UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

TIAGO TREVISAN JOST

RefreeMIPS - A CGRA-based MIPS architecture

Graduation Project.

Advisor: Prof. Dr. Luigi Carro
Co-advisor: Jorge Quadros

Porto Alegre
2014

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisors, Luigi Carro and Jorge Quadros, and my non-official advisors, Gabriel Nazar and Arthur Lorenzon, for the support and guidance during all the work.

Also, I would like to thank all my friends from Faxinal and Porto Alegre, and coworkers from the Laboratório de Sistemas Embarcados, for the support during the whole time.

Last but not least, I would like to thank my family and my girlfriend, Janaína, for being part of my life, for the encouragement and always believing in me.

RESUMO

CGRAs são dispositivos que exploram reconfigurabilidade de modo a obter alta performance e eficiente consumo de energia. São considerados tão potentes quanto ASICs, porém muito mais flexíveis. Inúmeros desses dispositivos foram propostos, entretanto, nenhum deles elimina o uso de banco de registradores. Register File Free, ReFree, é um CGRA que usa uma abordagem diferente na qual banco de registradores são substituídos por unidades de armazenamento internas aos Elementos de Processamento, e portanto, permitindo uma maior banda em um menor custo. ReFree foi primeiramente projetado para ser usado como um acelerador multimídia, por isso, instruções de fluxo de controle não são originalmente suportadas pela arquitetura.

Este trabalho tem como principal objetivo estender a funcionalidade do ReFree permitindo que instruções MIPS-I possam ser executadas e, portanto, adicionando suporte a instruções de controle de fluxo. Essa nova arquitetura, chamada ReFreeMIPS, pode executar duas ISAs diferentes, uma que permite a execução de instruções em paralelo e outra na qual código MIPS pode ser executado.

Além disso, nós avaliamos o impacto desta implementação em relação à arquitetura original, por meio de medições de área e frequência. Verificamos que houve um acréscimo de 45% no número de FFs e 14% no número de LUTs em relação à arquitetura original. Para fins de experimentação, também comparamos o desempenho de nossa arquitetura com uma implementação padrão MIPS, quando executando aplicações compiladas para este tipo de ISA. Nessa comparação, observamos um *slow-down* de aproximadamente 20% no desempenho de nossa arquitetura. Ressaltamos, porém, que o objetivo do trabalho não foi obter *speedup* em relação ao MIPS, mas, sim, estender a funcionalidade do ReFree.

Palavras-chave: Arquiteturas reconfiguráveis de grão grosso. Banco de registradores. Conjunto de instruções MIPS.

ABSTRACT

CGRAs are devices that exploit reconfigurability in order to achieve high performance and efficient power consumption. They are considered as powerful as ASICs and much more flexible. A myriad of these devices were proposed, though, none of them suppresses the use register files. Register File Free, ReFree, is a CGRA that uses a different approach in which register files are replaced by units of storage internal to the Processing Elements, therefore, allowing much higher bandwidth at a lower cost. ReFree was primarily designed to work as a multimedia accelerator component, thus, control-flow instructions are not originally supported by this architecture.

This work aims at extending ReFree functionality by allowing MIPS-I Instruction Set to run on ReFree and therefore, adding support for control-flow instructions. This new architecture, called ReFreeMIPS, can executes two different ISAs, one that allows multiple instructions in parallel and one in which MIPS code can be executed.

In addition, we evaluated the impact of this implementation in relation to the original architecture, by means of area and frequency. We verified an increase of 45% in FFs and 14% in LUTs when compared to the original architecture. For experimental purposes, we have also compared the performance of our architecture and a standard MIPS implementation, when executing MIPS applications. In this case, we noticed a 20% slowdown on average for our architecture regarding performance. It is important to emphasize, however, that our work does not aim at that, but rather in extending ReFree functionality.

Keywords: Coarse-grained reconfigurable architectures. Register File. MIPS Instruction Set.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ADRES	Architecture for Dynamically Reconfigurable Embedded System
ALU	Arithmetic Logic Unit
ASIC	Application-specific Integrated Circuit
CE	Computation Element
CGRA	Coarse-grained Reconfigurable Architecture
DFG	Data-flow Graph
EX/MEM	Execution/Memory
FF	Flip-Flop
FPCA	Fully Pipelined Composable Architecture
FPGA	Field-programmable Gate Array
FU	Functional Unit
GAM	Global Accelerator Manager
GPR	General Purpose Register
ID	Instruction Decode
IF	Instruction Fetch
ILP	Instruction Level Parallelism
IPC	Instruction Per Cycle
ISA	Instruction Set Architecture
LMU	Local Memory Unit
LUT	Look-Up Table
PE	Processing Element
PC	Program Counter
RC	Reconfigurable Cell
ReFree	Register File ReFree
VLIW	Very Long Instruction Word
WB	Write-back

SUMMARY

1 INTRODUCTION

During the past decades, there have been great advances in the microprocessor industry. Not only microprocessors have become more powerful and flexible, but also they are proofing to be much more power efficient. In spite of that, performance acceleration in microprocessors is not as high as in Application-Specific Integrated Circuits (ASICs). On the other hand, ASICs lack flexibility, since they are usually designed to execute a limited set of tasks and cannot be reprogrammed. To overcome this duality between flexibility and performance, different reconfigurable architectures were proposed (WONG; BROWN, 2008; MEI et al., 2003; SINGH et al., 2000; CONG et al., 2013).

Reconfigurable architectures combine the flexibility of microprocessors with as nearly as the high performance found in ASICs (BECK; CARRO, 2010) and are also proven to be extremely valuable to this ever-increasing time-to-market pressure. Their main characteristics are the ability for adaptation and reconfiguration, best fitting the application requirements, therefore, allowing improvements in performance and/or power efficiency. Designers can take advantage of these features to build products according to the consumer needs and reconfigure them whenever is needed.

In this work, we are considering the *Register file free* (ReFree) as our target archictecture, which is a *Coarse-Grained Reconfigurable Architecture* (CGRA) proposed here at the Laboratório de Sistemas Embarcados (UFRGS). It is composed of a set of dynamically reconfigurable and heterogeneous Processing Elements (PEs). ReFree was primarily designed to execute data-flow applications and, therefore, control-flow instructions are not supported by the original architecture.

In spite of exploring high *instruction level parallelism* (ILP), one of the main drawbacks of ReFree is code density. If an application does not provide good ILP, ReFree will have to store NOPs operations in its PEs, therefore, wasting memory resource. It is important to highlight that the compiler will perform an important role in this sense, as its main responsibility is to find ILP, so no resource is wasted. Having MIPS execute in ReFree will help solving this problem, since MIPS only executes one instruction per cycle at a cost of 32 bits. Balancing the use of both ISAs in ReFree will allow applications to execute fast without compromising memory utilization. For instance, if an application can only allocate a maximum of four instructions to execute in parallel, at least 12 instructions lanes will be storing NOP operations, and therefore, jeopardizing memory resources.

Considering the aforementioned scenario, in this work, we focus on developing and integrating MIPS to ReFree, also called ReFreeMIPS, allowing control-flow instructions to execute in ReFree. In addition to executing ReFree ISA, ReFree will be able to execute any MIPS-I legacy code, avoiding the need for code recompilation of such applications. Moreover, ReFreeMIPS is the part of ReFree where control-flow and MIPS instructions will be executed within the same organization.

The remaining of this work is organized as follows: Section 2 presents an overview of reconfigurable architectures and how they are categorized. It also describes some reconfigurable architectures proposed in the academia, such as, ADRES, ρ-VEX, MorphoSys and FPCA, highlighting their main characteristics regarding architecture, organization and compilation process. Section 3 details the original ReFree architecture and the motivation behind it, also describing the PE structures and the network that interconnects them. In section 4, we present the implementation of the proposed architecture, namely ReFreeMIPS, focusing on the main aspects of this new design compared to the original architecture. We cover aspects from the logic implementation to hazards detection mechanism. In section 5, we evaluate the solution by comparing with a standard MIPS implementation, as well as present the results regarding area and frequency of the solution. Finally, Section 6 concludes this work.

2 RECONFIGURABLE ARCHITECTURES

Reconfigurable architectures can be classified according to a different set of categories, such as, granularity, reconfiguration models and coupling. Granularity refers to the smallest block of which a reconfigurable system is made (VASSILIADIS; SOUDRIS, 2007). According to this, reconfigurable systems are divided into two groups, fine-grained and coarse-grained systems. Fine-grained architectures are systems that can be reconfigured at a bit level. They are characterized by being very flexible, however, with a high reconfiguration overhead due to the size of its configuration memory. On the other hand, CGRAs manipulate larger reconfigurable blocks, such as, *Arithmetic Logic Units* (ALUs) and Memories. Despite being less flexible, they are more suitable for performing word-level data processing (VASSILIADIS; SOUDRIS, 2007), since fewer bits are needed to reconfigure the system, improving performance and area utilization. CGRAs are especially good to execute data-flow kernels and usually come integrated to a main processor, working as an accelerator. Whenever an application must execute on the CGRA will be decided either by the compiler or on-the-fly throughout specialized hardware (FERREIRA et al., 2013). Examples of CGRAs are ADRES (MEI et al., 2003) and MorphoSys (SINGH et al., 2000).

According to the reconfiguration models, architectures can be classified as statically or dynamically reconfigurable. Statically reconfigurable architectures can only be programmed once for its running time. In order to reconfigure the system, it has to be halted and a new configuration has to be applied again. Dynamically reconfigurable architectures are much more flexible and a new configuration can be loaded or unloaded during the operation of the system, albeit costing the system run slower.

Regarding the host processor integration, reconfigurable architectures can be classified as tightly coupled or loosely coupled (SHANNON, 2008). In loosely coupled systems, the reconfigurable logic is used as an application-specific unit or external accelerator, connected to the main processor via a bus, and operating asynchronously. The main processor will enable its units and supply it with a set of parameters. The host will be able to perform other tasks while the reconfigurable system is also executing, improving performance. However, since they operate asynchronously, a resynchronization mechanism must be added when the reconfigurable units finish executing. In tightly coupled systems, the reconfigurable logic is treated as an internal functional unit of the microprocessor, avoiding the need for a bus interconnection and the communication latency between host and CGRA (BECK; CARRO, 2010).

The performance of a system is directly affected by which characteristics are being considered at the architecture. Choosing which characteristics to adopt require a deep understanding on the drawbacks and advantages of each type. A general rule of thumb states that using characteristics for improving performance will cost flexibility and vice-versa. Likewise, a good compiler is a key component on that process. If the compiler cannot find enough instructions to fill the PEs with, meaning there are too many dependencies, resources will be wasted, compromising area utilization. Different techniques are been explored to deal with those problems, such as, loop unrolling, software pipelining (RAU, 1994), among others. Applying such techniques improve hardware usability and performance.

Furthermore, the *principle of locality* states that

> programs tend to reuse data and instructions they have used recently. A program may spend 90% of its execution time in only 10% of the code and an implication of locality is that we can predict with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past (PATTERSON; HENNESSY, 1996).

Several reconfigurable architectures that take advantage of this program property to improve performance on applications were proposed. The following sub-sections describe the main aspects of them.

2.1 ADRES

Architecture for Dynamically Reconfigurable Embedded System (ADRES) consists of a *Very Long Instruction Word* (VLIW) processor, which is tightly coupled to a coarse-grained reconfigurable matrix (MEI et al., 2003). The main characteristic of the architecture is its tight integration between processor and reconfigurable matrix.

According to Mei et al. (2003), reduced communication cost and simplified programming model are two important advantages of the ADRES architecture in comparison to others. Most reconfigurable architectures are loosely coupled and require synchronization between the main processor and reconfigurable matrix.

2.1.1 Architecture and Organization

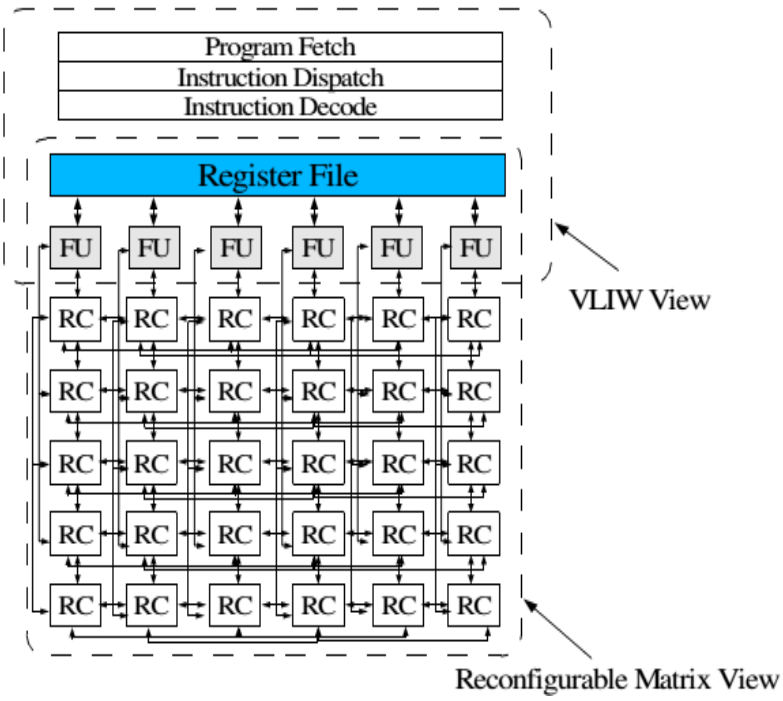
As depicted in Figure 2.1, the ADRES core is composed of functional units (FU) and a customizable number of register files. ADRES can be represented through two views, one from the VLIW and one from the reconfigurable matrix.

> These FU's perform two tasks: they make up the functional units of the general-purpose VLIW processor when this processor is executing code, and act as additional configurable FU's when the device is executing parallel code on the reconfigurable array. These FU's have direct connections to the two global register files through dedicated read and write ports. (MEI et al., 2013 *apud* KWOK; WILTON, 2005)

The VLIW core consists of basic functional units that execute control-flow and data-flow operations. The reconfigurable matrix is composed of reconfigurable and heterogeneous FUs and it is used to execute only data-flow operations. Both components have access to memory and register files and do not execute concurrently, therefore, no previous data preparation or synchronization are required between them.

Architecture properties such as, resource allocation, communication topology, are defined using an XML-based architectural description language.

Figure 2.1 - ADRES Core



Source: Mei et al. (2003, p. 4).

2.1.2 Compilation

ADRES uses the IMPACT compiler framework (CHANG et al., 1991) to generate an intermediate code representation called *lcode.* From this representation, the DRESC (MEI et al., 2002) compiler performs a set of optimizations and code analysis, trying to identify which loops can be accelerated by the reconfigurable matrix. The compilation process is divided in two paths, one for the VLIW processor and one for the reconfigurable matrix (MEI et al., 2002). Loops that can explore high ILP are schedule to execute on the reconfigurable matrix using a modulo scheduling algorithm (RAU, 1995).

2.2 ρ-VEX

ρ-VEX is an extensible and reconfigurable VLIW processor primarily designed to provide high ILP for multimedia applications (WONG; VAN AS; BROWN, 2008). This architecture is based on the VEX ISA (FISCHER et al., 2004), which already offered a qualified toolchain to compile applications.

2.2.1 Architecture and Organization

ρ-VEX is extremely flexible and allows two, four, and eight-issue configuration, meaning up to eight instructions can be executed in parallel. A four-stage pipeline was used consisting of *fetch*, *decode*, *execute,* and *writeback* stages. Figure 2.2 shows an example of a 4-issue configuration for ρ-VEX, composed by four ALU units, two multipliers, one branch control unit, one memory access unit, and 64 32-bit general purpose registers (GPR).

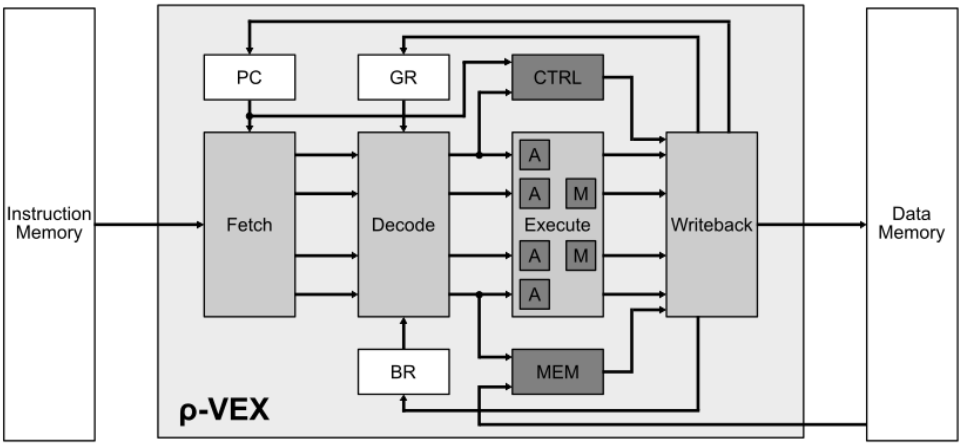
2.2.2 Compilation

A software development toolchain for ρ-VEX is provided by Hewlett-Packard, that goes from an extremely powerful compiler to a simulator (WONG; VAN AS; BROWN, 2008). The compiler is able to analyze and perform optimizations through loop-unrolling and modulo-scheduling algorithms.

Architecture properties such as, number of issues, GPR, branch registers as well as memory and type of units are all customizable and the compiler is completely aware of the

target architecture, meaning there is no need to add extra hardware to handle any type of dependency.

Figure 2.2 - 4-issue ρ-VEX example



Source: Wong; van As; Brown (2008, p. 2).

2.3 MORPHOSYS

The MorphoSys is a tightly coupled reconfigurable system targeted to be used in applications with inherent data-parallelism, high regularity, and high throughput requirements, such as, video compression, graphics and image processing, data encryption, and DSP transforms (SINGH et al., 2000). It was developed to investigate the impact and effectiveness of reconfigurable architectures when combined with a general-purpose processor.

2.3.1 Architecture and Organization

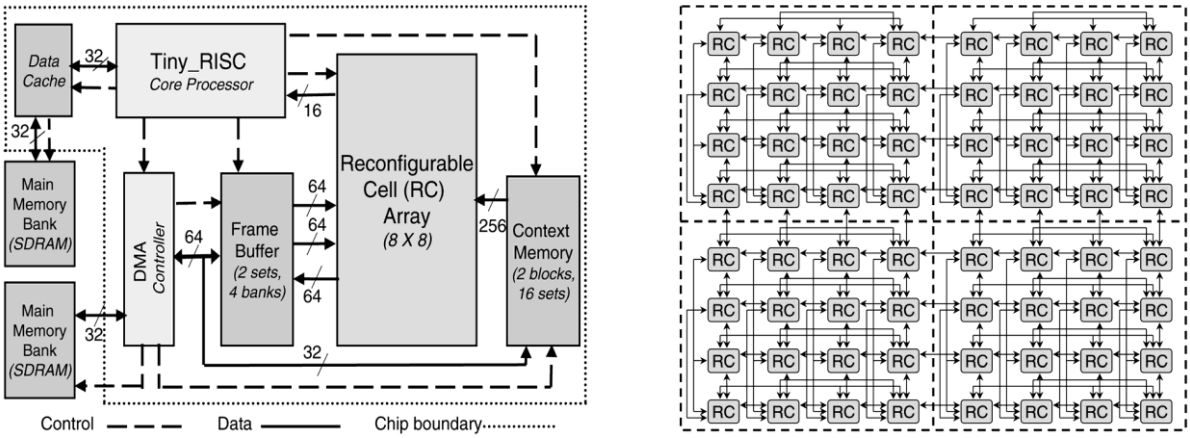
MorphoSys consists of an advanced-RISC processor, a reconfigurable array of processing cells and a memory interface unit (SINGH et al., 2000), as depicted in Figure 2.3. The reconfigurable component consists of an 8x8 array of identical processing elements or reconfigurable cells (RC). The frame buffer component interconnects the RCs with an external RAM memory using DMA for fast-speed data transferring.

Each RC is composed of an ALU-multiplier, a shift unit, and two multiplexers to select its inputs. It also has an output register and a register file. A Context Memory connected to the reconfigurable component defines the functionality of each RC.

2.3.2 Compilation

A programming environment was created in order to compile hybrid code for MorphoSys (SINGH et al., 2000). The compiler requires manually partitioning code informing which functions will execute on the reconfigurable component and which will execute on the main processor.

Figure 2.3 - MorphoSys System



(a) MorphoSys System interconnection (b) Reconfigurable Cells

Source: Singh et. al. (2000)

2.4 FPCA

Fully Pipelined Composable Architecture (FPCA) is a loosely coupled CGRA that enables full pipelining and dynamic composition to improve energy efficiency by taking full advantage of abundant transistors (CONG et. al. 2013). FPCA allows multiple applications to run at the same time, which contributes to improve resource utilization.

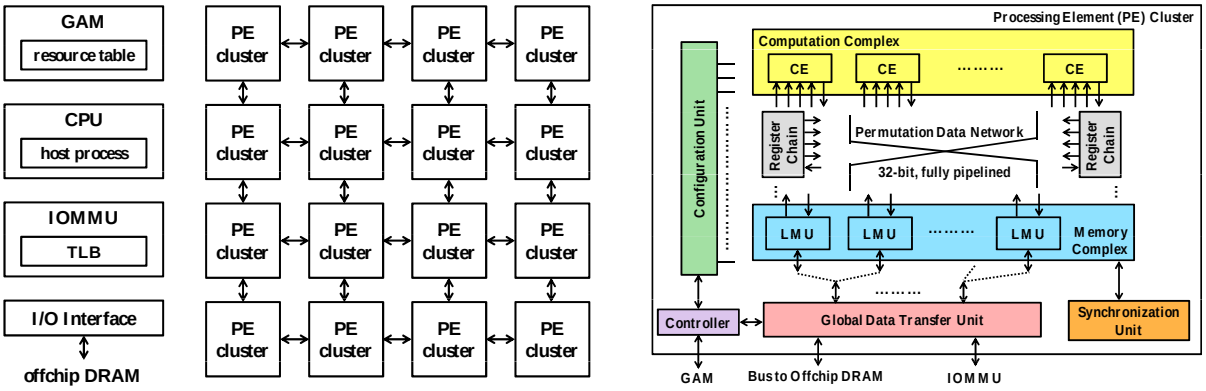
2.4.1 Architecture and Organization

The FPCA architecture consists of a general purpose CPU, a cluster of PEs, a resource table and memory units, as shows Figure 2.4. The type of the instruction determines where the instruction is scheduled to execute, e. g., control-flow instructions are more likely to execute in the CPU and data-flow operations are better explored in the reconfigurable logic.

The global accelerator manager (GAM) will receive all instructions that need executing in the reconfigurable component and allocate them in the clusters. Each cluster

contains a set of 32-bit heterogeneous PEs including computation elements (CEs), local memory units (LMUs) and register chains to act as ALUs, on-chip buffers and registers respectively (CONG et. al. 2013). As one can notice, this architecture allows a high ILP exploration, due to multi-application execution and its reconfigurable component size.

Figure 2.4 - FPCA System



(a) Organization (b) Internal PE Cluster

Source: Cong et al. (2013)

2.4.2 Compilation

An LLVM-based compiler was designed to map kernels into the FPCA platform. The compiler transforms the input kernels into a data flow graphs (DFG) and a series of analysis and optimization is performed. Due to the size of the architecture, the compiler may schedule multiple applications in parallel to maximize resource utilization and performance.

3. REFREE

Although all architectures described in Section 2 are reconfigurable, each of them has different characteristics. ADRES and MorphoSys belong to the class of tightly coupled CGRAs, FPCA is categorized as loosely coupled and ρ-VEX, though being a VLIW processor, has its reconfigurable nature. Our target architecture called ReFree (Register file Free) architecture, also fits in this type of classification. Those architectures also differ in area utilization and even performance. Furthermore, the type of applications in execution is another factor to take into account when one needs to choose the appropriate reconfigurable architecture to consider in its project.

3.1 Architecture

ReFree is a dynamically reconfigurable CGRA architecture proposed at the Laboratório de Sistemas Embarcados (LSE) - UFRGS, primarily designed to execute and speed up multimedia applications. ReFree is a tightly coupled architecture with a distinguishing feature, the complete absence of a register file.

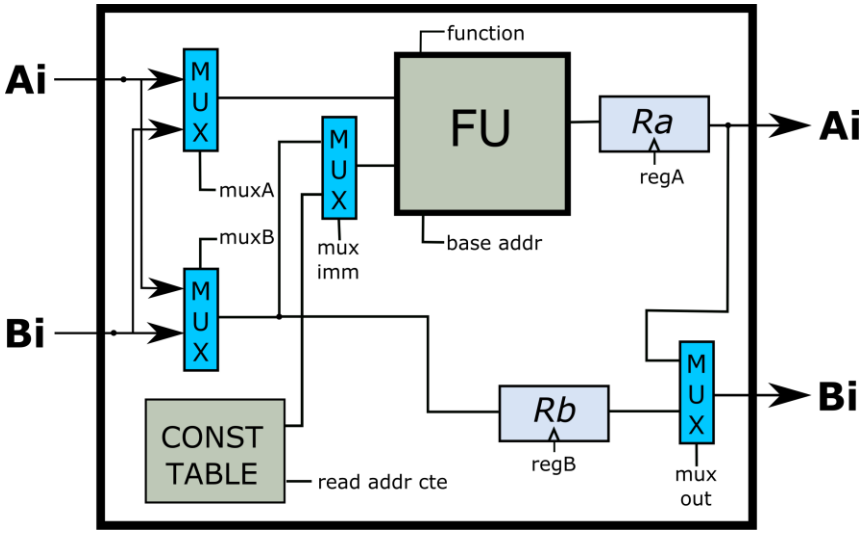
Register files are frequently considered one of the bottlenecks for exploring ILP in multiple-issued architectures. Kwok and Wilton (2005) shows that the numbers of read and write ports in register files are directly proportional to the increasing in area utilization. The bandwidth required also increases as the number of ports increases. An 8-issued VLIW processor, for example, requires sixteen read ports in order to provide two operands for each execution lane. Each lane potentially produces one result; therefore, eight write ports are also required. A register file able to meet such requirements is costly and it is not suitable for our architecture.

ReFree suppresses the need of a register file by spreading register through all its PEs. The architecture is composed of 16 heterogeneous processing elements and two registers are distributed throughout each unit. This is equivalent of having a total of 32 read ports and 16 write ports, without having the penalty of a costly register file and still having flexibility. Furthermore, 32 registers is the exact number of registers that architectures need to extend support to MIPS.

Figure 3.1 shows the original ReFree PE structure. The FU can operate as memory interface, multiplier unit and ALU. In general situations, there will be one memory unit, one or two multiplier units and the rest of them will operate as ALU, however, the configuration is

completely customizable. Internal multiplexers are used to invert the values from one network to another. As it will be shown the next section, this structure had suffered some modifications in order to adapt to MIPS.

Figure 3.1 - Original ReFree Structure

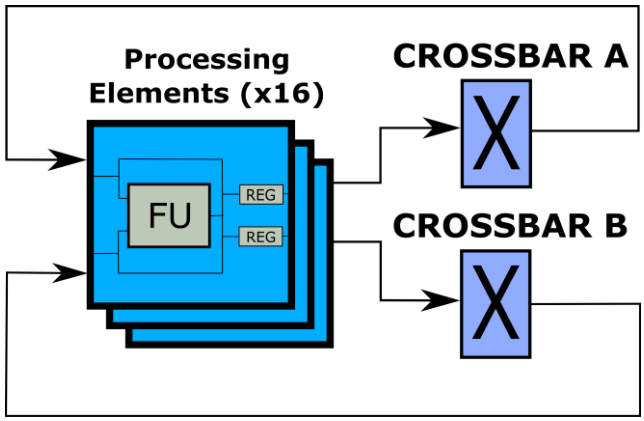


Source: Created by the author.

3.3 Interconnection Topology

Processing elements are interconnected using two crossbar networks, one for Registers *Ra* and one for Registers *Rb*, as illustrated in Figure 3.2. Such configuration reduces area cost of two if compared to full crossbar networks, mostly because rotating values between Crossbar A and Crossbar B can be done internally in the PE.

Figure 3.2 - Interconnection Topology



Source: Created by the author.

Additionally, the crossbar routing is O(1), and there is no placement problem since any PE could achieve any other PE (FERREIRA et al, 2013). Although the mapping is NP-complete even for crossbar-based CGRAs, experimental results in (FERREIRA et al., 2013) demonstrated it is possible to reduce compilation time up to 6 degree of magnitude when compared to mesh-based CGRAs. In general, PEs on mesh-based CGRAs connect to their neighbors, meaning an extra step would be required to verify if an output can be used as an input on a specific PE.

3.2 Compilation

Two different sets of tools are used to compile code for ReFree. First, we use the HP VEX compiler, same as used in ρ-VEX, to generate an assembly representation of the application. The generated code is used as input to another tool, developed at LSE, which will translate the ρ-VEX instructions into the ReFree binary code. It is worth mentioning that all code analysis is performed with the HP VEX compiler.

In terms of performance, ReFree may issue 16 instructions per cycle (IPC), which is significantly faster than a regular single-issued processor. Nonetheless, this performance depends on how efficient the compiler is and whether it is able to extract ILP from applications.

The following table summarizes the most important aspects of ReFree in comparison to the architectures from section 2.

Table 3.1 - Comparison between architectures

	Coupling	Type of Processor	Num. RUs	Num. Regs
ReFree	Tightly	MIPS	16 units	32
ADRES	Tightly	VLIW	Customized	Customizable
ρ-VEX	N/A	VLIW	2, 4 or 8	64
MorphoSys	Tighly	TinyRisc	64 units	N/S
FPCA	Loosely	N/S	64 units	N/S

N/A - Not Applicable, N/S - Not Specified

Source: Created by the Author

4. REFREEMIPS

ReFreeMIPS is the name given to the implementation and support of the MIPS-I ISA in ReFree. Originally, ReFree only supported data-flow instructions, mostly due to its characteristic of being a multimedia accelerator. However, it became indispensable to execute not only data-flow, but control-flow operations, as well. With this new afterthought, ReFree now can be used as the main processor of a system, not only as an accelerator.

Due to the high credibility in the academia, MIPS was the primary option of processor that adds the aforementioned features to our architecture. MIPS instructions will execute in one of the 16 PEs, depending on the type of instruction. Alongside this MIPS extension, it is important that no performance degradation occurs in the original ReFree ISA, i. e., the critical path of the system should not increase significantly as it interferes with the overall frequency of the system.

The following sections describe every important detail of ReFreeMIPS, going from architectural features to how instructions are mapped into PEs.

4.1 Operating Modes

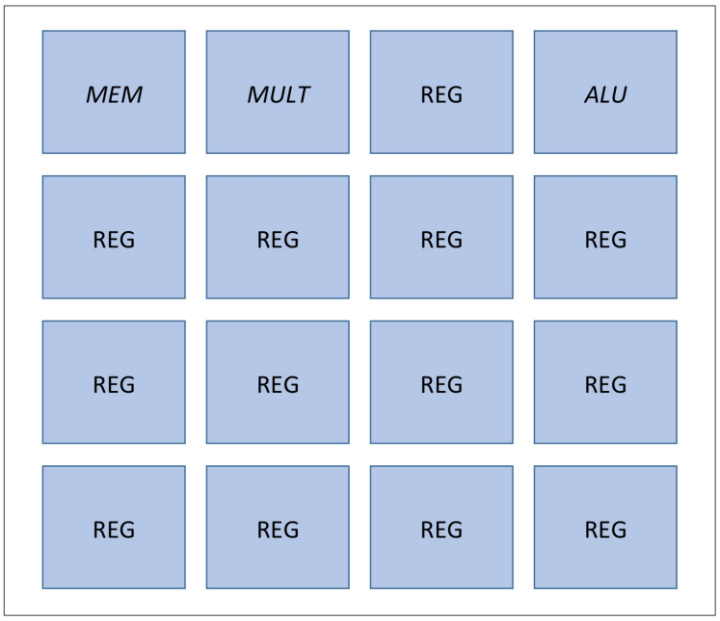
Original ReFree had no differentiation in terms of operating modes, as it could only operate in one single mode, the one executing data-flow instructions. One may notice that the architecture can now operate in two different modes: a single-issued MIPS instruction mode and a multi-issued ReFree instruction mode. The former, also called MIPS mode, is used to execute MIPS ISA code in which branch and jump instructions will be mapped. The later, called ReFree mode, is used to execute ReFree instructions. Furthermore, both operating modes use the same organization and compute operation using the same set of register, which are those inside the PEs.

ReFree mode is extremely powerful, though it needs a 512-bit instruction word. MIPS mode, on the other hand, issues only one instruction at a time and needs 32 bits of instruction word. Combining both modes appropriately may result in a substantial gain on code density without a high performance compromise. For now, let us assume that the system only operates in MIPS mode, which is the operating mode that ReFreeMIPS added in.

4.2 Processing Elements

Similar to original ReFree, ReFreeMIPS consists of 16 Processing Elements that can be classified according to their function on MIPS and ReFree Modes. Figure 4.1 illustrates one of the possible configurations of ReFreeMIPS. The location of *Mem*, *Mult* and *ALU* PEs are customizable through a configuration file. Operating modes will determine how PEs are used.

Figure 4.1 – Example of a possible ReFreeMIPS Structure



Source: Created by the author.

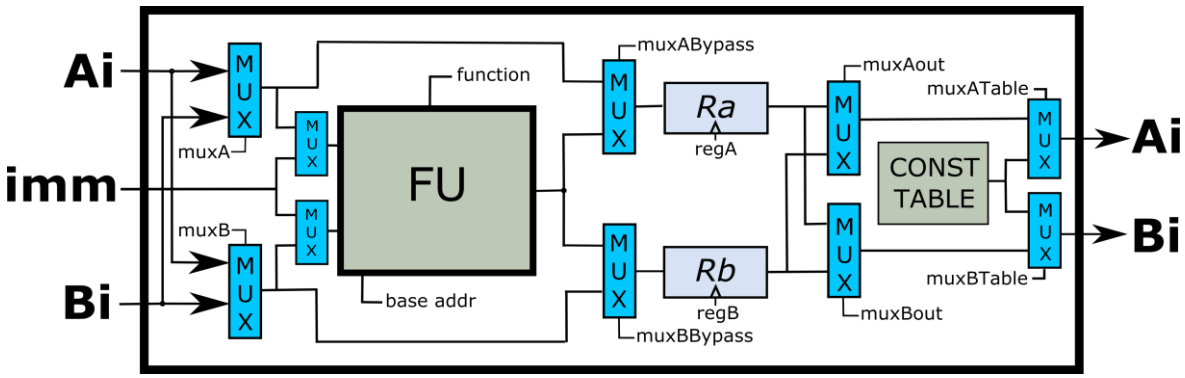
MIPS mode needs three special PEs, called MIPS PEs, to execute instructions, one to work as memory interface, one for multiplication, and one for ALU and branch operations. Others, namely non-MIPS PEs, are only needed to give access to the registers. *Mem*, *Mult* and *ALU* Units denote MIPS PEs, while *Reg* Units represent non-MIPS PEs.

Mem and *Mult* denote PEs that operate as both memory interface and multiplier unit, respectively. *Reg* units are those that only operate as Memory Interface, Multiplier or ALU in ReFree mode; in MIPS mode, *Reg* units are only used to give access to registers, and Functional Units are not used. *ALU* denotes a PE used in MIPS mode to execute ALU and branch instructions; and in ReFree mode, it only executes ALU instructions.

Some main modifications were required in the PEs to introduce compatibility with MIPS. Figure 4.2 and 4.3 illustrates a MIPS PE and a non-MIPS PE, respectively. The main

difference between them is the presence of an immediate input signal on MIPS PEs, which is necessary due to the immediate value come from the MIPS instruction word. Instructions such as loads, stores, and immediate ALU instructions will need this input signal. Nevertheless, the number of multiplexers, the position of the constant table, and the immediate signal input are the differences between the original PE structure, shown in Figure 3.1, and Figures 4.2 and 4.3.

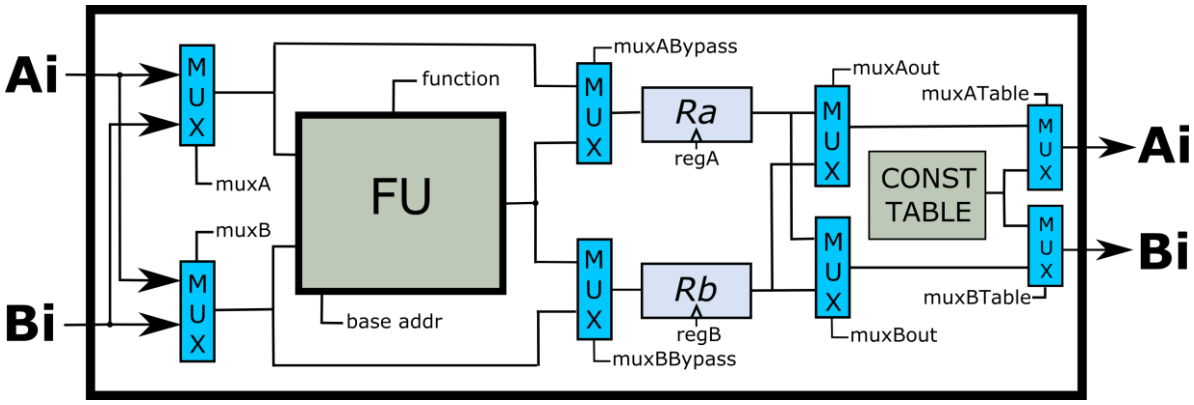
Figure 4.2 - Internal Structure of MIPS PE



Source: Created by the author.

In MIPS mode, non-MIPS PEs always bypass FUs to store new values in registers. A value stored in *Rb* can easily be used in any *Ra* and vice-versa, using the internal multiplexers.

Figure 4.3 - Internal Structure of non-MIPS PE



Source: Created by the author.

Common to all PEs are two aforementioned registers *Ra* and *Rb*, a constant table, and a functional unit (FU). Registers *Ra* and *Rb* output values directly to the network that interconnects the PEs. Placing registers at the output of the PE helps reducing cost on the

mapping process, since there is no need to choose where that value should be stored after computing the operation.

The constant immediate table holds up to eight predefined immediate values that may be used to operate as one of the sources in the FU. The compiler might use this table to perform operations without having to rotate immediate values coming from the current instruction, simplifying the mapping. Nonetheless, the constant table is only needed for fetching the constant 0 when the MIPS mode is on. This table is mostly used to fetch immediate values when in ReFree mode.

Furthermore, PEs are enumerate as PE_i, where $i = [0 - 15]$, for instance, figure 4.1 shows that Memory, Multiplier and ALU are located in PE_0, PE_1 and PE_3, respectively. Also, their registers are identified as $Ra_i = 2i$ and $Rb_i = 2i + 1$, e. g., PE0 has Registers 0 and 1, while PE1 holds values for Register 2 and 3, and so on. Crossbar A connects even-numbered registers and crossbar B connects odd-numbered registers. This definition is essential to identify where each register is located when executing MIPS instructions.

4.3 Pipelining

ReFreeMIPS pipeline consists of four stages, meaning that up to four instructions will be in execution during one single clock cycle. ReFreeMIPS uses one pipeline stage less than original MIPS, because the memory interface unit uses a dedicated adder to calculate the target address and, at the same time, reads/writes in memory. The pipeline is divided into instruction fetch (IF), instruction decode (ID), execution/memory (EX/MEM) and write-back (WB).

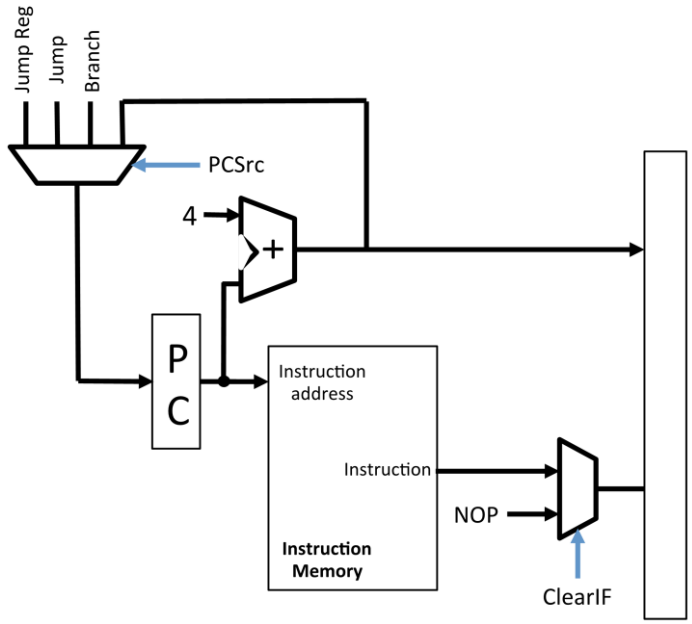
4.3.1 Instruction Fetch

The execution starts when a new instruction is fetched from instruction memory, having the Program Counter (PC) serve as the address. Alongside, the PC is incremented by 4 to prepare to the next fetching procedure.

Figure 4.4 shows how the fetching process occurs. According to MIPS ISA, the instruction address can come from four different places, a branch taken instruction, a jump instruction, a jump register instruction, and a regular PC+4 situation. The *PCSrc* signal selects which address will be used and come from the Control Unit that is located in the Decode

Stage. The *ClearIF* signal checks if the pipeline needs flushing, by replacing the current fetched instruction with NOP.

Figure 4.4 - Instruction Fetch Stage



Source: Created by the author.

4.3.2 Instruction Decode

The ID stage is where all signals used in ReFree PEs are generated. This stage will contain the logic necessary to decode instructions, explore forwarding and generate the control conditions of all instructions. The control unit is located at this stage and the logic behind it is detailed herein.

A regular MIPS architecture usually decodes register information from the instruction and uses it as input to a register file. This register file outputs the values that correspond to those registers and the control unit will select whether those values will be used as input to an FU. In the proposed architecture, however, this is not possible, mostly due to the absence of a register file. When a register needs to be read, crossbars control signals will be configured to rotate those registers to the proper PE. More than only crossbar signals, the decode stage needs to generate a set of signals to be used in EX/MEM and WB stages.

The following table lists those signals and their meaning:

Table 4.1 - Input Signals in ReFreePEs

Signal	Total of Bits	Bits per PE	Function
cfg_muxa_fu cfg_muxb_fu	3	1	Selection between an immediate value and a register on a MIPS PE
cfg_muxABypass cfg_muxBBypass	16	1	Selection between FU output and bypassed-FU signal
cfg_muxa cfg_muxb	16	1	Used to invert values coming from crossbars A and B
cfg_a_muxconst cfg_b_muxconst	16	1	Used to send a constant value to crossbar A or crossbar B
cfg_rega cfg_regb	16	1	Write enable signals for registers A and B
cfg_muxAOut cfg_muxBOut	16	1	Used to invert values coming that goes to crossbars A and B
cfg_rdaddr	48	3	Used to inform the constant table address
cfg_memaddrbase	32	Only in Memory PE	Used by the MEM PE to inform the base memory address of the memory (usually used as 0x00)
cfg_funct	128	8	Functional unit operation
cfg_neta cfg_netb	64	4	Used to manipulate crossbars A and B
imm	32	Common to all PEs	Used in MIPS PEs to send immediate data that comes from either MIPS instruction or PC address.

Source: Created by the author.

4.3.3 Execution/Memory

The EX/MEM stage happens inside the PEs, by receiving the input signals coming from the decode stage and manipulating registers, multiplexers, and FUs. As ReFree PEs consists of heterogeneous units, a PE can operate as memory interface, ALU, and multiplier.

As already mentioned, in MIPS mode only three units are used to execute the instructions, while the rest of them are needed only as unit storage.

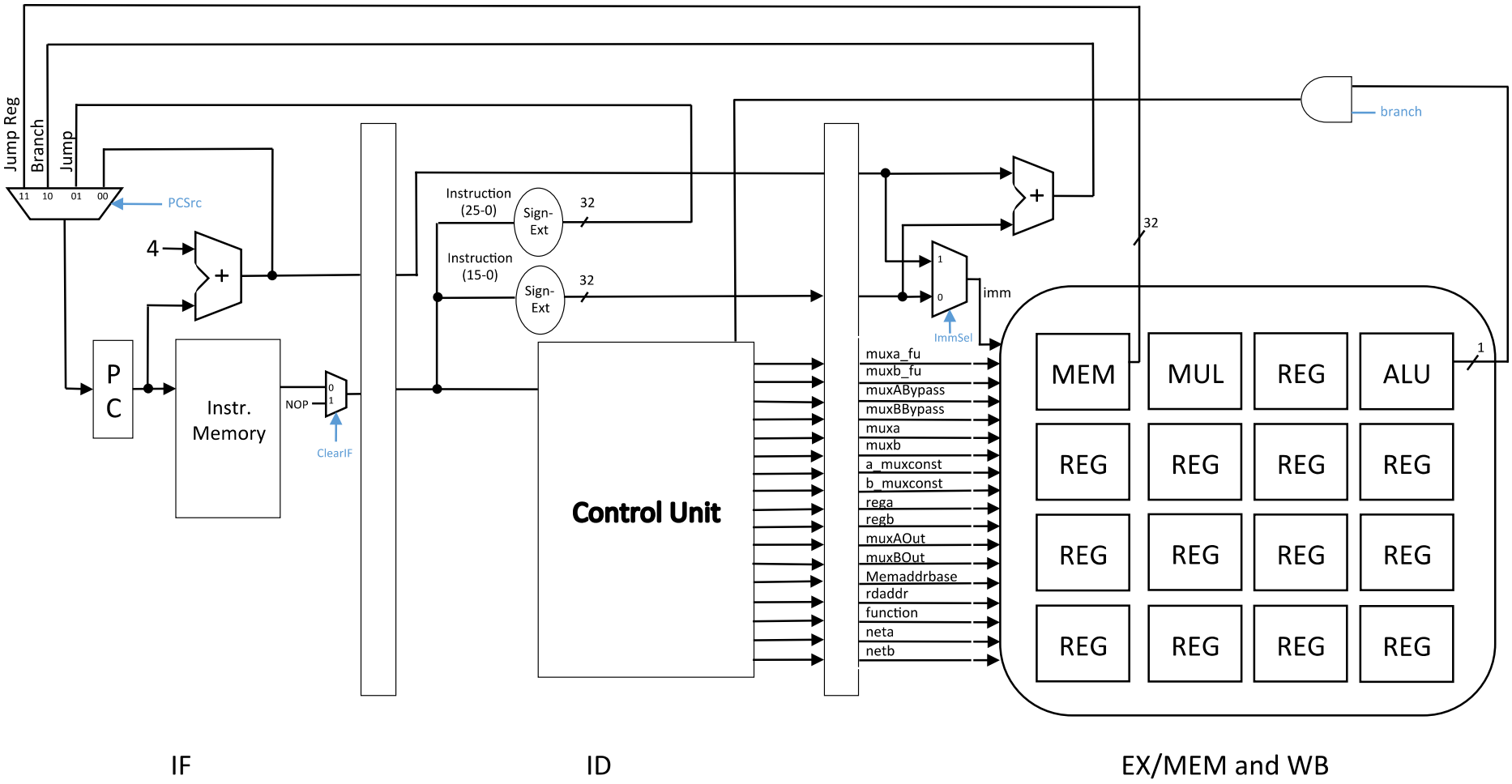
The regular EX and MEM stages from MIPS were joint in only one, because the MEM unit calculates the address and, at the same cycle, reads/writes in memory. Section 5 shows that the critical path of the system is composed by *multiplier* + *crossbar* path, therefore, having memory be mapped in BRAM blocks (XILINX INC, 2011) speeds up the access and, therefore, our system did not suffer performance loss because of it.

4.3.4 Write-back

The WB stage also occurs inside the PEs, by receiving the same input signals coming from ID. In a common operation flow, the WB stage is where the instruction result is written back to the correct register. In other situations, this stage does nothing, since the instruction completed executing during the EX/MEM stage.

These four stages compose ReFreeMIPS pipeline. Figure 4.5 illustrates the whole system. It is worth mentioning that at decode, the control unit will generate all signals needed in EX/MEM and WB stages.

Figure 4.5 - ReFreeMIPS Pipeline



Source: Created by the author.

4.4 Control Unit

In ReFree mode, the architecture needs no clever mechanism to use as control unit, mostly because every information comes directly from the instruction word. Mux-set bits, register-enabling bits, crossbars configuration are all handled and generated accurately by the compiler, as well as hazards detection, similar to most VLIW processors. The compiler spares the need for control logic as it can resolve most of the conflicts during compilation time. The only logic needed is to separate the instruction bits to the corresponding input signals in ReFree. This approach is easily achievable, since no control-flow instructions are executed in this mode.

On the other hand, MIPS mode needs more than just a spliting-instruction logic. Instruction needs both combinational and sequential circuits at the control logic to execute in ReFree PEs flawlessly. The logic behind the control unit is essential for MIPS operating mode, since ReFree configuration is not located at the intruction itself.

4.4.1 Logic

To understand the logic of the control unit, first one needs to take a look at how instructions are executed. Figures 4.2 and 4.3 show the internal structure of the two different PEs in ReFree. It is interesting to observe that PEs always store operation results in registers and those will be the same registers used. According to the configuration specified herein, memory instructions will execute in PE0, multiplication-like instructions in PE1, and ALU and Branch instructions in PE3. That creates an issue, because an operation like *ADD R2, R3, R4* would overwrite R6 (which is one of the output registers in PE3).

To address the aforementioned problem, one needs to remember, in MIPS, R0 holds the constant value of 0 and it cannot be overwritten. Therefore, it is useless to waste a register only to store a constant value. Instead, we will use a constant table to store the constant 0 and R0 will be used as temporary storage for situations like the one described above. Considering the example on the last paragraph *ADD R2, R3, R4* and the configuration presented in Figure 4.1, the EX stage will be mapped as two operations *ADD R6, R3, R4* and *MOV R0, R6.* Likewise, the WB stage will be mapped as *MOV R2, R6* and *MOV R6, R0*. Notice that we are only considering the execution of this single instruction and executing other instructions later might not result in the same mapping.

Additionally, the temporary value stored in R0 should remain there until the PE it belongs is in use or an instruction overwrites that register. This procedure serves not only for ALU instructions, but also when multiplication-like instructions are executed. Memory instructions do not need such treatment, because the memory is located at PE0.

When the destination register is located at the same PE where the operation is execute, that approach is not required. For instance, *ADD R6, R2, R4* and *ADD R7, R2, R4* will both overwrite the correct register (they both are output registers in PE3), thus there is no need to send values to R0. For cases like these, the WB stage does no operation.

4.4.1.1 Hazards

The logic also should be able to handle the three types of hazards: data, control, and structural hazards. Most of the data hazards can be dealt using the forwarding technique (PATTERSON; HENNESSY, 2009). This is possible because ReFreeMIPS has only four stages, one less than regular MIPS architecture, and there is no need to forward values two stages backwards. When a previously-written register needs to be used in the EX stage, it is always possible to forward that information from the previous instruction in the WB stage. The exception happens when a HILO write-to and read-from instructions are executed subsequently, as the multiplication units needed a two-cycled implementation design in order to maximize performance.

However, structural hazards will occur more frequently than data hazards. For instance, when the PE used in the EX stage is the same that needs to be used in the WB stage for the previous instruction. In this case, the logic needs to handle by stalling the instruction in the EX stage. For example, considering the configuration of Figure 4.1, when instruction ADD R2, R4, R7 is in WB stage and MULT R5, R10 is in EX stage, the instruction in WB needs to complete before the instruction in the EX stage continue.

In the case of control hazards, ReFreeMIPS always assumes that the next instruction should be fetched at PC+4, therefore, jump and branch-taken instructions in execution will cause the pipeline to fetch the incorrect instruction. When that happens, ReFreeMIPS must flush the instructions in the previous pipeline stages. Jump instructions need only to flush instructions in the IF stage, because it is possible to solve this unexpected event in the ID stage. Nevertheless, jump register and branch-taken instructions will only solve control situations in EX stage, therefore, we must flush instructions in IF and ID stages.

4.4.1.2 Control signals

Additionally to the signals shown in Table 4.1, the control unit needs to generate signals for multiplexers *PCSrc, ClearIF,* and *ImmSel*. The first is used to select the next PC address used to fetch in the instruction memory. *ClearIF* flushes the IF stage in case a control-flow instruction occurs and *ImmSel* selects which Immediate signal needs to be input in the PEs.

PCSrc signal operates as follows:

```
PCSrc <=  10   when InstructionEX = BranchType and BranchTaken = 1  else
          01   when InstructionEX = JumpRegType                     else
          11   when InstructionID =  Jump                          else
          00
```

Since Branch and Jump Register instructions are only resolved in EX stage, it is important to give priority to the older instruction, i. e., InstructionID will only be tested after the instruction in EX stage is neither a branch-taken nor jump register instruction.

ClearIF signal clears the IF instruction by replacing the instruction fetched from memory. The logic for this signal is:

```
ClearIF <=  1   when InstructionEX = BranchType and BranchTaken = 1  or
                InstructionEX = JumpRegType                          or
                InstructionID = Jump                                 else
            0
```

The logic for *ImmSel* is:

```
ImmSel <=  1   when InstructionEX = LinkType                         else
           0
```

Likewise, internally at the Control Unit, a *ClearID* signal is used to flush all signals the goes to the PEs. This signal obeys the logic:

```
ClearID <=  1   when InstructionEX = BranchType and BranchTaken = 1  or
                    InstructionEX = JumpRegType                      else
            0
```

Note that *ClearIF* and *ClearID* have different logic, because a jump event is resolved at the decode stage, therefore, there is no need to flush instructions at the ID stage when that type of instruction is executed.

5 RESULTS

Now that ReFreeMIPS has been presented, it is fundamental to validate our hardware design (TASIRAN; KEUTZER, 2001). The architecture was prototyped in a Xilinx Virtex-5 xc5vlx110t FPGA (XILINX INC, 2010) and simulations were performed in ISIM Simulator (XILINX INC, 2012). Results regarding frequency and area were measured and compared to the original ReFree implementation. For performance analysis, a standard MIPS was used as baseline.

5.1 Frequency and Critical Path

Frequency and critical path of the system are considered as a metric to measure a well-designed architecture. One of our goals was to enhance the system architecture without causing a significant loss in performance. Table 5.1 illustrates the result of frequency and critical path of the proposed solution in comparison to the original ReFree design, showing that frequency went from 74.00 MHz to 72.20 MHz. There had been a reduction in terms of operating frequency, mostly due to the new functionalities added to the FUs. For instance, Mul PE needs a 32-bit x 32-bit multipler, since original ReFree only performs 32-bit x 16-bit operations. Some airthmetic and logic operations also had to be implemented in ALU units.

In spite of some units had shown critical path increase, the overall critical path was not

Table 5.1 - Critical Path and Frequency Comparison

	ReFreeMIPS		Original ReFree	
	Critical Path (ns)	Frequency (MHz)	Critical Path (ns)	Frequency (MHz)
IF	3.46	289.0	1.46	684.93
ID	9.62	103.95	1.40	714.28
Crossbar	3.32	301.20	3.32	301.20
ALU PE	6.13	163.13	5.41	184.84
Mem PE	3.509	284.90	3.20	312.5
Mul PE	9.11	109.76	4.77	209.64
ReFree	13.85	72.20	13.513	74.00
System	13.85	72.20	13.513	74.00

Source: Created by the author.

compromise as Xilinx toolchain makes optimization, and, thus, minimizes the frequency gap between the two architectures. Moreover, table 5.1 demonstrates that our presumption of EX and MEM stage as a unique stage was true, as the critical path is represented by the ALU and Multiplier and not the memory.

5.2 Area

Another factor to be considered is how integrating MIPS into the architecture affected area utilization. Table 5.2 shows the results regarding area utilization for ReFreeMIPS and original ReFree, and also compares the overhead caused .

Table 5.2 - Area utilization on ReFreeMIPS and Original ReFree

	ReFreeMIPS		Original ReFree		ReFreeMIPS / ReFree Ratio	
	FFs	LUTs	FFs	LUTs	FFs	LUTs
IF	12	39	N/A	N/A	N/A	N/A
ID	368	1096	0	444	N/A	2.46
Crossbar	0	2560	0	2560	-	1
ALU PE	64	809	64	744	1	1.08
Mem PE	64	273	64	207	1	1.31
Mul PE	187	438	83	201	2.25	2.17
ReFree	1166	16550	1062	15088	1.09	1.04
System	1546	17685	1062	15532	1.45	1.14

N/A - Not applicable

Source: Created by the author.

No measurements for IF in Original Refree was done, since there is no logic nor circuit for this unit, as this stage basically consists of an instruction memory, an address input and instruction output signals. Furthermore, the overall area increase shows that 45% FFs and 14% look-up tables (LUTs) were added to the architecture, in order to add MIPS compatibility. This number

5.3 Performance

A set of benchmarks was used to attest the correctness of our system: Adpcm, Cjpeg, DFT, Matrix and x264. We have also used these benchmarks to compare our solution with a standard MIPS implementation, showing the performance our architecture achieves.

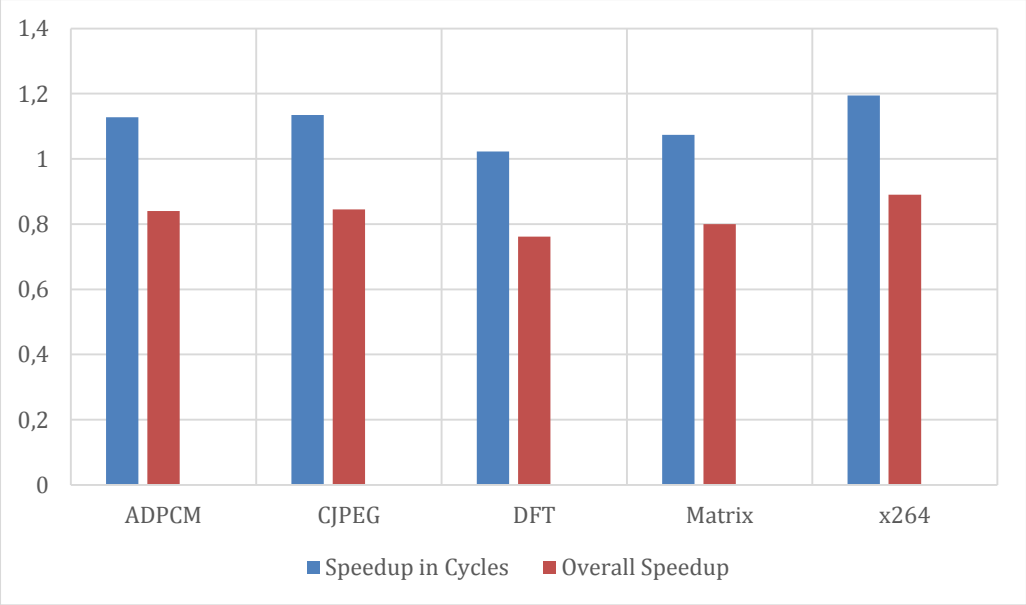
We have considered two scenarios: in the first, benchmarks were compiled using optimization flag *–O0* in GNU GCC Compiler, that is, with no code optimization. The second scenario considers benchmarks generated using the *–O2* optimization flag, hence, allowing GCC to optimize code. Results are displayed in terms of number of cycles and overall speedup, considering the standard MIPS as our baseline.

Figure 5.1 illustrates the first scenario. For benchmarks like x264 and Cjpeg, ReFreeMIPS might achieve as nearly as 20% better performance in number of cycles. However, the overall speedup is compromised due to its operating frequency, causing 20% slowdown on average.

Still, there are few considerations regarding this performance comparison:

1. The critical path of ReFreeMIPS is formed by a crossbar interconnection and a PE, therefore, ReFreeMIPS frequency will be considerably lower than a standard MIPS. Frequencies for ReFreeMIPS and the standard MIPS were measured as 72.2MHz and 96.61MHz, respectively.

Figure 5.1 - *–O0* Optimization Results



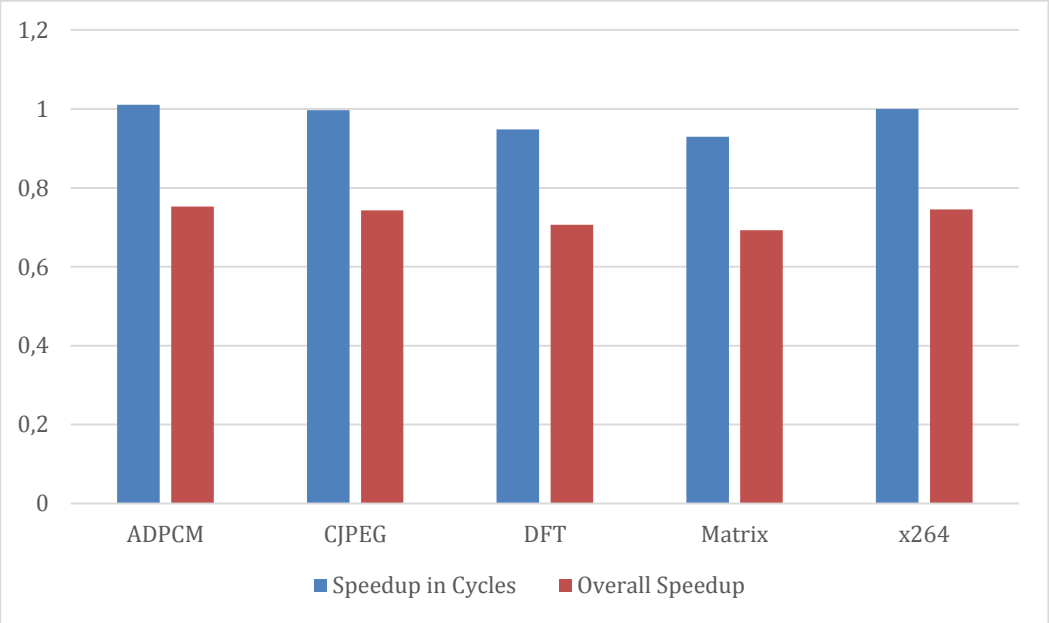
Source: Created by the author

2. Both architectures issue one instruction per cycle. There would be no huge performance increasing when executing in ReFreeMIPS over regular MIPS, as the major difference in the execution of MIPS instructions are how hazards are treated. A standard MIPS processor needs to stall the pipe when an instruction tries to read a register following a load instruction that writes the same register (PATTERSON; HENNESSY, 2009).
3. ReFreeMIPS is, in fact, a supplement to ReFree. MIPS mode should mostly be used in balance with ReFree mode. Using MIPS mode uniquely would mean wasting lots of resources only to issue a single instruction per cycle.

There is even more gap in the second scenario, shown in Figure 5.2. The compiler tries to eliminate data dependencies between instructions, favoring the standard MIPS implementation. Because of that, performance in MIPS increases and overtakes ReFreeMIPS, causing slowdown in overall speed up and number of cycles.

In terms of performance improvements, there appears to be no advantages regarding the integration of MIPS on ReFree. However, one must think that our architecture is still capable of executing up to 16 instructions per cycle, when operating in ReFree mode. Moreover, there is no need to reload registers with different values when switching between modes, since both operate using the same registers and the state of the processor is always preserved.

Figure 5.2 - *–O2* Optimization Results



Source: Created by the author

6 CONCLUSION

CGRAs normally include Global Register Files as a key component to share data between multiple PEs within the architecture. Although they offer flexibility, high bandwidth is required in order to provide a fast access to data. Refree, particularly, uses a different approach where no register file is needed. This architecture was primarily designed to explore parallelism in multimedia applications by executing only data-flow instructions.

This work proposed to extend the architecture by adding support to MIPS-I ISA, therefore, allowing the execution of control-flow instructions. This integration permits ReFree to be used not only as a coprocessor, but as the main processor of a system. In addition, the proposed architecture, ReFreeMIPS, executes instructions in the same PEs used for ReFree parallel execution mode.

ReFreeMIPS was prototype on the top of a FPGA and results shown that an increasing of 45% in FFs and 14% in LUTs were necessary to implement the solution. During performance analysis, the architecture was compared to a standard MIPS implementation and results shown an expected slowdown, since ReFreeMIPS frequency is around 25% less than a standard MIPS.

6.1 Future Works

Though it is very powerful, ReFree parallel mode needs a 512-bit instruction word to operate . If the compiler cannot find high ILP on the code, most PEs will be executing NOP operations, and therefore, waste memory resources. On the other hand, MIPS only needs 32-bit of instruction to execute.

As future work, we will combine the benefits of both ReFree and MIPS ISAs, which ReFreeMIPS executes, in order to provide a good balance between code density and performance, since ReFree ISA can issue up to 16 instructions per cycle and MIPS ISA instructions only occupies 32 bits per instruction. We may use code analysis to determine when each ISAs should be used, depending on how parallel our applications are. If one gets to find the optimal balance between two ISAs, applications may be both efficient in terms of performance and compact, when it comes to memory resource utilization.

REFERENCES

BECK, A. C. S.; CARRO, L. **Dynamic Reconfigurable Architectures and Transparent Optimization Techniques**. Berlin/Heidelberg: Springer, 2010.

CHANG, P. P. et al. IMPACT: An architectural framework for multiple-instruction-issue processors. **International Symposium on Computer Architecture**, May 1991. 266-275.

CONG, J. et al. A Fully Pipelined and Dynamically Composable Architecture of CGRA. **IEEE International Symposium on Field-Programmable Custom Computing Machines**, 2014. 9-16.

FERREIRA, R. et al. A Just-In-Time Modulo Scheduling for Virtual Coarse-Grained Reconfigurable Architectures. **International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)**, 2013. 188 - 195.

FISCHER, J.; Young, C. **Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools**. Morgan Kayfmann, 2004.

HEWLETT-PACKARD LABORATORIES. VEX Toolchain. Available at: <http://www.hpl.hp.com/downloads/vex/>. Acessed: October 17 2014.

KWOK, Z.; WILTON, S. J. E. Register file architecture optimization in a coarse-grained reconfigurable architecture. **IEEE Symposium on Field-Programmable Custom Computing Machines**, April 2005. 35-44.

LAM, M. S. Software pipelining: an effective scheduling technique for VLIW machines. **SIGPLAN Conference on Programming Language Design and Implementation**, 1988. 318-328.

MEI, B. et al. DRESC: a retargetable compiler for coarse-grained reconfigurable architectures. **IEEE Internarional Conference on Field-Programmable Technology**, 2002. 166-173.

MEI, B. et al. ADRES: An Architecture with Tightly Coupled VLIW Reconfigurable Processor and Coarse-Grained Reconfigurable Matrix. **International Conference on Field Programmable Logic and Applications (FLP)**, September 2003. 61-70.

MIPS TECHNOLOGIES, INC. MIPS32™ Architecture For Programmers Volume II: The MIPS32™ Instruction Set, 2001.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design - The Hardware and software Interface**. 4. ed. [S.l.]: Elsevier, 2009.

RAU, B. R. Iterative modulo scheduling: an algorithm for software pipelining loops. **International Symposium on Microarchitecture**, November 1994. 63-74.
SHANNON, L. Reconfigurable Computing Architectures. In: HAUCK, S.; DEHON, A. **Reconfigurable Computing - The Theory and Pratice**. Burlington: Elsevier, 2008. Cap. 2, p. 29-44.

SINGH, H. et al. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. **IEEE Transactions on Computers**, v. 49, n. 5, p. 465 - 481, May 2000.

TASIRAN, S.; KEUTZER, K. Coverage Metrics for Functional Validation of Hardware Designs. **IEEE Design & Test of Computers**, v. 18, n. 4, p. 36-45, 2001.

VASSILIADIS, S.; SOUDRIS, D. **Fine- and Coarse-Grain Reconfigurable Computing**. New York: Springer, 2007.

WONG, S.; VAN AS, T.; BROWN, G. ρ-VEX: A Reconfigurable and Extensible Softcore VLIW Processor. **IEEE International Conference on Field-Programmable Technology (ICFPT)**, December 2008. 369-372.

XILINX, INC. ISim User Guide, April 2012. Available at: <http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/plugin_ism.pdf>. Acessed: November 11 2014.

XILINX, INC. IP Processor Block RAM, March 2011. Available at: <http://www.xilinx.com/support/documentation/ip_documentation/bram_block.pdf> Accessed: November 12, 2014.

XILINX, INC. Virtex-5 FPGA Data Sheet: DC and Switching Characteristics, May 2010. Available at: <http://www.xilinx.com/support/documentation/data_sheets/ds202.pdf >. Accessed: November 11, 2014.

APPENDIX A: PROJECT DESCRIPTION (TG1)

RefreeMIPS - A CGRA-based MIPS architecture

Tiago Trevisan Jost

Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)

Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

tiagotrevisanjost@gmail.com

Abstract. *Reconfigurable computing is gaining lots of attentions lately, especially CGRA (Coarse-grained Reconfigurable Architecture), devices that exploit reconfigurability to achieve high performance and efficient power consumption. They are considered as powerful as ASICs and much more flexible, enabling reconfiguration on-the-fly to maximize performance and minimize power consumption. This project will focus on the development of the RefreeMIPS, a MIPS-based architecture that is tightly coupled to a register file free CGRA.*

1. Introduction

During the past decades, innumerable advances have been made in computer architecture and organization. Computer architects are taking advantage of today's nanometer technology to build smaller, more compact and powerful chips. Most of those chips are designed as ASICs (Application-specified Integrated Circuits) and microprocessors. ASICs are high performance devices used to execute specific-purpose tasks. Microprocessors, on the other hand, are more flexible than ASICs, but not quite as powerful. By combining the high performance of ASICs and the flexibility of microprocessors, FPGAs (Field-programmable Gate array) have made possible new types of applications, since they can be reprogrammed to perform theoretically any digital circuit [Beck and Carro 2010]. Traditional FPGAs have also made possible to design systems that adapt themselves according to the software they are executing. This new approach is best known as Reconfigurable Computing.

Reconfigurable Architectures can be classified according to a different set of categories, such as, granularity, reconfiguration models, among others. Granularity refers to the smallest block of which a reconfigurable device is made. Reconfigurable architectures can be divided into two major groups according to their granularity, fine-grain and coarse-grain systems.

Fine-grain reconfigurable architectures are systems that can be reconfigured at a bit level. They are characterized by being very flexible, however, with a high reconfiguration overhead due to the size of its configuration memory. On the other hand, Coarse-grain architectures manipulate larger reconfigurable blocks, such as, ALUs and Memories. Despite being less flexible, they are more suitable for performing word-level data processing [Vassiliadis and Soudris 2007], since fewer bits are needed to reconfigure the system, improving performance and area utilization.

According to the reconfiguration models, architectures can be classified as statically or dynamically reconfigurable. Statically reconfigurable architectures can only be programmed

once for its running time. In order to reconfigure the system, it has to be halted and apply a new configuration to it. Dynamically reconfigurable architectures are much more flexible and a new configuration can be loaded or unloaded during the operation of the system.

According to [Patterson and Hennessy 1996], most programs tend to reuse data and instructions they have used recently. A general rule of thumb is that most applications spend 90% of its running in 10% of the code. This special program property is known as the *Principle of Locality* and architects design systems to take advantage of this property. If one gets to find parallelism in this small piece of code, a considerable improvement in performance is achieved. Multimedia and digital processing applications make good use of inner-loops, therefore, CGRA is a good option to accelerate such applications.

In this work, we will be focusing on developing a MIPS architecture that is tightly coupled to a CGRA called Refree. Register file free, a.k.a. Refree, was proposed here at the *Laboratório de Sistemas Embarcados (UFRGS)* and is composed of a set of dynamically reconfigurable Functional Units that can be used to accelerate different parts of an application. We will discuss Refree later in this text. The main purpose of this project is to show how a regular processor like MIPS can benefit from the integration with a dynamically reconfigurable device.

The remaining of this work is organized as follows: Section 2 presents an overview of CGRAs and its main characteristics. Section 3 discusses the Refree Architecture. Section 4 gives a brief overview of the MIPS Architecture. Section 5 presents the project and the methodology of the work. Section 6 presents the chronogram regarding the next steps of this research. Finally, Section 7 concludes this work.

2. CGRA - Overview

Increasing performance of a system is not always an easy task to achieve. Architects utilize different hardware to accomplish that goal. GPUs, for example, are suitable for running vectorized applications and CPU are the best for control-based applications. When it comes to data-flow applications, Coarse Grain Reconfigurable Architectures can be used. CGRA are specialized hardware best suitable to run intensive inner-loop applications [Ferreira et al. 2014]. Since they are reconfigurable, a different configuration can be applied on-the-fly allowing the device to extend its functionality and even reducing power consumption. If compared to FPGAs, CGRAs reduce considerably the mapping process, at the cost of flexibility.

A CGRA is usually composed of a set of Processing Elements (PEs), also called Functional Units (FUs), a global register file and a configuration Memory, as depicted in Figure 1. The PEs can be connected through different networks, such as, crossbar and mesh. The configuration memory is responsible for configuring the units and register file to manipulate data that flows through the PEs. This configuration memory is previously generated using a compiler, which identifies the portion of code it can be parallelized, generates the proper configuration bits for each PE and handles any data dependency that might be associated with the code. As it will be seen shortly, a global register file is not required, however, most of the architectures use it for simplicity on their configuration.

A Processing Element can perform different operations, such as, ALU operations, memory operation, etc. Homogeneous systems are characterized by having PEs of the same type. Heterogeneous systems, however, have PEs of different type. As it will be shown later, the Refree Architecture can be used in either approach.

Regarding the host processor integration, CGRAs can be classified as *tightly coupled* or *loosely coupled*. In *loosely coupled* systems, the CGRA is used as an application-specific unit or external accelerator, connected to the main processor via a bus, and operating asynchronously. The main processor will enable its units and supply it with a set of

parameters. The host will be able to perform other tasks while the CGRA is also executing, improving performance. However, since they operate asynchronously, a resynchronization mechanism must be added when the CGRA units finish executing. In *tightly coupled* systems, the CGRA is treated as an internal functional unit of the microprocessor, avoiding the need for a bus interconnection and the communication latency between host and CGRA. When there is a reconfigurable unit working as functional unit in the main processor, it is called a Reconfigurable Functional Unit, or RFU [Beck and Carro 2010].

Choosing between a *loosely coupled* or *tightly coupled* approach is a design decision. *Loosely coupling* is often preferred when the bus connection is not a bottleneck and area limitation is not required, since loosely coupled approach consumes more area. *Tightly coupling* minimizes the overhead on the communication bus and allows the CGRA and microprocessor to share resources, minimizing area consumption [Beck and Carro 2010].

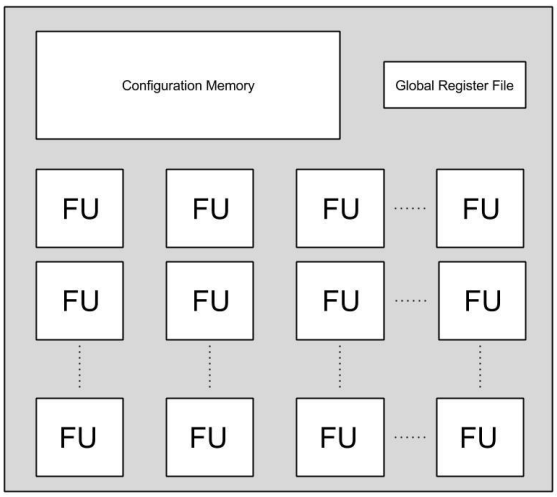


Figure 8. A typical CGRA structure

The beauty of CGRA is its capability of reconfiguration on-the-fly. As already mentioned, a good compiler is a key component on that process. If the compiler cannot find enough instructions to fill the PEs with, meaning there are too many dependencies, resources will be wasted, compromising area utilization. Different techniques are been explored to deal with those problems, such as, loop unrolling, software pipeline [Rau 1994], among others. Applying such techniques improve hardware usability and performance. Despite being of great interest in the academy, this will not be the focus of this work.

Countless reconfigurable architectures were proposed in the past decade to accelerate execution of data-flow application, however, they haven't been used in mainstream application mainly due to their architectural complexity and lack of automated tool support [Vassiliadis and Soudris 2007]. Two of the most successful architectures are ADRES (Architecture Dynamically Reconfigurable Embedded Systems) [Vernalde et al. 2003] and r-Vex.

ADRESS consists of a VLIW processor and a coarse-grain reconfigurable matrix that has direct access to register files, caches and memories [Vassiliadis and Soudris 2007], as illustrated in Figure 2. The architecture is configured using a XML template, where the characteristics of the architecture should be described, such as, communication topology, resource allocation, etc. [Vernalde et al. 2002]. This file is analyzed by DRESC (Dynamically Reconfigurable Embedded Systems Compiler), a retargetable compiler that uses a modulo scheduling (MS) algorithm to map efficiently the code, exploring the reconfigurable array. More information on ADRES and DRESC can be found in [Vernalde et al. 2003] and [Vernalde et al. 2002].

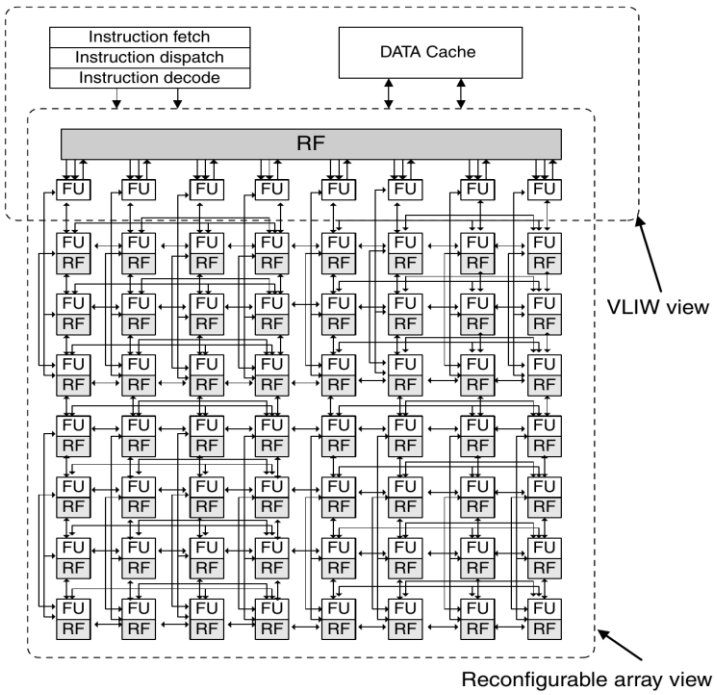


Figure 2. ADRES Architecture. Reprinted from [Vassiliadis and Soudris 2007]

Another important architecture is ρ-VEX, a reconfigurable and extensible Very Long Instruction Word (VLIW) processor. ρ-VEX architecture is based on the VEX Instruction Set Architecture (ISA), which offers a scalable technology platform that allows variation in many aspects, including instruction issue width, organization of functional units, and instruction set [Wong, van As and Brown 2008]. It uses a four-stage pipeline design, as illustrated in Figure 3. The figure shows a 4-issue configuration composed by 4 ALU units, 2 multiplier units, 1 branch control unit, 1 memory access unit, however, its issue width can vary between 1, 2 and 4. This architecture will be of particular interest, since it will be compared to our RefreeMIPS architecture. In spite of that, more detailed information about this architecture can be found in [Wong, van As and Brown 2008].

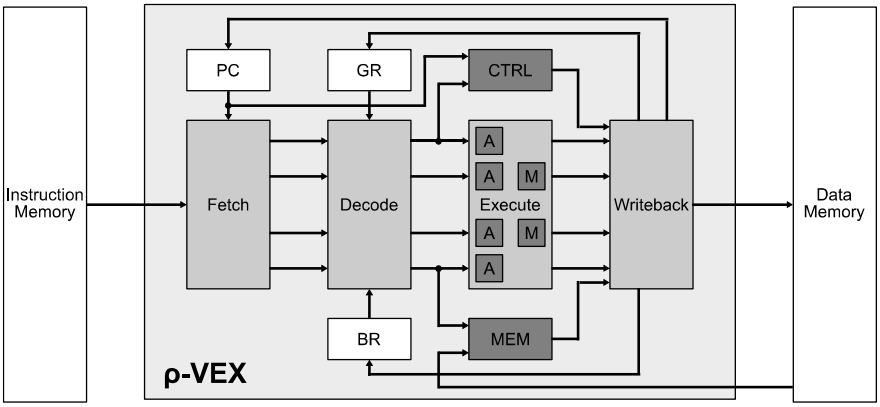


Figure 3. *ρ*-VEX Processor. Reprinted from [Wong, van As and Brown 2008]

3. Refree Architecture

The Register File Free Architeture, or Refree, was proposed here at the Instituto de Informática – UFRGS and designed to run in the top of an FPGA. The idea behind this new architecture is to eliminate the bottleneck caused by accessing a global register file (GRF), that would be shared among PEs in the CGRA. Instead of having a GRF visible for all PEs, each PE consist of an Internal Functional Unit, two registers and a table of constant immediate values, as depicted in Figure 4.

When a PE needs to be used to execute an operation, data is directly forwarded to those two registers and control signals are properly configured. It is worth mentioning that any register value can be forwarded to any other register inside the architecture, as it will be shown later on. Also, a value that comes from port A can easily be used in register B and vice-versa. This approach does not restrict the use of registers in any way, since data can be rotate to any port of a functional unit.

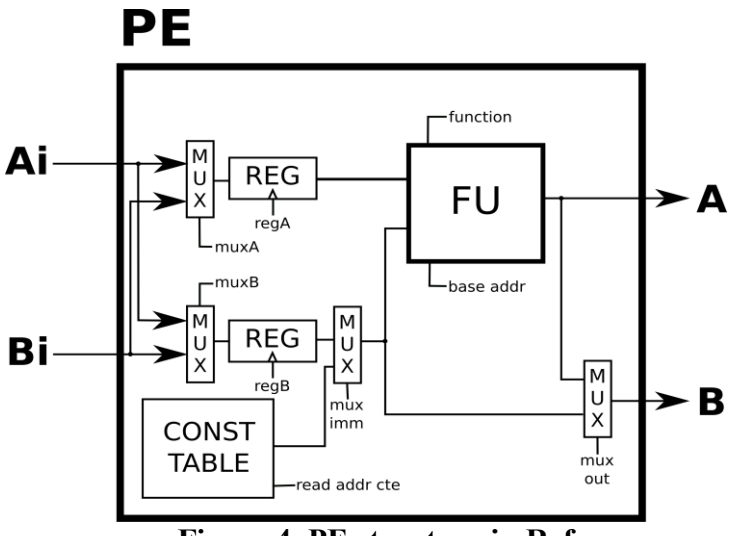


Figure 4. PE structure in Refree

Another important characteristic of Refree is related to the interconnection network. A crossbar network interconnects the PEs units, i.e., all PEs are connected to one another, simplifying the placement and routing process. Although the mapping is NP-complete even for crossbar-based CGRAs, experimental results in [Ferreira et al. 2013] demonstrate a huge reduction in compilation time and low area overhead if compared to mesh-based CGRAs. The

crossbar is divided into two networks, one for port A and one for port B. This approach reduces area utilization by a factor of four if compared to a full crossbar network.

The Functional Units can be configured to operate as memory interface, ALU or multiplier. If set to be a memory interface unit, *base addr* port defines the base address for memory operations. Registers A and B will carry the values for data to be written in memory and address offset, respectively. The constant immediate table holds up to eight predefined immediate values that may be used to operate as one of the sources in the ALU. The compiler might use this table to perform operations without having to rotate immediate values coming from the current instruction, simplifying the mapping process.

Because Refree will be integrated with MIPS, some necessary changes will have to be made inside the PEs to maintain compatibility. Other muxes will be added and some adaptations will be done. Section 5 is presented with a deeper focus on these modifications.

4. MIPS Architecture

Considering we will be designing a CGRA-based MIPS processor, it is fundamental to understand the basic characteristics of the MIPS architecture. MIPS (Microprocessor with Interlocked Pipeline Stages) is a 32-bit RISC-based (Reduce Instruction Set Computer) architecture designed primarily for academic purpose at Stanford University. Despite becoming a commercial product in middle 80's and many other improvements on the architecture were added, we will be focusing on the first ISA (Instruction Set Architecture) of MIPS.

MIPS includes 32 general-purpose register 32-bits wide (R0-R31). These registers are located in a Global Register File with two read ports and one write port, which might limit the architecture and cause control hazards in it. All instructions in MIPS are 32-bit long and they can be divided into three categories, as illustrated in figure 5. Each category represents a set of instructions on the ISA and bits on the instruction word are treated differently according to this representation. Basic ALU instructions with two registers as operands are from type R. I-type holds instructions with immediate operand and branches. Jump instructions belongs to the J-Type.

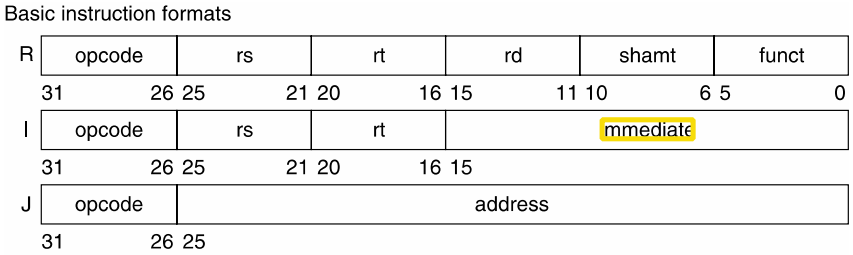


Figure 5. Instruction Categories

Pipelining is an important technique highly explored in almost any architecture nowadays. MIPS implements a five-stage pipeline that is briefly described below:

- Instruction Fetch - the Program Counter is used as the address of the instruction to be fetched in the instruction memory. The value is stored in the Instruction Register (IR) and the PC is incremented by 4 (the memory is byte-oriented and 4 bytes are needed for each memory);
- Instruction Decode and Register File Read - instruction is decoded and the register file is read to obtain the proper registers to be used in the next stage;
- Execution - stage where, in fact, the instruction gets executed;
- Memory access - load and store instructions use this stage to perform memory operations on the data memory;

- Write-back - this stage, if needed, update registers modified at the execution stage.

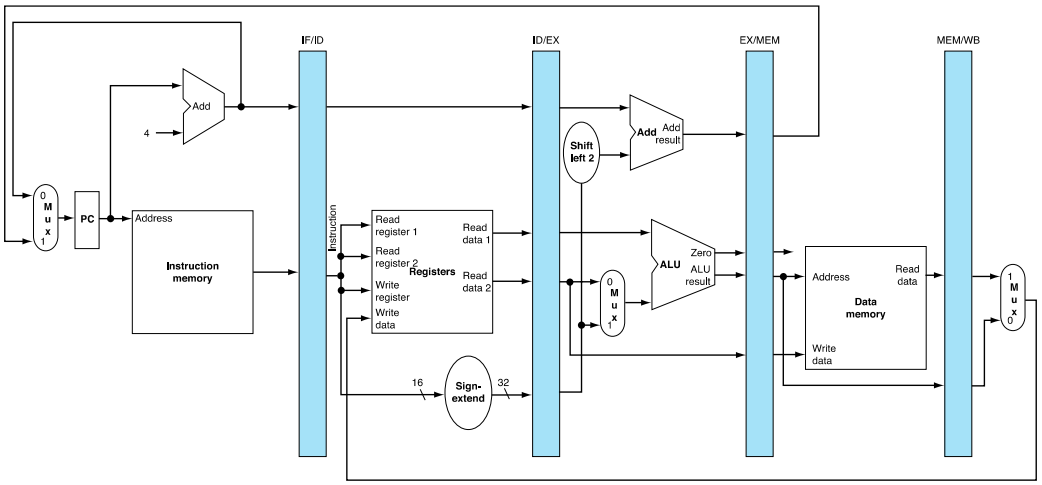


Figure 6. MIPS Pipeline. Reprinted from [Patterson and Hennessy 1998]

Figure 6 above illustrates the five-stage pipeline, however, it is only a simplification of the architecture. Hazard treatment, bypassing and forwarding are not represented in this figure. More information in MIPS architecture is found in [Patterson and Hennessy 1996] and [Patterson and Hennessy 1998]. Nonetheless, it gives a good idea of how the pipeline is used in MIPS and how it will be the basic pipeline structure of RefreeMIPS.

5. RefreeMIPS

In this section we will describe the methodology use to develop RefreeMIPS. The project will be divided into two distinct phases. The first phase will focus on the development of a monocycle RefreeMIPS processor. The second will consist on the development of a pipelined architecture, similar of what MIPS offers. The project will be implemented using VHDL language and Xilinx ISE Design Suite.

One of the main advantages of tightly coupled architectures is its proximity to the microprocessor, which decreases communication overhead. However, the area taken by the processor itself may be compromised. In order to minimize area, the global register file from MIPS will be exploded and will correspond to the 32 registers spread throughout the 16 PEs of Refree, namely PE0-PE15. With this new configuration, multiple writes can be performed at the same time, eliminating the limitation of the register file. PE0 consists of a memory interface unit to an external data memory and the others are configured as ALUs units.

RefreeMIPS will be fully compatible with regular MIPS instructions and also with the new CGRA instruction. When working as regular MIPS, two PEs will be used, one as an interface to an external data memory (PE0) and one for executing instructions (PE1), since only one instruction is issued at every cycle. When working as CGRA, as many as 16 instructions may be issued at every cycle, 15 performing ALU operations and one performing memory operation. The speedup when running as CGRA will depend on how easily the compiler can find instructions to fill the empty PEs.

Two instruction memories are necessary, one for regular MIPS instructions and another for the CGRA instructions. To be able differentiate when each memory should be accessed, two instructions will be used to inform that a new instruction should be fetched at a different memory. For example, a 32-bit instruction will be added to the MIPS ISA to inform

that the next instruction should be fetched from the CGRA instruction. This same approach will be used for the CGRA.

5.1. Monocycle RefreeMIPS

Some characteristics are specifically of a monocycle design. If one takes the MIPS stages as an example, one sees that only at the end of the last stage a register ought to be written. This means that all five stages will be executed at one single cycle. From a research perspective, it was useful to start developing a monocycle version of RefreeMIPS before adventuring into a pipeline approach. This will give a full understanding on the major problems faced during the implementation of the pipelined architecture.

At the decode process, signals will be generated to rotate the registers to the specific PEs and no bypassing is required for this case. It is very straightforward, since signals need not to propagate through other clock cycles. Nonetheless, modifications inside the PEs are needed for both monocycle and pipelined versions. Figures 7 illustrate these modifications. Muxes are added in both PEs types to ensure that bypassing can be performed, without the need of passing to a register. Figure 7a also shows muxes at the end of the PE due to necessity of bypassing the memory.

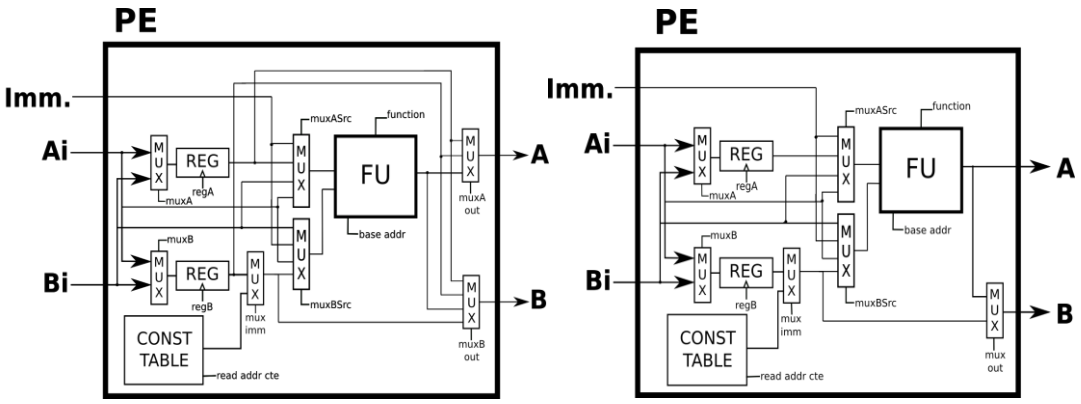


Figure 7. (a) Memory PE **(b) ALU PE**

5.2. Pipelined RefreeMIPS

After concluding the monocycle architecture, RefreeMIPS will be extended to a pipelined approach. More complexity will be added in the development of the project. A five-stage pipeline, much like depictured in Figure 5, characterizes the architecture. Nevertheless, stages 3 to 5 will be inside the Refree component and PEs will be similar to those described in Figure 7.

Pipes will be connected at the end of PEs 0 and 1, because those are only ones used by MIPS instruction. There is no need for adding pipes at the end of other PEs, mostly because synchronized registers are inside them. It is also worth mentioning that other changes might be considered during implementation, since one cannot estimate if the solution will work properly.

To ensure the good performance of the proposed architecture, a comparison with ρ-VEX will be done. Benchmarks are still to be chosen.

6. Chronogram

A chronogram of the activities that will be developed in this work is display in Table 1. It is important to clarify that Refree has already been implemented, however, modifications are needed to make it compatible with MIPS.

Table 1. Activities Chronogram

Activities	Jul.	Aug.	Sep.	Oct.	Nov.	Dec.
Adapting PEs from Original Refree	X					
Monocycle Architecture	X	X				
Pipelined Architecture		X	X	X		
Comparison with ρ-VEX				X	X	X

7. Conclusion

CGRAs normally include Global Register Files to share data between PEs. Refree, particularly, uses a different approach where no register file is used. By integrating Refree with MIPS, a range of applications and benchmarks can used to verify performance and efficiency. This project will attest whether this new CGRA integrated with MIPS is efficient and faster than ρ-VEX, a VLIW processor with reconfigurable instructions.

References

Beck, A. C. S. and Carro, L. (2010) “Dynamic Reconfigurable Architectures and Transparent Optimization Techniques”, In: Springer, Berlin/Heidelberg, Germany.

Vassiliadis, S. and Soudris, D. (2007) “Fine- and Coarse-Grain Reconfigurable Computing”, In: Springer, New York, NY, USA.

Ferreira, R., Denver, W., Pereira, M., Quadros, J., Carro, L. and Wong, S. (2014) “A Run-Time Modulo Scheduling by using a Binary Translation Mechanism”, In: SAMOS, Greece.

Rau B. R. (1994) “Iterative modulo scheduling: an algorithm for software pipelining loops,” In: Proc. MICRO, pp. 63–74.

Mei B., Vernalde S., Verkest D., De Man H., and Lauwereins R. (2003) “ADRES: An Architecture with Tightly Coupled VLIW Processor and Course-grained Reconfigurable Matrix”

Mei B., Vernalde S., Verkest D., De Man H., and Lauwereins R. (2002) “DRESC: a retargetable compiler for coarse-grained reconfigurable architectures,” in Proc. FPT, pp. 166 – 173

Ferreira R., Duarte V., Meireles W., Pereira M., Carro L., and Wong S. (2013) “A just-in-time modulo scheduling for virtual coarse-grained reconfigurable architectures,” In: Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII).

Patterson, D.A., Hennessy, J.L. (1996) “Computer Architecture: A Quantitative Approach”. Morgan Kaufmann Publishers, Inc.

Patterson, D.A., Hennessy, J.L. (1998) “Computer Organization and Design: The Hardware/Software Interface”. Morgan Kaufmann Publishers, Inc.

Wong, S., van As, T. and Brown, G. (2008) “ρ-VEX: A Reconfigurable and Extensible Softcore: VLIW Processor” In proceeding of: ICECE Technology, FPT 2008.