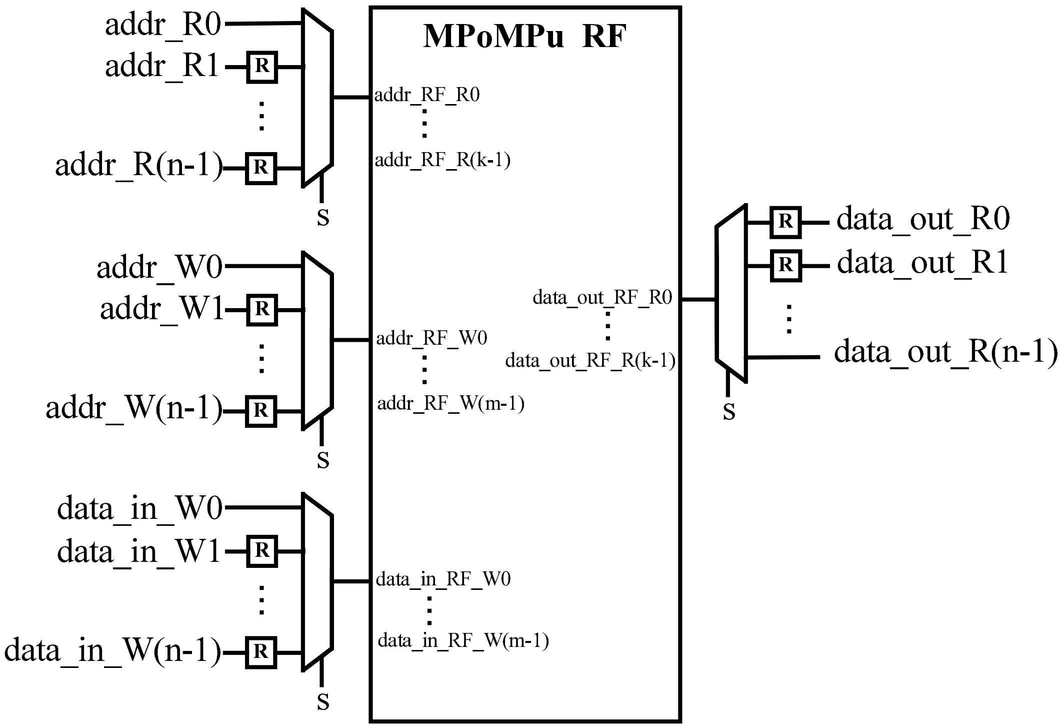


Figure 4.2. Multiplexer based MPoMPu-RF design.

the "read_address" (*addr_RF_Rx*), "write_address" (*addr_RF_Wx*) and "read_data" (*data_out_RF_Rx*) ports. Similarly a demultiplexer of n ports is connected to each of the "write_data" (*data_in_RF_Wx*) ports. In this way, we obtain a MPoMPu-RF with $k \times n$ read and $m \times n$ write ports. In this design, S is the common select input for all multiplexers with the size of $\lceil log\_2(n) \rceil$ where n is MPuF. Hence, in one clock cycle of the outer circuit, one can carry out $k \times n$ read and $m \times n$ write operations by incrementing S at each clock cycle of the inner circuit. In each cycle of the inner circuit, k read and m write operations are carried out through the ports selected by the S input of the RF. If internal operating frequency of multi-pumped RF is f Mhz and MPuF $= n$, then the speed of the outer circuit must be at most f/n. However, the architecture in Figure 4.2 is not scalable with MPuF due to long routing paths. In order to implement a multiplexer, built-in multiplexers inside the carry chain and look up tables are used in FPGAs. As the number of inputs in multiplexers increases, the propagation and wiring delays due to multiplexers and demultiplexer get so long that operating frequency of the inner circuit degrades dramatically. Hence, as a de-

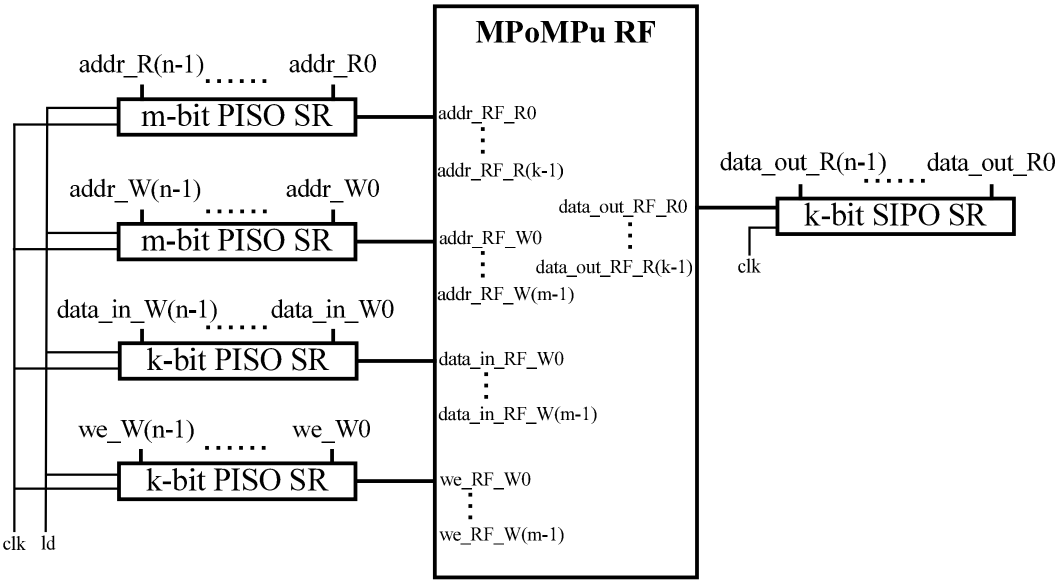
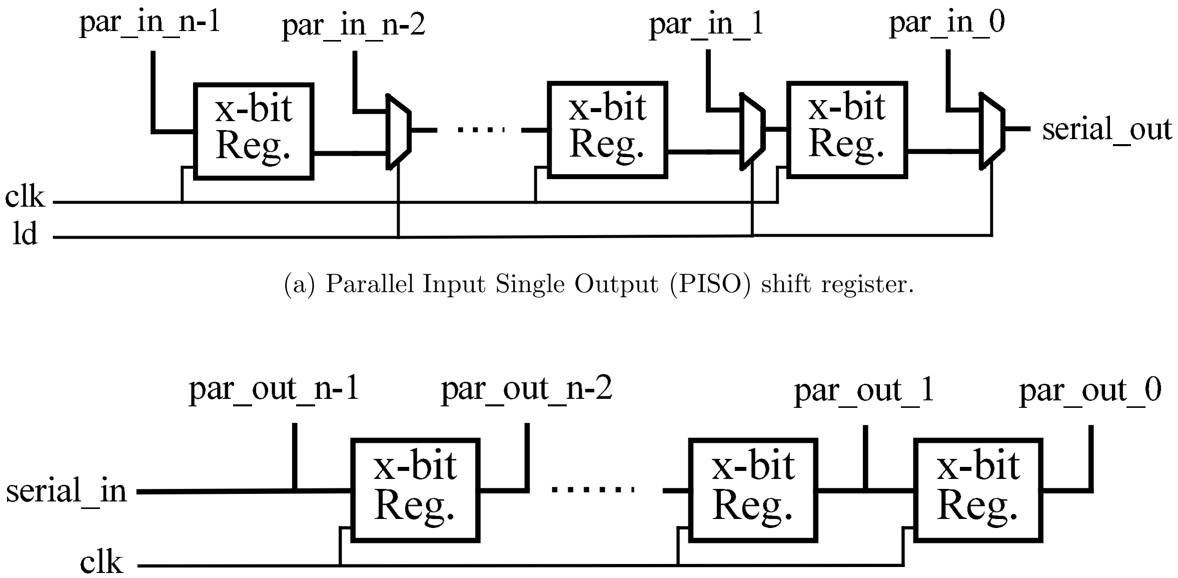


Figure 4.3. Shift register based MPoMPu-RF design.

signer tries to increase the port size by solely increasing MPuF, the internal operating frequency decreases significantly and this results in a drastic degradation in the speed of the outer circuit. This fact is also pointed out in LaForest's work [19].

4.3. Shift Register Based Multi-pumping

In order to eliminate the disadvantages of multi-pumping method explained in prior sections, we propose a new multi-pumping method for FPGAs. Figure 4.3 shows the design of shift register based multi-pumping with MPo-RF. SR-MPu approach uses the shift registers instead of multiplexers for the same functionality. In the proposed design, all of the input and output signals are connected to Parallel In Serial Out (PISO) and Single In Parallel Out (SIPO) shift registers as shown in Figures 4.4a and 4.4b respectively. In PISO, the two-to-one multiplexers select the signals which come from either the previous register or the outer circuit. All input signals are registered in order to be processed by RF. Keeping in mind that the outer circuit and the inner circuit have fully synchronized clocks, one access cycle from the outer circuit to the memory is realized as follows: Firstly, the outer circuit sets "load" (*ld*) at the rising edge of the outer circuit's clock, and holds it high for one cycle of the inner circuit. In this



(a) Parallel Input Single Output (PISO) shift register.

(b) Single Input Parallel Output (SIPO) shift register.

Figure 4.4. SIPO and PISO shift registers.

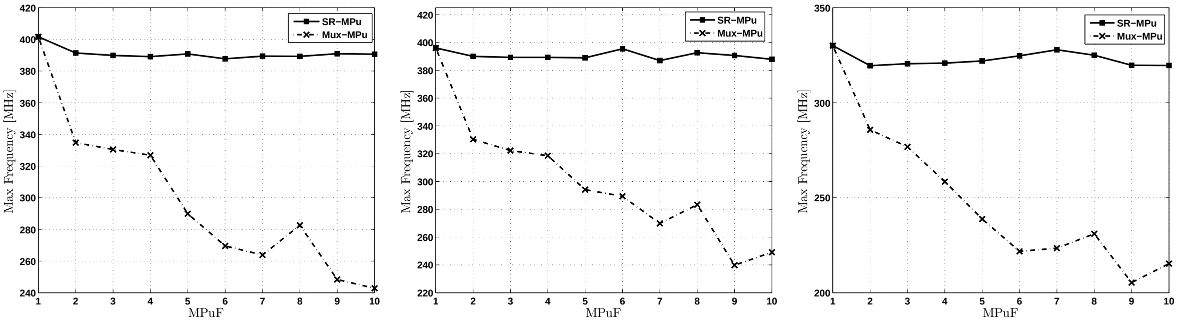
way, the input values coming from the outer circuit *(addr_Rx, addr_Wx, data_in_Wx, we_Wx)* are stored in the PISO shift registers. Note that the ones corresponding to $x = 0$ directly access to RF when $ld = 1$. At the end of this first cycle, the first read values are also loaded into the related SIPO shift registers. At the beginning of the next cycle of the inner clock, the outer circuit resets ld signal to logic-0 and keeps it at this level till the end of the clock period of the outer circuit. However, at each rising clock edge of the inner circuit, values in shift registers are moved towards the RF. In this way, the RF processes the next values of read address, write address, write data, write enable one-by-one. Similarly, RF produces one read value at each clock cycle and each read value is shifted through its corresponding SIPO shift regiter. After n cycles of the inner circuit, all PISO shift registers are empty and all SIPO shift registers are full. Hence at the end of the clock cycle of the outer circuit, the values are read from SIPO shift registers except the last ones, which are directly taken from the memory.

The SIPO at the output consists of only flip flops and does not include any additional combinational logic. In Mux-MPu RF design, width of the multiplexers increases with MPuF. The maximum width of the multiplexers in PISO shift register design is two and these are placed in front of the flip flops inside a slice hence, this design style is in coherence with the FPGA architecture as an FPGA slice consists of LUT, a multiplexer and finally flip flops in sequence. However in the Mux-MPu, this order is altered: a flip flop is followed by a multiplexer and this occupies twice more slices than SR-MPu approach.

4.4. Experimental Results

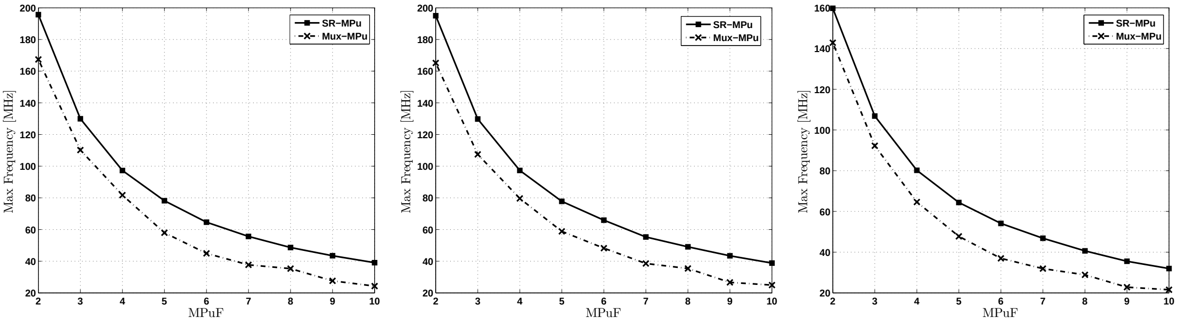
A set of experiments have been conducted to measure the performance, area and energy consumption of the designed MPoMPu-RFs. In the tests, Xilinx Virtex-5 XC5VLX110T FPGA are used. All HDL files that implement the corresponding RFs are automatically synthesized. All of the signals that a pure RF does not need however processor systems possibly need (write enable, RF enable, reset...) are connected for the fair measurements although they affect speed and area results adversely. In [19], these type of signals were left unconnected. For the proper timing measurement, all external ports of the RF are preserved and RF is considered as a part of bigger design so as to avoid I/O registers that cause long routing paths. All timing results are obtained from place and route report by constraints forcing until the constraints fail. Timing results represents the critical path delay of corresponding RF. In the experimental results, frequency is given in terms of MHz. Area results are given in terms of slices and BRAM separately because as far as we know, there is no conversion method from occupied BRAMs to occupied slices. For the power measurements, Xilinx Power Analyzer (XPA) is used by introducing an activity file as input. The activity file utilizes the register file with 100% load. Instead of power consumption, we specified the energy dissipation in terms of power delay product (nJ) because propagation delay and power dissipation generally form a design trade off, that is, improving one of them degrades the other one.

In Figure 4.5, Figure 4.6 and Figure 4.7, the detailed comparison between the Mux-MPu and SR-MPu is presented in terms of maximum internal frequency, maximum external frequency and area respectively. From the figure, it can be inferred that SR-based approach outperforms the Mux based type. Moreover, the internal operation frequency of the shift register based MPo-RF is nearly constant at all multi-pumping



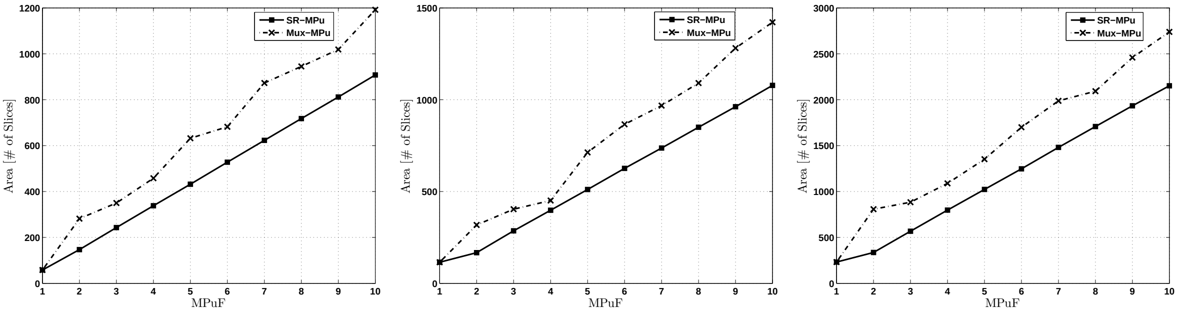
(a) (b) (c)

Figure 4.5. Maximum internal operating frequency of the RFs with respect to MPuF when base MPo-RF file comes with (a) 3R&2W, (b) 4R&2W, (c) 8R&4W ports.



(a) (b) (c)

Figure 4.6. Maximum external operating frequency of the RFs with respect to MPuF when base MPo-RF file comes with (a) 3R&2W, (b) 4R&2W, (c) 8R&4W ports.



(a) (b) (c)

Figure 4.7. Occupied area of the RFs with respect to MPuF when base MPo-RF file comes with (a) 3R&2W, (b) 4R&2W, (c) 8R&4W ports.

factors, even when MPuF=10. Also, as the number of RW ports in the base MPo-RF circuit increases, the internal frequency decreases in both designs. This is due to the increased multiplexer sizes in the base MPo-RFs.

These results are not the unique solutions because some advanced constraints and custom placement rules might affect the results. However the variations should be small and these results are sufficient to give an intuition. In our experiments, we tried to obtain the design under delay constraints. When we inspect the results for 3R&2W RFs with MPuF=2, the speed of the SR-based approach is about 390 Mhz while in Mux-MPu approach it is about 334 Mhz. As MPuF increases, maximum speed of the Mux-MPu RF design decreases as expected and shift-register based approach remains same at about 390 Mhz. This pattern is also valid for other RF combinations i.e. 4R&2W and 8R&4W. At some implementations, mapping and place&route tools gives the better results for a higher MPuF. This is due to the fact that at these points the designs are more compatible with FPGA architecture and the tools can make better placement. In Virtex 5, the smallest multiplexer that occupies one-pass combinational logic is 4:1 multiplexer. When wider multiplexers are required, the multiplexers inside the LUTs are connected to each other by the carry chain and this causes an increase in the longest path. For this reason, in Mux-MPu approach the maximum internal frequency undergoes a break off at every 4:1 multiplexer insertion. For example to obtain 9R&6W RF, we can use 3R&2W with MPuF=3. The internal and external operating frequencies for SR-MPu approach are 390 MHz and 130 MHz respectively. The same parameters correspond to 330 MHz and 110 MHz for Mux-MPu approach. In terms of area, SR approach always uses fewer resources then Mux-MPu counterpart. To summarize, SR-based approach is robust to changes in MPuF and this situation enables us to model RF in a synthesis tool easily. Moreover, SR-based approach is faster and fewer resource occupying compared to Mux-MPu.

Table 4.1 shows the comparison results for 4R&2W RFs with increasing width from 32-bit to 256-bits. In both both SR-MPu and Mux-MPu approaches, the maximum internal frequency decreases when width increases. However, when we compare their respective internal operation frequencies, SR-MPu approach provides nearly 20%

Table 4.1. Frequency and area comparison for 4R&2W RF with different word lengths.

		SR-MPu		Mux-MPu		Improvement (%)	
RF Width	BRAM Usage	Frequency	Area	Frequency	Area	Frequency	Area
32	4	408.330	104	344.116	196	18.66	46.94
64	8	390.016	167	330.360	318	18.06	47.48
128	16	393.701	297	329.489	617	19.49	51.86
256	32	335.909	552	299.401	1358	12.19	59.35

improvement over Mux-MPu approach. The area utilization of SR-MPu approach is nearly 50% better than the Mux-MPu approach. Hence SR-MPu approach provides smaller and faster design than Mux-MPu RF designs.

Table 4.2 depicts the resource utilization, maximum frequency and power delay product ($P \times D$) for 32-bit width and 64 deep RF implementations with 12R&6W. Previous experiments show that SR-MPu method seems to be a more reasonable choice than Mux-MPu method for a MPoMPu-RF implementations. However we also present the results of Mux-MPu RF implementations for more detailed comparison. As inferred from the table, using SR-MPu approach always outperforms the Mux-MPu approach. The distributed implementation (using slices) is the worst method because its frequency is the lowest and it occupies the largest area. Another disadvantage of using slices is long synthesis and implementation time. We have implemented the same RF by only using MPo method as mentioned in Section 4.1. The highest external frequency is given in this design. Using BRAMs with replication and banking is a more effective way of building RF unless it is small and has a few ports when compared with distributed implementation. If it has many ports, then the usage of implementation with BRAMs makes more sense. However the RF is divided into six parts when it is implemented by MPo method. This situation dictates that compiler should provide effort on write conflicts avoidance i.e. no two or more write should direct to the same bank. When RF with 6R&3W and MPuF two is used, internal speed of the RF is higher than previous design (about 384 Mhz) however due to the multi-pumping this design has to be driven at 192 Mhz by the outer circuit. If the processor using this RF works

Table 4.2. Different methods to design a 32-bit/64 deep RF with 12R&6W ports.

		SR-MPoMPu				Mux-MPoMPu				Improvement (%)		
RF Type	BRAM Usage	Internal Freq	External Freq	Area	PxD	Internal Freq	External Freq	Area	PxD	Freq	Area	PxD
Distributed(12R&6W)	0	187.793	187.793	5364	6.055	187.793	187.793	5364	6.055	0	0	0
MPo(12R&6W)	72	302.115	302.115	312	3.267	302.115	302.115	312	3.267	0	0	0
MPo(6R&3W) & MPuF=2	18	386.250	193.125	136	0.753	320.821	160.411	257	0.823	20.39	47.08	8.44
MPo(4R&2W) & MPuF=3	8	408.831	136.277	155	0.367	344.947	114.982	213	0.444	18.52	27.23	17.28
MPo(2R&1W) & MPuF=6	2	429.923	71.654	165	0.160	377.644	62.941	176	0.214	13.84	6.25	25.17

Table 4.3. Different methods to design a 64-bit/512 deep RF with 18R&12W ports.

		SR-MPoMPu				Mux-MPoMPu				Improvement (%)		
RF Type	BRAM Usage	Internal Freq	External Freq	Area	PxD	Internal Freq	External Freq	Area	PxD	Freq	Area	PxD
MPo(9R&6W) & MPuF=2	54	289.101	144.550	776	4.448	246.488	123.244	1177	4.828	17.29	34.07	7.86
MPo(6R&4W) & MPuF=3	24	349.650	116.550	486	1.862	276.472	92.157	707	2.029	26.47	31.26	8.24
MPo(3R&2W) & MPuF=6	6	387.747	64.625	528	0.619	269.542	44.924	682	0.735	43.85	22.58	15.74

at a frequency below 192 Mhz, this design is more advantageous because it occupies only nine BRAMs while the previous design uses 36 BRAMs. In addition to that, it is worth to mention that since RF is divided into three banks, the effort of the compiler is less than the previous case. Increasing MPu decreases the external speed. However the number of banks and the number of used BRAMs are decreased. So depending on the operation speed of the outer circuit and compiler effort, one can select the most suitable design from the listed MPo-RFs.In terms of energy, using BRAM and exploiting multi-pumping as much as possible decrease the energy consumption.

Table 4.3 shows different methods to implement a 64-bit/512 deep RF with 18R&12W ports. In these configuration, RF could not be implemented as pure MPo or distributed. Because implementing this RF requires 216 BRAMs (18 × 12) and our

Table 4.4. Different methods to design a 64-bit/512 deep RF with 32R&24W ports.

		SR-MPoMPu				Mux-MPoMPu				Improvement (%)		
RF Type	BRAM Usage	Internal Freq	External Freq	Area	PxD	Internal Freq	External Freq	Area	PxD	Freq	Area	PxD
MPo(8R&6W) & MPuF=4	48	301.205	75.301	1224	2.643	224.467	56.117	1436	3.58	34.19	14.76	26.13
MPo(4R&3W) & MPuF=8	12	376.790	47.099	1009	1.101	268.097	33.512	1243	1.24	40.54	18.83	11.06

FPGA has only 148 BRAMs. This situation also occurs in 32R&24W MPoMPu-RF implementations when MPuF is two as presented in Table 4.4. So, facilitating the multi-ported RF implementations that cannot be designed by pure MPoSPu-RF is one of the extra advantages of multi-pumping.

As inferred from the table results, using SR-based MPoMPu-RF provides improvement up to 43% in internal frequency, 47% in area and 26% in energy consumption over Mux-based MPoMPu-RF.

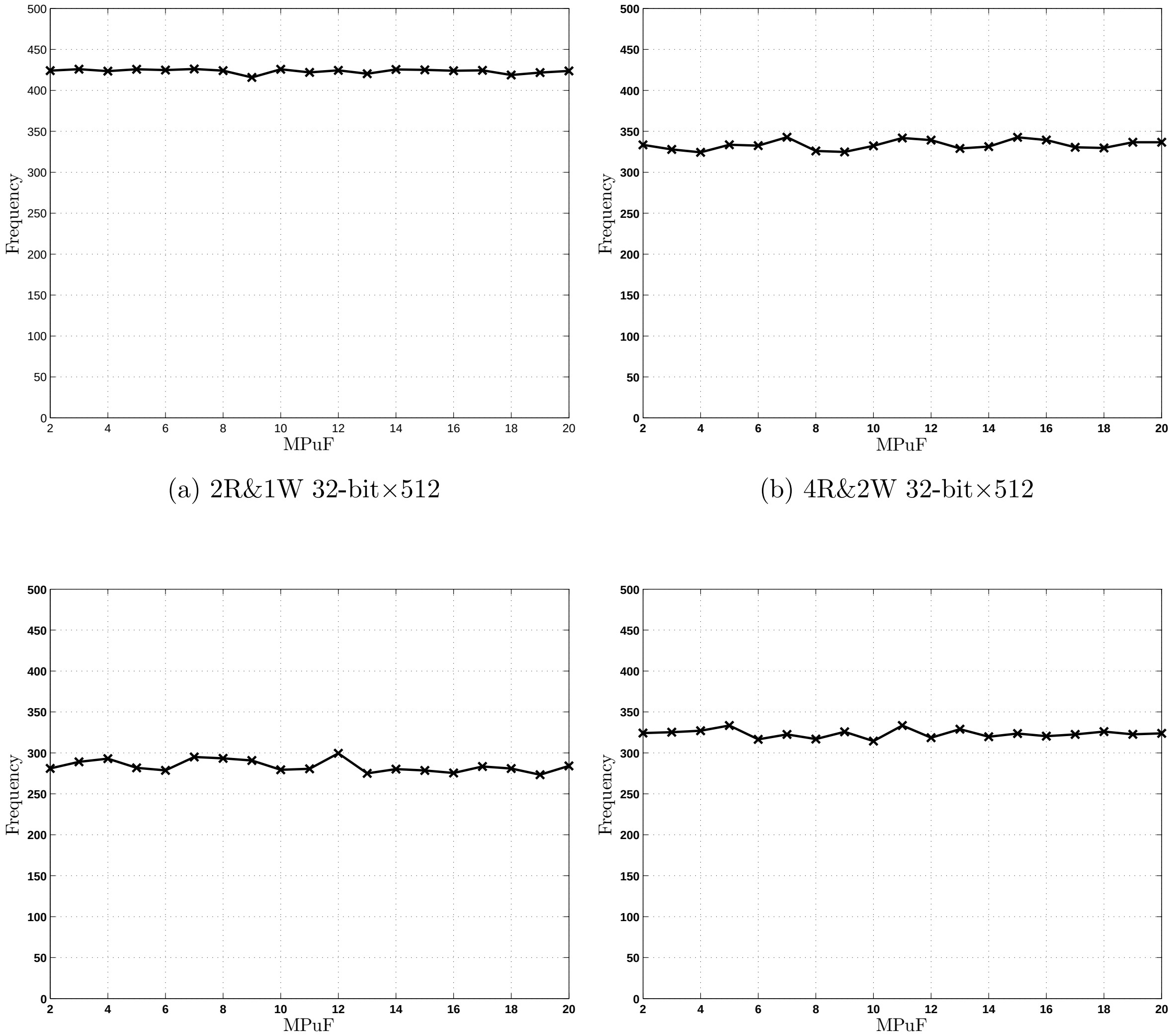
5. EMULATED REGISTER FILE MODELING FOR HLS TOOLs

High-level synthesis is an automated design process that generates the corresponding digital hardware from behavioral description of a system. HLS tools are very important because of the need for complex designs that meet the desired application specific criteria and time to market pressure. In HLS, each primitive part is modeled so as the features of generated systems can be computed rapidly and easily. The main features are delay (frequency), area and power consumption. For each high level system description, more than one digital system may be generated and best one is selected depending on the constraints.

For the usage of HLS tools, we modeled our register file described in Chapter 4. Our register file design can be used in HLS tools easily for the following reasons,

- External operation frequency of a SR-MPoMPu RF is independent from MPuF as touched a little bit in Chapter 4. Figure 5.1 shows internal operation frequency of four different register files with increasing MPuF. As shown in the figure, the frequency remains constant nearly.
- The delay and area models are in coherence with varying number of read and write ports. (see Figures 5.2 and 5.3).

We have grouped our register file by width and size. Depending on the number of read and write ports, we have generated operating frequency models. For each width and size variance, there is a different RF model. For surface fitting, Matlab Surface Fitting tool [35] has been used to generate a formulate for provided [Read Number, Write Number, Operating Frequency] points. These data have been collected by using the tool mentioned in Appendix A.



(a) 2R&1W 32-bit×512

(b) 4R&2W 32-bit×512

(c) 6R&4W 32-bit×512

(d) 9R&3W 32-bit×512

Figure 5.1. External operating frequencies of SR-MPoMPu RFs.

The Equation 5.1 defines the functions that give the operating frequency result for 32-bit×512 and 64-bit×512 respectively with MPuF=2 and varying read/write numbers. In the equations x is read number and y is write number. Figure 5.2 and 5.3 show the fitted functions and corresponding points for each RF respectively. The importance of these functions is that when frequency of a register file is known within a HLS tool, MPuF can be computed by considering the operating frequency of outer circuit hence external operating frequency remains nearly constant with MPuF (see Figure 5.1). For example, we have a 32-bit multiplication circuit running at 100 MHz. This circuits takes 8 inputs and returns 4 corresponding multiplication result of each input pairs so this circuit requires 8R&4W MPo-RF. From the equation, we can compute the external operating frequency of 2R&1W RF for MPuF=2 as f(2,1)=419.946. As shown in Figure 5.1a, internal operating frequency of this RF is nearly constant with increasing MPuF. From the ratio of the RF and multiplication circuit operating frequencies, we can apply MPuF=4 to the circuit. While internal frequency can be run at most 419 MHz, external operating frequency set as 100 MHz with the same as multiplication circuit and MPuF 4 is applied to the RF. So we can use 2R&1W RF as 8R&4W RF. This provides a significant area and power efficiency as stated in Chapter 4.

$$\begin{aligned} f(x,y) = p00 + p10*x + p01*y + p20*x^2 + p11*x*y + p02*y^2 + p30*x^3 \\ +p21*x^2*y + p12*x*y^2 + p03*y^3 + p40*x^4 + p31*x^3*y \\ +p22*x^2*y^2 + p13*x*y^3 + p04*y^4 + p50*x^5 + p41*x^4*y \\ +p32*x^3*y^2 + p23*x^2*y^3 + p14*x*y^4 + p05*y^5 \end{aligned} \tag{5.1}$$

where coefficients for 32-bit×512 RFs (with 95% confidence bounds and 0.9699 R-square are

p00 = 262.1 (258.2, 266.1), p10 = -10.33 (-17.13, -3.517), p01 = -54.36 (-61.1, -47.62), p20 = 2.91 (-3.82, 9.641), p11 = 0.7199 (-6.606, 8.046), p02 = -2.718 (-9.498, 4.063), p30 = -4.017 (-11.3, 3.262), p21 = 6.76 (-0.8082, 14.33), p12 = 6.504 (-1.116, 14.12), p03 = 18.64 (11.36, 25.92), p40 = 0.6986 (-1.89, 3.287), p31 = -2.139 (-5.822, 1.543), p22 = -3.008 (-7.602, 1.586), p13 = -0.665 (-4.338, 3.008), p04 = 11.71 (9.131, 14.3), p50 = 0.2793 (-1.761, 2.32), p41 = -0.6628 (-3.195, 1.869), p32 = -2.051 (-5.851, 1.75), p23 = -2.673 (-6.455, 1.109), p14 = -2.054 (-4.592, 0.4833), p05 = -8.873 (-10.92, -6.824).

where coefficients for 64-bit×512 RFs (with 95% confidence bounds and 0.9772 R-square are p00 = 232.4 (228.9, 236), p10 = -21.6 (-28.09, -15.11), p01 = -46.41 (-52.88, -39.93), p20 = -7.46 (-13.77, -1.15), p11 = -9.656 (-16.56, -2.747), p02 = 4.354 (-1.913, 10.62), p30 = -2.408 (-9.262, 4.445), p21 = -6.808 (-14.07, 0.4565), p12 = 2.433 (-4.748, 9.614), p03 = 3.319 (-3.503, 10.14), p40 = 4.204 (1.749, 6.659), p31 = 0.09755 (-3.343, 3.538), p22 = -4.002 (-8.205, 0.2017), p13 = 2.095 (-1.295, 5.484), p04 = 8.163 (5.768, 10.56), p50 = -0.7093 (-2.618, 1.199), p41 = 1.116 (-1.25, 3.483), p32 = 2.332 (-1.206, 5.87), p23 = 1.968 (-1.548, 5.485), p14 = -3.529 (-5.87, -1.189), p05 = -4.844 (-6.729, -2.958).

The Equation 5.2 defines the function that gives the number of occupied 18kb BRAMs to implement an emulated RF. As stated in Chapter 4, a MPo-RF implemented by replication and banking uses number of BRAMs as many as the multiplication of read port number and write port number when the word length is equals to 32-bit. When it is greater than 32-bit, a multiple 32-bit 18 kb BRAMs combine together side by side to form this word length as seen from the third term of the equation. When the address space (height) is greater than 512, BRAMs also combine together one under the other. This is also taken into account in the equation.

$$\begin{aligned} Number\ of\ Used\ BRAMs = {} & Read\ Number \times Write\ Number \\ & \times \left\lceil \frac{Word\ Length}{32} \right\rceil \times \left\lceil \frac{Height}{512} \right\rceil \end{aligned} \tag{5.2}$$

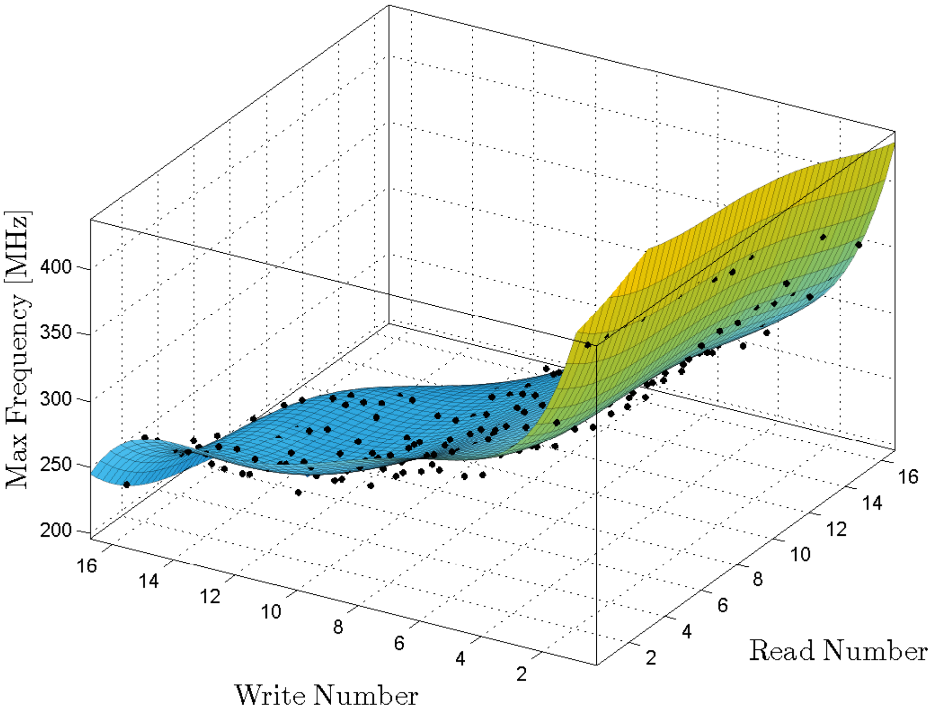


Figure 5.2. Surface fitting for 32-bit×512 RFs.

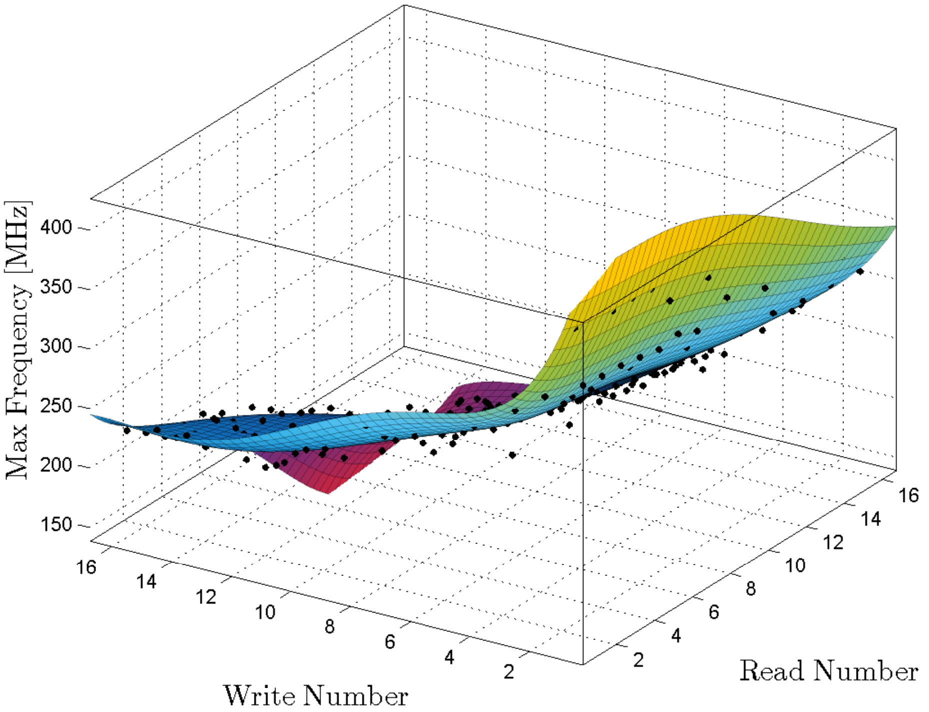


Figure 5.3. Surface fitting for 64-bit×512 RFs.

6. HETEROGENEOUS REGISTER FILE DESIGN FOR FPGAs

Our heterogeneous register file design is a combination of base heterogeneous MPo-RF and multi-pump circuits applied to ports of the base HRF. The resulting HRF can serve different processing elements having different characteristics. Each processing element can have its own clock domain (running frequency), number of read-/write ports, data width, address space, and endianness (big-endian or little-endian). Processing elements might be anything that can be implementable in FPGA like hard-core/softcore processors, DSP processors, custom logical circuits, etc. Figure 6.1 shows a representation of heterogeneous computing systems in FPGA with an abstract architecture of HRF. In the figure, clocks of each processing elements are generated from DCMs or PLLs existing in contemporary FPGAs. Number of processing elements, address spaces of the processing elements, number of read/write ports, data widths, multi-pumping factors and clock frequencies (DCM values) in the figure can be changed depending on the application running on heterogeneous system. The following sections will elaborate on the architectural details.

6.1. Base Heterogeneous Register File

As pointed in Chapter 4, an MPo-RF can be designed by replication and banking as Sagmir *et al.* suggested [17]. However, in this method, the RF is divided into equally sized address spaces and each BRAM holds full address space although only a portion of this address space is reachable. In addition, data width of the register file is fixed for all banks and cannot be changed. This architecture does not provide a heterogeneous structure since port widths and register file sizes are fixed. When used for heterogeneous systems, this method results in the waste of storage resources in FPGA. In a heterogeneous system, each processing element requires its own address space. Data widths of the processing elements may vary with characteristics of the processing elements. For example, Microblaze soft-core processor consumes 32-bit data and Pi-

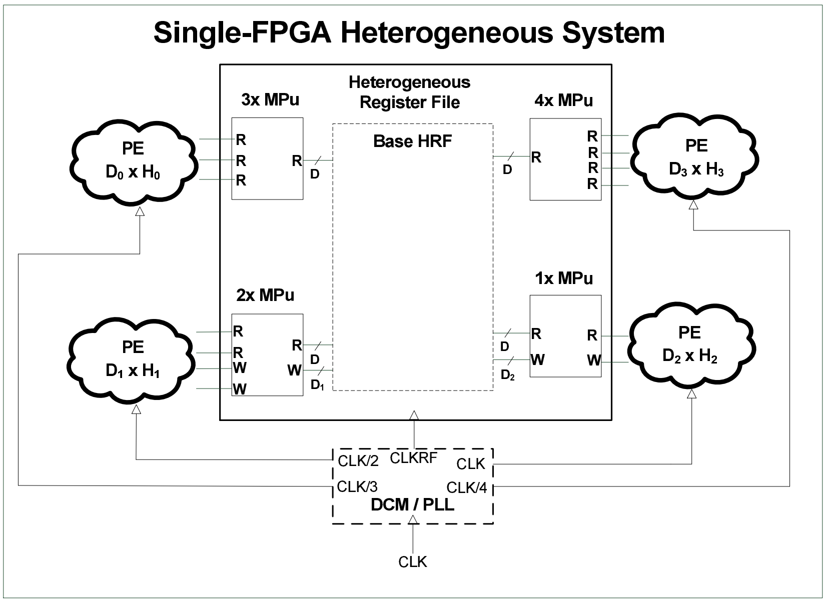


Figure 6.1. Single-FPGA heterogeneous system with HRF.

coblaze processor consumes 8-bit data. A heterogeneous system may consist of these processors and an encryption system like 128-bit AES core from OpenCores [36] to encrypt the results of their operations instantaneously. Additionally, the address space of a processing element should not be reachable from other processing elements to protect its data from accidental write operations and data corruptions. At the same time, processing elements may use computational results of each other so they can access the address spaces of other processing elements to get data. For this reason, address spaces of processing elements should be separated. Separating address space between the processing elements is useful because each processing element serviced from RF has its own address space for write operations and other processing elements cannot reach this address space to write. In other words, one processing element cannot write onto the register file of another processor. Even design of a basic compiler is challenging for heterogeneous systems [37]. Otherwise (if a processing element can write entire address space), this system requires a complex compiler that have to orchestrate all processing elements against data corruptions. Alternatively, this issue can also be handled by hardware mechanisms. In both cases, complexity of the heterogeneous system increases. For this reason, we propose a new heterogeneous register file architecture by inspiring from previous studies.

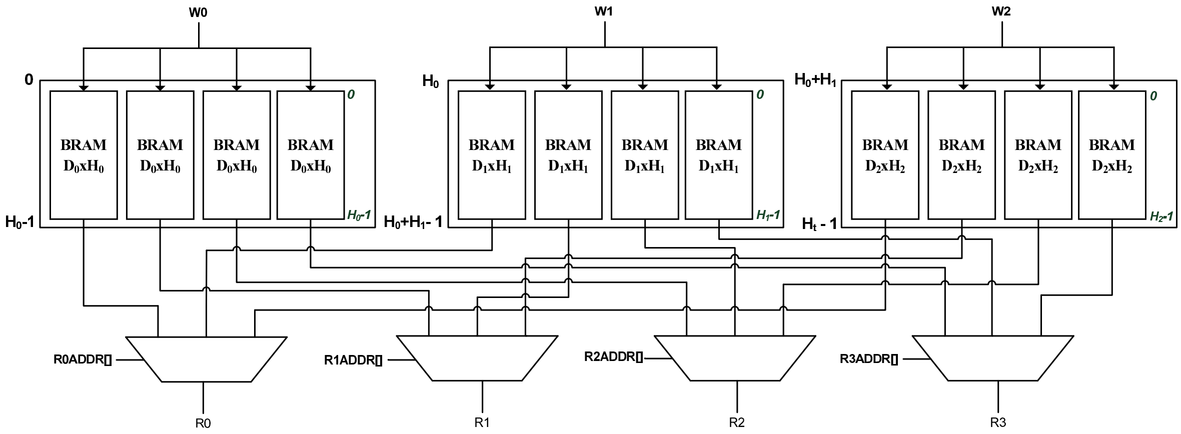


Figure 6.2. A base HRF with four read and three write ports (4R&3W).

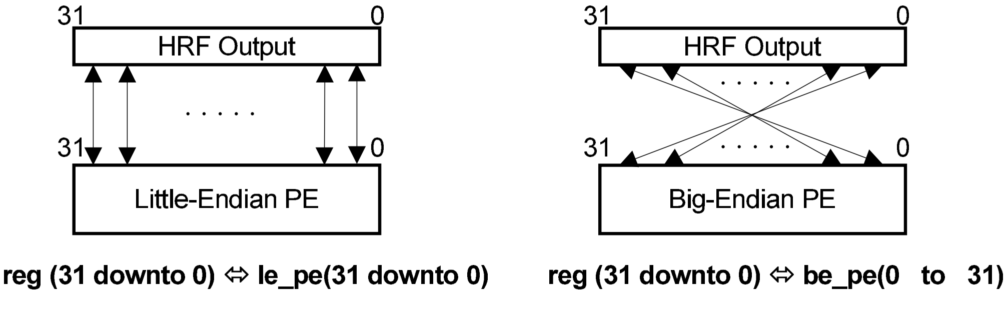
In our heterogeneous register file (HRF) design, each processing element has an address space and this address space corresponds to a portion/bank of the HRF with varying heights and data widths. Each bank consists of replicated BRAMs configured as simple dual port i.e. they have only one read and one write port. Each register bank is associated with a write port. Inside a bank, all BRAMs hold the same data and represent the same address range to increase the number of read ports. In other words, each replicated BRAM is associated with a read port. This address space is named as local address space. Union of all banks forms the global address space so the global address space is the sum of all local address spaces of processing elements. A processing element can only write its own address space and can read data from address spaces of all processing elements. Equation 6.1 shows the formula of the total address space. At the addressing perspective, width of the read address is wider than write address because read address space is higher. Equations 6.2 and 6.3 show the write address width and read address width respectively. Here, write addresses are the trimmed version of the read addresses.

$$Total\ Height\, H_t = \sum_{i=0}^{Bank\#} H_i \tag{6.1}$$

$$Write\ Address\ Width\ of\ PE_i = log_2(H_i) \tag{6.2}$$

$$Read\ Address\ Width\ for\ all\ PEs = log_2(H_t) \tag{6.3}$$

Figure 6.2 illustrates implementation of 4R&3W heterogeneous MPo-RF using BRAMs. In the example, there are three write ports so there should be three banks. Data widths and heights of the banks are determined according to the processing elements and they can be different. Each bank holds a local address space as shown inside of the banks. Global address correspondence of each bank is shown at the left corners of the banks. The first bank holds the data that corresponds the addresses between 0 and $H_1 - 1$ and the second bank contains the address range between H_1 and $H_1 + H_2 - 1$. The last bank holds the range between $H_1 + H_2$ and $H_t - 1$. In here, H_t equals to $H_0 + H_1 + H_2$. In this architecture, each read port can access the entire address space. During a read operation, the data that is pointed by given read address is directed to output port from BRAM that contains corresponding address space. This data direction is done by multiplexers connected to outputs of BRAMs. The select inputs of the multiplexers are the most significant digits of read address that is able to distinguish the smallest address space. The size of select inputs can be changed depending on the partition (i.e. number of write ports). Here it is worth noting that the height of a bank has to be a power of two, otherwise multiplexers would be very large. During a write operation, the value is written to the corresponding register bank. However it is not possible to realize more than one write operation to the same bank at the same time. All write operations should be mutually exclusive with respect to the associated bank. After applying multi-pumping (see Section 6.2), if the number of write ports is not sufficient for a processing element, local address space of a processing element is fragmented since there are more than one write ports dedicated to



(a) Little-endian

(b) Big-endian

Figure 6.3. Connection methods with different endianness.

this processing element. This problem can be handled by register renaming at compile time [18] when it is fragmented for a processing element.

6.1.1. Output Data Width

In HRF, read output data width of the base HRF should be set as the largest data width of the processing elements to guarantee one-cycle access to HRF for all processing elements. For example if there exists four processing elements with corresponding data widths 32-bit, 64-bit, 8-bit and 16-bit respectively; the read data output should be 64-bit wide. The processing elements with smaller data widths should use only a portion of the data. In such a case, processing elements can put data in a buffer and process it in parts. If a processing element reads smaller data, read data is normalized as described in following part.

6.1.2. Endianness

Endianness differences between the processing elements can be handled automatically in the design phase of our register file. The HRF holds all data values in little-endian format i.e. least significant bit is placed at first. Designer can specify the endianness of a processing element and routing of the HRF can be designed by regarding endianness. In such a case, order of the buses coming to/going from base-HRF can be reversed in order to provide compatibility. Figure 6.3 shows the little-endian

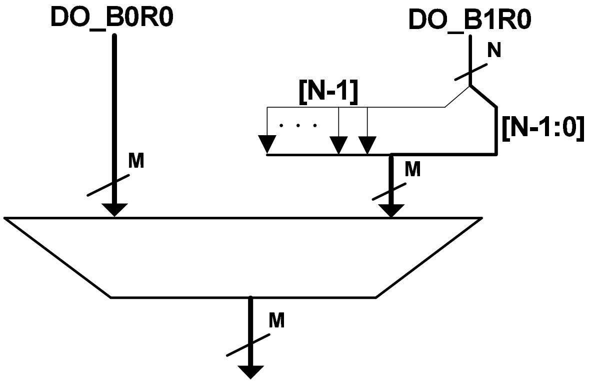


Figure 6.4. Sign extension between different width data.

and big-endian connection schemes for little-endian and big-endian processing elements respectively in an example 32-bit HRF design. For the byte endianness, byte orders for each processing element can also be configured.

6.1.3. Sign Extension

In some cases, a higher-bit processing element can require the results of lower-bit processing elements. To handle incompatibility between the number representations, data read from a bank should be fit to the target whilst passing through the multiplexer. A conversion mechanism is required to preserve the data sign. As stated in previous part, all outputs are given in maximum number of bits although register file can hold narrower data. In such a case, remaining bits of the data are complemented with the most significant bits of the read value to preserve the sign. Figure 6.4 shows this basic conversion.

6.2. Multi-pumping Implementation

In general, processing elements in a heterogeneous system runs at different frequencies and each of them belongs to different clock domains. Regarding the different clock domains, each processing element can exploit multi-pumping for area reduction. In HRF, different multi-pumping factors ($MPuF_i$) can be applied to different ports

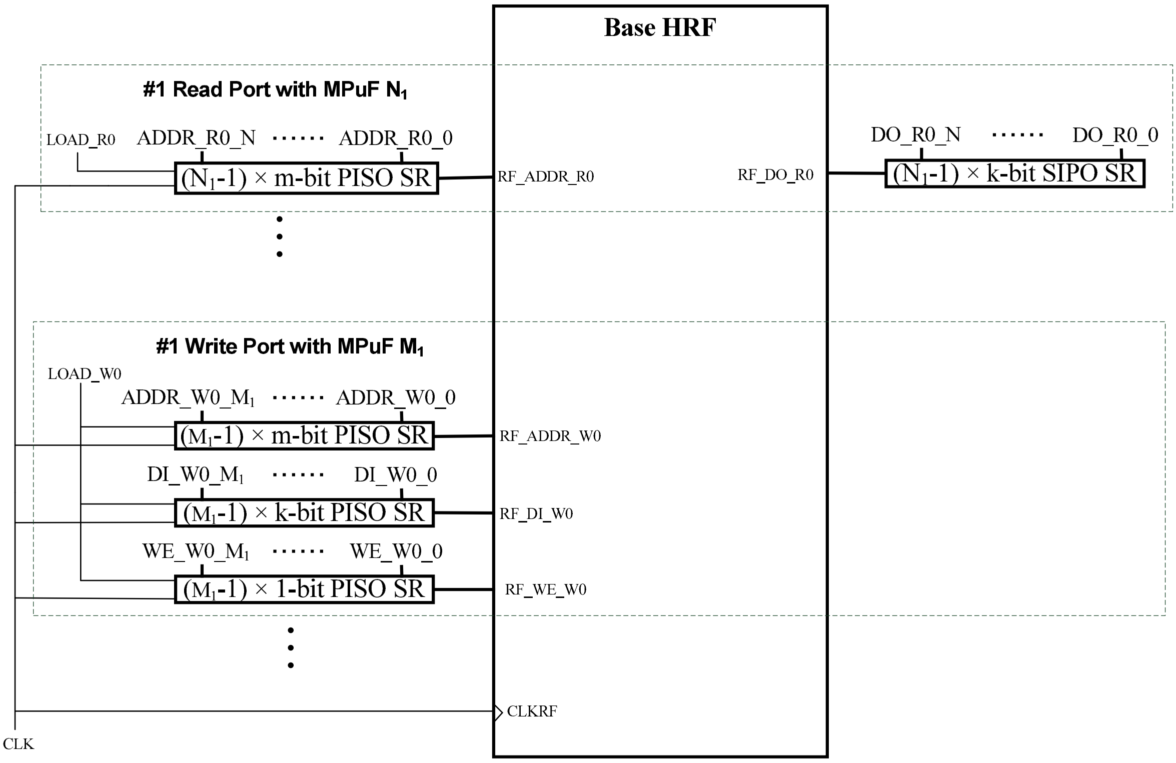


Figure 6.5. A multi-pumped HRF.

of the register file depending on characteristics of processing elements. When applied to the write ports, multi-pumping also diminishes address space fragmentation for a processing element as stated in Section 6.1. The multi-pumping factor (MPuF) is defined as the rate between the operation frequency of the register file and the frequency to read from or write to a port of corresponding processing element. The formula of MPuF is given in Equation 6.4.

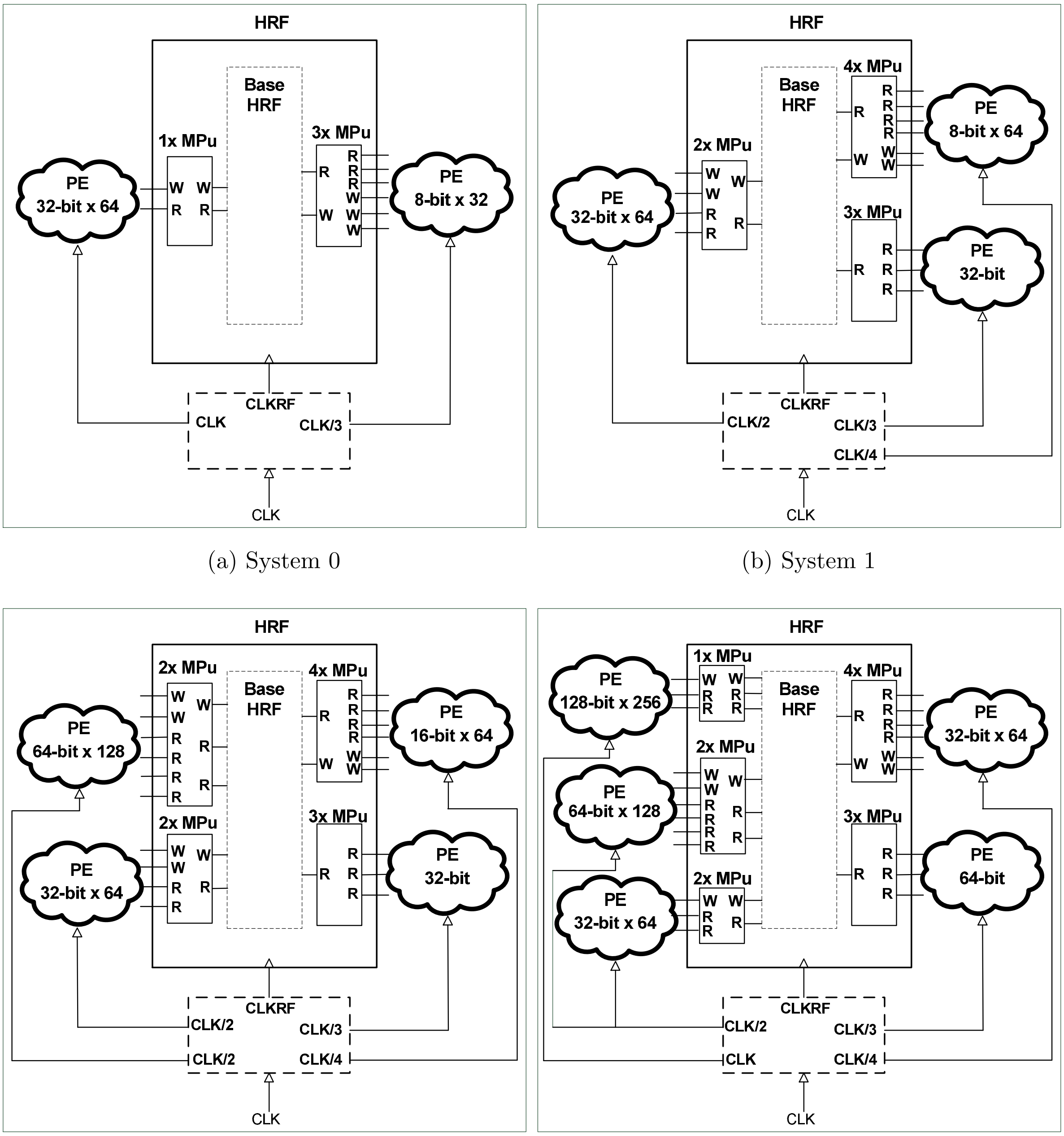
$$MPuF_i = \frac{Processing\ Element\ Period_i}{HRF\ Period} \tag{6.4}$$

With respect to variance in MPuFs, each connected processing elements run at most at a fraction of the HRF speed. For example in Figure 6.1, if MPuF=2 were applied to read port of the base HRF, the corresponding processing element that is getting service from first read port would run at most half of the HRF clock speed ($CLKRF$). In the designed HRF, each processing element can be dedicated more than one port (both read and write). However it is required that the MPuFs of these ports should be the same and equal to the clock speed ratio of HRF and processing element. If a processing element needs fewer ports than the multi-pumping offers, the processing element does not have to make connections to excess ports and they can be left unconnected.

Figure 6.5 shows the design of shift register based multi-pumping with base HRF. In the design, all of the input and output signals are connected to Parallel In Serial Out (PISO) and Single In Parallel Out (SIPO) shift registers as shown in Figures 4.4a and 4.4b respectively. In PISO, the two-to-one multiplexers select the signals which come from either the previous register or the processing element. Each processing element has its own load signal (*LOAD_Wy* and *LOAD_Ry*) if corresponding MPuF is greater than 1. Otherwise it is connected directly to HRF without using PISO and SIPO, that is, it can run at the frequency of the HRF. The processing elements control the flow of the data to/from HRF by this signal. All input signals are registered in order to be processed by RF. Keeping in mind that the processing element and the RF have fully synchronized the clocks, one access cycle from the processing element to the RF is realized as follows: Firstly, the processing element sets "LOAD_Rx" or "LOAD_Wx" at the rising edge of its clock, and holds it high for one cycle of the register file. In this way, the input values coming from the processing element *(ADDR_Ry_Rx, ADDR_Wy_Wx, DI_Wy_Wx, WE_Wy_Wx)* are stored in the PISO shift registers. In here, x corresponds the to port number that is connected to xth input of the shift registers and y corresponds to port number of HRF that the related SIPOs or PISOs is connected to. For example, *ADDR_W1_2* corresponds to the third address input of the SIPO that is connected to second port of the base-HRF. Note that the ones corresponding to x=0 directly access to base HRF when load signal is logic-1 because they are first input of SIPOs. In this way, the first read values are also loaded into the related SIPO shift registers. At the beginning of the next cycle of the inner clock, load signal is set to logic-0 and keeps it at this level till the end of the clock period of the processing element. However, at each rising clock edge of the base HRF, values in shift registers are moved towards the base HRF. In this way, the base HRF processes the next values of read address, write address, write data, write enable one-by-one. Similarly, base HRF produces one read value at each clock cycle and each read value is shifted through its corresponding SIPO shift register. After n cycles of the base HRF, all PISO shift registers are empty and all SIPO shift registers are full. Hence at the end of the clock cycle of the processing element, the values are read from SIPO shift registers except the last ones, which are directly taken from the HRF.

6.3. Experimental Results

A set of experiments have been conducted to measure the performance and area of the designed HRFs. In the tests, Xilinx Virtex-5 XC5VLX110T FPGA is used. All HDL files that implement the corresponding HRFs are automatically synthesized with proper features (data widths, address spaces, bank heights, endianness, port numbers, etc.). For the proper timing measurement, all external ports of the RF are registered and RF is considered as a part of bigger design so as to avoid I/O registers that cause long routing paths and reduce the speed. All timing results are obtained by constraints forcing until the constraints fail to find the maximum speed. In the experimental results, frequency is given in terms of MHz. Area results are given in terms of slices and BRAM separately because as far as we know, there is no conversion method from occupied BRAMs to occupied slices. For the power measurements, Xilinx Power Analyzer (XPA) is used by introducing an activity file as input. The activity file utilizes the register file with 100% load. Instead of power consumption, we specified the energy dissipation in terms of power delay product (nJ) because propagation delay and power dissipation generally form a design trade off, that is, improving one of them degrades the other one. To generate the clocks of the processing elements DCM is used. DCM can generate different clock frequencies with zero clock skew. All clock domain crossing operations is handled by XST. These results are not the unique solutions because some advanced constraints and custom placement rules might affect the results. However the variations should be small and these results are sufficient to give an intuition.



(a) System 0

(b) System 1

(c) System 2

(d) System 3

Figure 6.6. Single-FPGA heterogeneous systems with increasing complexity.

Table 6.1. Resource evaluation results of HRFs.

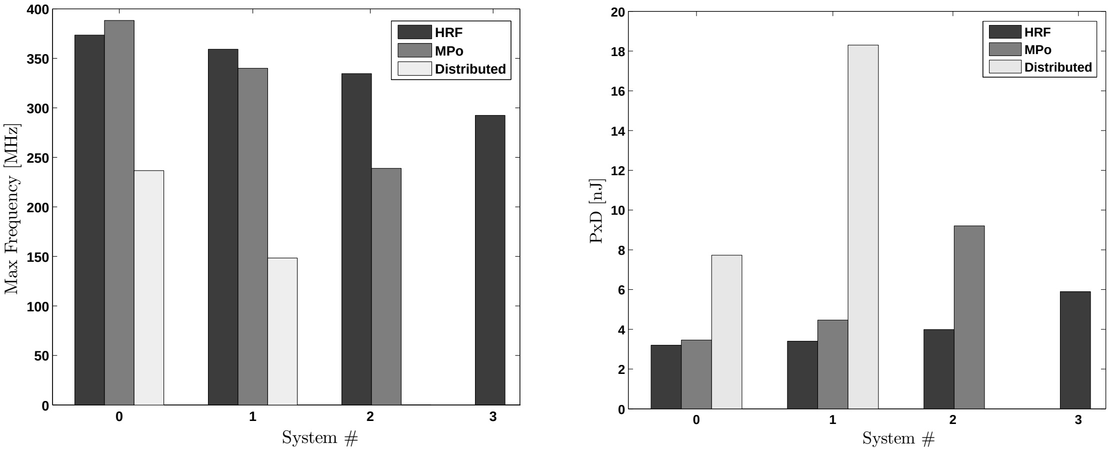
	HRF		MPo [17]		Distributed	
System	LUT-FF Pairs	BRAMs	LUT-FF Pairs	BRAMs	LUT-FF Pairs	BRAMs
System 0	146	2	132	16	10910	0
System 1	401	3	292	36	21397	0
System 2	1058	15	1670	156	NA	NA
System 3	2247	56	NA	NA	NA	NA

For the experiments, we have prepared four different heterogeneous systems with different features in FPGA. As system number increases, complexity of the system increases too. Heterogeneous systems are composed of different processing elements that show different characteristics. For example, System 2 has tree processing elements which are 32-bit, 32-bit and 8-bit respectively. Running frequencies of the processing elements vary and clocks are generated by DCM. Depending on the ratio between HRF speed and processing element speed, MPuFs are set. HRF has total 128 locations. The first processing element has 64 register locations and uses 2R&2W ports. Second one does only read operations so it has not got any address space. Third one is an 8-bit processor like Picoblaze and holds 64 registers. All processing elements can access entire HRF to read. For writing, they can write only their address spaces.

Table 6.1 shows the resource occupation of each HRF in terms of LUT-FF pairs and 18kB BRAM blocks. In terms of LUT-FF pairs, distributed approach is worst hence it is implemented by using slices. For System 0 and 1, HRF is slightly worse than MPo because HRF uses PISO and SIPO shift registers at the input and output ports. Nevertheless they are comparable. In BRAM usage, HRF consumes the lowest area because it exploits multi-pumping and world length of BRAM blocks can be variable. However in MPo approach, this structure is not suitable for different length blocks. It is also interesting that System 0 and System 1 had to use 4 and 6 18k BRAMs respectively according to our base RF design. During synthesis, XST uses slice-based memory resources like RAM32M for small blocks and number of occupied BRAMs decreases. This is also another reason to why LUT-FF pair usage is higher in HRF for System 0 and System 1. When the system complexity increases, our HRF outperforms

Table 6.2. Maximum operation frequency and energy consumption results of HRFs.

	HRF		MPo [17]		Distributed	
System	Frequency	PxD	Frequency	PxD	Frequency	PxD
System 0	373.413	3.198	388.199	3.460	236.686	7.728
System 1	359.195	3.402	339.905	4.463	148.368	18.299
System 2	334.560	3.990	238.949	9.207	NA	NA
System 3	292.398	5.893	NA	NA	NA	NA



(a)

(b)

Figure 6.7. Maximum operating frequency and energy consumption results.

MPo in both LUT-FF pairs and BRAMs (see System 2). In addition, our competitors become to be eliminated when the complexity increases. For example, distributed RF could not be implemented for System 2 because total chip area is not sufficient to route all RF signals in an allowable time so its results are not available (NA) as shown in table. Another disadvantage of using slices is long synthesis and implementation time. For System 3, MPo RF is non implementable too, because there is no enough BRAMs. In this configuration, RF could not be implemented as pure MPo or distributed. Because implementing this RF requires 540 BRAMs (see Equation 5.2) and our FPGA has only 296 BRAMs. So, facilitating the multi-ported RF implementations that cannot be designed by pure MPo-RF is one of the extra advantages of multi-pumping.

Figure 6.7a and 6.7b show the maximum operating frequencies and energy consumptions of the designed RFs respectively. Table 6.2 shows the corresponding numerical values of these figures. From the figure and table, it can be inferred that HRF consumes much less energy than others because it uses least number of BRAMs. In terms of energy, it is obvious that using BRAM and exploiting multi-pumping as much as possible decrease the energy consumption. For operating frequency, MPo is better in System 0 because shift registers in HRF dominates however for System 1 and System 2, HRF outperforms. The distributed implementation (using slices) is the worst method because its frequency is the lowest and it occupies the largest area.

7. AUTOMATIC CODE GENERATION AND SYNTHESIS

For FPGAs, outputs of an HLS tool are behavioral descriptions of digital system that represents the target architecture. For this reason, HLS tools should generate HDL files. In addition to models described in Chapter 5, we have also written TCL scripts to generate RFs automatically for HLS tools. Generally, EDA programs like Xilinx ISE, Altera Quartus support TCL stands for Tool Command Language and this is why we have used TCL for HDL generation. In all scripts, corresponding data and address vectors are computed as described in Equations 7.1 and 7.2 respectively.

$$data \ : \ std\_logic\_vector((Width-1) \ downto \ 0) \tag{7.1}$$

$$address \ : \ std\_logic\_vector(\lceil log_2(Depth)\rceil \ downto \ 0) \tag{7.2}$$

7.1. Emulated Register File (ERF)

Emulated register file generator (see Chapter 4) can be used in homogeneous architectures. As shown in the Figure 7.1, RegisterFile.tcl can be run and this initial script calls related scripts to generate a register file in specific features. As output, scripts generate RegisterFile.vhd, MultiPort.vhd, MultiPump.vhd and BRAM_SDP.vhd files. Optional TestBench.vhd file can also be generated for the given RF. Brief descriptions of each tcl file are given in the following paragraphs;

RegisterFile.tcl : This file is the top file to generates a emulated register file. The inputs of the script are read, write port number, width, size of base RF and MPuF applied to base RF. Path is optional that is the place where output files are placed. In order to create a registered RF circuit for speed measurement, RRegisterFile.tcl script

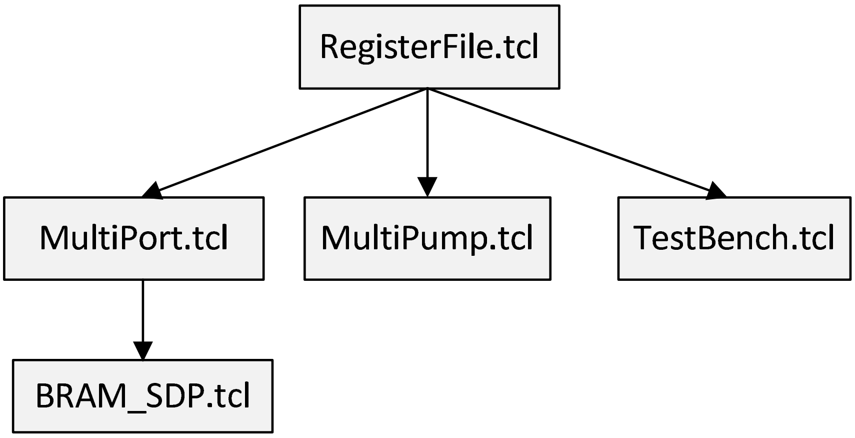


Figure 7.1. Call order for ERF generation.

can be used that registers all input and output ports.

MultiPump.tcl : There are two version of MultiPump.tcl file which are MultiPumpSR.tcl and MultiPumpMUX.tcl depending on the MPu type (M for MUX and S for SR). This file generates a MPu circuit with specified MPuF and base RF.

MultiPort.tcl : This script generates the base RF. Base RF consists of BRAMs connected together by replication and banking. (see Figure 6.2)

BRAM_SDP.tcl: This script generates simple-port BRAM that are used in base RF. This file generates a primitive simple-dual port BRAM with specified width and size.

7.2. Heterogeneous Register File (HRF)

Heterogeneous register file generator (see Chapter 6) can be used in heterogeneous architectures. Call order for HRF is the same as shown in Figure 7.1 As output, scripts generate RegisterFile.vhd, MultiPort.vhd, MultiPump.vhd and BRAM_SDP.vhd files. However this script generates BRAM_SDP.vhd files for each different processing elements. Optional TestBench.vhd file can be generated for given RF. Brief descriptions of each tcl file are given in the following paragraphs;

RegisterFile.tcl : This file is the top file to generate a heterogeneous register file. The inputs of the script are read, write port number, widths, sizes of base RFs and MPuFs applied to base RFs. Widths, sizes and MPuFs are given as array. Path is optional that is the place where output files are placed. In order to create a registered RF circuit for speed measurement, RRegisterFile.tcl script can be used that registers all inputs and outputs.

MultiPump.tcl : MultiPump.tcl file generates an HDL file that applies MPuFs to corresponding ports. For each read and write port a MPuF is applied.

MultiPort.tcl : This script generates the base RF. Base RF consists of BRAMs connected together by replication and banking. However each base RF has different widths and sizes. (see Figure 4.1)

BRAM_SDP.tcl: This script generates simple-port BRAMs for each PEs.

7.3. Distributed Register File

For the comparison, we have also written a tcl script for distributed RF generation. This script consists of single file (Distributed.tcl). For the speed measurement RDistributed.tcl can be used. This script exploits the distributed resources i.e. slices to implement a specific RF.

7.4. Automated Synthesis Process

For a digital system design targeted Xilinx FPGA devices, XST synthesizes HDL designs to create netlist files after design entry. These files include logical design. After the generation of netlist file, second phase is implementation. This phase consists of sequential translate, map and place & route. For all these steps, designer can provide constraints by user constraint files (UCF) to improve the design performance. These constraints direct the synthesis tool to optimize one or more specification of a design. These may be timing, placement, synthesis constraints. In translate step, incoming netlist files and constraints are merged into a Xilinx specific design file which represents the logical design in terms of Xilinx primitives. Map step fits the design available target device resources such as CLBs, DSPs. In place and route, these resources are placed. This process also routes the placed design. After each step, corresponding report files are generated in XML format. This file indicates status of the constraints which are whether met or not met in the corresponding process. For example, if we give 2 ns delay constraint for a 6R&2W MPo-RF design, this means that we want to operate this RF at 500 MHz. Implementation tool tries to fit the design this timing constraints. However, if the actual operating frequency does not fit to specified constraint, synthesis tool gives a warning to inform the designer. In some cases where constrains are not fit, synthesis tool may deviate and give much higher frequency values than the RF actually can operate. For this reason, the RF has to be resynthesized in order to find the best value i.e. maximum operating frequency by decreasing or increasing the constraint in accordance with the result reports until the best fit values is found.

When a designer wants to find the fastest operating frequency of a design, all these steps have to be done and this may take a long time until the best value is found. For this reason, we have developed an automation system for synthesis operations. Figure 7.2 shows our automated synthesis process aimed at finding maximum frequency and corresponding area results of the RFs. Although we have used it to obtain speed and area results of heterogeneous, emulated and distributed RF implementations, this process can be used for any digital system design implementations for FPGAs. Additionally, even though this automation process has been designed for Xilinx based tools, we can modify it for Altera devices since Altera also supports TCL [38]. Brief descriptions of each variables used in this process are given in the following items;

- target: This variable represents the next target delay value. In the next step, the RF design is synthesized by using this delay value in the UCF file.
- fit: This variable holds the lowest delay value that a RF can operate.
- unfit: This variable holds the highest delay value that a RF cannot operate.

- area: This variable holds the resource occupation in terms of LUT-FF pairs and number of BRAMs.
- delay: This variable indicates the delay result of recently synthesized RF.

In the initialization step, *target*, *fit*, *unfit* and *area* variables are set at their initial values. The value of *target* is 0. The *fit* value is a large value, 1000. The *unfit* value is 0 because no design can run at infinite frequency. After the initialization step, required HDL files are generated by using the TCL scripts. The target UCF file is generated with the *target* value 0. In this case, UCF generation function creates an empty UCF file. After UCF creation, these HDL files and UCF file are synthesized by using TCL interface of Xilinx tools. At the end of synthesis, XST gives detailed results in extensible markup language (XML) files. These XML files include all of the information about the designed system such as operating frequency, critical path, resource usage, etc. The *delay* and *area* variables are updated from these reports. According to the relationship between the *target* and read *delay* value, a new *target* is computed which is a value between the *fit* and highest *unfit* value. If the *delay* is smaller than the *target* i.e. target is met, *fit* value is updated as the *delay* value that is the best delay value ever found. *Target* is also reassigned to the better value. If the *target* is not met, in this case target is renewed to a bigger delay value. In some cases, the read *delay* value may be smaller than the *fit* value although the target cannot meet. The synthesis process also checks the relationships against such conditions. When a new *target* computed, the UCF file is created with the new *target* value and design is resynthesized. This process continues until the best *fit* value is found. If the synthesis process finds a value in an acceptable range (0.01), the value is recorded as best *fit* value.

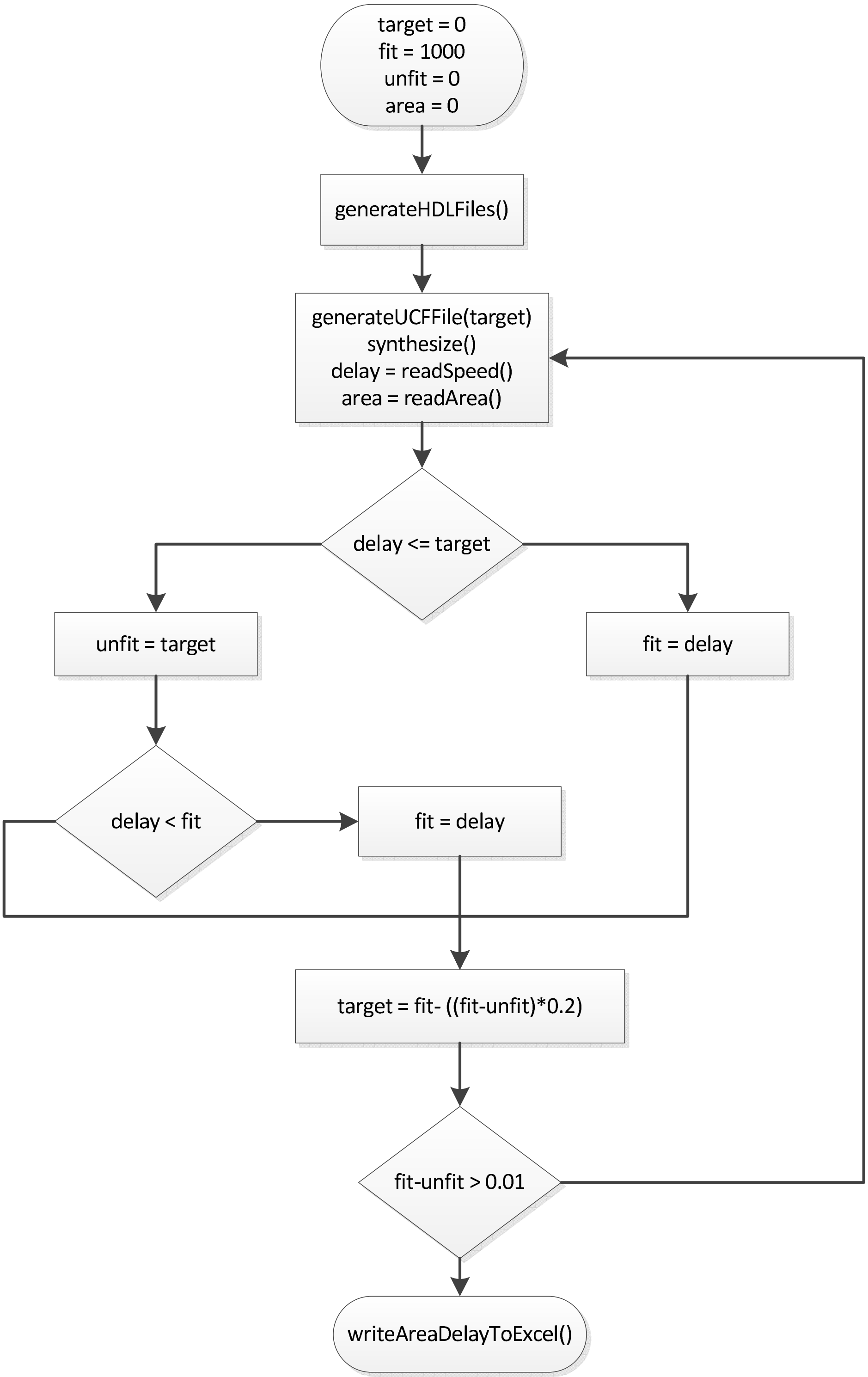


Figure 7.2. Flowchart of synthesis algorithm.

8. CONCLUSIONS

8.1. Contributions

In this paper, we presented the design and implementation of a MPoMPu-RF utilizing BRAMs by exploiting the MPu methodology. Compared to the slice register implementation, this method provides a considerable resource, speed and energy gain. We have pointed out that the multi-pumping techniques used in the literature are not compliant with the FPGA architecture. Hence, we have proposed SR-MPu approach which enables design of MPoMPu-RFs with aggressive multi-pumping. This is due to the fact that SR-MPu approach makes the internal frequency of the MPo-RF nearly independent from MPuF. Our MPoMPu-RF design technique can exploit the MPu as much as possible and results a reasonably profit in terms of speed and resource utilization. In addition to the RF design, the SR-MPoMPu method can be used for any resource which can work at faster internal frequencies compared to its external running frequency.

After the SR based MPu methodology, we have also proposed architecture for the design and implementation of a heterogeneous multi-port register file utilizing BRAMs by exploiting our MPu methodology. This heterogeneous register file can be used safely in single-FPGA heterogeneous systems. In the designed system, each processing element can use as many ports as it requires and bandwidth of the register file can be expanded to meet the data requirements of the processing elements. In addition to these, endianness differences between the processing elements can be handled by means of different wire routing orders to/from HRF. This HRF design occupies less area and consumes significantly lower energy than its alternatives.

APPENDIX A: XilinxTCL SYNTHESIS TOOL

Running TCL scripts from console is troublesome hence parameter list is long. In addition, finding maximum operating frequency (lowest delay) for a RF is very tedious because it requires many synthesis operations until optimum delay value is found. For these reasons, we have developed a program to handle these TCL scripts and for automatic synthesis of these RFs. Screenshot of this program is show in Figure A.1. This program can generate and synthesize emulated or heterogeneous register file. For the synthesis, a tcl script (Synthesis.tcl) sends commands to Xilinx ISE tool by using specific commands. Each synthesis operations create a thread so this program can run simultaneously as many synthesis operations as computer allows. In our server, we can run ten synthesis threads simultaneously. At the end of one synthesis, program reads area and speed results from corresponding Xilinx ISE output files and this process continues until best point is found (see Figure 7.4). For the speed measurement (speed box is checked), program generates registered RFs. The main features of the program are described in following paragraphs;

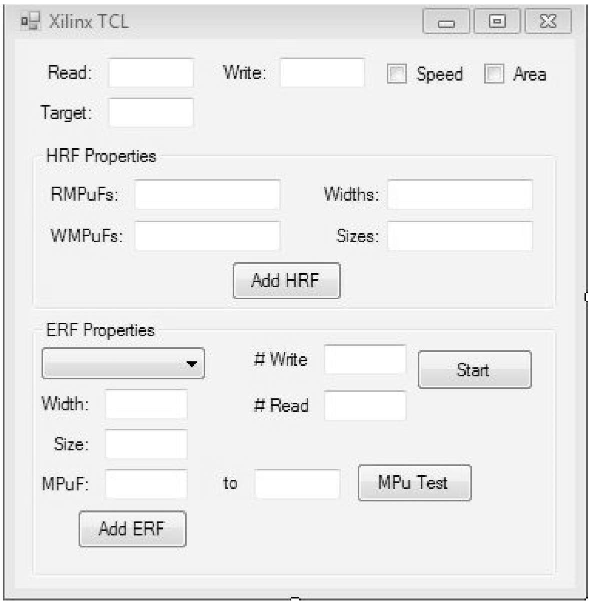


Figure A.1. Screenshot of XilinxTCL program.

Add ERF: This button generates and synthesizes an emulated or heterogeneous register file with specified inputs. If the speed box is checked, program finds the best result for delay. If area box is checked, program also finds the area results in terms of FF-LUT pairs. To find area results, unregistered RF is used.

ADD HRF: The same function with the Add ERF however this time HRF is synthesized.

Start: This button creates number of threads. Each thread synthesizes a register file with increasing read numbers [Read to Read+#Read]. Number of created threads is equal to number of write ports i.e. #Write. The maximum thread size is defined as 10.

MPuTest: This button is used to test the robustness of the RF to increasing MPuF. The same RF is synthesized with different MPuFs and each of them is synthesized.

REFERENCES

1. Woods, R., J. Mcallister, R. Turner, Y. Yi, and G. Lightbody, *FPGA-based Implementation of Signal Processing Systems*, Wiley Publishing, West Sussex, United Kingdom, 2008.

2. Xilinx, *Zynq-7000 All Programmable SoC Technical Reference Manual*, `http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`, accessed at December 2013.

3. Ham, T. J., B. K. Chelepalli, N. Xue, and B. C. Lee, "Disintegrated Control for Energy-efficient and Heterogeneous Memory Systems", *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 424–435, 2013.

4. Patterson, D. A. and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

5. Xilinx, *Virtex-5 FPGA Data Sheet: DC and Switching Characteristics*, `http://www.xilinx.com/support/documentation/data_sheets/ds202.pdf`, accessed at December 2013.

6. Altera, *Mercury Device Family*, `http://www.altera.com/products/devices/mercury/mcy-index.html`, accessed at December 2013.

7. Instruments, N., "Synchronous Communications and Timing Configurations in Digital Devices", `http://www.ni.com/white-paper/6552/en`, accessed at September 2013.

8. Xilinx, *XST User Guide for Virtex-4, Virtex-5, Spartan-3, and Newer CPLD Devices*, `http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/xst.pdf`, accessed at December 2013.

9. Flynn, M. J., "Some Computer Organizations and Their Effectiveness", *IEEE Trans. Comput.*, Vol. 21, No. 9, pp. 948–960, 1972.

10. Xilinx, *Xilinx UG384 Spartan-6 FPGA Configurable Logic Block User Guide*, `http://www.xilinx.com/support/documentation/user_guides/ug384.pdf`, accessed at September 2013.

11. Ghamari, R. and A. Yurdakul, "Register File Design in Automatically Generated ASIPs", *4th HiPEAC Workshop on Reconfigurable Computing (WRC10) , Pisa, Italy.*, 2010.

12. Jones, A. K., R. Hoare, D. Kusic, J. Fazekas, and J. Foster, "An FPGA-based VLIW Processor with Custom Hardware Execution", *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, FPGA '05, pp. 107–117, ACM, New York, NY, USA, 2005.

13. Lie-wen, L., G. Wei-hua, and H. Xiao-long, "Research on Low Power Design Methodology of Register Files Based on FPGA", *Electric Information and Control Engineering (ICEICE), 2011 International Conference on*, pp. 673–676, 2011.

14. Xilinx, *IP Processor Block RAM (BRAM) Block*, `http://www.xilinx.com/support/documentation/ip_documentation/bram_block.pdf`, accessed at December 2013.

15. Altera, *Internal Memory (RAM and ROM) User Guide*, `http://www.altera.com/literature/ug/ug_ram_rom.pdf`, accessed at December 2013.

16. Laforest, C. E., M. G. Liu, E. R. Rapati, and J. G. Steffan, "Multi-ported Memories for FPGAs via XOR", *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pp. 209–218, ACM, New York, NY, USA, 2012.

17. Saghir, M. A. R. and R. Naous, "A Configurable Multi-ported Register File Archi-

tecture for Soft Processor Cores", *Proceedings of the 3rd international conference on Reconfigurable computing: architectures, tools and applications*, ARC'07, pp. 14–25, Springer-Verlag, Berlin, Heidelberg, 2007.

18. Anjam, F., S. Wong, and F. Nadeem, "A Multiported Register File with Register Renaming for Configurable Softcore VLIW Processors", *Field-Programmable Technology (FPT), 2010 International Conference on*, pp. 403–408, 2010.

19. LaForest, C. E. and J. G. Steffan, "Efficient Multi-ported Memories for FPGAs", *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '10, pp. 41–50, ACM, New York, NY, USA, 2010.

20. Jolly, R. D., "A 9-ns, 1.4-gigabyte/s, 17-ported CMOS Register File", *Solid-State Circuits, IEEE Journal of*, Vol. 26, No. 10, pp. 1407–1412, 1991.

21. Garde, D., "Multi-port Register File with Flow-through of Data", U.S. Patent No:4811296, 1989.

22. Corre, Y., J.-P. Diguet, L. Lagadec, D. Heller, and D. Blouin, "Fast Template-based Heterogeneous MPSoC Synthesis on FPGA", *Proceedings of the 9th International Conference on Reconfigurable Computing: Architectures, Tools, and Applications*, ARC'13, pp. 154–166, Springer-Verlag, Berlin, Heidelberg, 2013.

23. Corre, Y., J.-P. Diguet, D. Heller, and L. Lagadec, "A Framework for High-level Synthesis of Heterogeneous MP-SoC", *Proceedings of the Great Lakes Symposium on VLSI*, GLSVLSI '12, pp. 283–286, ACM, New York, NY, USA, 2012.

24. Sawyer, N. and M. Defossez, *Quad-Port Memories in Virtex Devices*, `http://www.xilinx.com/support/documentation/application_notes/xapp228.pdf`, accessed at September 2013.

25. Manjikian, N., "Design Issues for Prototype Implementation of a Pipelined Su-

perscalar Processor in Programmable Logic", *Communications, Computers and signal Processing, 2003. PACRIM. 2003 IEEE Pacific Rim Conference on*, Vol. 1, pp. 155–158 vol.1, 2003.

26. Canis, A., J. H. Anderson, and S. D. Brown, "Multi-pumping for Resource Reduction in FPGA High-level Synthesis", *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pp. 194–197, 2013.

27. Venkateswaran, N., K. Saravanan, N. Nachiappan, A. Vasudevan, B. Subramaniam, and R. Mukundarajan, "Custom Built Heterogeneous Multi-core Architectures (CUBEMACH): Breaking the Conventions", *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1–15, 2010.

28. Yantir, H. E., S. Bayar, and A. Yurdakul, "Efficient Implementations of Multi-pumped Multi-port Register Files in FPGAs", *Digital System Design (DSD), 2013 Euromicro Conference on*, pp. 185–192, 2013.

29. Cruz, J.-L., A. González, M. Valero, and N. P. Topham, "Multiple-banked Register File Architectures", *SIGARCH Comput. Archit. News*, Vol. 28, No. 2, pp. 316–325, 2000.

30. Balasubramonian, R., S. Dwarkadas, and D. Albonesi, "Reducing the Complexity of the Register File in Dynamic Superscalar Processors", *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*, pp. 237–248, 2001.

31. Yongqiang Wu, H. S. E. G., Pengfei Zhu, "Register File Organization to Share Process Context for Heterogeneous Multiple Processors or Joint Processor", U.S. Patent No:0173865, 2013.

32. Rusten, L. T., *Implementing a Heterogeneous Multi-Core Prototype in an FPGA*, Master's thesis, Norwegian University of Science and Technology, Department of

Computer and Information Science, 2012.

33. Zhang, W.-T., L.-F. Geng, D. li Zhang, G.-M. Du, M.-L. Gao, W. Zhang, N. Hou, and Y.-H. Tang, "Design of Heterogeneous MPSoC on FPGA", *ASIC, 2007. ASICON '07. 7th International Conference on*, pp. 102–105, 2007.

34. Kuon, I. and J. Rose, "Measuring the Gap Between FPGAs and ASICs", *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, Vol. 26, No. 2, pp. 203–215, 2007.

35. MathWorks, *Matlab Curve Fitting Toolbox User's Guide*, `http://www.mathworks.com/help/releases/R2013b/pdf_doc/curvefit/curvefit.pdf`, accessed at January 2014.

36. Ruschival, T., "AES Core Project", `http://opencores.org/project,tiny_aes`, accessed at December 2013.

37. McKinley, K., S. Singhai, G. Weaver, and C. Weems, "Compiler Architectures for Heterogeneous Systems", Huang, C.-H., P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua (editors), *Languages and Compilers for Parallel Computing*, Vol. 1033 of *Lecture Notes in Computer Science*, pp. 434–449, Springer Berlin Heidelberg, 1996.

38. Altera, *Quartus II Scripting Reference Manual*, `http://www.altera.com/literature/manual/TclScriptRefMnl.pdf`, accessed at December 2013.